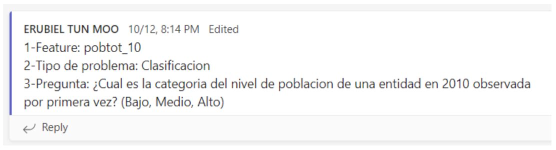
Implementing a Predictor from scratch

Erubiel Tun Moo
Computational Robotics Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 7193. CP 97357
Ucú, Yucatán. México
Email: 2009137@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

Implementing a Predictor from scratch

I. Screenshot of the Post



ERUBIEL TUN MOO 10/12, 8:14 PM Edited
1-Feature: pobtot_10
2-Tipo de problema: Clasificacion
3-Pregunta: ¿Cual es la categoria del nivel de poblacion de una entidad en 2010 observada por primera vez? (Bajo, Medio, Alto)

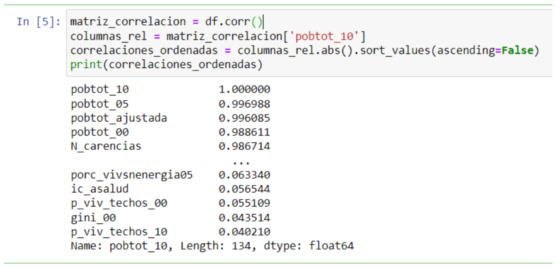
Fig. 1. First code

This is my post, in my particular case I chose as Target the feature: pobtot_10 which means the Total Population of the entity in 2010, using this feature as my Target I plan to implement supervised Machine Learning specifically for a classification problem which can show as a result what the population level category of an entity in 2010 would be observed for the first time; This level can be a low, medium or high level.

II. Steps followed to obtain the final result.

A. Dataset cleaning

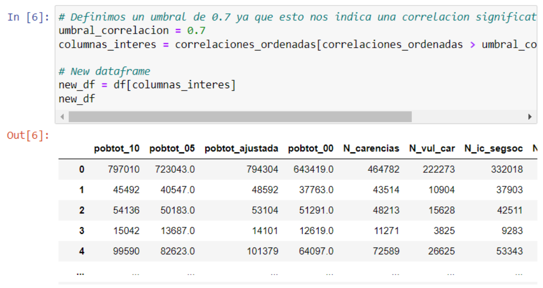
Cleaning the dataset is an extremely important step and consists of the process of preparing the data to subsequently apply any model; Because we had a dataset with 139 feature columns, I chose to apply a correlation matrix to the entire dataset and obtain how much relationship all the feature columns have with my column that I chose as feature target ('pobtot_10') such as shown in the following figure:



```
In [5]: matriz_correlacion = df.corr()
        columnas_rel = matriz_correlacion['pobtot_10']
        correlaciones_ordenadas = columnas_rel.abs().sort_values(ascending=False)
        print(correlaciones_ordenadas)

        pobtot_10              1.000000
        pobtot_05              0.996988
        pobtot_ajustada        0.996085
        pobtot_00              0.988611
        N_carencias            0.986714
                                 ...
        porc_vivsnenergia05    0.063340
        ic_asalud              0.056544
        p_viv_techos_00        0.055109
        gini_00                0.043514
        p_viv_techos_10        0.040210
        Name: pobtot_10, Length: 134, dtype: float64
```

Fig. 2. Correlation matrix

After obtaining the results of the correlation matrix, while investigating I realized that starting with a threshold of 0.7 indicates a significant correlation; In this way I established that from the results of the correlation matrix all feature columns above the 0.7 correlation threshold are preserved to eliminate unnecessary columns from the complete dataset and create a new dataset with only the columns of interest (new _df) in this way we are left with only 17 columns of features plus 1 column of the target, that is, 18 columns in total.

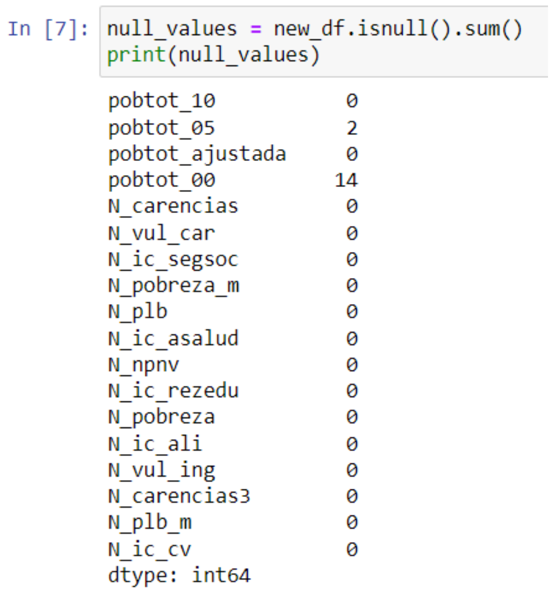


```
In [6]: # Definimos un umbral de 0.7 ya que esto nos indica una correlacion significat
        umbral_correlacion = 0.7
        columnas_interes = correlaciones_ordenadas[correlaciones_ordenadas > umbral_co

        # New dataframe
        new_df = df[columnas_interes]
        new_df

Out[6]:
```

	pobtot_10	pobtot_05	pobtot_ajustada	pobtot_00	N_carencias	N_vul_car	N_ic_segsoc	N
0	797010	723043.0	794304	643419.0	464782	222273	332018	
1	45492	40547.0	48592	37763.0	43514	10904	37903	
2	54136	50183.0	53104	51291.0	48213	15628	42511	
3	15042	13687.0	14101	12619.0	11271	3825	9283	
4	99590	82623.0	101379	64097.0	72589	26625	53343	
...	...	...	...	...	...	...	...	

Fig. 3. Correlation matrix

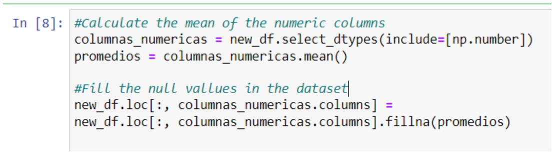
After removing unnecessary columns, the next thing is to check and identify null values; That is, it is first necessary to identify those empty values and then fill those values to avoid these null values in the dataset. In this process, it was identified which columns have null values and how many null values each column has, as shown in the following image in which we can see that the feature "pobtot_05" contains 2 null values and "pobtot_00" contains 14 null values.



```
In [7]: null_values = new_df.isnull().sum()
        print(null_values)

        pobtot_10           0
        pobtot_05           2
        pobtot_ajustada     0
        pobtot_00          14
        N_carencias         0
        N_vul_car           0
        N_ic_segsoc         0
        N_pobreza_m         0
        N_plb               0
        N_ic_asalud         0
        N_npnv              0
        N_ic_rezedu         0
        N_pobreza           0
        N_ic_ali            0
        N_vul_ing           0
        N_carencias3        0
        N_plb_m             0
        N_ic_cv             0
        dtype: int64
```

Fig. 4. Identify null values

Once we have identified the null values in each column, it is crucial to proceed to fill them to complete the data cleaning stage. In this context, it was decided to use the average of the respective column that contains the null value as the imputation method; This process means that, for example, in the "pobtot_10" feature, the missing values have been replaced by the mean calculated from all the values in that column. This process was applied consistently to address the replacement of all null values in the dataset that we will use later.



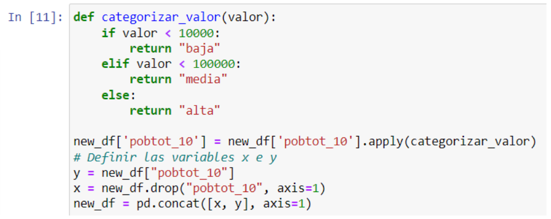
```
In [8]: #Calculate the mean of the numeric columns
        columnas_numericas = new_df.select_dtypes(include=[np.number])
        promedios = columnas_numericas.mean()

        #Fill the null vallues in the dataset
        new_df.loc[:, columnas_numericas.columns] =
        new_df.loc[:, columnas_numericas.columns].fillna(promedios)
```

Fig. 5.   Fill null values

Fig. 5 shows how the process was carried out to fill these null values with the average of the columns mentioned above. In my case, I only cleaned the data set in this way, but it is still important to detect outliers within the dataset we will work with to avoid errors and obtain inaccurate predictions; However, I did not carry out this process since when I was investigating I realized that outliers are not always bad and in the same way if we do not know the data it is better not to apply this process since we could lose important information when performing the training, I opted for Do not eliminate the ouliers since this data most of the time tells us the number of people with some characteristic or the number of people within a locality.

B. *Predictor training and performance evaluation*

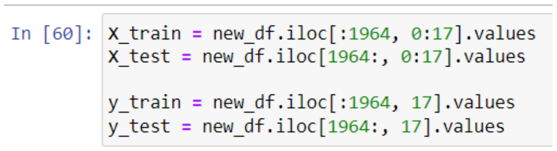
The next step after cleaning the data set, because my problem focuses on a classification problem, it is essential to change the numerical values to categories in my target feature; To do this, I established different limits in such a way that depending on the value of my target value, these will be classified as ”low”, ”medium” or ”high”, as can be seen in the following image.



```
In [11]: def categorizar_valor(valor):
             if valor < 10000:
                 return "baja"
             elif valor < 100000:
                 return "media"
             else:
                 return "alta"

         new_df['pobtot_10'] = new_df['pobtot_10'].apply(categorizar_valor)
         # Definir las variables x e y
         y = new_df["pobtot_10"]
         x = new_df.drop("pobtot_10", axis=1)
         new_df = pd.concat([x, y], axis=1)
```

Fig. 6.   Values-Categories

Since the dataset is ready to apply the model, now we must define our feature taget as ”y” and the other feature columns as ”x” to apply the KNN model. As can be seen in Fig. 7, we define the x and y data that will be used for training and testing; To carry out this step we divide the dataset into 80% for training and 20% for testing.

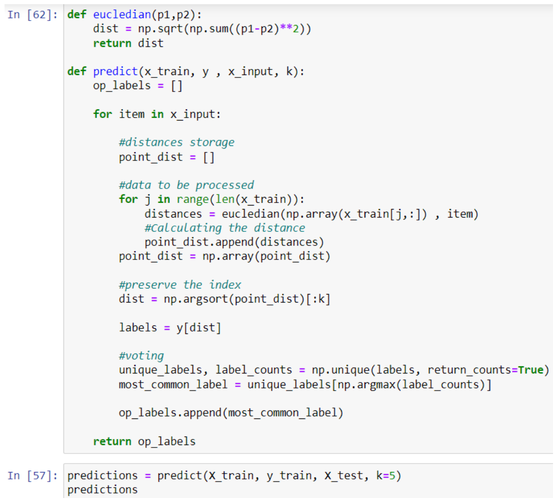


```
In [60]: x_train = new_df.iloc[:1964, 0:17].values
         x_test = new_df.iloc[1964:, 0:17].values

         y_train = new_df.iloc[:1964, 17].values
         y_test = new_df.iloc[1964:, 17].values
```

Fig. 7.   Train and test

From now on it is time to train the predictor, for this the ’eucledian’ function is established to calculate the Euclidean distance between two points p1 and p2 in the feature space, this Euclidean distance is a common metric to measure proximity between points in a Euclidean space; Subsequently, the predict function is used, taking as input the training set x_train, the training labels y, the new input instances x_input, and the value of k, which represents the number of nearest neighbors to consider for the classification; subsequently an empty list is created to store the predicted labels for the input instances, on the other hand a loop is created to iterate through each instance in x_input that you want to classify and the distance between that instance and all the instances is calculated. instances in the training set x_train to store in a list. For the selection of K nearest neighbors, the calculated distance values are used to find the indices of the k closest instances in the training set and these indices are stored in the ’dist’ variable using np.argsort to later recover the labels. corresponding to the k closest instances in the training set and store them in the variable labels, after this process the occurrences of each label in the set of nearest neighbors are counted and the most common label is selected as the predicted label for the current instance. This label is stored in the op_labels list and finally the op.labels list is returned which contains the predicted labels for all instances in ”x_input”.

In order to see the precision of the model, the following lines were simply used within the code: accuracy = accuracy_score(y_test, predictions)



```
In [62]: def eucledian(p1,p2):
             dist = np.sqrt(np.sum((p1-p2)**2))
             return dist

         def predict(x_train, y , x_input, k):
             op_labels = []

             for item in x_input:

                 #distances storage
                 point_dist = []

                 #data to be processed
                 for j in range(len(x_train)):
                     distances = eucledian(np.array(x_train[j,:]) , item)
                     #Calculating the distance
                     point_dist.append(distances)
                 point_dist = np.array(point_dist)

                 #preserve the index
                 dist = np.argsort(point_dist)[:k]

                 labels = y[dist]

                 #voting
                 unique_labels, label_counts = np.unique(labels, return_counts=True)
                 most_common_label = unique_labels[np.argmax(label_counts)]

                 op_labels.append(most_common_label)

             return op_labels

In [57]: predictions = predict(X_train, y_train, X_test, k=5)
         predictions
```

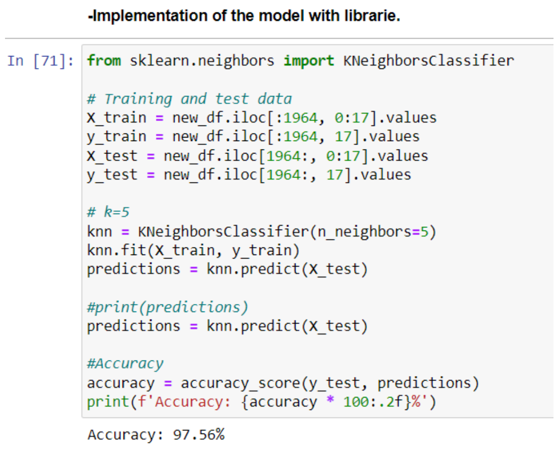
Fig. 8.   Predictor code

In the end I decided to use the KNN algorithm, because it is a simple and easy to understand algorithm and does not require a complex model to train; Likewise, this instance-based learning algorithm does not try to learn a mapping function from features to classes, but instead stores the training data and classifies new instances based on the similarity with the training instances in this particular case. This is very useful since even though some features have a close relationship and the only change is the year, we also have relationships between features and classes that we do not know.

C. *Predictor training with librarie*

Previously, the calculation was carried out, implementing the KNN algorithm step by step. This section explains the use of the sklearn library to perform the training of the same model; For this, the KNeighborsClassifier class from the neighbors module of scikit-learn is first imported with the objective of training and creating a KNN classifier. The next step is to divide the data from the clean data set in such a way that 80% are used for training and 20% for testing, this must be done for the characteristics ”x” and label ”y”, where X_train and y_train contain the training data, and X_test and y_test contain the test data.

Subsequently, a KNN classifier object is created with 5 nearest neighbors which fits the training data using the .fit() method, which means that the classifier will learn from this data to make predictions; As for the predictions on the test set, the features of (’X_test’) are used to make the rating predictions using the trained KNN model and the predicted labels are stored in ’predictions’.

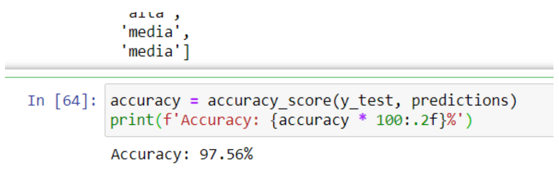


-Implementation of the model with librarie.

```
In [71]: from sklearn.neighbors import KNeighborsClassifier

         # Training and test data
         X_train = new_df.iloc[:1964, 0:17].values
         y_train = new_df.iloc[:1964, 17].values
         X_test = new_df.iloc[1964:, 0:17].values
         y_test = new_df.iloc[1964:, 17].values

         # k=5
         knn = KNeighborsClassifier(n_neighbors=5)
         knn.fit(X_train, y_train)
         predictions = knn.predict(X_test)

         #print(predictions)
         predictions = knn.predict(X_test)

         #Accuracy
         accuracy = accuracy_score(y_test, predictions)
         print(f'Accuracy: {accuracy * 100:.2f}%')

         Accuracy: 97.56%
```

Fig. 9. Predictor code using librarie

Finally, in order to compare this code with the previous one, the accuracy of the model is calculated by comparing the real labels of the test set (y_test) with the predicted labels (predictions) using the accuracy_score method of scikit-learn and by displaying the As a result of the precision of the model, we can realize that we obtained the same result in both cases, that is, a precision of 97.56% was obtained in both cases.



```
          'media',
          'media']

In [64]: accuracy = accuracy_score(y_test, predictions)
         print(f'Accuracy: {accuracy * 100:.2f}%')

         Accuracy: 97.56%
```

Fig. 10. Accuracy of previous model

III. Code in Github.

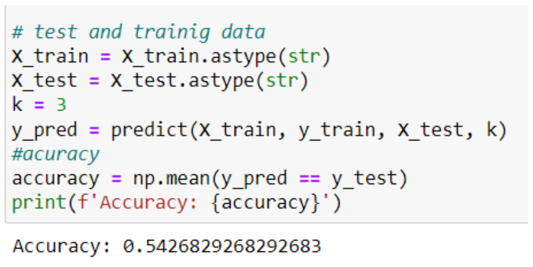
Here you can see the code made for this activity.

.ipynb: https://github.com/Erubiel-sudo/Machine-Learning-/blob/main/Implementing%20a%20predictor.ipynb

.py: https://github.com/Erubiel-sudo/Machine-Learning-/blob/main/Implementing%20a%20predictor.py

IV. Reflexion.

This activity has allowed me to learn more concepts about how the supervised part of Machnine learnig is implemented to solve classification problems; Throughout this task I had several complications in order to clean the dataset, at first I did not know how to eliminate unnecessary characteristics and when to fill in the null values, so I consider that cleaning the dataset was the most complicated part of this task. Another complication was when implementing the KNN algorithm since for this problem a perceptron model could also be implemented, however after deciding to use KNN I found myself with the difficult decision between whether to use the Euclidean metric or the Hamming metric, due to Because I didn’t know exactly which metric would have a better result, I decided to try both, obtaining a precision of 54.26% with the Hamming metric (Fig.11) and a precision of 97.56% with the Euclidean metric. Carrying out this activity has allowed me to learn more about this topic and the importance of cleaning the data in the dataset to obtain good precision as a result.



```
# test and trainig data
X_train = X_train.astype(str)
X_test = X_test.astype(str)
k = 3
y_pred = predict(X_train, y_train, X_test, k)
#acuracy
accuracy = np.mean(y_pred == y_test)
print(f'Accuracy: {accuracy}')

Accuracy: 0.5426829268292683
```

Fig. 11. Accuracy using Hamming metric