ALBUKHARY INTERNATIONAL UNIVERSITY

Artificial Intelligence

LAB THIRTEEN

Data Preprocessing

We deal with a lot of raw data in the real world. Machine learning algorithms expect data to be formatted in a certain way before they start the training process. In order to prepare the data for ingestion by machine learning algorithms, we have to preprocess it and convert it into the right format. Let's see how to do it.

Create a new Python file and import the following packages:

```
In [ ]: import numpy as np
        from sklearn import preprocessing
```

Let's define some sample data:

In [ ]:
```
input_data = np.array([[5.1, -2.9, 3.3],
[-1.2, 7.8, -6.1],
[3.9, 0.4, 2.1],
[7.3, -9.9, -4.5]])
```

We will be talking about several different preprocessing techniques. Let's start with binarization:

- Binarization
- Mean removal
- Scaling
- Normalization

1. Binarization

This process is used when we want to convert our numerical values into boolean values. Let's use an inbuilt method to binarize input data using 2.1 as the threshold value.

In [ ]:
```
# Binarize data
data_binarized = preprocessing.Binarizer(threshold=2.1).transform(input_data)
print("\nBinarized data:\n", data_binarized)
```

As we can see here, all the values above 2.1 become 1. The remaining values become 0.

2. Mean removal

Removing the mean is a common preprocessing technique used in machine learning. It's usually useful to remove the mean from our feature vector, so that each feature is centered on zero. We do this in order to remove bias from the features in our feature vector.

In [ ]:
```
# Print mean and standard deviation
print("\nBEFORE:")
print("Mean =", input_data.mean(axis=0))
print("Std deviation =", input_data.std(axis=0))
```

The preceding line displays the mean and standard deviation of the input data. Let's remove the mean:

In [ ]:
```
# Remove mean
data_scaled = preprocessing.scale(input_data)
print("\nAFTER:")
print("Mean =", data_scaled.mean(axis=0))
print("Std deviation =", data_scaled.std(axis=0))
```

As seen from the values obtained, the mean value is very close to 0 and standard deviation is 1.

3. Scaling

In our feature vector, the value of each feature can vary between many random values. So it becomes important to scale those features so that it is a level playing field for the machine learning algorithm to train on. We don't want any feature to be artificially large or small just because of the nature of the measurements.

In [ ]:
```
# Min max scaling
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled_minmax = data_scaler_minmax.fit_transform(input_data)
print("\nMin max scaled data:\n", data_scaled_minmax)
```

Each row is scaled so that the maximum value is 1 and all the other values are relative to this value.

4. Normalization

We use the process of normalization to modify the values in the feature vector so that we can measure them on a common scale. In machine learning, we use many different forms of normalization. Some of the most common forms of normalization aim to modify the values so that they sum up to 1. $L1normalization$, which refers to $Least Absolute Deviations$, works by making sure that the sum of absolute values is 1 in each row. $L2normalization$, which refers to $leastsquares$, works by making sure that the sum of squares is 1.

In general, $L1normalization$ technique is considered more robust than $L2normalization$ technique. $L1normalization$ technique is robust because it is resistant to outliers in the data. A lot of times, data tends to contain outliers and we cannot do anything about it. We want to use techniques that can safely and effectively ignore them during the calculations. If we are solving a problem where outliers are important, then maybe $L2normalization$ becomes a better choice.

In [ ]:
```
# Normalize data
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nL1 normalized data:\n", data_normalized_l1)
print("\nL2 normalized data:\n", data_normalized_l2)
```

Label encoding

When we perform classification, we usually deal with a lot of labels. These labels can be in the form of words, numbers, or something else. The machine learning functions in sklearn expect them to be numbers. So if they are already numbers, then we can use them directly to start training. But this is not usually the case. In the real world, labels are in the form of words, because words are human readable. We label our training data with words so that the mapping can be tracked. To convert word labels into numbers, we need to use a label encoder. Label encoding refers to the process of transforming the word labels into numerical form. This enables the algorithms to operate on our data.

In [ ]:
```
import numpy as np
from sklearn import preprocessing
```

Define some sample labels:

```
In [ ]: # Sample input labels
        input_labels = ['red', 'black', 'red', 'green', 'black', 'yellow', 'white']
```

Create the label encoder object and train it:

```
In [ ]: # Create label encoder and fit the labels
        encoder = preprocessing.LabelEncoder()
        encoder.fit(input_labels)
```

Print the mapping between words and numbers:

```
In [ ]: # Print the mapping
        print("\nLabel mapping:")
        for i, item in enumerate(encoder.classes_):
            print(item, '-->', i)
```

Let's encode a set of randomly ordered labels to see how it performs:

```
In [ ]: # Encode a set of labels using the encoder
        test_labels = ['green', 'red', 'black']
        encoded_values = encoder.transform(test_labels)
        print("\nLabels =", test_labels)
        print("Encoded values =", list(encoded_values))
```

Let's decode a random set of numbers:

In [ ]:
```python
# Decode a set of values using the encoder
encoded_values = [3, 0, 4, 1]
decoded_list = encoder.inverse_transform(encoded_values)
print("\nEncoded values =", encoded_values)
print("Decoded labels =", list(decoded_list))
```

In [ ]: