Multi-Layer Perceptron (MLP)

Lorenzo Tucceri Cimini

Introduction

This document outlines the process of creating a Python script for training multiple Multi-Layer Perceptron (MLP) models on a given dataset. The goal is to:

1. Define N different MLPs, each with N inputs and 1 output. Each MLP will be designed to predict one of the N symbolic variables based on the other variables as input.
2. Train each MLP on a specific dataset DS_i, which is derived from the original dataset DS. Each DS_i is used for learning the output variable X_i given all other variables as input.
3. Combine the trained MLPs by matching their inputs and joining their outputs. This results in an architecture with N inputs and N outputs, as depicted in the diagram.
4. Report on experimentation conducted to find optimal architecture hyper-parameters on the Letter Recognition Dataset and another dataset of your choice, including tests of the architecture's effectiveness in addressing the missing value problem.

Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP) is a type of feedforward artificial neural network that consists of multiple layers of neurons. Each neuron performs a weighted sum of its inputs, passes the result through an activation function, and propagates the output to the next layer. The key features of an MLP include:

- **Input Layer:** Receives input features from the dataset.
- **Hidden Layers:** Intermediate layers where computations are performed. More hidden layers allow the model to learn complex patterns.
- **Output Layer:** Produces the final output, which can be used for classification or regression.
- **Training:** The MLP is trained using a process called backpropagation, which involves adjusting the weights to minimize the error between the predicted and actual outputs.

MLPs are widely used for various tasks, including classification and regression, due to their ability to approximate complex functions and learn non-linear relationships.

Dataset overview

This section provides an overview of the datasets used in this analysis: the Letter Recognition dataset and another dataset of your choice.

Letter Recognition Dataset

The Letter Recognition dataset consists of 20,000 samples describing various features of uppercase English letters. Each feature is a measurement that contributes to the classification of the letter.

Descriptive Statistics

The following table summarizes key descriptive statistics for the numerical features in the dataset:

Table 1: Descriptive Statistics of Letter Recognition Dataset Features

Feature	Count	Mean	Std Dev	Min	Max
x-box	20000	4.02	1.91	0	15
y-box	20000	7.04	3.30	0	15
width	20000	5.12	2.01	0	15
high	20000	5.37	2.26	0	15
onpix	20000	3.51	2.19	0	15
x-bar	20000	6.90	2.03	0	15
y-bar	20000	7.50	2.33	0	15
x2bar	20000	4.63	2.70	0	15
y2bar	20000	5.18	2.38	0	15
xybar	20000	8.28	2.49	0	15
x2ybr	20000	6.45	2.63	0	15
xy2br	20000	7.93	2.08	0	15
x-ege	20000	3.05	2.33	0	15
xegvy	20000	8.34	1.55	0	15
y-ege	20000	3.69	2.57	0	15
yegvx	20000	7.80	1.62	0	15

The statistics reveal several key insights:

- The features are uniformly distributed, with minimum values of 0 and maximum values reaching up to 15.
- The mean values suggest that most features have average values clustered around 4 to 8:
  - This indicates a moderate range of values across the features.
- The standard deviation values indicate variability, with ‘y-box‘, ‘y-bar‘, and ‘xegvy‘ exhibiting higher variability relative to other features:
  - This spread of values may indicate different scales in feature measurements, which could impact the performance of machine learning algorithms.
  - Therefore, normalization or standardization may be necessary during preprocessing.

Correlation Analysis

To understand the relationships between the features, we can examine the correlation matrix. The correlation matrix illustrates how strongly the features are related to one another.

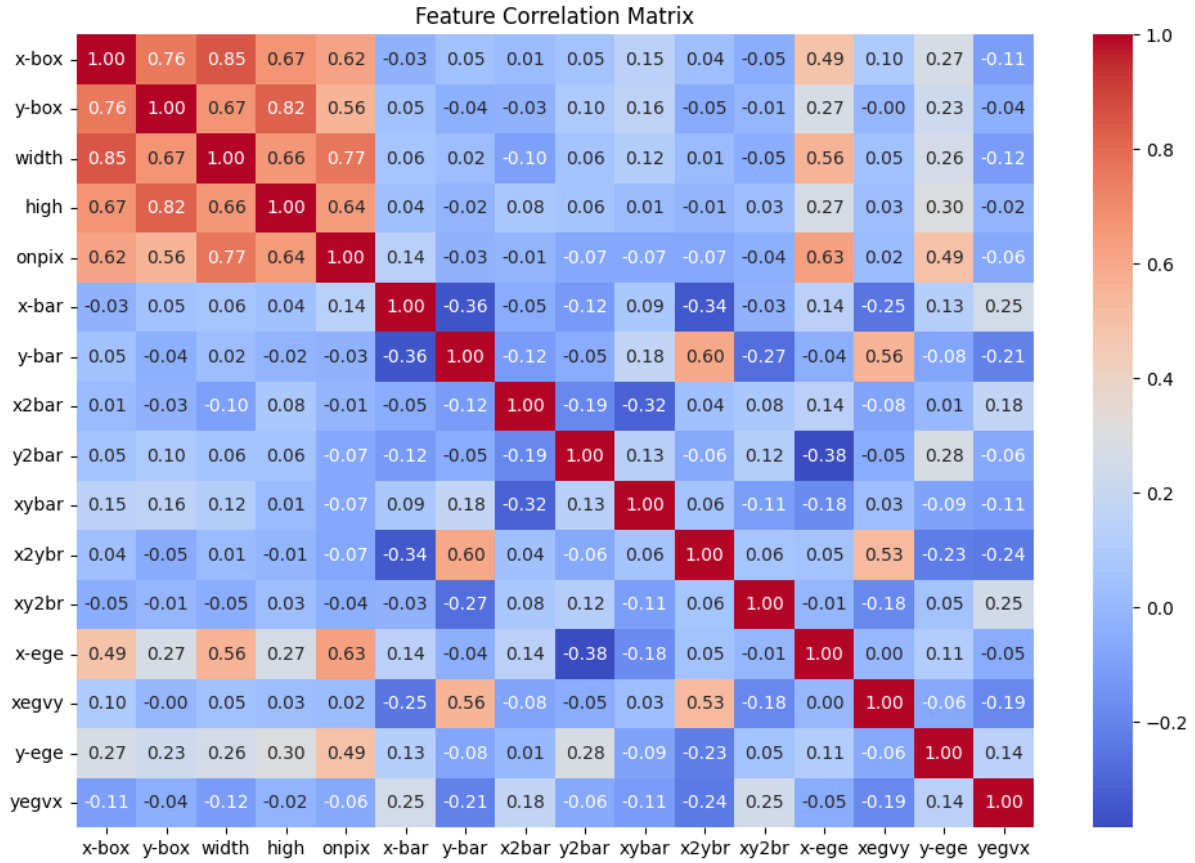

Figure 1: Correlation Matrix of Letter Recognition Dataset Features

From the correlation matrix, we can draw the following conclusions:

- Several strong correlations exist among certain features. For example:
  - 'x-box' and 'y-box' show a positive correlation with 'x-bar' and 'y-bar', suggesting that these features might share underlying factors contributing to their values.
- Features such as 'x-ege' and 'y-ege' display relatively weaker correlations with the target variable:
  - This implies that these features may contribute less to the classification task compared to others.
- High correlation between certain features can lead to multicollinearity issues in linear models:
  - Multicollinearity can affect model interpretability and performance.

Target Class Distribution

The target variable is categorical, representing one of the uppercase letters from A to Z. The distribution of the target classes is critical for understanding class balance in the dataset.

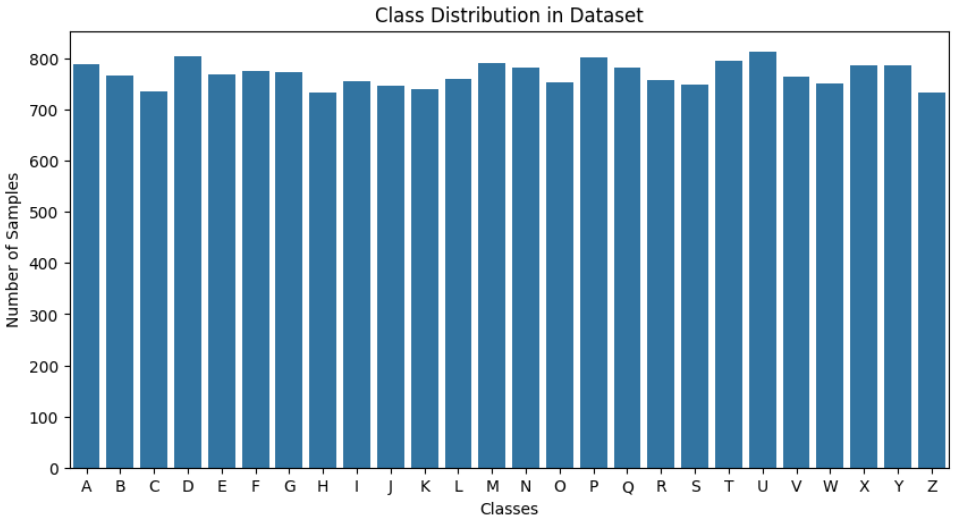

Figure 2: Distribution of Target Classes in Letter Recognition Dataset

- The distribution of target classes is relatively balanced across most categories.
- Counts for the target classes range from 734 to 813.
- Some classes, specifically C, H, and K, have slightly fewer instances.
- The underrepresentation of these classes may pose challenges for the learning algorithm.
- Overall, while the dataset is mostly balanced, certain classes might be underrepresented.
- Techniques such as oversampling for underrepresented classes or utilizing class weights during training may be beneficial to:
  - Ensure that the model does not become biased towards more frequent classes.
  - Improve the overall performance of the model on less frequent classes.

Diabetes Prediction Dataset

The Diabetes Prediction dataset consists of 100,000 samples with various features related to diabetes risk factors. Each feature provides insights into an individual's health and lifestyle choices that may contribute to the prediction of diabetes.

Descriptive Statistics

The following table summarizes key descriptive statistics for the numerical features in the dataset:

Table 2: Descriptive Statistics of Diabetes Prediction Dataset Features

Feature	Count	Mean	Std Dev	Min	Max
age	100000	41.89	22.52	0.08	80.00
hypertension	100000	0.07	0.26	0	1
heart_disease	100000	0.04	0.19	0	1
bmi	100000	27.32	6.64	10.01	95.69
HbA1c_level	100000	5.53	1.07	3.50	9.00
blood_glucose_level	100000	138.06	40.71	80	300
diabetes	100000	0.09	0.29	0	1

The statistics reveal several key insights:

- The age distribution is broad, with the mean age around 41.89 years, indicating a diverse age range in the dataset.

- The body mass index (BMI) shows variability, with an average of 27.32, suggesting that individuals vary significantly in weight categories.
- The HbA1c levels indicate average control of blood glucose, with a mean of 5.53. A level above 6.5 is typically indicative of diabetes.
- The binary features, 'hypertension', 'heart_disease', and 'diabetes', reflect the presence (1) or absence (0) of these conditions, with relatively low averages indicating that most individuals do not have these conditions.

Correlation Analysis

To understand the relationships between the features, we can examine the correlation matrix. The correlation matrix illustrates how strongly the features are related to one another.

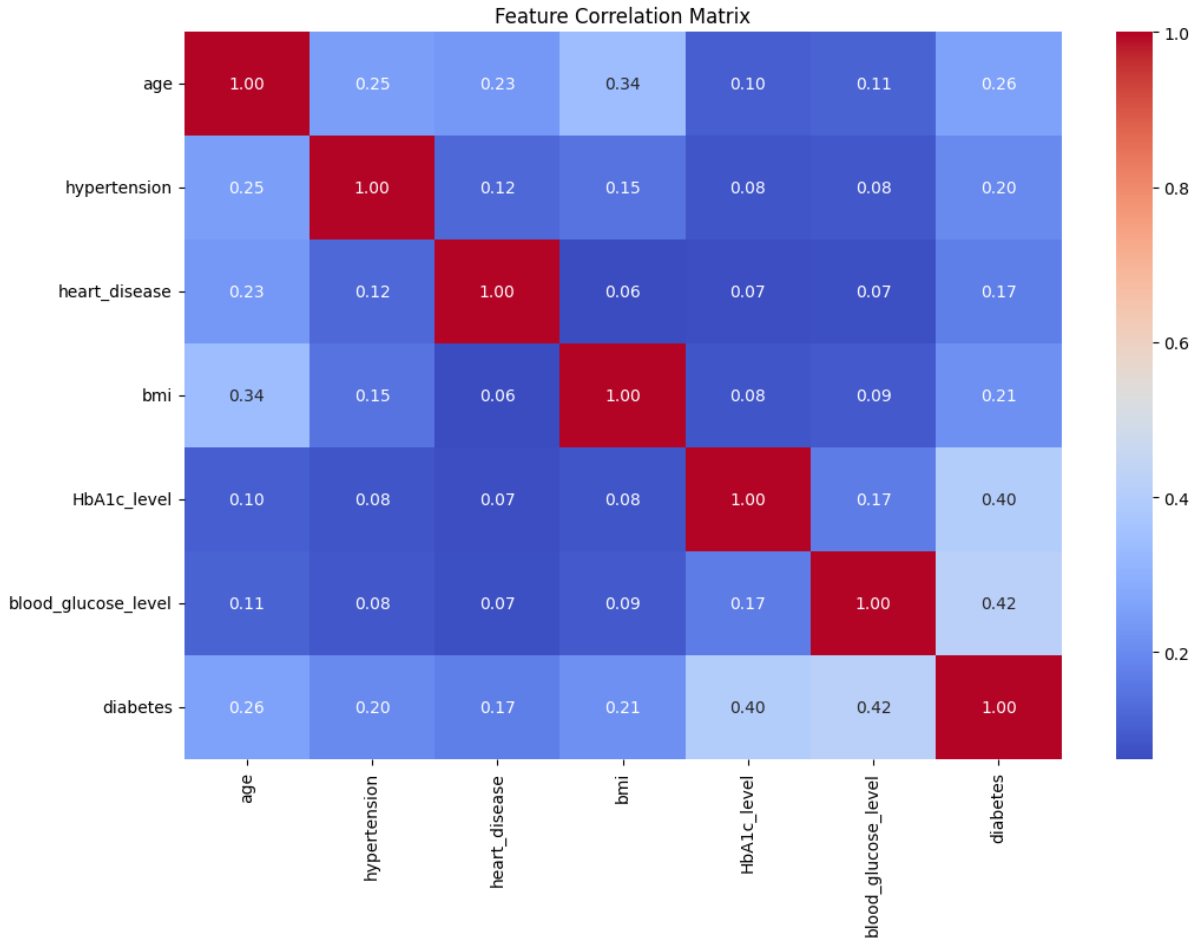

Figure 3: Correlation Matrix of Diabetes Prediction Dataset Features

From the correlation matrix, we can draw the following conclusions:

- Features such as 'age', 'bmi', and 'HbA1c_level' show moderate positive correlations with the 'diabetes' target variable.
- The 'hypertension' and 'heart_disease' features also show weak positive correlations with the diabetes diagnosis, suggesting they may contribute to diabetes risk.
- The correlation between 'blood_glucose_level' and 'HbA1c_level' is strong, indicating that glucose levels are a significant factor in diabetes management.
- High correlations among features like 'hypertension' and 'heart_disease' may indicate underlying health trends that could affect analysis and modeling.

Target Class Distribution

The target variable, 'diabetes', is binary, representing the presence (1) or absence (0) of diabetes. The distribution of the target classes is crucial for understanding class balance in the dataset.

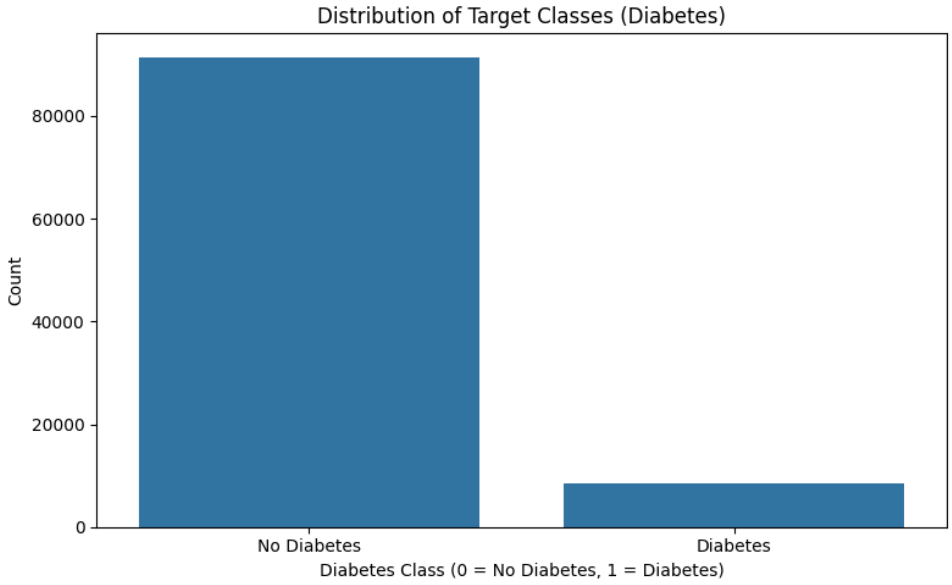

Figure 4: Distribution of Target Classes in Diabetes Prediction Dataset

- The distribution of the target classes is slightly imbalanced, with approximately 91.5% of samples indicating no diabetes and 8.5% indicating diabetes.
- The underrepresentation of the diabetic class may pose challenges for machine learning algorithms, potentially leading to biased predictions.
- Techniques such as oversampling the minority class or utilizing class weights during model training may help mitigate these issues and improve overall model performance.

Implementation

In this section, we describe the process of implementing the Python script for training Multi-Layer Perceptron (MLP) models. We will focus on key steps, dataset analysis, and model training, providing useful details for each phase.

1. Importing Required Libraries

To start, it is essential to import the necessary libraries for building and training the MLPs. The main libraries used include:

- `numpy` and `pandas` for data manipulation and analysis.
- `scikit-learn` for machine learning functionalities, particularly for creating MLP models and searching for optimal parameters.
- `matplotlib` and `seaborn` for data visualization and model performance assessment.
- `tqdm` for creating progress bars during training.

Below is an example of how the libraries are imported:

```
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputClassifier
```

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import warnings
warnings.filterwarnings('ignore')
```

2. Dataset Analysis

A crucial phase in preparing a machine learning model is exploratory data analysis (EDA). We have created a class called `DatasetAnalyzer` that is responsible for loading and analyzing the dataset. This class allows us to obtain detailed information about the dataset, such as:

- The dimensions of the dataset (number of samples and features).
- Descriptive statistics to better understand the data distribution.
- Identification of any missing values, which are essential for data cleaning.
- Visualization of feature distributions through histograms and correlation matrices.

This preliminary analysis provides a clear overview of the dataset, allowing us to identify potential issues and opportunities for model improvement.

3. Creation of the MLPModelTrainer Class

The `MLPModelTrainer` class is the core of the training process. It is designed to manage the entire workflow of training and evaluating MLP models. It includes several key attributes and methods:

Attributes

- `dataset`: Contains the loaded and pre-processed dataset.
- `mlp_models`: Stores the trained MLP models for each target column.
- `train_scores` and `test_scores`: Lists that contain the training and test scores for each model.
- `label_encoder`: Used for encoding categorical features.

Main Methods

- `fit_models()`: Trains an MLP model for each column in the dataset. During this process, the target is temporarily replaced with null values to avoid data contamination.
- `tuning_params(X, Y)`: Optimizes the model parameters using Grid Search, which explores different combinations of network architectures and regularization parameters to maximize performance.
- `evaluate_models()`: Assesses the performance of each trained model. If an optimized model provides a better score, the model is updated, and the change is recorded.
- `plot_scores()`: Visualizes the training and test scores of all models, providing a clear overview of their performance.
- `load_models(filepath, n_models)`: Loads previously saved models, allowing the reuse of results from past training sessions without the need to rerun them.

Usage Example

After defining the necessary classes and methods, using the `MLPModelTrainer` is intuitive. You can create an instance of the class by passing the dataset, then train the models, evaluate them, and visualize their performance through simple method calls. Here's an example workflow:

```
trainer = MLPModelTrainer(dataset)  # Create an instance
trainer.fit_models()                 # Train the models
trainer.evaluate_models()             # Evaluate the models
trainer.plot_scores()                 # Visualize the scores
```

This implementation provides a modular and clear structure for managing the machine learning process, facilitating the analysis and handling of MLP models.

4. Introduction of Null Values

The function `introduce_nulls(dfx, fraction=0.0275)` is designed to randomly introduce null values (NaN) into a given DataFrame. This is useful for testing the robustness of machine learning models against missing data. The function works as follows:

- `dfx (pd.DataFrame)`: The input DataFrame where nulls will be introduced.
- `fraction (float)`: The fraction of elements to be set as nulls, with a default value of 0.0275 (2.75%).

```
def introduce_nulls(dfx, fraction=0.0275):
    total_elements = dfx.size
    num_nulls = int(total_elements * fraction)
    null_rows = np.random.randint(0, dfx.shape[0], size=num_nulls)
    null_columns = np.random.randint(0, dfx.shape[1], size=num_nulls)

    for z1 in null_rows:
        z2 = random.choice(null_columns)
        dfx.iloc[z1, z2] = np.nan
```

This function calculates the total number of nulls to introduce based on the specified fraction and then selects random rows and columns to set as NaN.

5. ClassifierEnsembler Class

The `ClassifierEnsembler` class is responsible for managing multiple classifiers and making predictions based on their combined outputs. Below are the details of the class structure and methods.

Attributes

- `classifiers`: A list containing the classifiers that will be ensembled.
- `encoder`: An instance of `LabelEncoder` for transforming categorical labels.
- `results`: A dictionary to store the results of correct and incorrect predictions for each class label.

Main Methods

- `__init__(classifiers, dataset, outcome)`: Initializes the ensemble with the provided classifiers and dataset. It also encodes the outcome labels.
- `joined_arch(query)`: Makes predictions using the ensemble of classifiers for a given query.
- `predict(query)`: Aggregates predictions from all classifiers and returns the final predictions based on their average output.
- `prediction(Ds, testSet)`: Evaluates the performance of the ensemble classifiers against a test set and calculates the accuracy.

6. Cross-Validation Function

The `perform_cross_validation(classifier, X, y, cv=5)` function is used to evaluate the performance of a given classifier using cross-validation.

- `classifier`: The machine learning classifier to evaluate.
- `X (pd.DataFrame)`: The feature data used for training.
- `y (pd.Series or pd.DataFrame)`: The target labels corresponding to the features.
- `cv (int)`: The number of cross-validation folds (default is 5).

The function returns a list of accuracy scores obtained from the cross-validation process:

```
def perform_cross_validation(classifier, X, y, cv=5):
    scores = cross_val_score(classifier, X, y, cv=cv, scoring='accuracy')
    return scores
```

This function provides a straightforward way to assess the classifier's performance across multiple folds, helping to ensure the model's robustness and generalizability.

7. Class Definition: MLPMultiOutput

The `MLPMultiOutput` class encapsulates the entire workflow for training and predicting using a multi-output Multi-Layer Perceptron (MLP) classifier. This class handles the data preparation, introduces missing values, and facilitates the training and evaluation of the model.

Class Attributes

- `dataset`: Stores the original dataset for training and testing.
- `target_column`: Specifies the target variable for the model.
- `results`: A dictionary that keeps track of correct and incorrect predictions for each output feature.
- `le`: An instance of `LabelEncoder` for encoding categorical variables.
- `dataset_shuffled`: A randomized version of the dataset to ensure effective model training.

Methods

1. $_{init_{(self, dataset, target\_column=None)}}$ This method initializes the class by copying the dataset and shuffling it to ensure randomness. It also sets the target column and prepares the results dictionary for tracking predictions.

2. `introduce_nulls(self, fraction=0.0275)` This function simulates missing data by randomly introducing null values into the dataset based on a specified fraction. It is crucial for testing the model's robustness against missing data.

3. `prepare_datasets(self)` This method prepares the dataset for training and testing. It handles missing values, separates the training and test datasets, and initializes the results dictionary for each output feature. This step is essential to ensure that the model is trained on complete data.

4. `transform_query(self, query)` This function transforms the input query for predictions, handling categorical features by encoding them as necessary. It prepares the data in the same format expected by the trained model.

5. `joined_arch(self, classifiers, query)` This method utilizes an ensemble of classifiers to make predictions on the input query. It integrates the predictions from multiple classifiers to provide a comprehensive output while ensuring that categorical variables are appropriately handled.

6. `prediction(self, classifiers)` This function runs predictions on the dataset, calculates the accuracy of each model's predictions, and updates the results dictionary with correct and incorrect counts. The accuracy is then returned for further evaluation.

7. `visualize_results(self)` This method generates visual representations of the prediction results, including bar charts for correct and incorrect predictions. It computes the accuracy for each model, presenting the results clearly for analysis.

8. `multioutput_classifier_training(self)` This method is responsible for training the classifiers on the prepared dataset. It employs an MLP classifier to fit the model for each output feature, facilitating multi-output predictions.

Usage and Benefits

The `MLPMultiOutput` class streamlines the process of multi-output classification. By encapsulating data handling, model training, and evaluation in a cohesive framework, it simplifies the workflow for data scientists and machine learning practitioners. The class not only introduces robustness through the simulation of missing data but also provides insightful visualizations that enhance interpretability and assessment of model performance.

8. Example Workflow for Multi-Output MLP Training and Prediction

In this section, we outline the typical workflow for utilizing the `MLPMultiOutput` class to train and evaluate a multi-output MLP model. This includes loading the dataset, preparing it, training the model, and visualizing the results.

Step-by-Step Process

1. **Load the Dataset:** The dataset is loaded into the `DatasetAnalyzer` class using the `load_dataset()` method. This method initializes the dataset and prepares it for analysis.

   ```
   analyzer.load_dataset()
   ```

2. **Retrieve Classifiers:** After training the models using the `MLPModelTrainer` class, we obtain the trained classifiers. This is done by accessing the `mlp_models` attribute of the trainer instance.

   ```
   classifiers = trainer.mlp_models
   ```

3. **Initialize the Multi-Output Model:** An instance of the `MLPMultiOutput` class is created by passing the loaded dataset and the target column (in this case, `'x-bar'`). This instance will be responsible for the multi-output predictions.

   ```
   mlp_model = MLPMultiOutput(analyzer.df, 'x-bar')
   ```

4. **Prepare the Datasets:** The `prepare_datasets()` method is called on the `mlp_model` instance to preprocess the data. This includes introducing null values, handling missing data, and creating training and test datasets.

   ```
   mlp_model.prepare_datasets()
   ```

5. **Make Predictions:** The `prediction()` method is invoked to run predictions on the dataset using the trained classifiers. The method returns the overall accuracy of the predictions made by the ensemble of classifiers.

   ```
   accuracy = mlp_model.prediction(classifiers)
   ```

6. **Visualize Results:** Finally, the results of the predictions, including accuracy and counts of correct and incorrect predictions, are visualized using the `visualize_results()` method. This helps in assessing the model's performance visually.

```
mlp_model.visualize_results()
```

Conclusion

This workflow provides a clear and structured approach to training and evaluating a multi-output MLP model using the `MLPMultiOutput` class. Each step is designed to ensure proper data handling, model training, and performance evaluation, ultimately enhancing the effectiveness of the machine learning pipeline.

Analysis of Model Performance

In this section, we extend the analysis of the model performance by examining the accuracies of individual models along with an overall prediction accuracy.

Model Accuracies

The following are the accuracies obtained for each model:

Model	Accuracy
Model (x-box)	0.824
Model (y-box)	0.764
Model (width)	0.857
Model (high)	0.872
Model (onpix)	0.808
Model (x-bar)	0.772
Model (y-bar)	0.887
Model (x2bar)	0.961
Model (y2bar)	0.869
Model (xybar)	0.933
Model (x2ybr)	0.860
Model (xy2br)	0.756
Model (x-ege)	0.832
Model (xegvy)	0.976
Model (y-ege)	0.988
Model (yegvx)	0.763

Table 3: Accuracies of Various Models

The overall prediction accuracy across all models is:

Overall Prediction Accuracy: 0.858

Performance Insights

1. Top-Performing Models: - The model (y-ege) achieved the highest accuracy of 0.988, closely followed by model (xegvy) with an accuracy of 0.976. These models indicate exceptional performance and potential for use in applications requiring high reliability.

2. Models Requiring Attention: - On the lower end, model (xy2br) scored 0.756, suggesting a need for further optimization or reevaluation.

3. Overall Accuracy: - The overall prediction accuracy of 0.858 demonstrates a generally strong performance across models, although there remains room for improvement, especially for the underperforming models.

Visual Representation

To better understand the distribution of correct and incorrect predictions across the models, we can include visualizations. Below are two placeholder sections for images that represent correct and incorrect predictions.

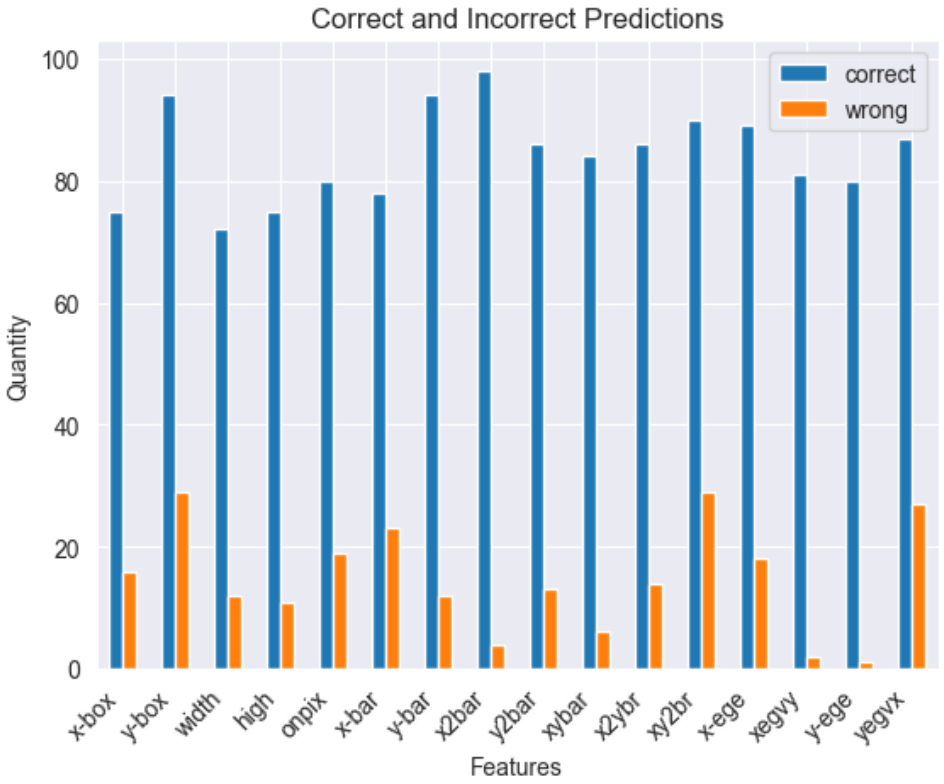

Figure 5: Visual Representation of Correct and Wrong Predictions

Conclusion

Conclusion

In conclusion, the performance of the MLP models indicates a promising overall prediction accuracy of 0.858. However, there are noticeable differences in effectiveness among the individual models. Models such as (y-ege) and (x2bar) stand out for their high performance, while models like (xy2br) may need more thorough evaluation. Implementing visual tools to depict predictions will aid in comprehending the variation in model performance, highlighting the specific areas that require enhancement.

Moreover, the flexibility and adaptability of these MLP models suggest that they can be effectively utilized on other datasets, such as diabetes data. This potential for broader application underscores the importance of ongoing oversight and adjustment to achieve the best possible results and maintain generalization across previously unseen datasets.

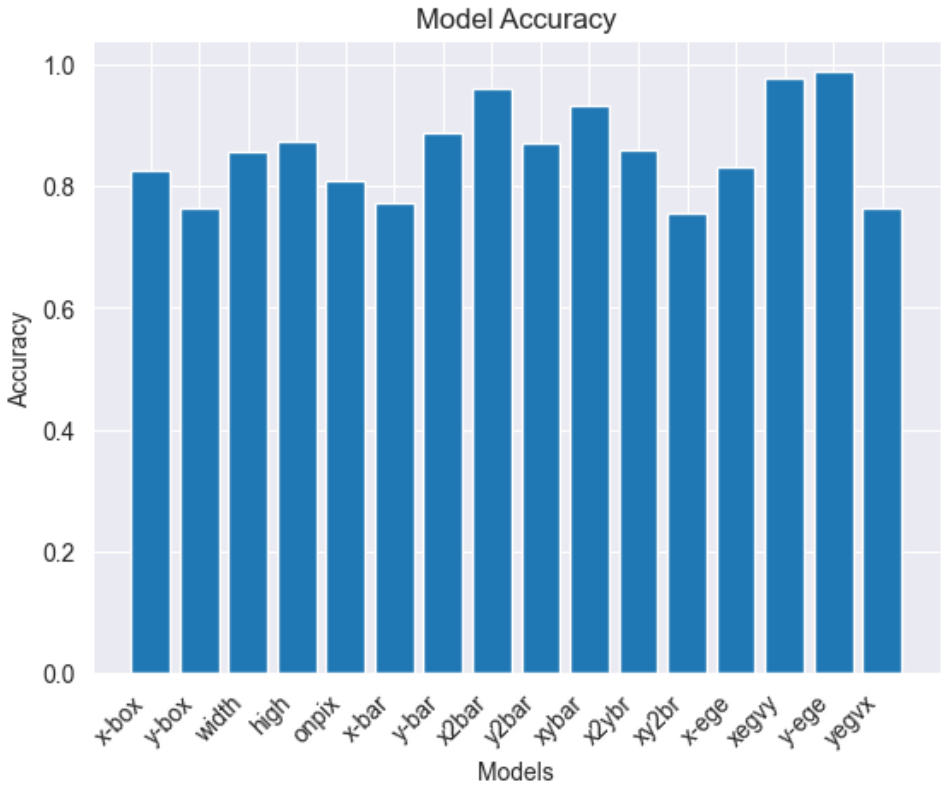

Figure 6: Visual Representation of Model accuracy