

Machine Learning

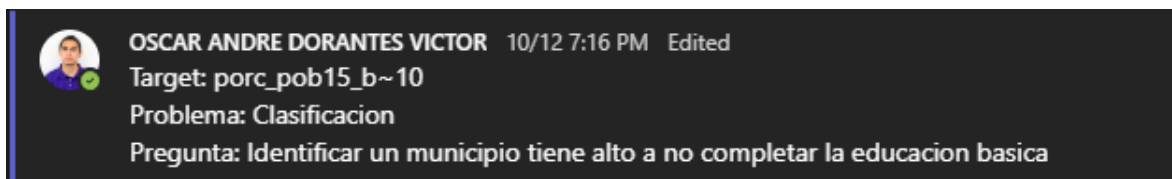
U2 - Implementing a Predictor from scratch

Oscar Andre Dorantes Victor

IRC 9B

10/23/2023

## Screenshot



A screenshot of a user profile and problem statement. The user's name is OSCAR ANDRE DORANTES VICTOR, with a profile picture and a green checkmark. The date and time are 10/12 7:16 PM, and the status is 'Edited'. The target is 'porc\_pob15\_b~10', the problem is 'Clasificacion', and the question is 'Identificar un municipio tiene alto a no completar la educacion basica'.

OSCAR ANDRE DORANTES VICTOR 10/12 7:16 PM Edited

Target: porc\_pob15\_b~10

Problema: Clasificacion

Pregunta: Identificar un municipio tiene alto a no completar la educacion basica

## Getting the dataset ready

```
df = pd.read_csv('/content/Indicadores_municipales_sabana_DA.csv', encoding='ISO-8859-1')
df.drop(df.columns[:5], axis=1, inplace=True)
```

The dataset is imported from a CSV file using the 'ISO-8859-1' encoding, which is often used for data that contains special characters. Then after importing, the first five columns are dropped. This is because the information is not relevant to the analysis or contain redundant information.

```
missing_values_before = df.isnull().sum().sum()
print(f"Total missing values before filling: {missing_values_before}")

for column in df.select_dtypes(include=['float64', 'int64']):
    df[column].fillna(df[column].median(), inplace=True)

for column in df.select_dtypes(include=['object']):
    df[column].fillna(df[column].mode()[0], inplace=True)

missing_values_after = df.isnull().sum().sum()
print(f"Total missing values after filling: {missing_values_after}")
```

Before addressing missing values, their total count is calculated and displayed. This provides a baseline to understand the extent of missing data.

For numerical columns, missing values are replaced with the median of the respective column. The median is a robust measure that's unaffected by extreme values or outliers.

For non-numerical columns, missing values are replaced with the most common value. This approach makes sure the substituted value doesn't alter the distribution of the data in the column.

After the filling process, the total count of missing values is recalculated and displayed to confirm the effectiveness of the imputation.

```
df.to_csv('/content/cleaned_dataset.csv', index=False)

print("Cleaned dataset saved to '/content/cleaned_dataset.csv'")
```

After all preprocessing steps, the cleaned dataset is saved to a new CSV file. By setting `index=False`, the default index of the dataframe is not saved to the file, ensuring the dataset remains concise and easy to work with.

## Steps taken to train the predictor

### K-Nearest Neighbors (KNN) Classifier

The first code is a KNN classifier, which predicts the class of a given input data point based on the majority class of its 'k' nearest neighbors in the training dataset.

#### Why Choose KNN?

- KNN is a non-parametric algorithm that doesn't make any assumptions about the underlying data distribution.
- KNN can be used for both classification and regression problems.
- It doesn't require a training phase, as it uses the training dataset directly during the prediction phase.

#### Characteristics of KNN:

- KNN uses a distance metric to find the nearest neighbors.
- The number of neighbors to consider for voting. In this code,  $k=3$ .
- Requires storing the entire training dataset.
- KNN can be sensitive to irrelevant features and the scale of the data.

### Perceptron Classifier

The second code is an implementation of the Perceptron algorithm, a binary classifier that iteratively adjusts its weights based on misclassified examples.

#### Why Choose the Perceptron?

- The Perceptron is foundational to neural networks, making it important for understanding more complex models.

- It's suitable for datasets that are linearly separable or nearly so.
- Can update its weights on-the-fly as new examples come in.

#### **Characteristics of the Perceptron:**

- Uses a weight vector and a bias to make predictions.
- Uses a step activation function to make binary decisions.
- Controls how much the weights are adjusted during training. In this code, the learning rate is set to 0.1.
- The number of epochs represents the number of times the Perceptron will iterate over the entire training dataset to update weights. Here, it's set to 100.

### **Steps taken to evaluate the predictor**

#### **K-Nearest Neighbors (KNN) Classifier**

##### **Evaluation Steps**

##### **Data Splitting**

- The dataset is randomly shuffled and split into training (80%) and testing (20%) subsets.

##### **Model Initialization and Training**

- A KNN classifier object is initialized with  $k=3$ .
- The training data is provided to the model, but KNN doesn't require a formal training phase.

##### **Accuracy Calculation**

- For each data point in the test set, the model predicts its class.
- The predictions are compared to the true labels to determine how many predictions were correct.
- The accuracy is computed as the fraction of correct predictions out of all predictions. This is done in two ways: using the `calculate_accuracy` method and manually using array comparisons.

##### **Single Data Point Prediction**

- A random data point from the test set is chosen.
- The model predicts its class and also provides distances to its 3 nearest neighbors in the training set.

## **Output**

- The accuracy of the model on the test data is printed.
- Details about the random test data point, distances to its neighbors, and its prediction are displayed.

## **Perceptron Classifier**

### **Evaluation Steps**

#### **Data Splitting**

- Similarly, the dataset is randomly shuffled and split into training (80%) and testing (20%) subsets.

#### **Model Initialization and Training**

- A Perceptron object is initialized with the number of features and default learning rate and epochs.
- The training data is fed into the perceptron to adjust its weights.

#### **Accuracy Calculation**

- The model predicts the class for each data point in the test set using the learned weights.
- Predictions are compared to the actual labels to calculate accuracy. This is done in two ways, through the `calculate_accuracy` method and manually via array comparisons.

#### **Single Data Point Prediction**

- A random data point from the test set is chosen.
- The perceptron predicts its class using the learned weights.

#### **Output:**

- The accuracy of the perceptron on the test data is printed.
- The prediction for the random test data point is displayed.

## **Steps taken to train the predictor using libraries**

### **K-Nearest Neighbors (KNN) Classifier**

#### **Steps to train and test the model**

##### **Data Preprocessing**

- Load the dataset.
- Select the features and target variable.
- Convert the target variable to binary based on the condition provided.

##### **Data Splitting**

- Use `train_test_split` to divide the dataset into training and testing subsets (80% training and 20% testing).

##### **Model Initialization**

- Create an instance of the `KNeighborsClassifier` with `k=3`.

##### **Training**

- Train the model using the `fit` method on the training data.

##### **Prediction and Evaluation**

- Use the trained model to predict the classes for the test set.
- Calculate the accuracy using the `accuracy_score` function.

##### **Single Data Point Prediction**

- Pick a random data point from the test set.
- Predict its class using the trained model.

##### **Output**

- Print the accuracy of the model on the test data.
- Display details about the random test data point and its prediction.

## **Perceptron Classifier**

### **Steps to train and test the model**

#### **Data Preprocessing**

- Same as KNN.

#### **Data Splitting**

- Same as KNN.

#### **Model Initialization:**

- Create an instance of the Perceptron class with the specified number of iterations and learning rate.

#### **Training**

- Train the perceptron using the training data.

#### **Prediction and Evaluation**

- Predict the classes for the test set.
- Calculate the accuracy.

#### **Single Data Point Prediction**

- Same as KNN.

#### **Output:**

- Print the accuracy of the perceptron on the test data.
- Display details about the random test data point and its prediction.

## **K-Nearest Neighbors (KNN) Comparison**

### **Accuracy**

Sklearn's KNN outperformed the custom KNN by about 1.02%. This difference, although small, might be due to optimization techniques used by sklearn or differences in the way distances are computed or sorted.

### **Efficiency**

Sklearn's methods are generally optimized for performance, so predictions using sklearn's KNN would likely be faster, especially on larger datasets. The custom KNN is educational but not as efficient in terms of computation time.

### **Simplicity & Ease of Use**

Using sklearn simplifies the process. With just a few lines of code, you can train, predict, and evaluate a model. The custom implementation, on the other hand, requires more code and a deeper understanding of the algorithm's inner workings.

### **Flexibility**

The custom KNN can be modified to experiment with different distance metrics, weighting strategies, or other tweaks. With sklearn, while there are many parameters to adjust, you are limited to the options provided by the library.

### **Consistency**

The results obtained using the method and mathematical calculation in the custom KNN matched exactly, ensuring that the custom method's implementation is consistent.

## **Perceptron Comparison:**

### **Accuracy**

The custom Perceptron outperformed sklearn's Perceptron by 6.3%. This discrepancy might be due to differences in the implementation details, the activation functions, or the way the weights are updated.



### **Efficiency**

Sklearn's methods are generally optimized for performance. However, given that the custom Perceptron yielded higher accuracy in this specific instance, it's worth considering its implementation for this dataset.

### **Simplicity & Ease of Use**

Using sklearn provides a more straightforward approach to training and predicting. The custom implementation, while providing insight into the inner workings of the algorithm, is more verbose.

### **Flexibility**

The custom Perceptron offers flexibility in terms of modifying the activation function, error calculation, or weight updates. With sklearn you are restricted to the parameters provided by the library.

### **Consistency**

As with the KNN, the results obtained using the method and mathematical calculation in the custom Perceptron matched exactly, ensuring that the custom method's implementation is consistent.

## **Conclusion**

This project encountered some difficulties. First, it was challenging to comprehend the data and select the pertinent pieces of information. Next, it was important to pay attention to every aspect when developing our own iterations of the KNN and Perceptron algorithms. It was unexpected to notice that occasionally our approaches performed better when compared to well-known tools like sklearn.

Robots may utilize techniques like KNN or Perceptron to make decisions depending on what they detect, so consider how this can aid in robotics. As an example, a robot may determine if it is safe to move or halt given certain inputs. Robots may also use these techniques to determine their location, such as whether they are indoors or outside. They can then modify their behavior as a result. Robots can utilize these techniques to engage with people to comprehend and anticipate human behavior. This makes interactions go more smoothly. Robots can also schedule their energy use or even foresee when repairs would be necessary.

**Github link**

<https://github.com/AndreDorantes/U2---Implementing-a-Predictor-from-scratch>