Journal of Machine Learning Research 0 (2023) 0-0 Submitted 0/00; Published 0/00

Evaluation of Naïve Null Model Predictor on Diverse Datasets

Tim Chen TCHEN124@JHU.EDU
Department of Computer Science
Johns Hopkins University
Baltimore, MA 21218, USA

Editor: None

Abstract

In this study, a machine learning pipeline was developed and applied to six diverse datasets from the UCI Machine Learning Repository. The pipeline included essential preprocessing steps, data splitting for train and test sets, cross-validation methods, and evaluation metrics. The emphasis was placed on implementing a naïve Null Model predictor for both classification and regression tasks. The performance of the Null Model was subsequently assessed across the datasets to ascertain its predictive ability. Results from the experiments demonstrate the inherent limitations and capabilities of such a simplistic model.

Keywords: Machine Learning, Naïve Null Model, Data Preprocessing, Cross-validation, UCI Machine Learning Repository

1. Introduction

In the age of data, machine learning has emerged as a dominant tool to decipher patterns, make predictions, and drive decisions. While sophisticated algorithms frequently gain attention, understanding the performance of elementary models, such as the Naïve Null Model predictor, remains crucial. By establishing a baseline, it sets the stage for more intricate algorithms to be compared against.

2. Data Preprocessing

Data preprocessing is an integral part of any machine learning pipeline, as it prepares the raw data to be suitable for the subsequent modeling process. In this work, a flexible and modular `DataPipeline` class was designed to handle a series of common preprocessing tasks. The following is a breakdown of the implemented preprocessing operations:

2.1 Handling Missing Values

One of the initial tasks in preprocessing is dealing with missing data. The `_fill_missing_values` method in the pipeline:

- Recognizes non-standard missing values such as "?", "x", spaces, and replaces them with NaNs.

- Attempts to convert non-numeric columns of type 'object' to a numeric format, which can be essential for certain algorithms that only operate on numerical data.
- Fills missing values in numeric columns with their mean. For non-numeric columns, the mode (most frequent value) is used to fill the gaps.

2.2 Ordinal Data Encoding

Ordinal data are categorical values that have a meaningful order. The method `_handle_ordinal_data` is designed to replace ordinal categories with specified numeric values, preserving their order.

2.3 Nominal Data Encoding

For nominal data, which consists of categories without inherent order, one-hot encoding is a popular technique. The method `_one_hot_encode` performs this transformation, expanding the dataset's feature space.

2.4 Data Discretization

Some algorithms benefit from having continuous attributes converted into discrete ones. Two discretization methods are implemented:

- `_discretize_equal_width`: This divides the range of a continuous feature into equal-width bins.
- `_discretize_equal_frequency`: This divides the feature into bins with approximately equal number of instances.

2.5 Standardization

Feature scaling is pivotal for algorithms that are sensitive to the magnitude of features, such as gradient descent-based methods. The `_standardize` method scales a numeric column to have a mean of zero and a standard deviation of one.

2.6 Function Transformation

A flexible transformation allows users to apply any callable function to a column. The method `_function_fransformer` facilitates this operation, proving useful for power transformations, logarithmic scaling, and more.

2.7 Executing the Pipeline

The `run` method drives the entire preprocessing operation. It iterates over the specified steps and applies the respective transformations in the given order, returning a cleaned and processed DataFrame ready for modeling.

The beauty of this pipeline lies in its modularity and scalability. New preprocessing operations can be easily added and integrated into the flow, ensuring that the pipeline remains adaptable to the ever-evolving demands of data preprocessing in machine learning.

3. Experimental Approach

This section describes the approach followed to validate and test the data processing pipeline.

3.1 Cross-Validation

To evaluate the performance of the model on unseen data, the dataset was split into training and test sets. To ensure the model was not overfitting to a particular training set and that the results were consistent across different train-test splits, the k-fold cross-validation technique was employed. In addition to traditional k-fold validation, the pipeline was also equipped with k×2 cross-validation, enhancing the robustness of the results. Tests primarily focused on 5×2 cross-validation, which provides a balance between computational efficiency and validation reliability.

3.2 Stratified Sampling

In datasets where the target classes might be imbalanced, stratified sampling ensures that each fold in cross-validation maintains the original class distribution. This approach guarantees that no fold is devoid of minority classes. The implemented `train_test_split` and `k_fold_split` functions in the `cross_validation.py` file support stratified sampling, which is especially useful for datasets with imbalanced class labels.

3.3 Standardization

Data standardization is crucial for machine learning models sensitive to feature scale. Standardizing features to have zero mean and unit variance makes sure that all features contribute equally to model training. Although the provided code has the capability for standardization in the preprocessing pipeline, in the main experiment with the breast cancer dataset, this step was not explicitly invoked.

3.4 Naïve Null Model

To establish a baseline performance, a Naïve Null Model was implemented. This model, defined in the `models.py` file, predicts the most common class in the case of classification or the mean in the case of regression. By comparing more sophisticated models' performance against this baseline, the real value added by such models can be ascertained.

3.5 Multiple Dataset Evaluation

The `main.py` script details the execution of the pipeline on the UCI datasets. The data undergoes preprocessing, where missing values are filled and other processes metioned in the data processing section. The dataset is then split into training and test sets in an 80-20 ratio. The performance of the Naïve Null Model is evaluated using 5-fold stratified cross-validation on the training set. Finally, the model's accuracy is determined on the test set.

3.6 Scalability

The modular architecture of the code means it is highly scalable. The `main.py` script contains placeholder functions like `exec_data2`, `exec_data3`, and so on. This structure suggests the pipeline's readiness to be tested on multiple datasets, ensuring its versatility and wide applicability.

In conclusion, the experimental approach is comprehensive, combining robust validation techniques with baseline model evaluations to assess the efficacy of the preprocessing pipeline and establish its reliability and scalability.

4. Results

Experiments were conducted on six datasets. The summarized performance metrics for each dataset, both on cross-validation and the test set, are provided in Table 4.

Dataset	**Task Type**	**Avg Perf. (5-fold CV)**	**Test Set Perf.**
Breast Cancer	Classification	ACC: 65.77%	ACC: 65.47%
Car Evaluation	Classification	ACC: 70.18%	ACC: 70.35%
Congressional Vote	Classification	ACC: 61.76%	ACC: 61.63%
Abalone	Regression	MSE: 10.435	MSE: 10.255
Computer Hardware	Regression	MSE: 24890	MSE: 31707
Forest Fires	Regression	MSE: 2.0021	MSE: 1.8079

Table 1: Performance of the Naïve Null Model on various datasets.

The table showcases the performance metrics obtained using the Naïve Null Model on various datasets. Classification tasks are evaluated based on accuracy, while regression tasks utilize the Mean Squared Error (MSE) metric. The results serve as a preliminary baseline, emphasizing the need for more complex models for certain datasets, especially in regression tasks like the Computer Hardware dataset where the MSE was particularly high.

5. Discussion

5.1 Performance Analysis

The Naïve Null Model displayed a varied performance across different datasets. As observed, classification tasks like Breast Cancer, Car Evaluation, and Congressional Vote achieved accuracies of 65.47%, 70.35%, and 61.63% respectively on the test sets. These values are indicative of the basic nature of the Naïve Null Model, which uses the most common class or mean value as its prediction.

On the regression front, the model faced challenges, especially with the Computer Hardware dataset, where the Mean Squared Error (MSE) on the test set reached a significant 31707.2717. This value indicates the model's limitations in capturing the intricacies of the underlying data distributions.

5.2 Baseline Importance

While the model's performance metrics may seem underwhelming compared to more sophisticated machine learning models, its primary purpose is to serve as a baseline. It provides a foundational benchmark upon which more complex models can be compared, highlighting their added value and sophistication.

6. Conclusion

In the world of machine learning, setting benchmarks is a critical step. Elementary models like the Naïve Null Model, despite their simplistic nature, play a pivotal role. They not only demonstrate the intricacies and nuances of the data but also set the stage for the performance assessment of more advanced algorithms. By understanding how the simplest models behave, researchers and developers can better appreciate the intricacies and capabilities of more sophisticated machine learning models.

Acknowledgments

Acknowledgments We extend our heartfelt gratitude to the developers of Scikit-learnPedregosa et al. (2011). The interface and structure of Scikit-learn have been instrumental in guiding our approach and methodology. The coherence, robustness, and versatility of its design have served as a remarkable benchmark and have provided consistent inspiration throughout our work. The extensive documentation and user-friendly design of Scikit-learn made our endeavors smoother, and for this, we remain indebted.

References

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*, 12:2825–2830, 2011.

Appendix A: Code Structure and Execution

The project's file structure is organized into main functional scripts, dataset results, source code, and tests, ensuring a streamlined process for data handling, model implementation, and evaluation. The primary execution of the model across the datasets is managed by the `main.py` script, which illustrates a modular and scalable code design.

File Structure

```
.
|-- README.md
|-- demo.ipynb
|-- main.py
|-- project1.pdf
|-- requirements.txt
|-- results
|   |-- main_log.txt
|-- setup.py
|-- src
|   |-- __init__.py
|   |-- cross_validation.py
|   |-- evaluation_metrics.py
|   |-- loader.py
|   |-- models.py
|   `-- preprocessing.py
`-- test
    |-- __init__.py
    |-- test_cross_validation.py
    |-- test_evaluation_metrics.py
    |-- test_loader.py
    |-- test_model.py
    `-- test_preprocessing.py
```

Setup and Execution Instructions

For a detailed setup and execution process, refer to the `README.md` file in the root directory. The key steps involve:

1. Setting up the Python environment, with the recommended usage of a virtual environment to prevent potential conflicts with other projects.
2. Installing necessary dependencies using `requirements.txt`.
3. Running the main script (`main.py`) to execute the model on various datasets.
4. Executing unit tests.

The actual content of the `README.md` file and other details have been omitted in this sample for brevity. Refer to the file directly for a comprehensive guide.