Implementing a Predictor from scratch

Mauricio Daniel Zaldívar Medina
Robotics Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: 2009147@upy.edu.mx

Victor Alejandro Ortiz Santiago
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: victor.ortiz@upy.edu.mx

Abstract

This report outlines a predictive modeling exploration using municipal indicators. The objective is to predict vulnerability levels, emphasizing key data science steps, such as preprocessing, custom algorithm implementation (K-NN and Perceptron), and performance evaluation. Challenges, including handling missing data and accommodating multiple vulnerability categories, are discussed. The report concludes with a reflection on the broader applicability of acquired skills in robotics, highlighting their potential impact on the development of intelligent systems for dynamic environments.

Index Terms

Predictive Modeling, Data Science Pipeline, Preprocessing, K-NN, Perceptron, Machine Learning, Dataset, Categorical Data Processing, Model Accuracy, Feature Selection, Scikit-Learn.

Implementing a Predictor from scratch

I. Introduction

This report encapsulates a data-driven journey into predictive modeling, centering on municipal indicators from the dataset. Our focus is on predicting vulnerability levels denoted by the 'vul-ing' column. From initial data preprocessing to the implementation and evaluation of custom machine learning algorithms, we navigate through essential steps in the data science pipeline.

Challenges, including the handling of missing values and the attempt to accommodate four vulnerability categories, are addressed. The report concludes with a reflection on the broader applicability of acquired skills in robotics, emphasizing their potential impact on the development of intelligent systems for dynamic environments.

II. Methods and Tools

A. Data Collection

The dataset used in this study comprises municipal indicators, providing valuable insights into vulnerability levels. The specific attributes were examined to identify key features, notably focusing on the `vul_ing` column, which represents vulnerability of the people by their earnings.

B. Data Preprocessing

1) Feature Selection: Initial exploration involved identifying and dropping irrelevant features, such as state and municipality names, to streamline the dataset.

2) Categorical Data Processing: Categorical columns (`gdo_rezsoc00`, `gdo_rezsoc05`, `gdo_rezsoc10`) were numerically encoded using predefined mappings to facilitate machine learning model training.

3) Handling Missing Values: Missing values were addressed by filling the `vul_ing` column with its mean and imputing other missing values with the mean of their respective columns.

C. Model Implementation

1) K-NN Algorithm: A custom K-Nearest Neighbors (K-NN) algorithm was implemented for its simplicity and suitability for classification tasks.

2) Perceptron Model: A Perceptron model was implemented to explore its effectiveness for binary classification, with adjustments made for compatibility with the dataset.

D. Performance Evaluation

1) Dataset Splitting: The dataset was divided into training and testing sets, with an 80-20 split ratio, ensuring the model's ability to generalize.

2) K-NN Model Evaluation: The custom K-NN model was evaluated using the test set, and accuracy was computed using the scikit-learn `accuracy_score` metric for comparison.

3) Perceptron Model Evaluation: The Perceptron model was trained on the training set and evaluated on the test set. Model accuracy was measured using the `accuracy_score` metric.

E. Libraries and Tools

The `scikit-learn` library was employed for critical tasks such as dataset splitting, feature scaling, and as a benchmark for model performance.

F. Challenges and Reflection

1) Handling Four Vulnerability Categories: An initial attempt to accommodate four vulnerability categories posed a challenge due to existing mapping limitations, leading to a simplified binary classification.

2) Algorithm Selection: The choice of algorithms (K-NN and Perceptron) was based on their simplicity and effectiveness for the initial exploration of vulnerability prediction.

3) Real-world Applicability: A reflection on challenges and acquired skills considers the broader applicability of the developed models, especially in the context of robotics and intelligent systems for dynamic environments.

III. Development

The development phase meticulously involved preparing the dataset, training predictors, and evaluating their performance, combining both custom algorithms and established libraries.

A. Dataset Preparation

Initiating with dataset loading, a thorough examination was conducted to decide feature inclusion or exclusion. Features like 'nom_ent' and 'nom_mun' were excluded for being non-contributory, while categorical columns (`gdo_rezsoc00`, `gdo_rezsoc05`, `gdo_rezsoc10`) underwent numerical encoding. The rationale behind each feature decision aimed at streamlining the dataset for effective predictive modeling.

B. Training Predictors

Two key predictors, a custom K-NN algorithm and a Perceptron model, were implemented. The choice of the K-NN algorithm was motivated by its simplicity and suitability for classification tasks, with Euclidean distance used for neighbor proximity. The Perceptron model, designed for binary classification, was selected for its adaptability. The training process involved iterative updates to weights and bias, facilitating the model's learning.

C. Performance Evaluation

To assess the efficacy of the models, a comprehensive performance evaluation was conducted using a dedicated test subset. The final accuracy of the K-NN model reached *92.28%*, and the Perceptron model achieved *96.75%*, demonstrating their ability to capture patterns and make accurate predictions.

D. Library Utilization and Comparison

Scikit-learn, being a widely adopted machine learning library, played a pivotal role in key tasks such as dataset splitting and standardization throughout the development phase. This ensured a robust foundation for the custom models, allowing for a reliable performance comparison with scikit-learn's implementations.

One of the notable challenges faced during the project was the need to accommodate four vulnerability categories. This factor significantly influenced decision-making processes, particularly in the realms of feature selection and model simplification. Navigating through these challenges became an integral part of the development journey, shaping the strategies employed in various stages, from dataset preparation to algorithm selection and performance evaluation.

It's worth highlighting that the custom models outperformed their scikit-learn counterparts in terms of accuracy. Specifically, the k-nearest neighbors (KNN) model achieved an accuracy of 92.28%, surpassing the scikit-learn implementation's 91.86%. Similarly, the perceptron model exhibited superior performance, achieving an accuracy of 96.75%, compared to the scikit-learn perceptron's accuracy of 96%.

This improvement underscores the efficacy of the custom models in addressing the specific challenges posed by the dataset and vulnerability categories. The meticulous approach taken in the development process, coupled with the performance benchmarks against established libraries, provides a holistic view of the predictive modeling journey undertaken in this project.

IV. Results

The results of the predictive modeling efforts are presented, highlighting the performance metrics of the custom K-NN and Perceptron models, along with a comparison to scikit-learn's implementations. The Perceptron model, designed for binary classification, achieved an accuracy of *93.41%*. Despite challenges, such as accommodating four vulnerability categories, the decision to simplify the classification task proved pragmatic. Reflection on the success of the models underscores their potential applications, particularly in urban planning and resource allocation. The acquired skills in custom algorithm implementation and model evaluation contribute to addressing complex real-world challenges.

V. Conclusion

In conclusion, this report outlines a journey through predictive modeling using municipal indicators to predict vulnerability levels. We tackled challenges, like handling missing values and accommodating multiple vulnerability categories, and implemented custom K-NN and Perceptron models.

Despite challenges, the models performed well, with the Perceptron model achieving above 90% accuracy, outperforming scikit-learn counterparts. The decision to simplify the classification task proved practical, showcasing potential applications in urban planning.

This project highlights the importance of a systematic data science approach and the effectiveness of custom models in real-world problem-solving. The skills acquired form a strong foundation for future exploration and improvement of predictive modeling techniques in practical data science applications.

Appendix A
Code

```python
def knn_predict(train_data, train_labels, new_data, K):
    distances_and_labels = []

    for data_point, label in zip(train_data, train_labels):
        distance = np.linalg.norm(new_data - data_point)  # Euclidean distance calculation
        distances_and_labels.append((distance, label))

    distances_and_labels.sort(key=lambda x: x[0])  # Sort by distance

    k_nearest_neighbors = distances_and_labels[:K]

    neighbor_labels = [neighbor[1] for neighbor in k_nearest_neighbors]

    predicted_label = max(set(neighbor_labels), key=neighbor_labels.count)  # Majority vote

    return predicted_label


def calculate_accuracy(predictions, true_labels):
    correct_predictions = np.sum(predictions == true_labels)
    total_predictions = len(true_labels)
    accuracy = (correct_predictions / total_predictions) * 100
    return accuracy

def step_function(y):
    return 1 if y >= 0 else 0
```

```python
def update_weights(weights, bias, error, features, learning_rate):
    updated_weights = weights + learning_rate * error * features
    updated_bias = bias + learning_rate * error
    return updated_weights, updated_bias

def perceptron_train(X_train, y_train, learning_rate=0.1, epochs=100):
    num_samples, num_features = X_train.shape
    weights = np.zeros(num_features)
    bias = 0

    for _ in range(epochs):
        errors = 0
        for i in range(num_samples):
            y = np.dot(X_train[i], weights) + bias
            prediction = step_function(y)
            error = y_train[i] - prediction
            errors += abs(error)
            if error != 0:
                weights, bias = update_weights(weights, bias, error, X_train[i], learning_rate)
        mean_error = errors / num_samples
        print("Epoch: {}, Mean Error: {:.2f}".format(_, mean_error))

    return weights, bias
```

```
[13] K = 5

    # Evaluate the K-NN model on the test set
    knn_predictions = [knn_predict(X_train, y_train, test_data, K) for test_data in X_test]
    knn_accuracy = calculate_accuracy(np.array(knn_predictions), y_test)

    # Print the accuracy
    print("Accuracy of the K-NN model on the test set: {:.2f}%".format(knn_accuracy))

    Accuracy of the K-NN model on the test set: 92.28%
```

```
KNN

[15] from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import accuracy_score
     from sklearn.model_selection import train_test_split

     knn_classifier = KNeighborsClassifier(n_neighbors=3)
     knn_classifier.fit(X_train, y_train)
     y_pred = knn_classifier.predict(X_test)

     # Paso 5: Evaluar la precisión del modelo
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy}")

     Accuracy: 0.9186991869918699
```

```python
def step_function(y):
    return 1 if y >= 0 else 0

def perceptron_train(X_train, y_train, learning_rate=0.1, epochs=100):
    num_samples, num_features = X_train.shape
    weights = np.zeros(num_features)
    bias = 0
    for _ in range(epochs):
        errors = 0
        for i in range(num_samples):
            y = np.dot(X_train[i], weights) + bias
            prediction = step_function(y)
            error = y_train.iloc[i] - prediction
            errors += abs(error)
            if error != 0:
                weights += learning_rate * error * X_train[i]
                bias += learning_rate * error
        mean_error = errors / num_samples
        print("Epoch: {}, Mean Error: {:.2f}".format(_, mean_error))
    return weights, bias

def perceptron_test(X_test, weights, bias):
    predictions = []
    for i in range(len(X_test)):
        y = np.dot(X_test[i], weights) + bias
        prediction = step_function(y)
        predictions.append(prediction)
    return np.array(predictions)
```

```
Epoch: 82, Mean Error: 0.02
Epoch: 83, Mean Error: 0.02
Epoch: 84, Mean Error: 0.02
Epoch: 85, Mean Error: 0.02
Epoch: 86, Mean Error: 0.02
Epoch: 87, Mean Error: 0.01
Epoch: 88, Mean Error: 0.02
Epoch: 89, Mean Error: 0.02
Epoch: 90, Mean Error: 0.02
Epoch: 91, Mean Error: 0.02
Epoch: 92, Mean Error: 0.02
Epoch: 93, Mean Error: 0.01
Epoch: 94, Mean Error: 0.01
Epoch: 95, Mean Error: 0.02
Epoch: 96, Mean Error: 0.02
Epoch: 97, Mean Error: 0.02
Epoch: 98, Mean Error: 0.02
Epoch: 99, Mean Error: 0.02
Accuracy of the Perceptron model on the test set: 96.75%
```

Perceptron

```python
[17] from sklearn.linear_model import Perceptron
     from sklearn.metrics import accuracy_score
     from sklearn.preprocessing import StandardScaler
     from sklearn.model_selection import train_test_split

     perceptron_classifier = Perceptron(max_iter=1000, random_state=42)
     perceptron_classifier.fit(X_train, y_train)

     # Realiza predicciones en el conjunto de prueba
     y_pred = perceptron_classifier.predict(X_test)

     # Evalúa la precisión del modelo
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.9573170731707317
```