

Implementing a Predictor from scratch

Enrique Arturo Emmanuel Chi Gongora
Robotics Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 7193. CP 97357
Ucú, Yucatán. México
Email: 1909040@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 report focuses on the implementation of a predictor using Perceptron and KNN algorithms in a supervised learning context. The main objective of the project is to thoroughly understand the process of creating a prediction model, from acquiring and preparing data to evaluating its performance. Specific objectives include preprocessing a data set, creating a predictor from scratch, and comparing results between the custom model and one implemented using libraries such as scikit-learn. Predictor implementation is addressed in both classification and regression problems, highlighting key challenges and decisions at each stage of the project. In addition, we reflect on how this experience can be applied to the field of robotics, where autonomous decision-making based on data plays a fundamental role. This report provides a detailed overview of the steps and processes involved in creating a predictor and its relevance in the context of data science and robotics.

Index Terms

Machine Learning, Perceptron, K-Nearest Neighbors (KNN), Prediction, Classification, Data Preprocessing, Model Evaluation, Supervised Machine Learning, Scikit-learn.



Implementing a Predictor from scratch

I. INTRODUCTION

THE purpose of this report is to explore the process of implementing a predictor using Perceptron and KNN algorithms. The goal is to gain an in-depth understanding of how a prediction model operates from its foundations. The process starts from data acquisition and preparation, addressing issues such as categorical label management and imputation of missing values. Throughout this report, the progress and challenges experienced in creating a custom predictor will be detailed. This experience not only contributes to improving the understanding of machine learning, but also establishes a solid foundation for future projects in this field.

II. OBJECTIVES

The purpose of the report is to facilitate the understanding of predictors in the context of supervised learning and their relevance in problem solving. In addition, it focuses on determining the level of access to electricity in the municipalities according to the selected target.

The specific objectives of the report are the following:

- Perform preprocessing on a data set.
- Develop a predictor from scratch.
- Compare and analyze the results obtained.

These objectives will guide the focus of the report, providing a clear view of the processes involved in creating and evaluating a predictor in a supervised learning environment.

III. STATE OF THE ART

In the field of implementing predictors from data sets, a diversity of approaches stands out, covering both regression and classification. In regression, there are traditional algorithms such as linear and logistic regression, which offer simplicity and the ability to model linear and logistic relationships in numerical data. Likewise, regularized regression methods are used, such as Lasso and Ridge, which help avoid overfitting and improve model generalization. In the more contemporary sphere, deep learning, such as neural networks, has gained relevance by allowing the modeling of complex relationships in large-scale data.

In classification, in addition to traditional algorithms such as k-Nearest Neighbors (KNN) and the Perceptron, there is an increase in advanced techniques such as support vector machines and convolutional neural networks, especially in image classification. Additionally, there is growth in the use of natural language processing and text-based algorithms, which has revolutionized the classification of unstructured data.

The increase in availability of open source machine learning libraries, such as scikit-learn and TensorFlow, has simplified the implementation of advanced algorithms, allowing developers and data scientists to take advantage of recent advances in their projects.

The current state of predictor implementation spans a wide variety of approaches, from traditional algorithms to deep learning techniques, with a strong emphasis on efficiency, accuracy, and adaptability to various types of data. Choosing the appropriate approach will largely depend on the nature of the data and the specific objectives of the project.

IV. METHODS AND TOOLS

The methodology followed in this report was developed in several stages. First, data was acquired from a data set suitable for analysis and implementation of a predictor.

Next, a data preprocessing process was carried out that included data cleaning and preparation. This included outlier removal, feature normalization, and imputation of missing values. Additionally, categorical labels were managed to be compatible with supervised learning algorithms.

Subsequently, two supervised learning algorithms were implemented, namely the Perceptron and K-Nearest Neighbors (KNN). These algorithms were programmed from scratch. The hyperparameters were adjusted and the models were trained with the prepared data.

To evaluate the predictive ability of the models, performance tests and cross-validation were carried out. Specific metrics, such as accuracy, were used to measure the performance of the models.

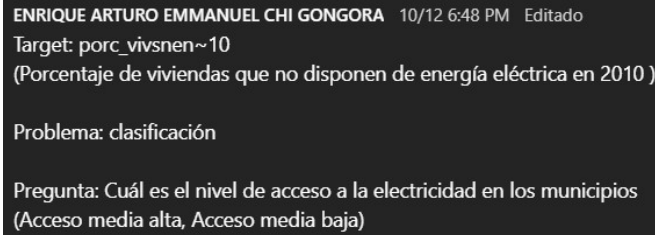
Finally, the results obtained with both algorithms were compared and analyzed to determine which of them worked more effectively in the context of the data used. This methodology provided a solid structure for the development of the report and the achievement of the stated objectives.

Tools:

- Python: The Python programming language is used due to its wide variety of libraries and frameworks for machine learning, such as scikit-learn and NumPy.
- Jupyter Notebooks: Jupyter notebooks are created to facilitate developing code and displaying results interactively.
- scikit-learn: This Python library is used to implement the supervised learning algorithms and perform model preprocessing and evaluation tasks.
- NumPy: NumPy is used to perform efficient numerical operations on matrices and vectors of data.
- Pandas: The Pandas library is used for data manipulation and cleaning, as well as for the management of categorical labels.
- Matplotlib: This visualization library is used to create graphs and visualize results.

V. DEVELOPMENT

The objective of the project is to classify the level of access to electricity in municipalities, according to the percentage of homes without electricity in 2010. The problem focuses on determining whether a municipality has "High access", or "Medium/Low access" to electricity.



ENRIQUE ARTURO EMMANUEL CHI GONGORA 10/12 6:48 PM Editado
 Target: porc_vivsnen~10
 (Porcentaje de viviendas que no disponen de energía eléctrica en 2010)
 Problema: clasificación
 Pregunta: Cuál es el nivel de acceso a la electricidad en los municipios
 (Acceso media alta, Acceso media baja)

Fig. 1. Selected Target.

Next, the steps that were carried out to achieve the main objective will be detailed. In order to improve the appreciation of the code in the images provided, it was structured in an unconventional way to facilitate its understanding.

A. Import libraries and load dataset

The first step to carry out this activity is to import the libraries

```
1 #Import libraries
2
3 import pandas as pd
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from sklearn.preprocessing import LabelEncoder
7 from sklearn.linear_model import Perceptron
8 from sklearn.preprocessing import
   StandardScaler
9 from sklearn.metrics import accuracy_score
10 from sklearn.impute import SimpleImputer
```

- pandas for manipulating tabular data.
- numpy for numerical operations.
- matplotlib.pyplot for creating graphs and visualizations
- LabelEncoder from sklearn.preprocessing to encode categorical labels.
- Perceptron from sklearn.linear_model to create a perceptron model.
- StandardScaler from sklearn.preprocessing to standardize features.
- accuracy_score from sklearn.metrics to evaluate the accuracy of the model.
- SimpleImputer from sklearn.impute to impute missing values in the data.

These libraries are used in the process of loading data, preprocessing, and evaluating machine learning models.

```
1 df = pd.read_csv('
2   Indicadores_municipales_sabana_DA.csv',
3   index_col=0, encoding='latin-1')
```

This code is used to upload a data file in CSV format. The 'pandas' library, an essential tool in data analysis, is used to perform this task. The file in question, called 'Indicadores_municipales_sabana_DA.csv', is loaded into a DataFrame, which is a tabular data structure. This DataFrame is saved in a variable called "df".

B. Prepare the dataset

The next step is the preparation of the data set, which includes converting categorical values to numeric values, removing unwanted features, and filling gaps to ensure the quality of the input data.

```
1
2 label_encoder_ent = LabelEncoder()
3 df['Estado'] = label_encoder_ent.fit_transform
   (df['nom_ent']) + 1
4 label_encoder_mun = LabelEncoder()
5 df['Municipio'] = label_encoder_mun.
   fit_transform(df['nom_mun']) + 1
```

In these lines of code, categorical data is being converted into numeric data using the tag encoding technique.

- A LabelEncoder object called label_encoder_ent is created for the 'ent_name' column, which represents the state.

- The fit_transform method of label_encoder_ent fits and transforms the 'ent_name' column of the DataFrame df. This means that it will assign a unique number to each unique category in 'ent_name', allowing states to be represented numerically. Additionally, 1 is added to the resulting values to ensure that the numbered values start from 1 instead of 0.

- A LabelEncoder object called label_encoder_mun is created for the 'nom_mun' column, which represents the municipality.

- As in the previous case, the fit_transform method of label_encoder_mun adjusts and transforms the column 'nom_mun'. Each unique category in 'nom_mun' is associated with a unique number, converting the municipality names into numeric values. 1 is also added to the resulting values to ensure that the numbered values start from 1 instead of 0.

The purpose of this process is to transform categorical data (such as names of states and municipalities) into numerical data so that it can be used in machine learning models that require numerical data as input.

```
1
2 df = df.drop(['nom_ent', 'nom_mun', '
   gdo_rezsoc00', 'gdo_rezsoc05', '
   gdo_rezsoc10'], axis=1)
3 dataset = df.copy()
```

These lines of code remove the columns 'nom_ent', 'nom_mun', 'gdo_rezsoc00', 'gdo_rezsoc05' and 'gdo_rezsoc10' from the DataFrame df. The elimination of 'nom_ent' and 'nom_mun' is justified because new columns have been added with numerical categories that represent those same variables, which simplifies the DataFrame and avoids redundancies.

The elimination of 'gdo_rezsoc00', 'gdo_rezsoc05' and 'gdo_rezsoc10' is justified because they are categorical data and, in the context of the activity, we sought to simplify the data set to focus on the variables of interest and reduce complexity. The elimination of these columns was done with the purpose of creating a simpler and more understandable prediction model, focusing on the variables selected for the objective of the activity.

Also, the DataFrame "df" is copied to a new one called "dataset"

```
1
2 imputer = SimpleImputer(strategy='median')
3 dataset = pd.DataFrame(imputer.fit_transform(
    dataset), columns=dataset.columns)
```

In these lines of code, the SimpleImputer class is used to replace missing values in a DataFrame. The 'median' imputation strategy is employed, replacing missing values with the median of each column. This ensures that the data is complete and ready for processing, improving data quality for use in machine learning models. The result is a new DataFrame with the missing values replaced by the medians of the respective columns.

```
1
2 valor_maximo = dataset['porc_vivsnenergia10'].
    max()
3 valor_minimo = dataset['porc_vivsnenergia10'].
    min()
4
5 print(f"El valor m s alto en 'porc_vivsnen
    ~10' es: {valor_maximo}")
6 print(f"El valor m s bajo en 'porc_vivsnen
    ~10' es: {valor_minimo}")
```

These lines of code calculate and display the maximum and minimum value in the column 'porc_vivsnenergia10' of a DataFrame called 'dataset'. This is essential for sorting the column, since knowing the maximum and minimum values allows you to divide the range of values into meaningful categories.

```
1
2 dataset['Nivel de Acceso'] = pd.cut(dataset['
    porc_vivsnenergia10'], bins=[0, 10, 69],
    labels=['Acceso alto', 'Acceso medio/bajo'
    ])
3
4 mapping = {'Acceso alto': 0, 'Acceso medio/
    bajo': 1}
5 dataset['Nivel de Acceso'] = dataset['Nivel de
    Acceso'].map(mapping)
6
7 dataset
```

These lines of code perform data preparation and create a new column called 'Nivel de acceso' in the DataFrame 'dataset'.

pd.cut(dataset['porc_vivsnenergia10'], bins=[0, 10, 69], labels=['Acceso alto', 'Acceso medio/bajo']): This part of the code splits the column 'porc_vivsnenergia10' into intervals (bins) defined by the values [0, 10, 69] and assigns categorical

labels to each interval. If a value in 'porc_vivsnenergia10' is between 0 and 10, it is labeled 'Acceso alto', and if it is between 10 and 69, it is labeled 'Acceso medio/bajo'. These labels reflect different levels of access to electricity.

mapping = 'Acceso alto': 0, 'Acceso medio/bajo': 1: Here, a dictionary called 'mapping' is created that maps the numeric values 0 and 1 to the categorical labels 'Acceso alto', and 'Acceso medio/bajo', respectively.

dataset['Nivel de acceso'] = dataset['Nivel de acceso'].map(mapping): The mapping defined in the 'mapping' dictionary is applied to the 'Nivel de acceso' column, replacing the categorical labels with the values corresponding numbers. This is useful so that machine learning algorithms can work with these labels numerically.

C. Data train and test

In machine learning, the key to building accurate models lies in dividing data into training and test sets. This allows us to train the model on a portion of the data and then evaluate its performance on unseen data, avoiding problems such as overfitting. This practice is essential in the development of machine learning models.

```
1 # 80% para train
2 X_train = dataset.iloc[:1965, 0:134].values
3 y_train = dataset.iloc[:1965, 135].values
4
5 # 20% para test
6 X_test = dataset.iloc[1966:, 0:134].values
7 y_test = dataset.iloc[1966:, 135].values
```

The provided code is used to split a data set into two essential parts: a training set (train) and a test set (test). This split is common practice in machine learning, and its purpose is to evaluate how well a machine learning model can generalize to unseen data. Specifically, 80% of the data is assigned to the training set, where X_train contains the training features (the first 1965 rows and the first 134 columns of the original dataset), while y_train stores the labels corresponding to those rows. The remaining 20% is assigned to the test set, where X_test contains the test features (from row 1966 to the end of the original data set), and y_test stores the corresponding labels. This division is essential to evaluate the performance of the model on unobserved data and detect overfitting problems, thus helping to estimate its generalization ability in real-world situations.

D. Perceptron from scratch

The Perceptron, a fundamental concept in the field of machine learning, is a simple neural network model used to perform binary classification tasks. Despite its simplicity, the Perceptron provides a solid foundation for understanding how more complex neural networks work. At its core, a Perceptron takes a set of features as input and produces a binary output, that is, it classifies the data into two distinct categories. To achieve this, it uses an activation function called the step function. The code is explained below.

```

1
2 def step_function(x):
3     return 1 if x >= 0 else 0

```

In the code, the `step_function` function is being used to perform activation on a Perceptron. This function takes the weighted sum of the inputs and decides whether to activate or deactivate the Perceptron (assigning 1 or 0) based on whether that sum is greater than or equal to zero. In essence, it is used to model the binary decision that a Perceptron makes when classifying data into two distinct categories.

```

1
2 def train_perceptron(X, y, learning_rate,
3                     epochs):
4     num_features = X.shape[1]
5     num_samples = X.shape[0]
6     weights = np.zeros(num_features)
7     bias = 0
8
9     for _ in range(epochs):
10        for i in range(num_samples):
11            prediction = step_function(np.dot(
12                X[i], weights) + bias)
13            error = y[i] - prediction
14            weights += learning_rate * error *
15                X[i]
16            bias += learning_rate * error
17
18    return weights, bias

```

The `train_perceptron` function in the code is responsible for training a Perceptron, a simple machine learning model used for binary classification tasks. It receives as input the training features (X) and the target labels (y) that are used for training the model. Additionally, it takes two essential hyperparameters: `learning_rate`, which controls the size of the fits at each training step, and `epochs`, which determines how many times all training data will be traversed. At each iteration (epoch), the Perceptron computes a prediction for each training sample using the `step_function` activation function and adjusts its weights and bias to minimize errors between the predictions and the actual labels. Once training is complete, the function returns the adjusted weights and bias, representing the Perceptron trained and ready to perform binary classifications with greater accuracy.

```

1
2 def predict_perceptron(X, weights, bias):
3     num_samples = X.shape[0]
4     y_pred = np.zeros(num_samples, dtype=int)
5     for i in range(num_samples):
6         prediction = step_function(np.dot(X[i],
7             weights) + bias)
8         y_pred[i] = prediction
9     return y_pred

```

The function `predict_perceptron` is intended to make predictions using a Perceptron that has been previously trained. It receives three arguments: the input feature matrix (X), the vector of weights learned during Perceptron training (weights), and the bias that was also learned during training. The function iterates through each sample in the input set, computes the

weighted sum of the features multiplied by the weights and adjusted for bias, applies a step activation function to determine whether the prediction is 0 or 1, and stores these predictions in an array. The end result is a set of binary predictions that classify samples into two categories based on the Perceptron's decisions.

```

1 learning_rate = 0.01
2 epochs = 10
3 weights, bias = train_perceptron(X_train,
4 y_train, learning_rate, epochs)

```

This part of the code is used to train a Perceptron. The learning rate (`learning_rate`) is specified, which determines the magnitude of the adjustments in the Perceptron weights in each iteration, and the number of epochs (`epochs`), which indicates how many times the training set will be traversed to adjust the weights. The training data, `X_train` and `y_train`, contain the features and class labels respectively. The `train_perceptron` function will perform Perceptron training using this data and produce the final values of the weights and bias. These weights and bias can later be used to make predictions on new data. Make sure you have the `train_perceptron` function defined and the correct training data before running this code.

```

1 accuracy = np.mean(y_test == y_pred)
2 print("Precisi n del Perceptr n:", accuracy)

```

This calculates the accuracy of the Perceptron model on a test set and displays this value on screen. First, the actual labels of the test set (`y_test`) are compared with the predictions made by the Perceptron (`y_pred`), generating an array of Boolean values that represent whether the predictions are correct or not. Accuracy is then calculated by averaging these Boolean values, where a value of 1 indicates 100% accurate predictions and 0 indicates completely wrong predictions. Finally, the accuracy is printed on the screen accompanied by an informational message.

E. KNN from scratch

k-Nearest Neighbors is a classification and regression algorithm widely used in the field of machine learning. It is known to be one of the simplest and most effective methods for addressing classification and regression problems in various applications. The core idea behind KNN is that similar objects or examples tend to cluster together in a feature space. Therefore, if we know the label of the examples close to an unknown one, we can predict its label or value. The code is explained below

```

1
2 def euclidean_distance(x1, x2):
3     return np.sqrt(np.sum((x1 - x2) ** 2))

```

The function `euclidean_distance(x1, x2)` calculates the Euclidean distance between two points in a multidimensional space. To do this, take two points `x1` and `x2`, subtract the

components of x_2 from those of x_1 , square those differences, and then add these components to the square. Finally, take the square root of that sum to obtain the Euclidean distance between the two points. This function is fundamental in the KNN algorithm for determining the closeness of points and therefore essential for making predictions based on nearest neighbors in the feature space.

```

1
2 def predict_knn(X_train, y_train, x_test, k):
3     distances = [euclidean_distance(x_test, x)
4                   for x in X_train]
5     k_indices = np.argsort(distances)[:k]
6     k_nearest_labels = [y_train[i] for i in
7                          k_indices]
8     most_common = np.bincount(k_nearest_labels
9                               ).argmax()
10    return most_common

```

The `predict_knn` function is responsible for predicting the label of a test sample in the KNN (k-Nearest Neighbors) algorithm in a series of steps. First, calculate the Euclidean distance between the test sample and all training samples, thus creating a list of distances. Then, identify the indices of the k smallest distances, representing the k training samples closest to the test sample. Then retrieve the labels corresponding to those closest samples. Finally, determine which label is most common among the nearest neighbors and assign it as the final prediction for the test sample.

```

1
2 y_pred_knn = [predict_knn(X_train, y_train,
3                           x_test, k=3) for x_test in X_test]

```

This line of code applies the algorithm to predict the labels of a set of test samples in `X_test`. It uses a list comprehension to iterate through each test sample, passing each one to the `predict_knn` function. This function calculates the predicted label for each test sample taking into account the three nearest neighbors in the training set `X_train`. The predicted labels are stored in a list called `y_pred_knn`. In summary, this line automates the label prediction process for all test samples using KNN with a value of " k " equal to 3, and stores the predictions in the `y_pred_knn` list for later evaluation or use.

```

1
2 accuracy_knn = np.mean(y_test == y_pred_knn)
3 print("Precisin del KNN:", accuracy_knn)

```

This calculates the accuracy of the algorithm's predictions compared to the actual labels. First, it compares the actual labels (`y_test`) with the labels predicted by KNN (`y_pred_knn`) and generates a series of boolean values indicating whether the predictions are correct or not for each test sample. These boolean values are then averaged using `np.mean`, providing the overall accuracy of the model. Finally, the accuracy is printed on the screen with a message indicating "KNN Accuracy."

F. Perceptron with libraries

By using libraries such as Scikit-Learn, the implementation of the Perceptron becomes simpler and more efficient. Scikit-

Learn provides an easy-to-use interface for training, tuning and evaluating Perceptron models. This includes the possibility of working with multidimensional features and applying regularization strategies to avoid overfitting. The code is explained below

```

1
2 imputer = SimpleImputer(strategy='
3 most_frequent')
4 y_train = imputer.fit_transform(y_train.
5 reshape(-1, 1)).ravel()

```

This code snippet is responsible for handling null values in training labels. To do this, an imputer called "SimpleImputer" is used with the 'most_frequent' strategy, which means that it will replace null values with the most frequent label in the training set. First, the imputer is fitted to the training labels and the set of labels is transformed to replace null values. The labels are then flattened to make sure they are the same shape as before. This ensures that the training labels do not have null values and are ready to be used in training the model.

```

1
2 model = Perceptron(eta0=1.0, max_iter=1000,
3 random_state=0)

```

This creates a Perceptron model for binary classification. The specified parameters are as follows:

- `eta0=1.0`: This sets the initial learning rate, determining how large the weight adjustments are during training.
- `max_iter=1000`: Limit the maximum number of training iterations to 1000 to prevent the process from extending indefinitely.
- `random_state=0`: Sets a random seed to 0, ensuring that the results are reproducible in future runs. In summary, this code configures the Perceptron model with initial parameters, ready to be trained and used in binary classification tasks, guaranteeing consistency and control in the training process.

```

1
2 scaler = StandardScaler()
3 X_train = scaler.fit_transform(X_train)
4 X_test = scaler.transform(X_test)

```

These lines of code apply a normalization technique called `StandardScaler` to the training and test sets. Normalization adjusts the features so that they have a mean of zero and a standard deviation of one. This is useful for ensuring that features are on the same scale, making it easier to train and evaluate the Perceptron model, especially when features have different magnitudes. It is first applied to the training set to learn its statistical properties and then the same normalization is used on the test set to maintain consistency in feature scaling.

```

1
2 model.fit(X_train, y_train)

```

This trains the model using the training set. The method fits the model to the input data `X_train` and the training labels `y_train`. During the training process, the Perceptron adjusts its

weights and biases to learn how to perform classification on training samples so that accurate predictions can be made on new data. It is the crucial phase in building the model and is where it is adapted to make classifications based on the pattern of the training data, which will allow it to make predictions on the test set or on unseen data.

```
1
2 y_pred_per = model.predict(X_test)
```

This line is used to make predictions on the test set. The predict function takes the test set X_test as input and uses the trained model to predict the corresponding labels. The resulting predictions are stored in the variable y_pred_per, allowing them to be compared with the actual labels of the test set and to evaluate the performance of the model in terms of accuracy and classification ability on unseen data.

```
1
2 accuracy_lib_per = accuracy_score(y_test,
3 y_pred_per)
4 print(f'Accuracy: {accuracy_lib_per}')
```

Finally, in this line of code, the accuracy of the model is calculated. First, the accuracy_score function is used to compare the actual y_test labels with the model predictions y_pred_per. Accuracy measures how many of the predictions made by the model are correct relative to the actual labels. The accuracy value is then stored in the accuracy_lib_per variable.

G. KNN with libraries

The fundamental process of K-Nearest Neighbors involves finding the “k” nearest neighbors to a test data point and determining its label based on the majority of nearest neighbor labels. Scikit-Learn offers an intuitive interface for building, training and evaluating KNN models. Additionally, it provides functions to measure the distance between data points, select the optimal value of “k”, and apply cross-validation strategies to evaluate model performance.

```
1
2 knn = KNeighborsClassifier(n_neighbors=3)
```

This part of the code configures a KNN classifier using the Scikit-Learn library in Python. In this case, the classifier is instantiated and the three nearest neighbors (n_neighbors=3) are considered when making predictions. This means that the model will take into account the three samples closest to a test point to decide its classification. The classifier configured in this way is ready to be trained and used in data classification tasks, where labels will be assigned to new data points based on their proximity to the closest training samples.

```
1
2 knn.fit(X_train, y_train)
```

This performs the model training process. The knn object is the previously configured classifier instance. The fit function

is used to fit this model to the training data. In this case, the training data is located in X_train, which contains the features of the samples, and y_train, which contains the class labels associated with those samples. During the training process, the model analyzes how features relate to class labels in the training set, allowing it to learn patterns and relationships. Once training is complete, the model will be ready to make accurate predictions on new data, based on the information acquired during the training process.

```
1
2 y_pred_lib_knn = knn.predict(X_test)
```

This code is responsible for making predictions using the previously trained model. The KNN instance, named “knn”, takes the test data set, represented by “X_test”, which contains the characteristics of the samples we want to predict, and uses the “predict” method to assign class labels to these samples. The resulting predictions are stored in the variable “y_pred_lib_knn”, allowing you to evaluate how well the model classifies test samples based on similarities to the training samples it has previously learned.

```
1
2 accuracy_lib_knn = accuracy_score(y_test,
3 y_pred_lib_knn)
4 print("Precisi n del kNN:", accuracy_lib_knn)
```

Lastly, this calculates the accuracy of the KNN model on the test set. Use the accuracy_score function to compare the actual labels of the test set, represented by y_test, with the labels predicted by the KNN model, stored in y_pred_lib_knn. Precision is obtained as the percentage of correct predictions in relation to the total predictions.

VI. RESULTS

In the results section, graphs were generated to compare the predictions with the actual data on the test set for both labels. This approach was consistently applied across all models, using standardized code for creating these graphical representations.

The following code is used for all graphs.

```
1
2 unique_labels_pred, counts_pred = np.unique(
3 y_pred_categories, return_counts=True)
4 unique_labels_actual, counts_actual = np.
5 unique(y_test_categories, return_counts=
6 True)
7 labels = ['Acceso alto', 'Acceso medio/bajo']
8
9 for label in labels:
10     if label not in unique_labels_pred:
11         unique_labels_pred = np.append(
12             unique_labels_pred, label)
13         counts_pred = np.append(counts_pred,
14                                 0)
15     if label not in unique_labels_actual:
16         unique_labels_actual = np.append(
17             unique_labels_actual, label)
```

```

13     counts_actual = np.append(
14         counts_actual, 0)
15 sorted_labels = ['Acceso alto', 'Acceso medio/
16     bajo']
17 sorted_pred_counts = [counts_pred[
18     unique_labels_pred == label][0] for label
19     in sorted_labels]
20 sorted_actual_counts = [counts_actual[
21     unique_labels_actual == label][0] for
22     label in sorted_labels]
23
24 x = np.arange(len(sorted_labels))
25 width = 0.4
26
27 fig, ax = plt.subplots(figsize=(10, 6))
28 rects_pred = ax.bar(x - width/2,
29     sorted_pred_counts, width, label='
30     Etiquetas Predichas', color='orange')
31 rects_actual = ax.bar(x + width/2,
32     sorted_actual_counts, width, label='
33     Etiquetas Reales', color='g')
34
35 ax.set_xticks(x)
36 ax.set_xticklabels(sorted_labels)
37
38 ax.set_xlabel('Niveles de Acceso')
39 ax.set_ylabel('Cantidad de Muestras')
40 ax.set_title('Etiquetas Predichas vs Reales (
41     Perceptron)')
42
43 ax.legend()
44
45 print("Predicciones:", sorted_pred_counts)
46 print("Datos Reales:", sorted_actual_counts)
47
48 plt.show()

```

This code creates a bar chart that compares the labels predicted by a model to the actual labels in a binary classification problem, specifically related to access levels. First, the unique tags and their counts are obtained for both the predictions and the actual data. Then, the target tags are defined, which are “Acceso alto” and “Acceso medio/bajo”. The code checks whether these tags are present in the predictions and actual data, adding them if they are not. Consistent order in the graph is ensured by creating an ordered list of labels. Subsequently, the bar chart is configured, assigning different colors to the predicted labels and the actual labels. Labels are set on the x-axis to represent access levels, and labels and a title are added to the graph. Finally, the prediction counts and actual data are printed to the screen, and the graph is displayed. In summary, this code visualizes how a model’s predictions compare to actual data in a binary classification context, specifically focused on access levels, providing a clear graphical representation of model performance.

Next, plots of the models will be displayed, providing a clear visualization of the comparison between actual data and predictions. The corresponding accuracy will also be provided, along with the exact number of predicted data and actual data.

A. Perceptron from scratch

Precision: 0.9571428571428572

Precisión del Perceptrón: 0.9571428571428572
 Predicciones: [490, 0]
 Datos Reales: [469, 21]

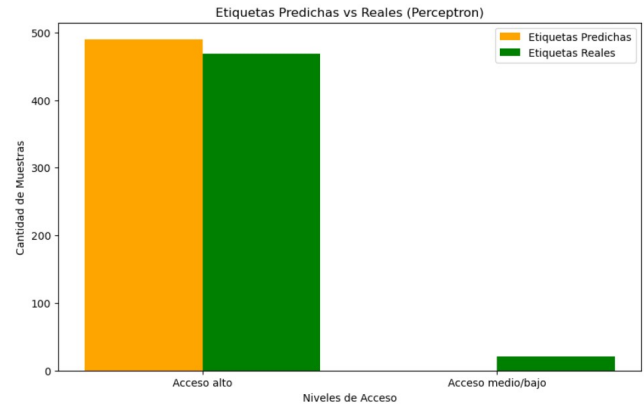


Fig. 2. Perceptron from scratch result

B. KNN from scratch

Precision: 0.8530612244897959

Precisión del KNN: 0.8530612244897959
 Predicciones: [423, 67]
 Datos Reales: [469, 21]

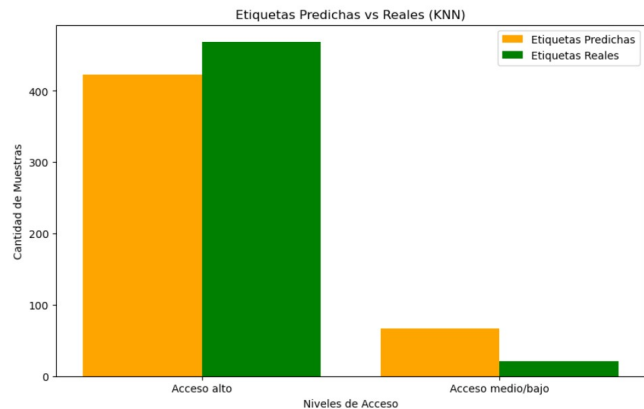


Fig. 3. KNN from scratch result

C. Perceptron with libraries

Precision: 0.9795918367346939

Precisión del Perceptrón: 0.9795918367346939
Predicciones: [461, 29]
Datos Reales: [469, 21]

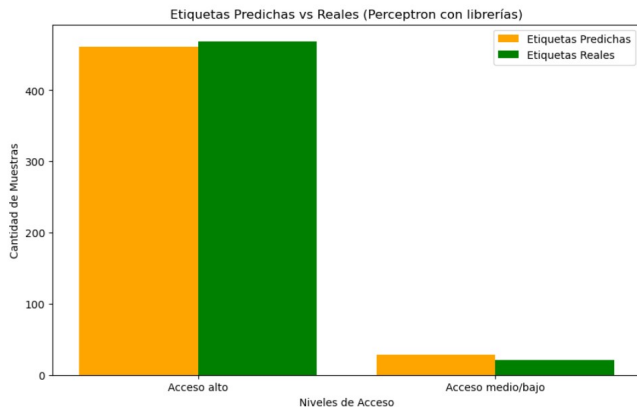


Fig. 4. Perceptron with libraries result

D. KNN with libraries

Precision: 0.9551020408163265

Precisión del KNN: 0.9551020408163265
Predicciones: [473, 17]
Datos Reales: [469, 21]

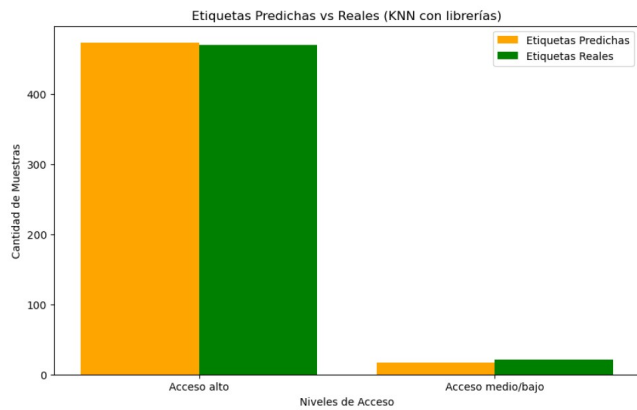


Fig. 5. KNN with libraries result

As seen in the images, the Perceptron model using libraries exhibits higher accuracy and is presented as a simpler option compared to other approaches. The visual representations show that the Perceptron achieves better performance in data classification, with a straightforward and effective approach. This suggests that, in this particular context, the simplicity of the Perceptron coupled with its accuracy makes it a favorable choice for the task at hand.

VII. CONCLUSION

The implementation of predictors with supervised learning algorithms, whether in regression or classification, has evolved significantly. Data preparation is crucial, including managing null values and selecting relevant features. The choice of algorithms should be based on the nature of the data and

the objectives of the project, from traditional models to deep learning.

Open source libraries like scikit-learn make it easy to deploy and evaluate models, saving time. Additionally, interpretation and visualization tools provide deeper understanding.

Regarding the application in the field of robotics, the implementation of predictors has great potential. Modern robots often face complex and changing situations in their environment. Predictors can help robots make autonomous decisions based on real-time data. For example, an autonomous robot navigating an unknown environment could use a predictor to anticipate obstacles and plan safe routes. Furthermore, the application of predictors in robotics also extends to computer vision, object recognition, and natural language processing tasks. Robots can use classification models to understand and respond to human language, or to identify objects in their environment.

APPENDIX GITHUB REPOSITORY

You can then access the repository on GitHub that includes the Jupyter notebook, the dataset used and the dataset dictionary.

Link: <https://github.com/EnriqueCG/Machine-learning.git>

REFERENCES

- [1] A. Sumayli, "Development of advanced machine learning models for optimization of methyl ester biofuel production from papaya oil: Gaussian process regression (GPR), multilayer perceptron (MLP), and K-nearest neighbor (KNN) regression models", *Arabian J. Chemistry*, vol. 16, n.º 7, p. 104833, julio de 2023. Accessed on October 18, 2023. [Online]. Available: <https://doi.org/10.1016/j.arabjc.2023.104833>
- [2] B. Shetty, "5 Classification Algorithms for Machine Learning". Built In. Accessed on October 18, 2023. [Online]. Available: <https://builtin.com/data-science/supervised-machine-learning-classification>
- [3] J. Brownlee, "Develop k-Nearest Neighbors in Python From Scratch - MachineLearningMastery.com". MachineLearningMastery.com. Accessed on October 18, 2023. [Online]. Available: <https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/>
- [4] J. Brownlee, "How To Implement The Perceptron Algorithm From Scratch In Python - MachineLearningMastery.com". MachineLearningMastery.com. Accessed on October 18, 2023. [Online]. Available: <https://machinelearningmastery.com/implement-perceptron-algorithm-scratch-python/>