Implementing a Predictor from Scratch

Marco Alejandro González Barbudo
Computacional Robotics Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 7193. CP 97357
Ucú, Yucatán. México
Email: 1909073@upy.edu.mx

Víctor 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 project aims to predict the national poverty ranking (percentage) of the city of Mérida, Mexico, based on the dataset "Indicadores de pobreza, pobreza por ingresos, rezago social y gini a nivel municipal, 1990, 2000, 2005 y 2010," provided by the Consejo Nacional de Evaluación de la Política de Desarrollo Social (CONEVAL). The project was executed in a Jupyter Notebook using the Anaconda software environment. It employs the K-Nearest Neighbors (KNN) machine learning algorithm to address the regression problem. Two different implementations of KNN are explored: one developed from scratch and another utilizing the scikit-learn library. A comparison between both codes will be sought with the objective of generating a better understanding not only of the data set but also of the resolution of regression problems through the implementation of KNN algorithms. Finally, the executed code as well as the files used can be found in a GitHub repository attached to the document.

Index Terms

Machine Learning, Predictive Model, Regression Analysis, Nearest Neighbor Methods.

Implementing a Predictor from Scratch

I. Introduction

The purpose of this report is to explain the development process of a K-Nearest Neighbors algorithm aimed at predicting the placement of the city of Mérida, Yucatán, Mexico in the National Poverty Ranking as provided by the dataset "Poverty Indicators, Poverty by income, social backwardness and gini at the municipal level, 1990, 2000, 2005, and 2010," which has been generated by the National Council for the Evaluation of Social Development Policy (CONEVAL). The fact that CONEVAL is a decentralized public organization of the Federal Public Administration, possessing autonomy and technical capacity for the generation of objective information regarding the state of social policy and poverty measurement in Mexico, should be emphasized. This capability enables the enhancement of decision-making in this field.

By the utilization of this dataset, not only will this regression problem be addressed through the application of a KNN algorithm, but also the exploration of its functionality will be pursued through its development from the ground up, particularly without drawing upon or relying on any prior work or libraries for assistance.

Subsequently, the implementation of scikit-learn libraries will be employed to execute another prediction algorithm, thereby allowing for the eventual comparison of performance and results.

Understanding the development process and utilization of the K-Nearest Neighbors (KNN) algorithm enriches the knowledge of the algorithm's practical application in addressing regression problems, while also highlighting the critical role of data independence and the capability to construct algorithms from the ground up.

II. Objectives

General objective:

To predict the national Poverty Ranking of the city of Mérida, Mexico, based on the provided dataset.

1) Develop a K-Nearest Neighbors (KNN) machine learning algorithm from scratch to address the regression problem of predicting Mérida's national poverty ranking.
2) Implement the scikit-learn library to create an alternative prediction algorithm for Mérida's national poverty ranking and facilitate a performance and results comparison with the custom KNN implementation.
3) Enhance understanding of the practical application of the K-Nearest Neighbors (KNN) algorithm in addressing regression problems and emphasize the significance of data independence in algorithm construction.

III. State of the Art

K-Nearest Neighbors (KNN) algorithm

The k-Nearest Neighbor (kNN) classification algorithm is a fundamental and widely used method in machine learning and pattern recognition. The core idea behind kNN is to classify a test object by considering the class labels of its k-nearest neighbors in the training dataset. This algorithm is simple yet effective and has found applications in various domains, and fits perfectly to solve this regression problem.

A. Key Components of kNN

The kNN algorithm comprises three key components:

1. A Set of Labeled Objects: This is the training dataset, consisting of labeled data points. Each data point has attributes and a corresponding class label.

2. Distance or Similarity Metric: A distance metric, such as Euclidean distance or cosine similarity, is used to compute the similarity or dissimilarity between data points.

3. The Number of Nearest Neighbors (k): The parameter k determines how many nearest neighbors should be considered when classifying a test object.

B. The kNN Classification Process

The kNN classification process can be summarized through the following pseudocode:

When given a training set D and a test object $x = (x', y')$, the algorithm calculates the distance or similarity between z and all the training objects $(x, y) \in D$ to establish its nearest-neighbor list, D_z. Here, x represents the data of a training object, and y denotes its class label. Similarly, x' represents the data of the test object, and y' signifies its class label. This process is performed for a specified value k.

Input: D, The set of k training objects, $z = x', y'$

Process:

Compute $d(x', x)$, the distance between z and every object, $(x, y) \in D$.

Select $D_z \subseteq D$, the set of k closest training objects to z.

Output:

$$y' = \underset{v}{\operatorname{argmax}} \sum_{(x_i, y_i) \in D_z} I(v = y_i)$$

where v is a class label, y_i is the class label for the ith nearest neighbors, and $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

This pseudocode outlines the kNN classification process, which is based on finding the k nearest neighbors of the test object and determining its class label through majority voting. The choice of distance metric and the value of k are essential parameters that affect the performance of the kNN algorithm. [1]

IV. Methods and Tools

A. Dataset

The dataset "Indicadores de pobreza, pobreza por ingresos, rezago social y Gini a nivel municipal, 1990, 2000, 2005 y 2010" is published by the Consejo Nacional de Evaluación de la Política de Desarrollo Social (CONEVAL). CONEVAL is a public decentralized body of the Federal Public Administration in Mexico with autonomy and technical capacity to generate objective information about the social policy situation and poverty measurement in Mexico, aiming to improve decision-making in this area.

Source: The data is sourced from CONEVAL and can be found at https://www.coneval.org.mx/Paginas/principal.aspx.

Frequency: The data is updated every 10 years (R/P10Y) and covers the years 1990, 2000, 2005, and 2010.

Language: The dataset is in Spanish (es).

Modified Date: The dataset was last modified on October 6, 2015.

Publisher: The dataset is published by CONEVAL, which is an Institutional Data Administrator focused on local governments.

Data Dictionary: The dataset contains a wide range of indicators at the municipal level in Mexico, including:

- Demographic information (e.g., population)
- Poverty indicators (e.g., percentage of people in poverty, poverty by income, poverty by capabilities, and poverty by patrimony)
- Social deprivation indicators (e.g., access to health services, security, housing quality, and basic services)
- Social development indices (e.g., social lag index and social deprivation degree)
- Gini coefficient values for the years 1990, 2000, and 2010

The dataset covers various aspects of poverty, social development, and income inequality in Mexican municipalities.

It also provides a detailed explanation of the data sources and methodologies used for different years, including information on the estimation of poverty and Gini coefficients.

Some indicators are not available for all years, and the 2010 results may not coincide with multidimensional poverty measurements due to differences in data sources and methodologies. [2]

B. Anaconda Software

Anaconda is a widely used open-source platform and distribution for data science and machine learning that simplifies the process of managing and deploying complex software environments. It was developed by Anaconda, Inc., a software company focused on data science and machine learning tools. Anaconda is designed to streamline the management of packages, libraries, and dependencies commonly used in data science and scientific computing projects. [3]

Fig. 1. Anaconda Logo

C. Libraries

NumPy (np): NumPy is considered a fundamental package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions for operation on these arrays. It is frequently utilized for numerical computations and data manipulation. [4]

Pandas (pd): Pandas is recognized as a data manipulation and analysis library for Python. It offers data structures such as data frames for the handling of structured data. It is commonly employed for tasks related to data cleaning, exploration, and transformation. [5]

Scikit-Learn (sklearn): Scikit-Learn is a widely used machine-learning library for Python. It encompasses a diverse range of tools designed for tasks including classification, regression, clustering, model selection, and more. Additionally, it incorporates utilities for data preprocessing and the evaluation of machine learning models.

LabelEncoder (from sklearn.preprocessing): LabelEncoder, part of Scikit-Learn, is utilized to encode categorical labels into numerical values. It is commonly employed when addressing categorical data in machine learning models.

KNeighborsRegressor (from sklearn.neighbors): KNeighborsRegressor, a regression model within Scikit-Learn, implements the k-nearest neighbors algorithm for regression tasks. Predictions of the target variable are made by averaging the values of its k-nearest neighbors in the training data.

train_test_split (from sklearn.model_selection): The train_test_split function, available in Scikit-Learn, is employed for the division of a dataset into training and testing sets. This is an essential step in evaluating the performance of machine learning models.

StandardScaler (from sklearn.preprocessing): StandardScaler is a preprocessing technique used to standardize features by eliminating the mean and scaling them to have unit variance. This technique is particularly valuable when the features within a dataset possess differing scales or units. [6]

V. Development

First, the code for notebook initialization was developed. It included the importation of the numpy library as np, the pandas library as pd, and the matplotlib.pyplot library as plt. For dataset loading, a data frame named df was created. The dataset was read from the file located at C: disk using the specified encoding 'latin-1'. The 'latin-1' encoding, also known as ISO 8859-1, is used to interpret text data that includes characters from a wide range of Western European languages, making it suitable for datasets that may contain characters with diacritics and special symbols. This encoding ensures that the data is properly interpreted and displayed, especially when dealing with diverse text content.

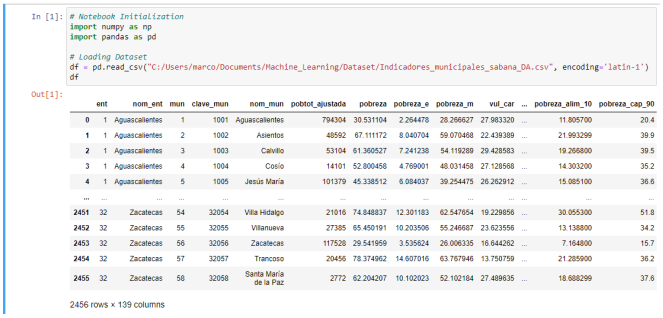


Fig. 2. Initialize Libraries and Loading Dataset

The next step, implied the use of df.iloc function in pandas, which facilitates data access in a DataFrame using numerical positions, allowing the selection of specific rows and columns using indices. With this, it was possible to select the analyzed data corresponding to the city of Mérida and place them at the end to be able to manipulate the dataset in an easier way later.

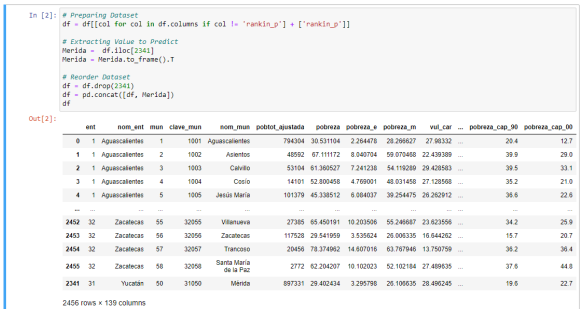


Fig. 3. Dataset Rearrangement

The code segment used LabelEncoder from the sklearn.preprocessing module. It was used to transform categorical text data into numerical values in specific columns in the DataFrame. Those are'nom_ent' (state name), 'nom_mun' (municipality name), and 'gdo_rezsoc00', 'gdo_rezsoc05', and 'gdo_rezsoc10' (degree of social underdevelopment in 2000, 2005, and 2010).

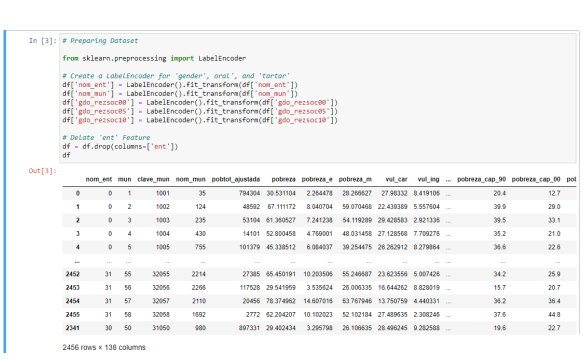


Fig. 4. Apply Label Encoder

Blank spaces with "NaN" values were found in the dataset. A mask could be implemented using the 'df.isna()' function. This mask took the form of a table identical in shape to the original df, where each entry was either marked as True if the corresponding element in df was a NaN value or False if it was not.

Then, each row in this mask was examined to determine if it contained at least one True value. This check effectively assessed the presence of NaN values in each row. The outcomes of this operation were stored in a new structure referred to as "NaN," essentially constituting a list of rows containing at least one NaN value. With this, the ability to locate and print the NaN values in the dataset was facilitated.

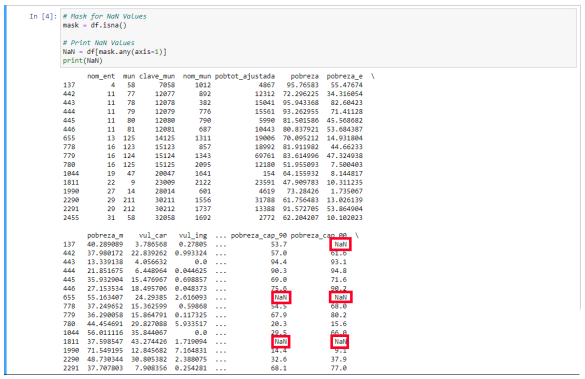


Fig. 5. Locate NaN Values

After that, the NaN values within the data frame were addressed by setting them to a mean value. This operation was carried out through the utilization of the statement df. fillna(df.mean(), inplace=True).

A comparative analysis was conducted and the results were printed. Specifically, a data frame named "NaN" was generated, consisting of rows in which the mask exhibited at least one True value. This data frame was then printed to provide information and confirm the absence of NaN values.

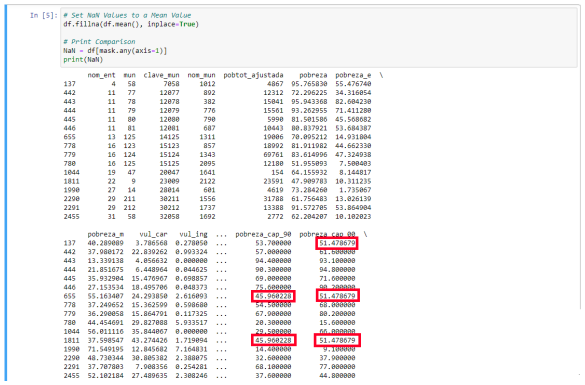


Fig. 6. Replace NaN Values with Mean Values

Subsequently, the data selection process was carried out using the iloc function to choose the data corresponding to the Features and the Target Variable. X was defined as "df.drop(columns=['rankin_p'])[:-1].values", encompassing all columns except the one associated with the Target Variable 'rankin_p' and the final row corresponding to the Features of Mérida.

Furthermore, Y was established as "df['rankin_p'][:-1].values," involving the extraction of all rows from the

'rankin_p' column, except for the last one, which pertained to the Target Variable intended for prediction.

```
In [6]:  # Features
         X = df.drop(columns=['rankin_p'])[:-1].values
         X

Out[6]: array([[0.00000000e+00, 1.00000000e+00, 1.00100000e+03, ...,
                 4.73000000e-01, 4.25000000e-01, 4.22628105e-01],
                [0.00000000e+00, 2.00000000e+00, 1.00200000e+03, ...,
                 3.70000000e-01, 5.33000000e-01, 3.43879312e-01],
                [0.00000000e+00, 3.00000000e+00, 1.00300000e+03, ...,
                 4.14000000e-01, 4.65000000e-01, 3.86708888e-01],
                ...,
                [3.10000000e+01, 5.60000000e+01, 3.20560000e+04, ...,
                 5.28000000e-01, 4.98000000e-01, 4.36338514e-01],
                [3.10000000e+01, 5.70000000e+01, 3.20570000e+04, ...,
                 3.80000000e-01, 4.83000000e-01, 3.65387093e-01],
                [3.10000000e+01, 5.80000000e+01, 3.20580000e+04, ...,
                 4.31000000e-01, 5.00000000e-01, 3.85065089e-01]])

In [7]: # Target Variables
        y = df['rankin_p'][:-1].values
        y

Out[7]: array([2351, 1340, 1601, ..., 2364,  837, 1570], dtype=int64)
```

Fig. 7. Define Features(x) and Target Variable(y)

Next, the Features corresponding to the prediction were declared, that is, the data related to the city of Mérida. Again the iloc function was used to take the last row of the dataset with the exception of the 'rankin_p' column belonging to the Target Variable.

```
In [8]: # Preparing Features for Prediction
        x_predict = df.iloc[-1, :-1].values
        x_predict

Out[8]: array([ 3.00000000e+01,  5.00000000e+01,  3.10500000e+04,  9.00000000e+02,
                9.97331000e+05,  2.94024339e+01,  3.29579000e+00,  2.61006347e+01,
                2.84962446e+01,  9.28258821e+00,  3.28187347e+01,  1.62292290e+01,
                2.07848168e+01,  4.14415120e+01,  8.02500825e+00,  1.05973159e+01,
                1.79631368e+01,  5.78086782e+01,  1.44037001e+01,  3.86850223e+01,
                7.68906192e+00,  2.31800000e+03,  2.63837000e+05,  2.95740000e+04,
                2.34263000e+05,  2.55770000e+05,  8.32900000e+04,  2.94493000e+05,
                1.45630000e+05,  1.86509000e+05,  3.71869000e+05,  7.20190000e+04,
                9.58930000e+04,  1.61180000e+05,  5.19543000e+05,  1.29240000e+05,
                3.47133000e+05,  6.82780000e+04,  1.60000000e+01,  4.90000000e+01,
                2.23650145e+00,  3.77261043e+00,  2.04344392e+00,  1.73050249e+00,
                2.29231238e+00,  2.58924127e+00,  2.22893857e+00,  3.26581144e+00,
                3.18025013e+00,  2.98879062e+00,  1.98695457e+00,  3.58365321e+00,
                1.70306361e+00,  2.32131100e+00,  7.05055000e+05,  7.81146000e+05,
                8.30732000e+05,  4.51180983e+00,  3.80778480e+00,  3.23050893e+00,
                3.91403007e+00,  2.58807397e+00,  2.51128006e+00,  4.13428017e+01,
                3.45963707e+01,  2.95828304e+01,  3.48741608e+01,  2.94262505e+01,
                2.32321000e+01,  1.52652999e+00,  6.66799128e-01,  8.12279959e-01,
                8.97474956e+00,  1.05878601e+01,  4.42426014e+00,  5.66362000e+00,
                3.86552889e+00,  2.42534995e+00,  1.59324799e+01,  8.11628056e+00,
                4.45038022e+00,  1.68393000e+00,  5.44163704e+00,  6.63950826e-01,
                2.90573692e+01,  2.33826504e+01,  2.02843800e+01,  1.60807099e+01,
                1.13477270e+00,  7.81405018e+00,  1.79238701e+00, -1.52265704e+00,
               -1.58490506e+00,  4.00000000e+00,  4.00000000e+00,  4.00000000e+00,
                2.38700000e+03,  2.38400000e+03,  2.39700000e+03,  2.15000000e+01,
                1.69000000e+01,  1.33105202e+01,  1.50000000e+01,  2.00878201e+01,
                3.70000000e+00,  1.10000000e+00,  6.06120978e-01,  3.90000000e+00,
                2.30000000e+00,  8.45049077e-01,  1.18000000e+01,  5.10000000e+00,
                8.28339978e-01,  1.97000000e+01,  1.22000000e+01,  7.47844082e+00,
                2.12000000e+01,  5.30000000e+00,  3.37759995e+00,  3.00000000e+01,
                1.78000000e+01,  4.65597010e+00,  4.30000000e+00,  1.30000000e+00,
                7.16769993e-01,  1.29000000e+01,  1.59000000e+01,  7.57110024e+00,
                1.96000000e+01,  2.27000000e+01,  1.38619001e+01,  4.22000000e+01,
                4.40000000e+01,  9.52354998e+01,  4.82000000e-01,  5.41000000e-01,
                4.15663126e-01])
```

Fig. 8. Prepare Features for Prediction

To implement the KNN algorithm from scratch, four functions were developed to build the Regression model. The first function, euclidean_distance(point1, point2), was responsible for computing the Euclidean distance between two data points, point1 and point2. This distance represented the straight-line distance between the two points in a multi-dimensional space. The code squared the differences between corresponding elements of the two data points using (point1 - point2) ** 2. It then summed these squared differences with np.sum and took the square root of the sum using np.sqrt, resulting in the Euclidean distance.

The second function, knn_regression(X_train, y_train, x_predict, k), was designed for KNN regression, which involved predicting continuous target variables. The function took four parameters: X_train (the training dataset with feature vectors), y_train (the corresponding target values for the training data), x_predict (a single data point for prediction), and k (the number of nearest neighbors to consider). It began by computing the Euclidean distances between x_predict and all data points in the X_train dataset. These distances were stored in a list. The k nearest neighbors were determined by sorting these distances and selecting the first k indices. The corresponding target values for these nearest neighbors were then retrieved from y_train. The function returned the mean of these target values as the predicted value for x_predict.

The third function, train_test_split_custom(X, y, test_size, random_state), was used to split a dataset into training and testing sets. It accepted the feature matrix X and the target values y, as well as two parameters: test_size (indicating the proportion of the dataset allocated for the testing set) and random_state (used for ensuring reproducible random shuffling). The function initialized the random number generator using np.random.seed(random_state), shuffled the indices of the dataset randomly, calculated the size of the test set, and selected the appropriate indices for the test and training sets. It then returned the feature and target data for both the training and testing sets.

The fourth function, standardize_features(X_train, X_test), standardized the features in both the training and testing datasets. This standardization process involved calculating the mean and standard deviation for each feature in the training set using np.mean and np.std. The mean was then subtracted from each data point, and the result was divided by the standard deviation for each feature in both datasets. This ensured that all features had a mean of 0 and a standard deviation of 1, making the data more suitable for modeling. The function returned the standardized training and testing feature datasets for use in subsequent modeling tasks.

```
In [9]: # Euclidean Distance between Two Points
        def euclidean_distance(point1, point2):
            return np.sqrt(np.sum((point1 - point2) ** 2))

        # Create a KNN Regression model
        def knn_regression(X_train, y_train, x_predict, k):
            distances = [euclidean_distance(x_predict, x) for x in X_train]
            nearest_indices = np.argsort(distances)[:k]
            nearest_neighbors = [y_train[i] for i in nearest_indices]
            return np.mean(nearest_neighbors)

        # Split Dataset
        def train_test_split_custom(X, y, test_size, random_state):
            np.random.seed(random_state)
            shuffled_indices = np.random.permutation(len(X))
            test_set_size = int(len(X) * test_size)
            test_indices = shuffled_indices[:test_set_size]
            train_indices = shuffled_indices[test_set_size:]
            X_train, X_test = X[train_indices], X[test_indices]
            y_train, y_test = y[train_indices], y[test_indices]
            return X_train, X_test, y_train, y_test

        # Standardize the features
        def standardize_features(X_train, X_test):
            mean = np.mean(X_train, axis=0)
            std_dev = np.std(X_train, axis=0)
            X_train = (X_train - mean) / std_dev
            X_test = (X_test - mean) / std_dev
            return X_train, X_test
```

Fig. 9. Scratch KNN Model Functions

To execution of the model began by splitting the dataset into training and testing sets using the custom 'train_test_split_custom' function. Approximately 20% of the data was designated for the testing set, and the random seed was set to 0 to ensure the reproducibility of the split. The training dataset was stored in 'X_train', the training target data in 'y_train', the testing feature data in 'X_test', and the testing target data in 'y_test'.

Subsequently, the features in both the training and testing sets were standardized using the 'standardize_features' function. This process was executed to normalize the data, making it more suitable for modeling.

To assess the performance of the K-nearest neighbors (KNN) Regressor model under various 'k_neighbors' values, the code initialized an empty list named 'mse_values' to hold the Mean Squared Error (MSE) values.

The code then entered a loop, traversing different values for 'k_neighbors' within the range of 1 to 10. For each value of 'k', the code followed a similar procedure. It employed list comprehension to compute predictions denoted as 'y_pred' for the testing set using the 'knn_regression' function, considering the specific 'k_neighbors' value. Following this, the code calculated the MSE by determining the mean of the

squared differences between the actual target values, ‘y_test‘, and the predicted values, ‘y_pred‘. The computed MSE was subsequently appended to the ‘mse_values‘ list.

After the loop, the code progressed to visualize the MSE values. The x-axis of the plot was labeled with the ‘neighbor_values‘ (the range of values for ‘k_neighbors‘), and the corresponding MSE values were presented on the y-axis. The plot was displayed with an accompanying title, axis labels, and grid lines.

Continuing with the code, it once again divided the dataset into training and testing sets and standardized the features to maintain consistency. Subsequently, a new function, ‘knn_predict(X_train, y_train, X_test, k)‘, was defined for generating predictions using the KNN Regressor model. This function calculated predictions, ‘y_pred‘, for the testing set based on the specified value of ‘k_neighbors‘.

The code then proceeded to make predictions using the ‘knn_predict‘ function, specifying a value of 5 for ‘k_neighbors‘. The predicted values were stored in the ‘y_pred‘ variable.

To evaluate the model’s performance, the code computed the MSE once more by determining the mean of the squared differences between the actual target values, ‘y_test‘, and the predicted values, ‘y_pred‘. Finally, the MSE was printed to the console, providing an assessment of the model’s accuracy with the message ”Mean Squared Error: mse”.

```
In [10]: # Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split_custom(x, y, test_size=0.2, random_state=0)

# Standardize the features
X_train, X_test = standardize_features(X_train, X_test)

# Store MSE values
mse_values = []

# Try different values for k_neighbors
neighbor_values = range(1, 11)

for k in neighbor_values:
    y_pred = np.array([knn_regression(X_train, y_train, x, k) for x in X_test])

    # Calculate the MSE
    mse = np.mean((y_test - y_pred) ** 2)
    mse_values.append(mse)

# Plot the MSE values
plt.figure(figsize=(8, 6))
plt.plot(neighbor_values, mse_values, marker='o')
plt.title('MSE for Different Values of n_neighbors')
plt.xlabel('Number of Neighbors (KNN)')
plt.ylabel('Mean Squared Error (MSE)')
plt.grid()
plt.show()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split_custom(x, y, test_size=0.2, random_state=0)

# Standardize the features
X_train, X_test = standardize_features(X_train, X_test)

# Create a KNN Regressor model
def knn_predict(X_train, y_train, X_test, k):
    y_pred = [knn_regression(X_train, y_train, x, k) for x in X_test]
    return y_pred

# Make predictions
y_pred = knn_predict(X_train, y_train, X_test, 5)

# Evaluate the model
mse = np.mean((y_test - y_pred) ** 2)
print(f"Mean Squared Error: {mse}")
```

Fig. 10. Scratch KNN Model Implementation

To perform the predictions, the x_predict data was reshaped to (1, -1) using the x_predict.reshape(1, -1) method to ensure that the input data was in the appropriate format. This reshaping ensured that the data was treated as a single sample with multiple features.

Subsequently, the reshaped x_predict data was standardized. This involved subtracting the mean of the training feature data, X_train, from the x_predict data. The subtraction was carried out as (x_predict - np.mean(X_train, axis=0)). Additionally, the result was divided by the standard deviation of the training feature data, achieved by using / np.std(X_train, axis=0). The standardization process was employed to make the x_predict data compatible with the standardized training data.

After standardization, a prediction was made. The knn_predict function was called with X_train and y_train for the training data, the standardized x_predict data, and a specified value of 5 for k_neighbors. The result was stored in the variable predicted_ranking.

Finally, the predicted Mérida Poverty Ranking was printed to the console using the statement print(”Predicted Mérida Poverty Ranking:”, predicted_ranking). This step provided the predicted value for the given input data and allowed for the interpretation of the model’s performance.

```
In [11]: # Prediction
x_predict = x_predict.reshape(1, -1)
x_predict = (x_predict - np.mean(X_train, axis=0)) / np.std(X_train, axis=0)
predicted_ranking = knn_predict(X_train, y_train, x_predict, 5)
print("Predicted Mérida Poverty Ranking:", predicted_ranking)

Predicted Mérida Poverty Ranking: [2277.8]
```

Fig. 11. Scratch KNN Model Prediction

To perform a regression model using the scikit-learn library for machine learning first was necessary to insert the necessary modules, including ‘KNeighborsRegressor‘ for KNN regression, ‘train_test_split‘ for data splitting, ‘StandardScaler‘ for feature standardization, and ‘mean_squared_error‘ for calculating the Mean Squared Error (MSE).

In the subsequent section, the code conducted a dataset split. It utilized the ‘train_test_split‘ function to divide the dataset into training and testing sets. Approximately 20% of the data was allocated to the testing set, and a random seed of 0 was specified for the purpose of reproducibility. The feature data for training and testing were denoted as ‘X_train‘ and ‘X_test‘, while the target data was represented by ‘y_train‘ and ‘y_test‘.

Following the data split, the code proceeded to standardize the features. It employed the ‘StandardScaler‘ from scikit-learn to achieve this. The training feature data, ‘X_train‘, was standardized using ‘scaler.fit_transform(X_train)‘, and the testing feature data, ‘X_test‘, was standardized using ‘scaler.transform(X_test)‘. This standardization was necessary to ensure that all features had a mean of 0 and a standard deviation of 1, preparing the data for modeling.

The code then prepared to store the Mean Squared Error (MSE) values in the ‘mse_values‘ list. To assess the model’s performance under various ‘k_neighbors‘ values, a range of values from 1 to 10 was established using the ‘neighbor_values‘ variable.

A loop was initiated to iterate through the range of ‘k_neighbors‘ values. Within each iteration, a KNN model (‘knn_model‘) was created using the ‘KNeighborsRegressor‘ class with the specified value of ‘k_neighbors‘. The model was then fitted with the standardized training data, ‘X_train‘, and the corresponding target data, ‘y_train‘.

Subsequently, predictions were made using the fitted KNN model. Predictions for the testing data, ‘X_test‘, were calculated with ‘y_pred = knn_model.predict(X_test)‘.

The code also calculated the MSE for each value of ‘k_neighbors‘ by comparing the predicted values, ‘y_pred‘, to the actual target values, ‘y_test‘, using the ‘mean_squared_error‘ function. The computed MSE for each value was appended to the ‘mse_values‘ list.

For visual assessment, the MSE values were plotted using the ‘matplotlib‘ library. A plot was created, illustrating the relationship between the number of neighbors (‘k_neighbors‘)

and the corresponding MSE values. The plot was configured with appropriate labels and grid lines.

Subsequently, a KNN Regressor model (‘knn_model‘) was constructed with a specified value of 5 for ‘k_neighbors‘. The model was fitted with the standardized training data, ‘X_train‘.

A predictor, ‘y_pred‘, was defined, and it calculated predictions using the KNN model for the testing data, ‘X_test‘.

To evaluate the model’s performance, the code calculated the MSE by comparing the predicted values, ‘y_pred‘, to the actual target values, ‘y_test‘, and stored it in the variable ‘mse‘.

Finally, the code printed the MSE to the console, providing an assessment of the model’s accuracy with the message ”Mean Squared Error: mse”.

```
In [12]: from sklearn.neighbors import KNeighborsRegressor
         from sklearn.model_selection import train_test_split
         from sklearn.preprocessing import StandardScaler
         from sklearn.metrics import mean_squared_error

         # Split Dataset
         X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)

         # Standardize the features
         scaler = StandardScaler()
         X_train = scaler.fit_transform(X_train)
         X_test = scaler.transform(X_test)

         # Store MSE values
         mse_values = []

         # Try different values for k_neighbors
         neighbor_values = range(1, 11)

         for k in neighbor_values:
             knn_model = KNeighborsRegressor(n_neighbors=k)
             knn_model.fit(X_train, y_train)

             # Make predictions
             y_pred = knn_model.predict(X_test)

             # Calculate the MSE
             mse = mean_squared_error(y_test, y_pred)
             mse_values.append(mse)

         # Plot the MSE values
         plt.figure(figsize=(8, 6))
         plt.plot(neighbor_values, mse_values, marker='o')
         plt.title('MSE for Different Values of n_neighbors')
         plt.xlabel('Number of Neighbors (KNN)')
         plt.ylabel('Mean Squared Error (MSE)')
         plt.grid()
         plt.show()

         # Create a KNN Regressor model
         knn_model = KNeighborsRegressor(n_neighbors=5)
         knn_model.fit(X_train, y_train)

         # Define Predictor
         y_pred = knn_model.predict(X_test)

         # Evaluate the model
         mse = mean_squared_error(y_test, y_pred)
         print(f"Mean Squared Error: {mse}")
```

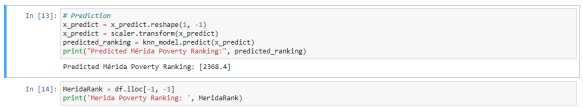
Fig. 12. KNN Model Development

The prediction task was carried out using the scikit-learn library. The process began with the reshaping of x_predict, where it was transformed into a format suitable for prediction by employing the .reshape(1, -1) operation. This allowed x_predict to be treated as a single sample with multiple features, facilitating compatibility with the prediction model.

Following the reshaping, the next step involved standardizing x_predict. The previously created scaler was utilized for this purpose, which had been previously fit and transformed based on the training data. The standardized x_predict data was obtained using the scaler.transform(x_predict) operation. This step ensured that the input data adhered to the same standardization as the training data, making it suitable for use with the trained model.

Subsequently, a prediction was made using the KNN Regressor model, knn_model. The knn_model.predict(x_predict) function was employed to calculate the predicted values. These predictions represented the model’s estimate of Mérida’s Poverty Ranking based on the input data x_predict.

The code concluded by printing the predicted Mérida Poverty Ranking to the console as same as the Target Value of the prediction found in the dataset, in order to compare it with the results generated by both models. For this, the iloc function was used to take the last row of the last column of the dataset.



```
In [13]: # Prediction
         x_predict = x_predict.reshape(1, -1)
         x_predict = scaler.transform(x_predict)
         predicted_ranking = knn_model.predict(x_predict)
         print("Predicted Mérida Poverty Ranking:", predicted_ranking)

         Predicted Mérida Poverty Ranking: [2368.4]

In [14]: MeridaRank = df.iloc[-1, -1]
         print('Merida Poverty Ranking: ', MeridaRank)

         Merida Poverty Ranking:  2367
```

Fig. 13. KNN Model Prediction

VI. Conclusion

In the previous sections, two different approaches to predicting the national poverty ranking of Mérida, Mexico, were discussed. The first approach involved building a K-Nearest Neighbors (KNN) regression model from scratch, while the second approach utilized the scikit-learn library to implement a KNN regressor. The results from both methods were compared to understand the performance and outcomes of the models.

The custom KNN implementation yielded a predicted poverty ranking of 2277.8 for Mérida. On the other hand, the scikit-learn KNN regressor predicted a ranking of 2368.4. The actual poverty ranking of Mérida was 2367.

The difference in predicted rankings between the two methods is relatively small, especially considering the range of poverty rankings. This suggests that both approaches provide reasonably accurate predictions of Mérida’s national poverty ranking based on the provided dataset. However, the scikit-learn implementation appears to be slightly more accurate, as it is closer to the actual ranking.

Several factors could contribute to the small disparities between the predicted and actual rankings. One key factor is the choice of the number of neighbors (k) in the KNN algorithm. In both implementations, a value of 5 was selected for k, but tuning this parameter might lead to improved accuracy. Additionally, the quality and representativeness of the dataset play a crucial role in the accuracy of the predictions. Further data preprocessing and feature engineering might enhance the model’s performance.

During the project, one of the main challenges was dealing with missing data in the dataset. Missing values were identified and replaced with mean values to ensure the dataset’s completeness. Moreover, standardizing features and splitting the dataset into training and testing sets were essential steps in model development. Balancing the need for computational efficiency and model accuracy was another challenge, especially when working with large datasets.

In the context of robotics, machine learning models like KNN regressors can be applied in various ways. For instance, in the field of robotics, these models can be used for robot localization and navigation tasks. Robots can utilize regression algorithms to predict their position based on sensor data and environmental features. Additionally, KNN-based regression models can assist in object recognition, enabling robots to identify and interact with objects in their environment. The ability to predict and adapt based on data is a critical aspect of robotics, and machine learning techniques like KNN regression can contribute to improved decision-making and autonomy in robotic systems.

APPENDIX A
GITHUB REPOSITORY

In the following link, it is possible to access the files used and generated during this project. The GitHub repository contains a readme.txt document whose content is the description of the other files:

https://github.com/Maages09/ScratchCoding.git

APPENDIX B
PROPOSAL EVIDENCE

Here is a screenshot of the comment made in the Teams publication as a proposal for the development of the exercise, where the target, the problem (regression or classification), and the question that the predictor answered were selected.

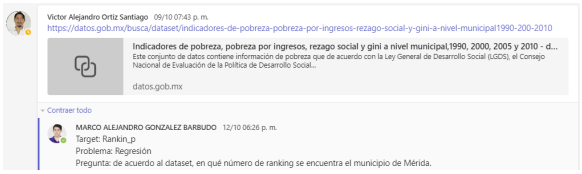


Fig. 14. Proposal Evidence

REFERENCES

[1] W. Xindong and K. Vipin, Eds., The top ten algorithms in data mining. Philadelphia, PA: Chapman & Hall/CRC, 2009.

[2] CONEVAL. "Indicadores de pobreza, pobreza por ingresos, rezago social y gini a nivel municipal, 1990, 2000, 2005 y 2010", Datos.gob.mx, 2017-10-12. [Online]. Available: https://datos.gob.mx/busca/dataset/indicadores-de-pobreza-pobreza-por-ingresos-rezago-social-y-gini-a-nivel-municipal1990-200-2010

[3] Anaconda Inc., "Anaconda Software Distribution," Anaconda Documentation, Vers. 2-2.4.0, 2020. [Online]. Available: https://docs.anaconda.com/. Accessed on: [Accessed Oct. 16, 2023].

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," Nature, vol. 585, no. 7825, pp. 357-362, Sep. 2020. doi: 10.1038/s41586-020-2649-2.

[5] The pandas development team, "pandas-dev/pandas: Pandas," Zenodo, Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134. [Accessed Oct. 16, 2023].

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.