Data Pre-processing

Dummy Variables, Imputing Missing Values

Need for Data Pre-processing

- Many times data is not compatible to be passed to any function in the libraries like scikit-learn
- Data can be
  - Categorical
  - With some genuinely missing values
  - With variables of different scales
  - Too much dispersed

Categorical Data

- Some functions will not accept the data in categorical form
- Hence we require to create a dummy data
- This is also called One Hot Encoding

Categorical Variable

Type
Small
Medsize
Small
Compact
Small
Medsize
Compact

Dummy Variables

Small	Medsize	Compact
1	0	0
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0
0	0	1

Dummy Variables

- Dummy variables all taken at a time may introduce linear relationship within the predictors which also isn't allowed
- Hence we need to drop one of the variables

Categorical Variable

Type
Small
Medsize
Small
Compact
Small
Medsize
Compact

Dummy Variables

Small	Medsize
1	0
0	1
1	0
0	0
1	0
0	1
0	0

Dummy Variables in pandas

- Dummy variables in pandas can be created with the function get_dummies()

Syntax: DataFrame.get_dummies(DataFrame Object, drop_first)

```
dum_cars = pd.get_dummies(cars, drop_first=True)
```

Label Encoding

- A Multi-Class responses requires label encoding

```
In [55]: from sklearn.preprocessing import LabelEncoder
    ...: lbcode = LabelEncoder()
    ...: y = ['a','b','a','a','c','a','b','b','a','c','a']
    ...: trny = lbcode.fit_transform(y)
    ...: print(trny)
[0 1 0 0 2 0 1 1 0 2 0]
```

Genuinely Missing Values

- Missing Values can be missing not just because of negligence, but also because the information wasn’t collected due to some reasons
- Our functions / algorithms in ML cannot tolerate missing values
  - Either we remove them. If it doesn’t matter
  - Or we impute them

Dropping NA values

Syntax: DataFrame.dropna(axis,how, …)

Where

axis: 0 for rows; 1 for column

how: “any” : if any NA values are present, drop that label(row/column)

“all” : if all values are NA, drop that label(row/column)

```
In [22]: carsMissing = pd.read_csv("F:/Python Material/ML with Python/
Datasets/Cars93Missing.csv")
    ...: carsMissing.shape
Out[22]: (93, 26)

In [23]: carsDropNA = carsMissing.dropna()
    ...: carsDropNA.shape
Out[23]: (76, 26)
```

Sane's STATS Academy of Statistics

Imputation

- We can make an educated guess on the nan values like imputing mean, median in case of numeric data or imputing mode in case of categorical data
- We require to import class SimpleImputer from sklearn.impute

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy='mean')
carsImputed = imp.fit_transform(dum_cars_miss)
```

Sane's STATS Academy of Statistics

Variables with different scales

- Sometimes, in the data in the data, we may get two variables of totally different scales. Say rating between 1 to 10 and Sales figure in crores
- In cluster analysis or PCA kind of algorithms, we require all variables to be treated equally
- This causes an imbalance as Sales figures will influence the whole analysis and rating variable won't have any role
- Hence we need to bring them all to one scale. This is called scaling

Scaling in Python

- We require to import StandardScaler from sklearn.preprocessing
- We consider here dataset milk for example

$$ScaledX = \frac{X - mean(X)}{Std(X)}$$

```
In [45]: milk.head()
Out[45]:
           water  protein  fat  lactose   ash
HORSE       90.1      2.6  1.0      6.9  0.35
ORANGUTAN   88.5      1.4  3.5      6.0  0.24
MONKEY      88.4      2.2  2.7      6.4  0.18
DONKEY      90.3      1.7  1.4      6.2  0.40
HIPPO       90.4      0.6  4.5      4.4  0.10

In [46]: np.mean(milk), np.std(milk)
Out[46]:
(water       78.1840
 protein      6.2120
 fat         10.3080
 lactose      4.1320
 ash          0.8632
 dtype: float64, water       12.558939
 protein      3.578751
 fat         10.305491
 lactose      1.794819
 ash          0.494625
 dtype: float64)
```

Sane's STATS Academy of Statistics

Scaling in Python

```
In [63]: from sklearn.preprocessing import StandardScaler
    ...: scaler = StandardScaler()
    ...: scaler.fit(milk)
    ...: milkscaled=scaler.transform(milk)
    ...: np.mean(milkscaled[:,0]), np.std(milkscaled[:,0])
Out[63]: (-9.237055564881303e-16, 0.9999999999999999)

In [64]: np.mean(milkscaled[:,1]), np.std(milkscaled[:,1])
Out[64]: (2.6645352591003756e-17, 0.9999999999999998)

In [65]: np.mean(milkscaled[:,2]), np.std(milkscaled[:,2])
Out[65]: (1.7763568394002505e-17, 1.0)

In [66]: np.mean(milkscaled[:,3]), np.std(milkscaled[:,3])
Out[66]: (-2.575717417130363e-16, 1.0)

In [67]: np.mean(milkscaled[:,4]), np.std(milkscaled[:,4])
Out[67]: (4.440892098500626e-18, 1.0)
```

Sane's STATS Academy of Statistics

Min Max Scaling

- There is often a need for scaling the variables between the values 0 to 1
- We can import MinMaxScaler from sklearn.preprocessing

$$MinMax\ X = \frac{X\ - \min(X)}{\max(X) - \min(X)}$$

```
In [6]: from sklearn.preprocessing import MinMaxScaler
   ...: minmax = MinMaxScaler()
   ...: minmax.fit(milk)
   ...: minmaxMilk = minmax.transform(milk)
   ...: minmaxMilk[1:5,]
Out[6]:
array([[0.95824176, 0.06837607, 0.06097561, 0.86956522, 0.06363636],
       [0.95604396, 0.13675214, 0.04146341, 0.92753623, 0.03636364],
       [0.9978022 , 0.09401709, 0.0097561 , 0.89855072, 0.13636364],
       [1.        , 0.        , 0.08536585, 0.63768116, 0.        ]])
```

Questions?