Data preprocessing
Custom Dataset for Tabular data

Data: Pumpkin Seeds Dataset

MD ISTIAK AHAMMED
Kyungpook National University

Data preprocessing

Dataset (example)

Features

Labels or Targets

Area	Perimeter	Major_Axis_Length	Minor_Axis_Length	Convex_Area	Equiv_Diameter	Eccentricity	Solidity	Extent	Roundness	Aspect_Ration	Compactness	Class
56276	888.242	326.1485	220.2388	56831	267.6805	0.7376	0.9902	0.7453	0.8963	1.4809	0.8207	Çerçevelik
76631	1068.146	417.1932	234.2289	77280	312.3614	0.8275	0.9916	0.7151	0.844	1.7811	0.7487	Çerçevelik
71623	1082.987	435.8328	211.0457	72663	301.9822	0.8749	0.9857	0.74	0.7674	2.0651	0.6929	Çerçevelik
66458	992.051	381.5638	222.5322	67118	290.8899	0.8123	0.9902	0.7396	0.8486	1.7146	0.7624	Çerçevelik
66107	998.146	383.8883	220.4545	67117	290.1207	0.8187	0.985	0.6752	0.8338	1.7413	0.7557	Çerçevelik
73191	1041.46	405.8132	231.4261	73969	305.2698	0.8215	0.9895	0.7165	0.848	1.7535	0.7522	Çerçevelik
73338	1020.055	392.2516	238.5494	73859	305.5762	0.7938	0.9929	0.7187	0.8857	1.6443	0.779	Çerçevelik
69692	1049.108	421.4875	211.7707	70442	297.8836	0.8646	0.9894	0.6736	0.7957	1.9903	0.7067	Çerçevelik
95727	1231.609	488.1199	251.3086	96831	349.118	0.8573	0.9886	0.6188	0.793	1.9423	0.7152	Çerçevelik
73465	1047.767	413.6504	227.2644	74089	305.8407	0.8356	0.9916	0.7443	0.8409	1.8201	0.7394	Çerçevelik
83429	1114.561	438.5827	242.8826	84126	325.9219	0.8327	0.9917	0.7019	0.844	1.8057	0.7431	Çerçevelik
85461	1136.125	446.2935	245.1551	86344	329.8671	0.8356	0.9898	0.7457	0.832	1.8205	0.7391	Çerçevelik
71393	1096.533	459.2091	199.1305	72203	301.4969	0.9011	0.9888	0.6	0.7461	2.3061	0.6566	Çerçevelik
80151	1088.349	420.8842	244.2649	80854	319.4549	0.8144	0.9913	0.7285	0.8503	1.7231	0.759	Çerçevelik
68078	1016.821	403.0626	215.6027	68709	294.414	0.8449	0.9908	0.7377	0.8274	1.8695	0.7304	Çerçevelik
57934	933.357	368.7807	201.2084	58651	271.595	0.838	0.9878	0.7124	0.8357	1.8328	0.7365	Çerçevelik
61138	953.256	371.2713	211.3706	61753	279.0042	0.8221	0.99	0.7391	0.8455	1.7565	0.7515	Çerçevelik
61519	964.694	382.1808	205.6436	62227	279.8722	0.8429	0.9886	0.6728	0.8307	1.8585	0.7323	Çerçevelik
76073	1064.233	430.7576	225.3286	76576	311.222	0.8523	0.9934	0.7692	0.844	1.9117	0.7225	Çerçevelik
56882	926.303	368.015	197.4554	57544	269.1178	0.8439	0.9885	0.7403	0.8331	1.8638	0.7313	Çerçevelik
69350	1037.403	418.2706	211.9446	70249	297.1517	0.8621	0.9872	0.7469	0.8098	1.9735	0.7104	Çerçevelik
82196	1141.067	466.2324	225.8543	82991	323.5046	0.8748	0.9904	0.6702	0.7933	2.0643	0.6939	Çerçevelik
62165	936.716	356.8281	222.3935	62647	281.3378	0.782	0.9923	0.7237	0.8903	1.6045	0.7884	Çerçevelik
85913	1120.778	435.7355	251.4643	86577	330.7383	0.8167	0.9923	0.7386	0.8595	1.7328	0.759	Çerçevelik
68683	1021.57	409.83	214.3968	69337	295.7193	0.8522	0.9906	0.6526	0.827	1.9115	0.7216	Çerçevelik
61215	951.511	369.4055	212.7202	61936	279.1798	0.8176	0.9884	0.6779	0.8497	1.7366	0.7558	Çerçevelik
72713	1039.375	401.8137	231.1989	73336	304.2714	0.8179	0.9915	0.6861	0.8458	1.738	0.7572	Çerçevelik
58684	917.482	355.8436	210.4503	59202	273.3474	0.8064	0.9913	0.7156	0.8761	1.6909	0.7682	Çerçevelik
59652	932.197	359.1674	213.7165	60204	275.5926	0.8037	0.9908	0.675	0.8626	1.6806	0.7673	Çerçevelik
89091	1138.972	443.718	256.1033	89964	336.7999	0.8166	0.9903	0.7407	0.863	1.7326	0.759	Çerçevelik

Data preprocessing

1. LabelEncoder
2. Data Normalization Using StandardScaler
3. Data Normalization Using MinMaxScaler

Data preprocessing

1. LabelEncoder

2. Data Normalization Using StandardScaler

3. Data Normalization Using MinMaxScaler

Data preprocessing

LabelEncoder

LabelEncoder is a class from the sklearn.preprocessing module in scikit-learn. It is used to encode categorical labels into numerical values

Why it needs?

In machine learning, many algorithms require numerical inputs, and categorical labels need to be converted into numerical form for such algorithms to work properly. LabelEncoder provides a simple and effective way to accomplish this task.

The encoded target values are returned as a *numpy array*. The numerical values assigned by LabelEncoder are based on the alphabetical order of the labels

Here's how LabelEncoder works:

from sklearn.preprocessing import LabelEncoder

- Initialize an instance of LabelEncoder: label_encoder = LabelEncoder()
- Fit the encoder to the target labels: *label_encoder.fit(target)*. This step calculates the unique labels in the target and assigns a numerical value to each unique label.
- Transform the target labels into encoded values: *encoded_target = label_encoder.transform(target)*. This step replaces each label with its corresponding numerical encoded value.

Data preprocessing

LabelEncoder (Example)

```
from sklearn.preprocessing import LabelEncoder

# Example target labels
target = ['cat', 'dog', 'bird', 'cat', 'dog']

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the target labels
encoded_target = label_encoder.fit_transform(target)

print("Original labels:", target)
print("Encoded labels:", encoded_target)
```

Output

```
Original labels: ['cat', 'dog', 'bird', 'cat', 'dog']
Encoded labels: [0 1 2 0 1]
```

Inverse Transform

```
decoded_labels = label_encoder.inverse_transform(encoded_target)
print("Decoded labels:", decoded_labels)

Decoded labels: ['cat' 'dog' 'bird' 'cat' 'dog']
```

Data preprocessing

1. LabelEncoder
2. Data Normalization Using StandardScaler
3. Data Normalization Using MinMaxScaler

Data preprocessing

Data Normalization Using StandardScaler

StandardScaler:

StandardScaler scales the data to have zero mean and unit variance. It calculates the mean and standard deviation of the data and transforms the values based on the following formula:

```
x_scaled = (x - mean) / standard_deviation
```

This transformation results in the distribution of the data having a mean of 0 and a standard deviation of 1.

Example: Let's say we have a feature with the following values: `[2, 4, 6, 8, 10]`. Applying StandardScaler to this feature would transform it to `[-1.414, -0.707, 0, 0.707, 1.414]`. The mean of the transformed data is 0, and the standard deviation is 1.

Code:

```
self.scaler = StandardScaler()

self.features = self.scaler.fit_transform(self.features)
```

The fit_transform method of `StandardScaler returns a numpy array`. This array contains the transformed features after applying the scaling operation. The transformed features will have zero mean and unit variance as a result of the scaling process.

Data preprocessing

Data Normalization Using StandardScaler

```python
# Create Custom Dataset
class CustomDatasetv31(Dataset):
    def __init__(self, xlsx_file, transform=None):
        self.data = pd.read_excel(xlsx_file)
        self.transform = transform
        self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
        self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

        self.label_encoder = LabelEncoder()
        self.target = self.label_encoder.fit_transform(self.target)

        self.scaler = StandardScaler()
        self.features = self.scaler.fit_transform(self.features)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        x = torch.tensor(self.features[index], dtype=torch.float32)  # Convert features to tensor
        y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

        if self.transform:
            x = self.transform(x)

        return x, y
```

Output

```
Features: tensor([-1.7847, -2.2158, -2.3202, -0.2385, -1.7932, -1.9212, -2.7299,  0.2028,
         0.8554,  1.8738, -1.7751,  2.1973])
Label: tensor(0.)
```

Data preprocessing

1. LabelEncoder
2. Data Normalization Using StandardScaler
3. Data Normalization Using MinMaxScaler

Data preprocessing

Data Normalization Using MinMaxScaler

`MinMaxScaler` scales the data to a specified range, typically between 0 and 1. It calculates the minimum and maximum values of the data and transforms the values based on the following formula:

`x_scaled = (x - min) / (max - min)`

This transformation linearly maps the original values to the desired range.

Example:

Consider a feature with the following values: `[2, 4, 6, 8, 10]`. Applying MinMaxScaler to this feature would transform it to `[0, 0.25, 0.5, 0.75, 1]`, assuming a `range of 0 to 1`. The minimum value becomes 0, and the maximum value becomes 1.

To apply `MinMaxScaler`, you can use the `MinMaxScaler` class from the `sklearn.preprocessing` module. Here's an example of how to use it:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaled_features = scaler.fit_transform(features)
```

In the code snippet above, `MinMaxScaler` is instantiated as scaler. Then, the fit_transform method is called on scaler, passing in the features data. This method first fits the scalar to the data by calculating the minimum and maximum values of each feature. Then, it transforms the data by scaling each feature to the range `[0, 1]` using the following formula:

`scaled_value = (value - min_value) / (max_value - min_value)`

The transformed features are returned as scaled_features, which will be a `numpy array`.

Data preprocessing

Data Normalization Using MinMaxScaler

```python
# Create Custom Dataset
class CustomDatasetv4(Dataset):
    def __init__(self, xlsx_file, transform=None):
        self.data = pd.read_excel(xlsx_file)
        self.transform = transform
        self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
        self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

        self.label_encoder = LabelEncoder()
        self.target = self.label_encoder.fit_transform(self.target)

        self.scaler = MinMaxScaler()
        self.features = self.scaler.fit_transform(self.features)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        x = torch.tensor(self.features[index], dtype=torch.float32)  # Convert features to tensor
        y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

        if self.transform:
            x = self.transform(x)

        return x, y
```

Output

```
Features: tensor([0.0941, 0.0286, 0.0156, 0.4430, 0.0940, 0.1213, 0.5384, 0.9446, 0.7669,
        0.8875, 0.1665, 0.7553])
Label: tensor(0.)
```

Data preprocessing

Original Vs Normalize Outputs

Original

```
Features: tensor([5.6276e+04, 8.8824e+02, 3.2615e+02, 2.2024e+02, 5.6831e+04, 2.6768e+02,
        7.3760e-01, 9.9020e-01, 7.4530e-01, 8.9630e-01, 1.4809e+00, 8.2070e-01])
Label: tensor(0.)
```

StandardScaler

```
Features: tensor([-1.7847, -2.2158, -2.3202, -0.2385, -1.7932, -1.9212, -2.7299,  0.2028,
         0.8554,  1.8738, -1.7751,  2.1973])
Label: tensor(0.)
```

MinMaxScaler

```
Features: tensor([0.0941, 0.0286, 0.0156, 0.4430, 0.0940, 0.1213, 0.5384, 0.9446, 0.7669,
        0.8875, 0.1665, 0.7553])
Label: tensor(0.)
```

Indexing Error in Pandas DataFrame

```python
# Create Custom Dataset
class CustomDatasetv2(Dataset):
  def __init__(self, xlsx_file, transform=None):
    self.data = pd.read_excel(xlsx_file)
    self.transform = transform
    self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
    self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

    self.label_encoder = LabelEncoder()
    self.target = self.label_encoder.fit_transform(self.target)

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    # x = torch.tensor(self.features.iloc[index], dtype=torch.float32)  # Convert features to tensor
    x = torch.tensor(self.features[index], dtype=torch.float32)  # Convert features to tensor
    y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

    if self.transform:
      x = self.transform(x)

    return x, y
```

Indexing Error in Pandas DataFrame

2.3.1 Lets create instance of custom dataset again

```
[95]
custom_datasetv2 = CustomDatasetv2(xlsx_file)
print(custom_datasetv2)
```

```
<__main__.CustomDatasetv2 object at 0x7f397d04eec0>
```

2.3.2 Check the feature and labels in any index

```
index = 2000  # Index of the sample you want to access

sample = custom_datasetv2[0]
features = sample[0]  # Access features
label = sample[1]  # Access label

print("Features:", features)
print("Label:", label)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   3801             try:
-> 3802                 return self._engine.get_loc(casted_key)
   3803             except KeyError as err:

                                    5 frames

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
   3802                 return self._engine.get_loc(casted_key)
   3803             except KeyError as err:
-> 3804                 raise KeyError(key) from err
   3805             except TypeError:
   3806                 # If we have a listlike key, _check_indexing_error will raise

KeyError: 0
```

Error occurred because of index error

Indexing Error in Pandas DataFrame

```
[97]
# Create Custom Dataset
class CustomDatasetv2(Dataset):
  def __init__(self, xlsx_file, transform=None):
    self.data = pd.read_excel(xlsx_file)
    self.transform = transform
    self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
    self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

    self.label_encoder = LabelEncoder()
    self.target = self.label_encoder.fit_transform(self.target)

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    x = torch.tensor(self.features.iloc[index], dtype=torch.float32)  # Convert features to tensor
    y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

    if self.transform:
      x = self.transform(x)

    return x, y
```

2.3.1 Lets create instance of custom dataset again

```
[98]
custom_datasetv2 = CustomDatasetv2(xlsx_file)
print(custom_datasetv2)
```

```
<__main__.CustomDatasetv2 object at 0x7f397d168fa0>
```

2.3.2 Check the feature and labels in any index

```
index = 2000  # Index of the sample you want to access

sample = custom_datasetv2[0]
features = sample[0]  # Access features
label = sample[1]  # Access label

print("Features:", features)
print("Label:", label)
```

```
Features: tensor([5.6276e+04, 8.8824e+02, 3.2615e+02, 2.2024e+02, 5.6831e+04, 2.6768e+02,
        7.3760e-01, 9.9020e-01, 7.4530e-01, 8.9630e-01, 1.4809e+00, 8.2070e-01])
Label: tensor(0.)
```

No Error and Output is showing

But what is the Difference?

[97]

```python
# Create Custom Dataset
class CustomDatasetv2(Dataset):
  def __init__(self, xlsx_file, transform=None):
    self.data = pd.read_excel(xlsx_file)
    self.transform = transform
    self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
    self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

    self.label_encoder = LabelEncoder()
    self.target = self.label_encoder.fit_transform(self.target)

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    x = torch.tensor(self.features.iloc[index], dtype=torch.float32)  # Convert features to tensor
    y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

    if self.transform:
        x = self.transform(x)

    return x, y
```

features.iloc[index]

features.[index]

[100]

```python
# Create Custom Dataset
class CustomDatasetv2(Dataset):
  def __init__(self, xlsx_file, transform=None):
    self.data = pd.read_excel(xlsx_file)
    self.transform = transform
    self.features = self.data.iloc[:, :-1]  # Select all columns except the last one
    self.target = self.data.iloc[:, -1]  # Select the last column as the target variable

    self.label_encoder = LabelEncoder()
    self.target = self.label_encoder.fit_transform(self.target)

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    # x = torch.tensor(self.features.iloc[index], dtype=torch.float32)  # Convert features to tensor
    x = torch.tensor(self.features[index], dtype=torch.float32)  # Convert features to tensor
    y = torch.tensor(self.target[index], dtype=torch.float32)  # Convert target to tensor

    if self.transform:
        x = self.transform(x)

    return x, y
```

In this case direct index has been applied to catch the rows and columns. The values of the default rows supposed to show. But the index was not similar to the original dataset.

features.[index] VS features.iloc[index]

Feature 1	Feature 2	Feature 3	Target
0.2	0.5	0.1	A
0.6	0.9	0.3	B
0.4	0.7	0.2	A

Let's consider an example dataset to explain the difference. In this example, the dataset has three feature columns (**Feature 1, Feature 2, and Feature 3**) and a target column (Target)

To summarize, when you use ***self.features[index]***, you access the row *based on the default index (e.g., row number)*, while using ***self.features.iloc[index]*** explicitly specifies the indexing *based on the integer location* of the row.

Accessing using the default index:

- If you use ***self.features[index]***, where the index is an integer, it will access the row based on the default index. For example, self.features[0] will return the values for the first row, which are [0.2, 0.5, 0.1].

- The default index in this case is [0, 1, 2], which corresponds to the row numbers of the dataset.

Accessing using the .iloc indexer:

- If you use ***self.features.iloc[index]***, where index is an integer, it will explicitly specify the indexing **based on the integer location of the row**. For example, self.features.iloc[0] will also return [0.2, 0.5, 0.1], which are the values of the first row.

- The ***.iloc*** indexer is used to access rows and columns of a pandas DataFrame based on their integer location. It disregards the default index and directly selects rows by their position.