Implement Machine Learning Pipeline From Scratch

Noboru Hayashi NHAYASH3@JHU.EDU

1. Introduction

This paper describes the process to build a Machine Learning Pipeline from scratch using Python with the functionalities of data preprocessing: missing value handling, categorical feature encoding, discretization & standardization. It also contains the implementation of K-fold function, evaluation metrics and a simple learning algorithm (Majority Predictor) to experiment with the entire machine learning pipeline.

With this implementation of the machine learning toolkit, in this paper, the demonstration of the pipeline will be shown using the actual datasets. And in the future, this toolkit can be reused for more complex tasks.

2. Implementation

2.1 Data Loading

Since the data files are available as a text file and each feature value of the samples are separated by commas, python built-in package csv is used to load the data. And a 2-D list is chosen as the data structure given the 6 datasets are tabular data.

Also additional functionalities such as custom column headers and log transformation are implemented by taking extra arguments to the function

2.2 Data Preprocessing

The python file of data preprocessing (data_handler.py) contains multiple features as follows. Machine learning pipelines can utilize these functionalities before loading the data to the learning algorithms.

2.2.1 Handle Missing Values

Handle_missing() function is also implemented to fill the missing value stored in the dataset, such as '?' label stored in 1984 United States Congressional Voting Records Database.

Given the argument of the label indicating the missing values, the function scans through the entire column list and detects the missing data, then calculates the average values with available data. After the missing fields are filled with the mean value. As a limitation, currently the function only supports numeric fields imputation.

2.2.2 Handle Categorical Values

When handling categorical features, some logic to encode the non-numerical category is required for ML. So handle_categorical_ordinal() and handle_categorical_nominal() are implemented and available for each type of categorical feature.

Each function takes the list of categories with the length of N. handle_categorical_ordinal() will map this category list to a numerical list, 0, 1, … , N, while handle_categorical_nominal() will generate N additional columns with 0 or 1 to indicate the categorical value.

2.2.3 Discretization

Given the numerical feature, equal-width & equal-frequency discretizations are implemented so that we can transform the data into a series of discretized values. Depends on the type of discretization, the function will decide the width (the range of the feature values for each bin) or the size of the bin (the number of element stored in each bin), then use the key-sorted dataset to update the column's value with the bin index.

2.2.4 Standardization

Standardization will convert the feature values to be on the Z-score scale, and in some cases this will benefit machine learning algorithms. The implemented standardization() function reads the training data and calculates the training mean and standard deviation, then apply z-score calculation: $z(x) = \frac{x-\mu}{\sigma}$ to training and test datasets.

2.3 K-Fold

K-Fold sampling and cross validation will help the machine learning performance by maximizing the amount of data for training. The implemented function partitions the data into k folds by random sampling. It also has a feature of generating a validation set that samples 20% of the entire dataset for validation then begins to sample for K folds.

Since the data for each fold are sampled randomly, the class label for classification tasks is approximately equally distributed but not strictly.

2.4 Evaluation Metrics

Once the predicted data and true data are prepared with the machine learning pipeline, it's crucial to have the function of calculating evaluation metrics such as accuracy for classification and mean squared error for regression. So that we can run model selection analysis or hyperparameter tuning.

In the implemented toolkit, evaluation_metric() calculations the following metrics for classification and regression tasks:

- For classification:
    - Accuracy
    - Precision
    - Recall
    - F1 Score
- For regression
    - Mean Squared Error
    - Mean Absolute Error
    - R2 coefficient
    - Pearson's correlation

2.5 Learning Algorithm

In this toolkit, the very naive learning algorithm is built for testing, which is the Majority predictor. This algorithm will learn from the training dataset and return the most common case label in classification task, and the average value in regression tasks.

The function very_naive_algo() takes train data, output data index and mode (classification or regression) as arguments, and scans through the data to get the most common value or calculate the mean value.

3. Results

As a demonstration, I created the test file (test.py) to test the features of data loading, data handling, sampling, learning and evaluation. In the test file, the 'Car Evaluation Database' data is used for testing. The dataset contains the data with all categorical features: buying, maint, doors, persons, lug_boot, safety, and class. With this dataset, handling categorical features, K-fold, majority predictor and accuracy calculation are used by implemented functions.

For demonstrating the encodings of categorical features, I chose ‘buying’ feature for ordinal encoding, and ‘lug_boot’ for nominal one-hot encoding. As we can see from the partial output of the script below, the two categorical features are successfully encoded.

pj1 % python test.py

The head of unprocessed data:

[['vhigh', 'vhigh', 2.0, 2.0, 'small', 'low', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'small', 'med', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'small', 'high', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'med', 'low', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'med', 'med', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'med', 'high', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'big', 'low', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'big', 'med', 'unacc'], ['vhigh', 'vhigh', 2.0, 2.0, 'big', 'high', 'unacc'], ['vhigh', 'vhigh', 2.0, 4.0, 'small', 'low', 'unacc']]

The head of processed data:

[[0, 'vhigh', 2.0, 2.0, 1, 0, 0, 'low', 'unacc'], [0, 'vhigh', 2.0, 2.0, 1, 0, 0, 'med', 'unacc'], [0, 'vhigh', 2.0, 2.0, 1, 0, 0, 'high', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 1, 0, 'low', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 1, 0, 'med', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 1, 0, 'high', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 0, 1, 'low', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 0, 1, 'med', 'unacc'], [0, 'vhigh', 2.0, 2.0, 0, 0, 1, 'high', 'unacc'], [0, 'vhigh', 2.0, 4.0, 1, 0, 0, 'low', 'unacc']]

Regarding the evaluation metric, since in the file ‘car.names’, the class distribution is described as below:

Class Distribution (number of instances per class)

class	N	N[%]
unacc	1210	(70.023 %)
acc	384	(22.222 %)
good	69	(3.993 %)
v-good	65	(3.762 %)

We can observe from the script output that the majority predictor will return an ‘unacc’ label since 70% of the source data belong to this class. And the accuracy measures (correct label / total data size) on training and test data sets are close to 70%:

```
pj1 % python test.py

...

The majority of the train_data:  unacc

Training Accuracy:  0.702819956616052

Training Accuracy:  0.6898550724637681
```

4. Conclusion

In this paper, I described my approach to implement data preprocessing, learning algorithms and evaluation from scratch. Also the toolkit is tested with the actual data sample and we can observe that it’s functional so that I can reuse this package for the later assignments or tasks.