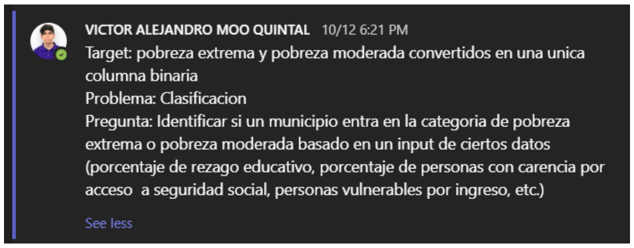
# Course: Machine Learning
# Unit 2: Implementing a Predictor from scratch.

Name: Victor Alejandro Moo Quintal
Group: IRC 9B
e-mail: 2009098@upy.edu.mx

## *Selected target and problematic*



VICTOR ALEJANDRO MOO QUINTAL 10/12 6:21 PM
Target: pobreza extrema y pobreza moderada convertidos en una unica columna binaria
Problema: Clasificacion
Pregunta: Identificar si un municipio entra en la categoria de pobreza extrema o pobreza moderada basado en un input de ciertos datos (porcentaje de rezago educativo, porcentaje de personas con carencia por acceso a seguridad social, personas vulnerables por ingreso, etc.)

As the problem describes, the targets selected for this implementation were the columns 'pobreza_e' and 'pobreza_m', because these columns represent the percentage of poverty in the specific municipality; thus, and algorithm to predict if a municipality belongs to "no poverty", "moderate poverty" or "extreme poverty" could be created using any multi-classification model. However, these features are numerical, so the idea was to transform these values into categorical by doing a pre-classification by math.
This process consisted in create a label column with information from the features 'pobreza', 'pobreza_e' and 'pobreza_m'; then, classify the municipalities by the following conditions:

No poor (label 0): if the municipality has less than 50% in the 'pobreza' feature.
Moderate poor (label 1): if the municipality has 50% in the 'pobreza' feature, and more percentage of 'pobreza_m' than 'pobreza_e'.
Extreme poor (label 1): if the municipality has 50% in the 'pobreza' feature, and more percentage of 'pobreza_e' than 'pobreza_m'.

## *Journal Description*

### i. *Preparing the dataset:*

"Indicadores de pobreza, pobreza por ingresos, rezago social y gini a nivel municipal,1990, 2000, 2005 y 2010" dataset consists in 139 features (divided in 134 numerical and 5 categorical), and 2'456 observations which include 56 missing values which had to be handled to fill the information. This dataset represents and analyze different socio-economic indicators per municipality in Mexico. Most of the values are related mainly to poverty, income, economic conditions, vulnerability, etc.

After generating the label column and extracting the required information, some columns were erased: 'ent', 'nom_ent', 'mun', 'clave_mun', 'nom_mun'. These features are not necessary because they include information related to identification of the municipalities and states, however this information could affect the results by overfitting the results. First, the categorical information should be converted to numerical, it means that 'hot-encoding' would be applied to the previously mentioned features, converting thousands of new features to put more attributes into the dataset. These would happen for the bast majority of these features, also, this information is not needed because the poverty does not depend on the name of the municipality, nor states or identification number; It depends on the other attributes. Another feature that had to be deleted is 'gdo_rezsoc00' because it included empty values, and these could not be filled in the following step.
Now, as was mentioned previously, there are 56 missing values that could cause problems to the model. The way to handle these values was by median using Pandas, because this dataset has a non-symmetric distribution. After filling these values, the last step for the preprocessing was to convert the 2 categorical values to numerical. The way to address this was by an ordinal mapping, scoring the categorical values 'Muy bajo', 'Bajo', 'Medio', 'Alto', 'Muy alto' as 1, 2, 3, 4 and 5, respectively.

*ii. Training the predictor:*

Since this is a multi-class classification, the 'KNN' and the Perceptron model were chosen. These algorithms are capable of predicting various classes using only math.

KNN model:

The 'KNN' model is basic, it uses the "Euclidean Distance" to measure the distance of all the data and the dataset, and selecting a 'K' number of data (49 in this case, because is obtained from the root of the total of observations) as the neighbors in order to determine its class, the higher number of neighbors with one label will determine the label of this observation. The "Euclidean Distance" formula is defined by:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In terms of Python code, the "Euclidian Distance" is defined by:

```
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))
```

Where the distance is computed by the square root of the sum of squared distances between the two coordinates. Once these distances are calculated, they are sorted in ascending order. From this sorted list, the '*k*' training samples closest to the test point are selected. The function then counts the number of occurrences of each class label among these '*k*' nearest neighbors. The final prediction for the test sample is the class label that appears most frequently among these neighbors. In essence, the majority class among the '*k*' nearest neighbors determine the class of the test sample.

Perceptron Model:

The perceptron is a binary classification algorithm that computes a linear combination of its input features and associated weights, then passes this value through an activation function to produce a binary output. In this implementation, the perceptron was used as a multi-class classification by applying the model two times.

The basic perceptron formula is defined by:

$$z = w1x1 + w1x2 + \ldots + wnxn = X^T W$$

The perceptron function defined in the code has the following components:

```
def train_perceptron(data, labels,
learning_rate=0.05, n_iters=1000):
    weights = np.zeros(data.shape[1])
    bias = 0
    for _ in range(n_iters):
        for idx in range(len(data)):
            linear_output =
np.dot(data[idx], weights) + bias
            y_predicted = 1 if
linear_output >= 0 else 0
            error = labels[idx] -
y_predicted
            weights += learning_rate *
error * data[idx]
            bias += learning_rate * error
    return weights, bias
```

Where the function initializes weights as zeros and bias as zero, then through a series of iterations, the perceptron adjusts its weights and bias based on the error between its predictions and the true labels. The weight updates are calculated by the learning rate and the calculated error.

Then, two perceptrons must be prepared. The first set of binary labels, 'is_poor_labels', distinguishes between the class '0' (no poor) and all other classes. The first perceptron is trained on this label set. The second perceptron is trained on a subset of the data, specifically data labeled as '1' (moderate poor) by the first perceptron, to further classify it. The label '2' does not belong to any specific perceptron because it is the remaining data.

```
is_poor_labels = np.array([0 if label ==
0 else 1 for label in train_labels])
weights1, bias1 =
train_perceptron(train_data,
is_poor_labels)
poor_data = train_data[is_poor_labels ==
1]
poor_labels = np.array([0 if label == 1
else 1 for label in
train_labels[is_poor_labels == 1]])
```

```
weights2, bias2 = 
train_perceptron(poor_data, poor_labels)
```

The prediction function computes the output of the perceptron for a given input:

```
def predict(x, weights, bias):
    linear_output = np.dot(x, weights) + 
bias
    return 1 if linear_output >= 0 else 0
```

### iii. *Evaluating the performance of the predictor*

KNN model:

The code evaluates the k-NN model's performance by testing its accuracy on a set of test data. For every sample in this test dataset, the model predicts a class label using the predict function. This predicted label is then compared to the true label associated with the test sample. Each time a prediction matches the true label, a counter is incremented. After processing all the test samples, the model accuracy is computed. This is done by taking the ratio of the number of correct predictions to the total number of test samples. The resultant accuracy is then presented in percentage form to provide a clear measure of the model's performance.

```
correct = 0
for i in range(len(test_data)):
    prediction = predict(test_data[i], 
train_data, train_labels, k)
    if prediction == test_labels[i]:
        correct += 1

accuracy = correct / len(test_data)
print(f"Accuracy: {accuracy*100:.2f}%")
```

Perceptron Model:

For evaluation, each test data point is passed through the first perceptron. If its prediction is ‘0’, that becomes the final prediction. If it predicts ‘1’, the data point is then classified by the second perceptron. The accuracy of this model is then computed by comparing the predictions to the true test labels.

```
correct = 0
for i in range(len(test_data)):
    is_poor = predict(test_data[i], 
weights1, bias1)
    if is_poor == 0:
        predicted_label = 0
    else:
        predicted_label = 
predict(test_data[i], weights2, bias2) + 
1
    true_label = test_labels[i]
    if predicted_label == true_label:
        correct += 1
accuracy = correct / len(test_data)
print(f"Accuracy: {accuracy*100:.2f}%")
```

### iv. *Predictor using libraries*

The libraries are the easiest form to apply these algorithms because these algorithms were previously coded, and Scikit Learn includes many of these in the package.

KNN model:

This model use the function ‘KNeighborsClassifier’ from Scikit Learn. Then, the number of neighbors, ‘k’, is set to 49 and the training is done by using the ‘fit’ method.

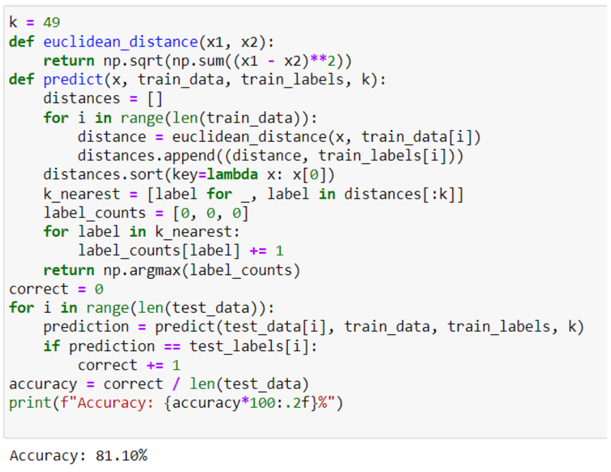
```
k = 49
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(train_data, train_labels)
```

With the k-NN model ready, predictions are made on the test data using the ‘predict’ method. The accuracy of these predictions is then computed by comparing them with the true labels of the test data using the ‘accuracy_score’ function. Finally, the calculated accuracy is printed out as a percentage.

```
predicted_labels = knn.predict(test_data)
accuracy = accuracy_score(test_labels, 
predicted_labels)
print(f"Accuracy: {accuracy*100:.2f}%")
```

KNN models comparisons:

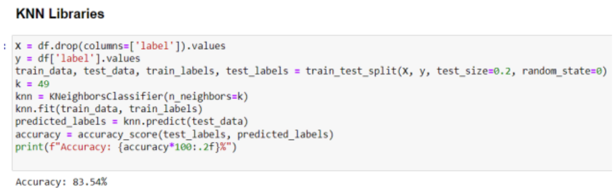
The results are similar, around 81% for the algorithm from scratch and around 83.5% for the algorithm using libraries. This shows that there is not a large difference between both operations, probably because this algorithms cannot be optimized and the operation performed is the same, however, the algorithm using libraries still has a better performance.



```
k = 49
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))
def predict(x, train_data, train_labels, k):
    distances = []
    for i in range(len(train_data)):
        distance = euclidean_distance(x, train_data[i])
        distances.append((distance, train_labels[i]))
    distances.sort(key=lambda x: x[0])
    k_nearest = [label for _, label in distances[:k]]
    label_counts = [0, 0, 0]
    for label in k_nearest:
        label_counts[label] += 1
    return np.argmax(label_counts)
correct = 0
for i in range(len(test_data)):
    prediction = predict(test_data[i], train_data, train_labels, k)
    if prediction == test_labels[i]:
        correct += 1
accuracy = correct / len(test_data)
print(f"Accuracy: {accuracy*100:.2f}%")
```

```
Accuracy: 81.10%
```

***FIgure 1. Accuracy of KNN model from scratch.***



KNN Libraries

```
X = df.drop(columns=['label']).values
y = df['label'].values
train_data, test_data, train_labels, test_labels = train_test_split(X, y, test_size=0.2, random_state=0)
k = 49
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(train_data, train_labels)
predicted_labels = knn.predict(test_data)
accuracy = accuracy_score(test_labels, predicted_labels)
print(f"Accuracy: {accuracy*100:.2f}%")
```

```
Accuracy: 83.54%
```

***FIgure 2. Accuracy of KNN model using libraries.***

Perceptron model:

This model uses the function 'Perceptron' from Scikit Learn. The code begins by transforming the labels in the 'train_labels' array into a binary format. Specifically, the label '0' remains '0', while all other labels are converted to '1'. This transformed label set is stored in the 'is_poor_labels' array. Then, 'perceptron1' is created and trained.

```
is_poor_labels = np.array([0 if label == 0
else 1 for label in train_labels])

perceptron1 = Perceptron()
perceptron1.fit(train_data,
is_poor_labels)
```

The data corresponding to the label '1' in 'is_poor_labels' is extracted and stored in the 'poor_data' variable. A new set of binary labels, named 'poor_labels', is then generated from this subset of data. In this new label set, the label '1' remains '1', and all other labels are converted to '0'. The second perceptron, named 'perceptron2', is then instantiated and trained using the 'poor_data' and its corresponding 'poor_labels'.

```
poor_data = train_data[is_poor_labels ==
1]
poor_labels = np.array([0 if label == 1
else 1 for label in
train_labels[is_poor_labels == 1]])
perceptron2 = Perceptron()
perceptron2.fit(poor_data, poor_labels)
```

To evaluate the model on the test data, predictions are first made using 'perceptron1'. For data points that 'perceptron1' predicts as '1' (indicating they belong to the subset that 'perceptron2' was trained on), 'perceptron2 'is used to further classify them. The predictions from 'perceptron2' are incremented by 1 to ensure they align with the original label scheme. The final predictions are then combined into the 'final_predictions' array.

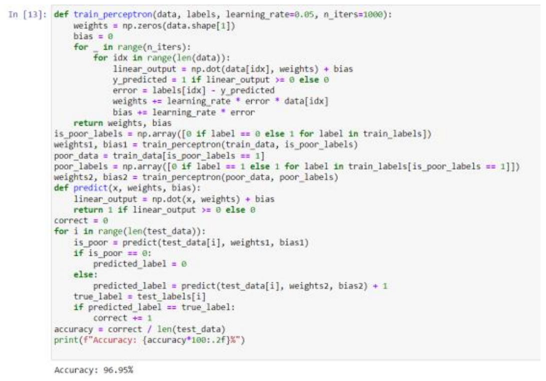
```
predictions_stage1 =
perceptron1.predict(test_data)
predictions_stage2 =
perceptron2.predict(test_data[predictions
_stage1 == 1]) + 1
final_predictions =
predictions_stage1.copy()
final_predictions[predictions_stage1 ==
1] = predictions_stage2
```

The accuracy is given by comparing the 'final predictions' with 'test_labels'.

```
accuracy = accuracy_score(test_labels,
final_predictions)
print(f"Accuracy: {accuracy*100:.2f}%")
```

Perceptron models comparisons:

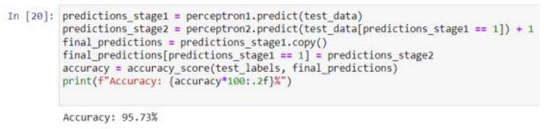
The results are very close, the difference is minimal, however it is very interesting the fact that the model from scratch obtained a higher result.



```
In [13]: def train_perceptron(data, labels, learning_rate=0.05, n_iters=1000):
             weights = np.zeros(data.shape[1])
             bias = 0
             for _ in range(n_iters):
                 for idx in range(len(data)):
                     linear_output = np.dot(data[idx], weights) + bias
                     y_predicted = 1 if linear_output >= 0 else 0
                     error = labels[idx] - y_predicted
                     weights += learning_rate * error * data[idx]
                     bias += learning_rate * error
             return weights, bias
         is_poor_labels = np.array([0 if label == 0 else 1 for label in train_labels])
         weights1, bias1 = train_perceptron(train_data, is_poor_labels)
         poor_data = train_data[is_poor_labels == 1]
         poor_labels = np.array([0 if label == 1 else 1 for label in train_labels[is_poor_labels == 1]])
         weights2, bias2 = train_perceptron(poor_data, poor_labels)
         def predict(x, weights, bias):
             linear_output = np.dot(x, weights) + bias
             return 1 if linear_output >= 0 else 0
         correct = 0
         for i in range(len(test_data)):
             is_poor = predict(test_data[i], weights1, bias1)
             if is_poor == 0:
                 predicted_label = 0
             else:
                 predicted_label = predict(test_data[i], weights2, bias2) + 1
             true_label = test_labels[i]
             if predicted_label == true_label:
                 correct += 1
         accuracy = correct / len(test_data)
         print(f"Accuracy: {accuracy*100:.2f}%")

         Accuracy: 96.95%
```

***FIgure 3. Accuracy of Perceptron model using scratch.***



```
In [20]: predictions_stage1 = perceptron1.predict(test_data)
         predictions_stage2 = perceptron2.predict(test_data[predictions_stage1 == 1]) + 1
         final_predictions = predictions_stage1.copy()
         final_predictions[predictions_stage1 == 1] = predictions_stage2
         accuracy = accuracy_score(test_labels, final_predictions)
         print(f"Accuracy: {accuracy*100:.2f}%")

         Accuracy: 95.73%
```

***FIgure 4. Accuracy of Perceptron model using libraries.***

*Github code:*

[Cokitoquintal/MachineLearning: Repository exclusive for Machine Learning curse from Universidad Politecnica de Yucatan. (github.com)]

## v. Conclusion

In this project, the initial challenges were faced during data preparation. The dataset was found to have both numerical and categorical values, along with missing entries. To prepare the data for modeling, missing values were filled, unnecessary columns were removed, and categorical values were converted to numerical ones.

Two algorithms, KNN and Perceptron, were built from scratch. By developing these algorithms manually, a deeper understanding of their workings was gained. However, when the same tasks were performed using ready-made libraries, the process was observed to be much faster and easier. The models created could be useful in real-world scenarios. In robotics, understanding regions based on their socio-economic conditions can be crucial for tasks such as optimizing delivery routes for a mobile robot.

The comparison between building models from scratch and using libraries showed the importance of choosing the right approach for different project needs, sometimes basic approaches could be beneficial for a project.