HPCC Systems Machine Learning Preprocessing Bundle

HPCC Systems Summer Internship 2020

Vannel Zeufack

Supervised by Xu Lili

1. Introduction

It is well known in the Machine Learning community that data preparation is the most time-consuming phase of a Machine Learning Project. To make that phase smoother on HPCC Systems platform, we developed a Preprocessing Bundle. The current version of the bundle includes modules and functions allowing to easily handle categorical features, scale your data and split it in various ways. In this blog, we describe each module and functions of the Preprocessing bundle.

2. Bundle description

The current version of Preprocessing Bundle has:

- 2 modules for handling categorical features: **LabelEncoder, OneHotEncoder**
- 2 modules and 1 function for scaling: **StandardScaler, MinMaxScaler, Normaliz**
- 2 functions for dataset splitting: **Split and StratifiedSplit**.

2.1. LabelEncoder

LabelEncoder allows to convert categorical features' values into unique numbers in the range $[0, numberOfCategories - 1]$. The LabelEncoder module includes 4 functions: **GetKey, GetMapping, Encode and Decode.** Figure 1 illustrates label encoding:

Id	Level	Color
1	high	red
2	low	blue
3	medium	yellow

Decode / Encode

Id	Level	Color
1	2	1
2	0	0
3	1	2

Figure 1 – LabelEncoding Illustration

2.1.1. GetKey(baseData, featureList)

This function computes the labelEncoder key. Encoding and decoding are done based on a key. The key stores the categories of each categorical feature in *baseData*. *baseData* is the data from which the categories of each feature in *featureList* are extracted.

featureList is used to provide the names of the categorical features. For this parameter, **a dataset of SET OF STRINGs** is expected. Each SET OF STRING in the record structure must be identified by the name of a categorical feature. When constructing the *featureList*, the programmer may provide the categories of some features for which he would like to enforce an ordering.

2.1.2. GetMapping(key)

This function allows to get the mapping between the categories of each feature in the *key* and the values assigned to them.

2.1.3. Encode (dataToEncode, key)

This function converts the categorical features' values into numbers according to the key mapping. *dataToEncode* is the dataset to encode and can be any record-oriented dataset. *Key* is the key generated through the GetKey function or manually built. If an unknown category (a category not in the key) is found, it is encoded to -1.

2.1.4. Decode (dataToDecode, key)

This function converts back the categorical features' values into labels according to the key mapping. *dataToDecode* is the dataset to decode and can be any record-oriented dataset. *Key* is the key generated through the GetKey function or manually built. An unknown category (-1) is decoded to empty string.

2.1.5. Example

```
IMPORT ML_Core.Preprocessing.LabelEncoder as Encoder;

//layout for the key
KeyLayout := RECORD
  SET OF STRING level;
  SET OF STRING color;
END;

//we impose categories for level but let the categories for color
//to be extracted from our dataset
partialKey := ROW({['low', 'medium', 'high'], []}, KeyLayout);
key := encoder.GetKey(baseData, partialKey);

//Encoding
encodedData := encoder.encode(sampleData, key);

//Decoding
decodedData := encoder.decode(encodedData, key);

//Getting the mapping between numbers and categories if need be.
mapping := encoder.GetMapping(key);
```

2.2. OneHotEncoder

OneHotEncoding allows to convert categorical features into new binary features where a 1 indicates the presence of a value and 0 indicates its absence. The OneHotEncoder module has a constructor and 3 functions: **GetKey, Encode and Decode**. OneHotEncoding is illustrated in Figure 2:

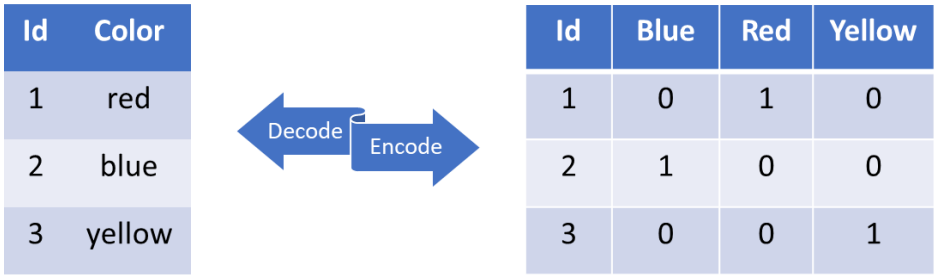

Figure 2 - OneHotEncoding Illustration

2.2.1. OneHotEncoder(baseData, featureIds, Key)

The oneHotEncoder could be initialized either by passing it *baseData* and *featureIds*, or by only passing it a precomputed *key*. *baseData* is a NumericField [1] dataset from which the categories listed in *featureIds (SET OF UNSIGNED)* are extracted.

2.2.2. GetKey()

This function allows to get the mapping between the features listed in *featureIds* and their respective categories.

2.2.3. Encode(dataToEncode)

This function allows to perform oneHotEncoding on the given data. An unknown category is encoded to all zeros.

2.2.4. Decode(dataToDecode)

This function allows to revert the encoded data to their original values. If a value has been encoded to all zeros, it will be decoded to -1.

2.2.5. Example

```
IMPORT ML_Core.Preprocessing;;

//initializing the encoder
encoder := Preprocessing.OneHotEncoder(sampleData, [1]);

//Encoding
encodedData := encoder.encode(sampleData);

//Decoding
decodedData := encoder.decode(encodedData);

//Getting the mapping between features and categories if need be.
mapping := encoder.GetKey();
```

2.3. StandardScaler & MinMaxScaler

Standard scaling allows you to scale each feature such that they have a zero mean and standard deviation equal to 1. MinMaxScaler allows to scale the data in some range (default = [0,1]).

2.3.1. Constructors

The constructors for StandardScaler and MinMaxScaler are as follows:

2.3.1.1. StandardScaler(baseData, key)

The standard scaler could be initialized either by passing it *baseData or* a precomputed *key.*

2.3.1.2. MinMaxScaler(baseData, lowBound = 0, highBound = 1, key)

The MinMaxScaler can be initialized either by passing it *baseData*, *lowBound (default = 0)* and *highBound (default = 1)* or by passing only a precomputed *key*.

2.3.2. GetKey()

This function allows to get the scaler's key. StandardScaler's key contains means and standard deviation of each feature is *baseData*. MinMaxScaler's key contains minimums and maximums of each feature in *baseData*.

2.3.3. Scale(dataToScale)

This function scales the given data and returns the scaled data.

2.3.4. Unscale(dataToUnscale)

This function reverts the scale operation.

2.3.5. Example

```
IMPORT ML_Core.Preprocessing;

//initializing the scaler
scaler := Preprocessing.MinMaxScaler(trainData);
//scaler := Preprocessing.StandardScaler(trainData);

//Scaling
scaledData := scaler.scale(sampleData);

//Unscaling
unscaledData := scaler.unsacle(scaledData);

//Getting the Key
key := scaler.GetKey();
```

2.4. Normaliz (dataToNormalize, norm)

The Normaliz function allows to scale either following the **L1 norm** (every value is divided by the sum of absolute values of elements of its row), the **L2 norm** (every value is divided by the square root of sum of squares of elements of its row) or **L-infinity norm** (every value is divided by the maximum absolute value of elements of its row).

The first parameter of the function is the data you wish to normalize and the second specifies the *norm* to use (default = "l2").

2.5. Split and Stratified Split

These functions allow to split datasets into training and test sets.

2.5.1. Split (dataToSplit, trainSize, testSize, shuffle)

To perform a simple split, you would provide either trainSize or the testSize. Those are values in the range (0, 1). The *shuffle* parameter is a Boolean variable based on which the data would be shuffled or not before splitting. Its default value is *False*.

2.5.2. StratifiedSplit (dataToSplit, trainSize, testSize, labelId, shuffle)

StratifiedSplit allows to split the data while maintaining the proportions of a given feature. For example, if your label feature has two values Yes and No with proportions 30% and 70%, stratified split will keep those proportions in both the training and test data.

Compared to the split function, StratifiedSplit has an extra parameter *labelId* which is the id of the field whose proportions must be maintained after splitting.

2.5.3. Example

```
IMPORT ML_Core.Preprocessing;

//Splitting into half
splitResult := Preprocessing.Split(someData, 0.5);
trainData := splitResult.trainData;
testData := splitResult.testData;
```

3. How to Install

The Preprocessing Bundle will be included into the ML_Core library. You can read about how to install the ML_Core library from [1]. After installation the Preprocessing Bundle could be accessed through the following import:

IMPORT ML_Core.Preprocessing;

You can access the beta version of the Preprocessing bundle at https://github.com/vzeufack/ML_Core.

Conclusion and Future Work

The current version of the preprocessing bundle allows you to handle categorical features (**LabelEncoder, OneHotEncoder**), scale (**MinMaxScaler, StandardScaler, Normaliz**) and split data into training and test sets (**Split, StratifiedSplit**). It has been made to help machine learning engineers speed up the data preparation of phase of Machine Learning projects undertaken on the HPCC Systems platform.

Future improvements will include the addition of more modules like an impute module for handling missing values. It should be noted that future changes or improvements will be mainly made from developers' remarks, needs or requests. Hence, please feel free to use and provide any feedback.

References

1. Learn how to use HPCC Systems Machine Learning Library https://hpccsystems.com/blog/HPCC-Sytems-Machine-Learning
2. Preprocessing Bundle Beta version: https://github.com/vzeufack/ML_Core