CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

scikit learn

Machine Learning

Machine Learning

Part 2 – Data Preparation

Ted Scully

Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- Dealing with Missing Values
- **Handling Categorical Data**
- Scaling Data
- Handling Imbalance
- Feature Selection
- Dimensionality Reduction

Encoding Categorical Features

- When dealing with **categorical** data, it is important to distinguish between **nominal** and **ordinal** values.

- Ordinal values are variables that have a **logical ordering**. For example, a letter grade for a student, 'A', 'B', 'C', …

- In contrast nominal features have **no inherent ordering**. For example, a features that report's the colour of a car 'Red', 'Blue', etc.

- In the example below we have two categorical features. Department being a nominal feature and Grade being an ordinal feature

	Age	DegreeYear	Department	Grade
0	21	4	Computing	A
1	18	1	Biology	C
2	19	1	Chemistry	B

Ordinal Variables

- To make sure that our learning algorithms interpret the ordinal features correctly we need to convert the **categorical string values into integers**.

- Unfortunately, the ordering of ordinal values is typically related to the domain and as such there is no automatic mechanism of encoding this information.

- Therefore, you need to **specify the mapping manually**, which can be time consuming. In the example on the next slide we specify the mapping for the ordinal feature Grades.

- This involves creating a **dictionary** to specify the direct mapping and using a **dataframe** method call **map**.

In the following examples we will use the dataframe below

```
import pandas as pd
import numpy as np

seriesA = pd.Series(['A', 'C', 'B'])
seriesB = pd.Series([21, 18, 19])
seriesC = pd.Series([4, 1, 1])
seriesD = pd.Series(['Computing', 'Biology', 'Biology'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
            'Department' : seriesD})
print (df)
```

```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1    Biology     B
```

```
# continued from previous slide

grade_mapping = {'F':0, 'D':1, 'C':2, 'B':3, 'A':4}

df['Grade'] = df['Grade'].map(grade_mapping)

print (df)
```

```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B

   Age  DegreeYear Department  Grade
0   21           4  Computing      4
1   18           1    Biology      2
2   19           1  Chemistry      3
```

Notice we create a dictionary that provides a mapping from letter grades to numerical values.

We then provide call the map method for the Grade column, which takes the dictionary as an argument.

Nominal Values – Encode as Integer Values

- For nominal values we could directly **encode them into integer values** using the **OrdinalEncoder** from Scikitlearn. After the transform operation the OrdinalEncoder will return an array of numerical values.

```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder()

df["Department"] = enc.fit_transform(df[["Department"]])

print (df)
```

```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1    Biology     B
   Age  DegreeYear  Department Grade
0   21           4         1.0     A
1   18           1         0.0     C
2   19           1         0.0     B
```

Nominal Values

- For nominal values we could directly encode them into integer values using the **OrdinalEncoder** from Scikitlearn. After the transform operation the OrdinalEncoder will return an array of numerical values.

```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder()

df[["Grade", "Department"]] = enc.fit_transform(df[["Grade", "Department"]])

print (df)
```

```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1    Biology     B
   Age  DegreeYear  Department  Grade
0   21           4         1.0    0.0
1   18           1         0.0    2.0
2   19           1         0.0    1.0
```

Encoding Categorical Features – Nominal Values

	Age	DegreeYear	Department	Grade
0	21	4	Computing	A
1	18	1	Biology	C
2	19	1	Chemistry	B

	Age	DegreeYear	Department	Grade
0	21	4	2.0	A
1	18	1	0.0	C
2	19	1	1.0	B

- Inadvertently, we have now specified a ordering on a nominal categorical feature.
- Although the Department feature has no specific ordering, a learning algorithm will view the encoding as an ordering. Notice that that Computing is closer to Chemistry then it is to Biology.

Encoding Categorical Features – Nominal Values

- A common way of addressing this problem is to use a technique referred to as '**one-hot encoding**'.
- One hot encoding transforms each categorical feature with **n** possible values into **n** binary features
- In the example on the previous slide we could convert the **Department** feature into three new binary features: Computing, Biology and Chemistry.
  - Binary values can then be used indicate the presence of a particular department.
  - For example, a Computing sample would be encoded as Computing = 1, Chemistry = 0 and Biology = 0

One Hot Encoding

	Age	DegreeYear	Department	Grade
0	21	4	Computing	A
1	18	1	Biology	C
2	19	1	Chemistry	B

- **Scikitlearn** provides a **OneHotEncoder** class that allows us to implement a One-Hot Encoder method.

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder


seriesA = pd.Series(['A', 'C', 'B'])
seriesB = pd.Series([21, 18, 19])
seriesC = pd.Series([4, 1, 1])
seriesD = pd.Series(['Computing', 'Biology', 'Biology'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
            'Department' : seriesD})
print (df)


encoder = OneHotEncoder(sparse=False)
departmentEncoded =  encoder.fit_transform( df[["Department"]] )
```

```
  Grade  Age  DegreeYear Department
0     A   21           4  Computing
1     C   18           1    Biology
2     B   19           1    Biology
3     D   18           2  Computing
4     C   17           3  Chemistry
```

```
[[0. 0. 1.]
 [1. 0. 0.]
 [1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
```

Notice when we call fit_transform on the department column it returns the one-hot encoding of this column.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder


seriesA = pd.Series(['A', 'C', 'B'])
seriesB = pd.Series([21, 18, 19])
seriesC = pd.Series([4, 1, 1])
seriesD = pd.Series(['Computing', 'Biology', 'Biology'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
          'Department' : seriesD})
print (df)



encoder = OneHotEncoder(sparse=False)
categEncoded = encoder.fit_transform(df[["Department", "Grade"]])
print (categEncoded)
```

```
  Grade  Age  DegreeYear Department
0     A   21           4  Computing
1     C   18           1    Biology
2     B   19           1    Biology
3     D   18           2  Computing
4     C   17           3  Chemistry
```

```
[[0. 0. 1. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 1. 0.]]
```

Using ColumnTransformer

- This ColumnTransformer is a class that allows **different columns** of the input to be **transformed separately** and the results combined into a single feature space. This is useful when dealing with datasets that contain heterogeneous data types.
- Let's illustrate using a slightly different dataset. You will notice there are two missing values in the Age column, one in the degree and one in the Department column. In this example we are going to use ColumnTransformer to perform impute for missing numerical and categorical features and merge the result.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer

seriesA = pd.Series(['A', 'C', 'B', 'B'])
seriesB = pd.Series([21, 18])
seriesC = pd.Series([4, 1, 1])
seriesD = pd.Series(['Computing', 'Biology',  "Chemistry"])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
          'Department' : seriesD})
print (df)
```

```
  Grade   Age  DegreeYear Department
0     A  21.0         4.0  Computing
1     C  18.0         1.0    Biology
2     B   NaN         1.0  Computing
3     B   NaN         NaN        NaN
```

Using ColumnTransformer

- The main parameter in ColumnTransformer is called **transformers**, which is a list of tuples **(name, transformer, column(s))** specifying the transformer objects to be applied to subsets of the data. See here for a full list of supported transformers.

```
#separate the categorical and numerical data
categoricalFeatures = ['Grade', 'Department']
numericalFeatures = ['DegreeYear', 'Age']

print ()
# On creating the ColumnTransformer we specify a list of transformer operations
preprocessor = ColumnTransformer(     transformers=
   [ ('cat1', SimpleImputer(strategy='mean'), numericalFeatures),
    ('num', SimpleImputer(strategy="most_frequent"), categoricalFeatures) ] )


transformedData = preprocessor.fit_transform(df)
df = pd.DataFrame(data= transformedData)

print (df)
```

	0	1	2	3
0	4	21	A	Computing
1	1	18	C	Biology
2	1	19.5	B	Computing
3	2	19.5	B	Computing

Using ColumnTransformer

- Another useful parameter we can specific for the ColumnTransformer is **remainder='passthrough'.** By default, only the specified columns in transformers are transformed and combined in the output, and the non-specified columns are dropped. By specifying remainder='passthrough', all remaining columns that were not specified in transformers will be automatically passed through.

```
from sklearn.impute import SimpleImputer

seriesA = pd.Series(['A', 'C', 'B', 'B'])
seriesB = pd.Series([21, 18, 17, 14])
seriesC = pd.Series([4, 1, 1, 3])
seriesD = pd.Series(['Computing', 'Biology',  "Computing", "Biology"])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
            'Department' : seriesD})
print (df)
```

```
  Grade  Age  DegreeYear Department
0     A   21           4  Computing
1     C   18           1    Biology
2     B   17           1  Computing
3     B   14           3    Biology
```

Using ColumnTransformer

```
categoricalFeatures = ['Grade', 'Department']
numericalFeatures = ['DegreeYear', 'Age']

print ()
# On creating the ColumnTransformer we specify a list of transformer operations
preprocessor = ColumnTransformer(
   [ ('num', OneHotEncoder(sparse="False", categories='auto’),
        categoricalFeatures)], remainder='passthrough')

transformedData = preprocessor.fit_transform(df)
df = pd.DataFrame(data= transformedData)

print (df)
```

```
     0    1    2    3    4     5    6
0  1.0  0.0  0.0  0.0  1.0  21.0  4.0
1  0.0  0.0  1.0  1.0  0.0  18.0  1.0
2  0.0  1.0  0.0  0.0  1.0  17.0  1.0
3  0.0  1.0  0.0  1.0  0.0  14.0  3.0
```

One-hot Encoding

- The disadvantage of one-hot encoding is that if there are a very large number of distinct values for a feature, which can consequently mean that we end up with a very large number of additional features.

- In turn this can lead to a very long training time or underperformance of some models due to the rapid increase in dimensionality.

- One approach that is used to mitigate the impact of this is dimensionality reduction (techniques such as PCA), which can allow us to in some cases dramatically reduce the overall number of features.

Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- Dealing with Missing Values
- Handling Categorical Data
- **Scaling Data**
- Handling Imbalance
- Feature Selection
- Dimensionality Reduction

Scaling Data

- Feature scaling is a important step in pre-processing for machine learning. The majority of ML and optimization algorithms behave better if features are on the same scale.

- The two most common data transformation techniques used are:
  - Normalising Data
  - Standardizing Data

Normalising Data

- Normalization is the rescaling of the features into the range between **0 and 1**.

- This can improve the performance for algorithms that assign a weight to features such as linear regression and in particular for algorithms that utilize geometric distance such as KNNs.

- In the following dataset I print out the first few rows of the feature data. You will notice that the second feature has a much greater range than any of the other features.

```
[[  5.          67.          3.          5.        ]
 [  4.          43.          1.          1.        ]
 [  5.          58.          4.          5.        ]
 [  4.          28.          1.          1.        ]
 [  5.          74.          1.          5.        ]
 [  4.          65.          1.          2.79627601]]
```

Normalising Data

- Scikit allows us to normalize all features using the MinMaxScaler class in sklearn.preprocessing.
- Note the fit_transform function takes as input a **NumPy** array or a **DataFrame**.
- It returns NumPy array as an argument.

```
from sklearn import preprocessing

scalingObj = preprocessing.MinMaxScaler()
newX = scalingObj.fit_transform(allValues)
```

```
[[ 0.09090909  0.62820513  0.66666667  1.        ]
 [ 0.07272727  0.32051282  0.          0.        ]
 [ 0.09090909  0.51282051  1.          1.        ]
 [ 0.07272727  0.12820513  0.          0.        ]
 [ 0.09090909  0.71794872  0.          1.        ]
 [ 0.07272727  0.6025641   0.          0.449069  ]]
```

```
import pandas as pd
from sklearn import preprocessing
Import numpy as np

seriesA = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])
seriesB = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA, 'two' : seriesB, three' : seriesC})
print (df)

scalingObj = preprocessing.MinMaxScaler()
df[['one', 'two']]= scalingObj.fit_transform( df[['one', 'two']]  )
print (df)
```

```
        one      three        two
a  20.668265  36.253693  13.657378
b  72.221698  67.915196  76.266970
c  78.236288  70.865698   2.496296
d   5.661663  58.946655  47.480441

       one      three       two
a  0.206775  36.253693  0.151294
b  0.917125  67.915196  1.000000
c  1.000000  70.865698  0.000000
d  0.000000  58.946655  0.609784
```

In the example above we perform normalization on two columns from our dataframe object.

Standardizing Data

- Standardization facilitates the transformation of features to **a standard Gaussian(normal) distribution** with a **mean of 0** and a **standard deviation of 1**.
- The scaling happens independently on each individual feature by computing the relevant statistics on the samples in the training set.
- Standardization of a dataset is a common requirement for dealing with dataset features with different ranges but also for many machine learning estimators such as clustering techniques , logistic regression, neural networks, SVMs.
- One point to note is that standardization is less sensitive to outlier than normalization.

$$z = \frac{x - \mu}{\sigma}$$

Input	Standardized	Normalized
0.0	-1.33	0.0
1.0	-0.80	0.2
2.0	-0.26	0.4
3.0	0.26	0.6
4.0	0.80	0.8
5.0	1.33	1.0

Standardizing Data

- In the example below we standardize the training data from the iris data
- The values for each attribute now have a mean value of 0 and a standard deviation of 1.
- As with the MinMaxScaler object the StandardScaler can accept either a **NumPy** array or a Pandas **dataframe**.

```
[[ 0.37   0.8    0.23   1.44]
 [-0.2   -0.87  -1.41  -1.18]
 [ 0.37   0.17   1.05   1.44]
 [-0.2   -1.9   -1.41  -1.18]
 [ 0.37   1.28  -1.41   1.44]
 [-0.2    0.66  -1.41   0.  ]]
```

```
from sklearn import preprocessing

scaler = preprocessing.StandardScaler()
newX = scaler.fit_transform(allValues)
```

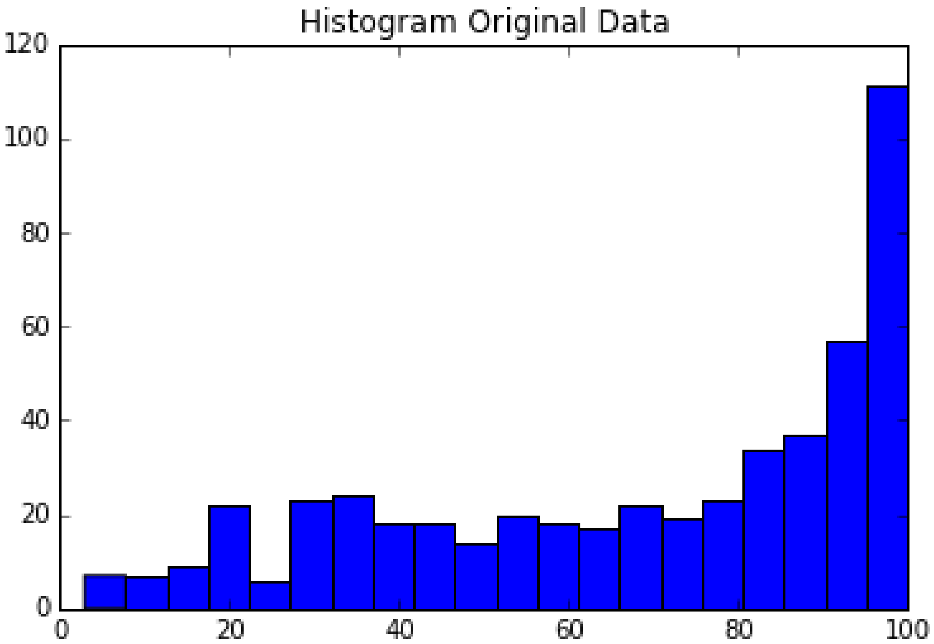
If you apply **MinMaxScaler** or **StandardScaler** to a dataframe containing multiple features with different data types it will generate a warning.

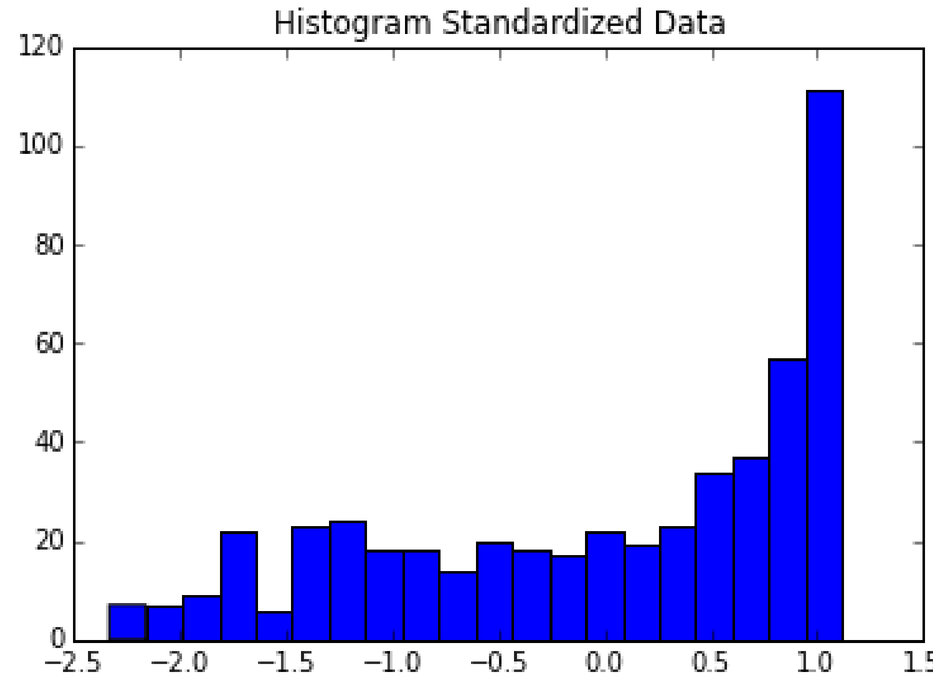
The scaling is still perfomed. To view the datatypes of each column in a dataframe you can use df.info(). If you then wish to change the datatype of any column in a dataframe you can use pd.astype().

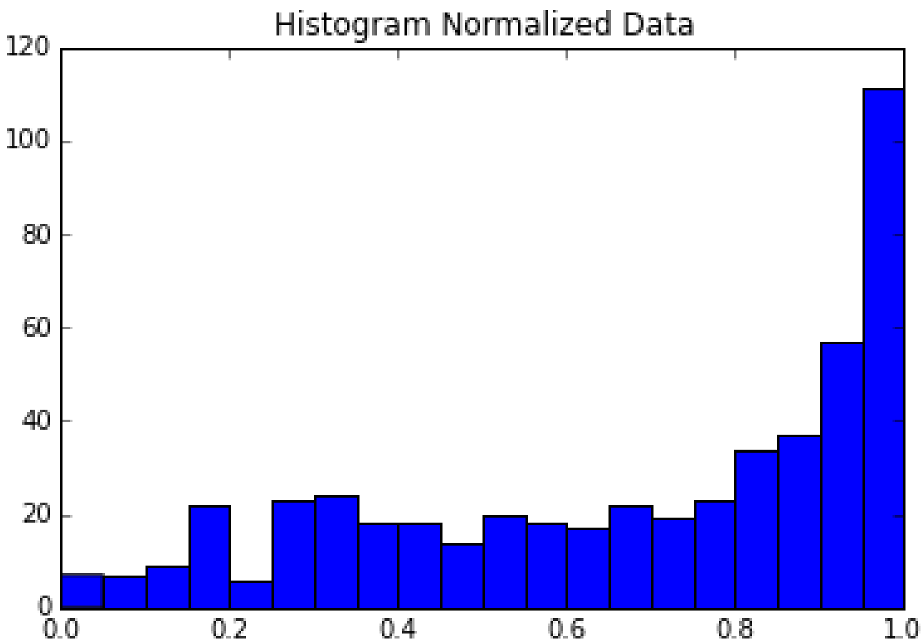
```
import pandas as pd
import numpy as np
from sklearn import preprocessing

seriesA = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])
seriesB = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(4)*100, index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA, 'two' : seriesB, 'three' : seriesC})
print (df)

scaler = preprocessing.StandardScaler()
df[['one', 'two']]= scaler.fit_transform( df[['one', 'two']]  )
print (df)
```

```
         one      three        two
a  20.668265  36.253693  13.657378
b  72.221698  67.915196  76.266970
c  78.236288  70.865698   2.496296
d   5.661663  58.946655  47.480441

        one      three       two
a  0.206775  36.253693  0.151294
b  0.917125  67.915196  1.000000
c  1.000000  70.865698  0.000000
d  0.000000  58.946655  0.609784
```

In the example above we perform standardization on two columns from our dataframe object.







- Unless you fully understand in-depth all the ML algorithms you use it can be very difficult to know if you should use normalization or standardization when scaling your data.

- It is generally recommended that try both approaches to determine which provides the best accuracy value.