University of Salford, MSc Data Science

Module: Machine Learning & Data Mining

Session: Workshop Week 2

Topic: Data Pre-processing

Tools: Jupyter Notebook

Objectives:

After completing this workshop, you will be able to:

- Create and manipulate pandas Data Frames to explore your data
- Identify missing values in your dataset, and use a range of imputation strategies for both numerical and categorical variables
- Use one hot and ordinal encoding methods for categorical variables
- Standardize data using normalization or min-max scaling methods

@Dr Azadeh Mohammadi

MLDM Week 2 Workshop: Data pre-processing

Introduction

Data comes from a variety of sources, forming the foundation of any analysis or model. However, for models to be reliable, the data must be clean, relevant, and well-structured. Poor-quality data can lead to misleading results, often due to errors or inconsistencies introduced during the data collection process.

In this workshop, we review loading and handling data in python. Then we will cover the following issues and potential solutions using python libraries.

- Missing values
- Categorical variables
- Standardization

Data manipulation

Pandas is a Python library used for working with datasets. It has many functions for:

- Loading
- Analyzing
- Exploring
- Manipulating
- Cleaning

Important Note: Saving Your Notebook to F Drive

By default, when you launch Jupyter Notebook, the starting directory is in the C drive. If you are using a university desktop or laptop to complete this workshop, you should make sure that at the end of session you save your notebook to F drive so you can access it from other university devices.

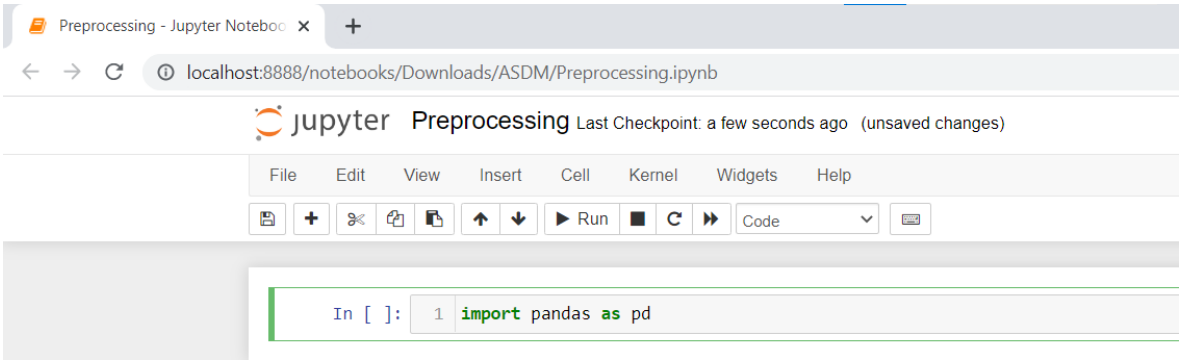
Alternatively, you can launch Jupyter Notebook with the F drive as the starting directory by opening the Anaconda Prompt and using the below command:

```
jupyter notebook --notebook-dir=F:/
```

1) Open Jupyter and open a new notebook by clicking on New> notebook. Select Python3 as Kernel
2) To install pandas' package, you can run the following command in your notebook: (It is already preinstalled in Anaconda so there is no need to reinstall it)

```
In [ ]: !pip install pandas
```

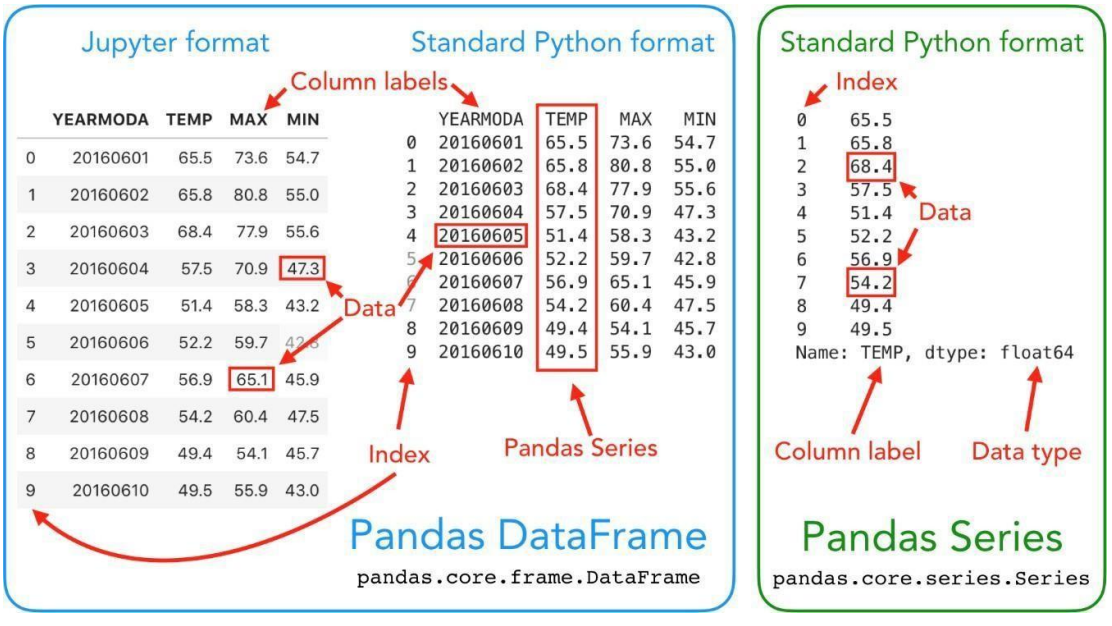
3) After installation, pandas should be imported as follows:

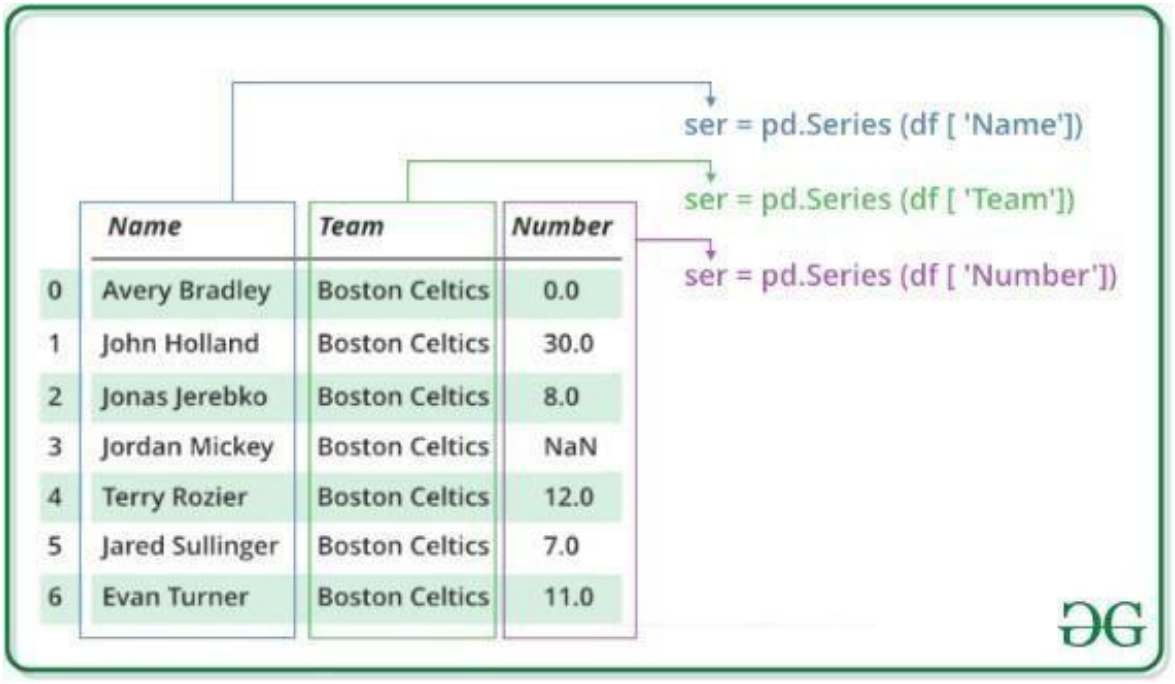


Pandas' library has many functions to load data in different formats; after loading, pandas keeps them as a structured and tabular object namely **data frames**. Using the following function we can load a comma-separated values file (CSV) which is a widely adopted format in data science:

```
In [ ]: 1 df = pd.read_csv('file-name.csv')
```

DataFrames are a set of series. Each set is denoted by the columns name.





Example

This is a sample dataset containing 10 records of car insurance claims. They show whether the owners of these cars have claimed for an accident or not. There are 6 features that are categorical and integer values and a binary target variable denoting whether they are claimed or not. Download the 'Car insurance.csv' file from Blackboard.

Make	Age	Mileage	Fuel	Gearbox	Colour	Claimed
Toyota	2	27000	P	A	Red	Yes
Ford	4	30500	P	M	Black	Yes
Toyota	15	120000	D	A	White	No
Nissan	13	53000	D	M	White	No
Nissan	2		D	M	Black	No
Ford	8	73000		M	Green	No
Toyota		138000	P	M		Yes
Nissan	20	38000	D		Green	Yes
Toyota	13	67000	D	A	Blue	No
Nissan	7	36000	P	M	Black	Yes

4) Using pandas, we load the file into a Data Frame. (Important Note: Before loading the dataset to a Data Frame you should upload the dataset in the same directory as your notebook)

```
In [2]:  1  df = pd.read_csv('Car Insurance.csv')
```

Alternative method: Instead of uploading the dataset to the notebook's directory, you can load the dataset to a Data Frame from its location using the following code:

df = pd.read_csv (r'Path where the CSV file is stored\File name.csv')

5) We can show the content of the data frame by calling its name

```
In [187]:  1  df
```

Out[187]:

	Make	Age	Mileage	Fuel	Gearbox	Colour	Claimed
0	Toyota	2.0	27000.0	P	A	Red	Yes
1	Ford	4.0	30500.0	P	M	Black	Yes
2	Toyota	15.0	120000.0	D	A	White	No
3	Nissan	13.0	53000.0	D	M	White	No
4	Nissan	2.0	NaN	D	M	Black	No
5	Ford	8.0	73000.0	NaN	M	Green	No
6	Toyota	NaN	138000.0	P	M	NaN	Yes
7	Nissan	20.0	38000.0	D	NaN	Green	Yes
8	Toyota	13.0	67000.0	D	A	Blue	No
9	Nissan	7.0	36000.0	P	M	Black	Yes

6) Data frame elements are addressed through an index and a column label. We can use row and column indices to show the content of a cell. In addition, we can call the column name to return a python series.

DataFrame.iloc[row,col] shows the value of denoted element.

Indices start from 0.

7) Showing all elements using iloc:

```
In [9]:  1 df.iloc[:,:]
```

Out[9]:

	Make	Age	Mileage	Fuel	Gearbox	Colour	Claimed
0	Toyota	2.0	27000.0	P	A	Red	Yes
1	Ford	4.0	30500.0	P	M	Black	Yes
2	Toyota	15.0	120000.0	D	A	White	No
3	Nissan	13.0	53000.0	D	M	White	No
4	Nissan	2.0	NaN	D	M	Black	No
5	Ford	8.0	73000.0	NaN	M	Green	No
6	Toyota	NaN	138000.0	P	M	NaN	Yes
7	Nissan	20.0	38000.0	D	NaN	Green	Yes
8	Toyota	13.0	67000.0	D	A	Blue	No
9	Nissan	7.0	36000.0	P	M	Black	Yes

8) Showing the first column

```
In [10]:  1 df.iloc[:,0]
```

```
Out[10]: 0    Toyota
         1      Ford
         2    Toyota
         3    Nissan
         4    Nissan
         5      Ford
         6    Toyota
         7    Nissan
         8    Toyota
         9    Nissan
         Name: Make, dtype: object
```

9) Showing the second row

```
In [23]: df.iloc[1]
```

```
Out[23]: Make          Ford
         Age            4.0
         Mileage    30500.0
         Fuel             P
         Gearbox          M
         Colour       Black
         Claimed        Yes
         Name: 1, dtype: object
```

10) Calling the third element of the first column:

```
In [11]:  1 df.iloc[2,0]
```

```
Out[11]: 'Toyota'
```

11) Returning the Age column:

```
In [33]: 1 df['Age']

Out[33]: 0     2.0
         1     4.0
         2    15.0
         3    13.0
         4     2.0
         5     8.0
         6     NaN
         7    20.0
         8    13.0
         9     7.0
         Name: Age, dtype: float64
```

12) Returning the Age column:

```
In [34]: 1 df.Age

Out[34]: 0     2.0
         1     4.0
         2    15.0
         3    13.0
         4     2.0
         5     8.0
         6     NaN
         7    20.0
         8    13.0
         9     7.0
         Name: Age, dtype: float64
```

13) Listing the unique values of the first column.

```
In [13]: 1 df.iloc[:,0].unique()

Out[13]: array(['Toyota', 'Ford', 'Nissan'], dtype=object)
```

14) Counting each value in the first column.

```
In [15]: 1 df.iloc[:,0].value_counts()

Out[15]: Nissan    4
         Toyota    4
         Ford      2
         Name: Make, dtype: int64
```

15) Checking where (in which rows) 'Nissan' exists in the first column?

```
In [19]: 1 df.iloc[:,0].isin(['Nissan'])

Out[19]: 0    False
         1    False
         2    False
         3     True
         4     True
         5    False
         6    False
         7     True
         8    False
         9     True
         Name: Make, dtype: bool
```

16) Sorting ascendingly the data frame based on values of the second column.

```
In [24]: 1 df.iloc[:,1].sort_values()

Out[24]: 0     2.0
         4     2.0
         1     4.0
         9     7.0
         5     8.0
         3    13.0
         8    13.0
         2    15.0
         7    20.0
         6     NaN
         Name: Age, dtype: float64
```

17) Sorting descendingly the data frame based on values of the second column.

```
In [26]: 1 df.iloc[:,1].sort_values(ascending=False)

Out[26]: 7    20.0
         2    15.0
         3    13.0
         8    13.0
         5     8.0
         9     7.0
         1     4.0
         0     2.0
         4     2.0
         6     NaN
         Name: Age, dtype: float64
```

18) Calculating min, max and mean value of Age column.

```
In [43]: 1 df['Age'].min()
Out[43]: 2.0

In [44]: 1 df['Age'].max()
Out[44]: 20.0

In [46]: 1 df['Age'].mean()
Out[46]: 9.333333333333334
```

Missing Values

Missing values are the most common issues in gathered data. They can raise serious errors and exceptions during working with data. Therefore, you need to manage them before taking any other action in data mining. In dataframes, missing values are shown by NaN.

```
In [9]: 1 df.iloc[:,:]
```

Out[9]:

	Make	Age	Mileage	Fuel	Gearbox	Colour	Claimed
0	Toyota	2.0	27000.0	P	A	Red	Yes
1	Ford	4.0	30500.0	P	M	Black	Yes
2	Toyota	15.0	120000.0	D	A	White	No
3	Nissan	13.0	53000.0	D	M	White	No
4	Nissan	2.0	NaN	D	M	Black	No
5	Ford	8.0	73000.0	NaN	M	Green	No
6	Toyota	NaN	138000.0	P	M	NaN	Yes
7	Nissan	20.0	38000.0	D	NaN	Green	Yes
8	Toyota	13.0	67000.0	D	A	Blue	No
9	Nissan	7.0	36000.0	P	M	Black	Yes

19) Checking which rows has null value in the Age column (second column).

```
In [28]: 1 df.iloc[:,1].isnull()
Out[28]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         7    False
         8    False
         9    False
         Name: Age, dtype: bool
```

20) Checking whether the second column has any null value.

```
In [29]:  1  df.iloc[:,1].isnull().values.any()
Out[29]: True
```

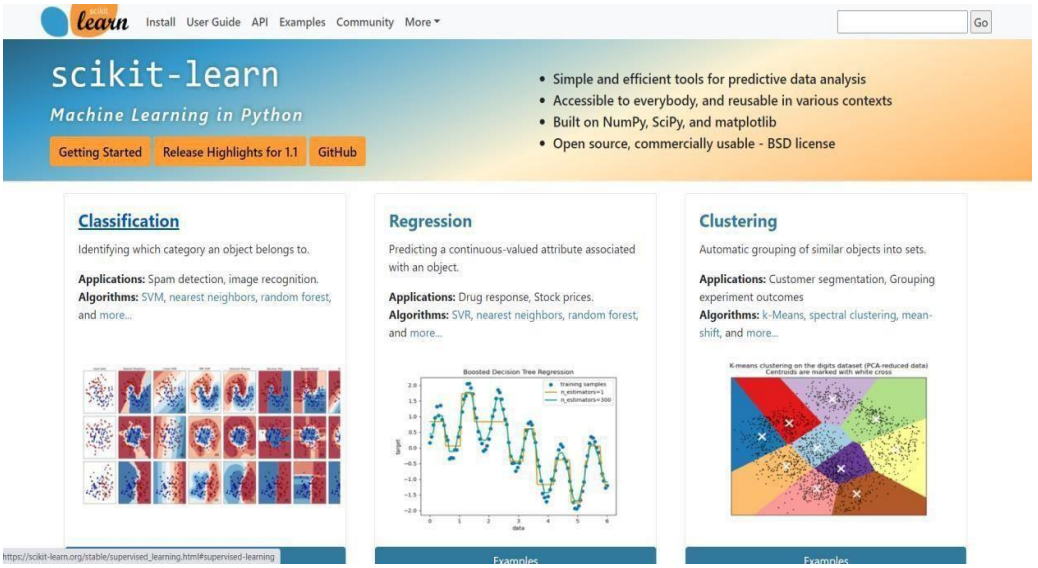
21) Checking which columns contain null values.

```
In [52]:  1  df.isnull().sum()
Out[52]: Make       0
         Age        1
         Mileage    1
         Fuel       1
         Gearbox    1
         Colour     1
         Claimed    0
         dtype: int64
```

which means we have missing values in Age, Mileage, fuel, Gearbox and Colour columns, but there is no missing value in Make and Claimed

For imputing missing values, we can use Scikit-learn library.

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms. This library is also one of the most adopted tools for pre-processing data (source: https://scikit-learn.org/stable/).



Scikit learn can be installed using:

```
!pip install scikit-learn
```

22) To impute data, we need to import SimpleImputer class from sklearn.impute library.

```
In [47]:
1 from sklearn.impute import SimpleImputer
```

Regarding whether data are numerical or categorical you can select appropriate strategies for imputing. For numerical variables the existing strategies are mean, median, most_frequent, and constant values.

23) As Age and Mileage are numerical and contain null values, we can use SimpleImputer to estimate the null value. The following code uses "mean" to get the null values filled. It replaces the null value of Age and Mileage columns (NaN) by mean value of those columns.

```
In [49]:
1 # numerical variables
2 # strategies: mean, median, most_frequent, constant
3 import numpy as np
4 numImputer = SimpleImputer(missing_values=np.nan, strategy='mean')
5 numImputer = numImputer.fit(df[['Age','Mileage']])
6 new_serr=numImputer.transform(df[['Age','Mileage']])
7 new_serr
```

```
Out[49]: array([[2.00000000e+00, 2.70000000e+04],
                [4.00000000e+00, 3.05000000e+04],
                [1.50000000e+01, 1.20000000e+05],
                [1.30000000e+01, 5.30000000e+04],
                [2.00000000e+00, 6.47222222e+04],
                [8.00000000e+00, 7.30000000e+04],
                [9.33333333e+00, 1.38000000e+05],
                [2.00000000e+01, 3.80000000e+04],
                [1.30000000e+01, 6.70000000e+04],
                [7.00000000e+00, 3.60000000e+04]])
```

24) The following uses median to impute null values.

In [53]:
```
# numerical variables
# strategies: mean, median, most_frequent, constant
import numpy as np
numImputer = SimpleImputer(missing_values=np.nan, strategy='median')
numImputer = numImputer.fit(df[['Age','Mileage']])
new_serr=numImputer.transform(df[['Age','Mileage']])
new_serr
```

Out[53]:
```
array([[2.00e+00, 2.70e+04],
       [4.00e+00, 3.05e+04],
       [1.50e+01, 1.20e+05],
       [1.30e+01, 5.30e+04],
       [2.00e+00, 5.30e+04],
       [8.00e+00, 7.30e+04],
       [8.00e+00, 1.38e+05],
       [2.00e+01, 3.80e+04],
       [1.30e+01, 6.70e+04],
       [7.00e+00, 3.60e+04]])
```

25) Mode or the most frequent is used in the following code to impute missing values:

In [54]:
```
# numerical variables
# strategies: mean, median, most_frequent, constant
import numpy as np
numImputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
numImputer = numImputer.fit(df[['Age','Mileage']])
new_serr=numImputer.transform(df[['Age','Mileage']])
new_serr
```

Out[54]:
```
array([[2.00e+00, 2.70e+04],
       [4.00e+00, 3.05e+04],
       [1.50e+01, 1.20e+05],
       [1.30e+01, 5.30e+04],
       [2.00e+00, 2.70e+04],
       [8.00e+00, 7.30e+04],
       [2.00e+00, 1.38e+05],
       [2.00e+01, 3.80e+04],
       [1.30e+01, 6.70e+04],
       [7.00e+00, 3.60e+04]])
```

26) A designated constant value is used in the following code (imputation using default values):

In [56]:
```
# numerical variables
# strategies: mean, median, most_frequent, constant
import numpy as np
numImputer = SimpleImputer(missing_values=np.nan, strategy='constant',fill_value=-1)
numImputer = numImputer.fit(df[['Age','Mileage']])
new_serr=numImputer.transform(df[['Age','Mileage']])
new_serr
```

Out[56]:
```
array([[ 2.00e+00,  2.70e+04],
       [ 4.00e+00,  3.05e+04],
       [ 1.50e+01,  1.20e+05],
       [ 1.30e+01,  5.30e+04],
       [ 2.00e+00, -1.00e+00],
       [ 8.00e+00,  7.30e+04],
       [-1.00e+00,  1.38e+05],
       [ 2.00e+01,  3.80e+04],
       [ 1.30e+01,  6.70e+04],
       [ 7.00e+00,  3.60e+04]])
```

27) For the categorical variables **the most frequent** and **constant** strategies are applicable:

In [58]:
```
# Categorical variables
# strategies: most_frequent, constant
catImputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
catImputer = catImputer.fit(df[['Fuel','Gearbox','Colour']])
new_serr=catImputer.transform(df[['Fuel','Gearbox','Colour']])
new_serr
```

Out[58]:
```
array([['P', 'A', 'Red'],
       ['P', 'M', 'Black'],
       ['D', 'A', 'White'],
       ['D', 'M', 'White'],
       ['D', 'M', 'Black'],
       ['D', 'M', 'Green'],
       ['P', 'M', 'Black'],
       ['D', 'M', 'Green'],
       ['D', 'A', 'Blue'],
       ['P', 'M', 'Black']], dtype=object)
```

In [59]:
```
# Categorical variables
# strategies: most_frequent, constant
catImputer = SimpleImputer(missing_values=np.nan, strategy='constant', fill_value='Red')
catImputer = catImputer.fit(df[['Fuel','Gearbox','Colour']])
new_serr=catImputerA.transform(df[['Fuel','Gearbox','Colour']])
new_serr
```

Out[59]:
```
array([['P', 'A', 'Red'],
       ['P', 'M', 'Black'],
       ['D', 'A', 'White'],
       ['D', 'M', 'White'],
       ['D', 'M', 'Black'],
       ['Red', 'M', 'Green'],
       ['P', 'M', 'Red'],
       ['D', 'Red', 'Green'],
       ['D', 'A', 'Blue'],
       ['P', 'M', 'Black']], dtype=object)
```

28) To manage all existing values in the dataset we can run the following:

```
In [61]:
# All together
numImputer = SimpleImputer(missing_values=np.nan, strategy='mean')
numImputer = numImputer.fit(df[['Age','Mileage']])
df[['Age','Mileage']]=numImputer.transform(df[['Age','Mileage']])
catImputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
catImputer = catImputer.fit(df[['Fuel','Gearbox','Colour']])
df[['Fuel','Gearbox','Colour']]=catImputer.transform(df[['Fuel','Gearbox','Colour']])
df
```

Out[61]:

	Make	Age	Mileage	Fuel	Gearbox	Colour	Claimed
0	Toyota	2.000000	27000.000000	P	A	Red	Yes
1	Ford	4.000000	30500.000000	P	M	Black	Yes
2	Toyota	15.000000	120000.000000	D	A	White	No
3	Nissan	13.000000	53000.000000	D	M	White	No
4	Nissan	2.000000	64722.222222	D	M	Black	No
5	Ford	8.000000	73000.000000	D	M	Green	No
6	Toyota	9.333333	138000.000000	P	M	Black	Yes
7	Nissan	20.000000	38000.000000	D	M	Green	Yes
8	Toyota	13.000000	67000.000000	D	A	Blue	No
9	Nissan	7.000000	36000.000000	P	M	Black	Yes

Categorical Data

29) Sometimes features are categorical. For example, a person could have features ["male", "female"], ["from Europe", "from US", "from Asia"]. Such features can be efficiently coded as integers, for instance ["male", "from US", "uses Internet Explorer"] could be expressed as [0, 1, 3]. To convert categorical features to integer codes, the **OrdinalEncoder** is an option. This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1):

```
In [65]:
from sklearn import preprocessing
enc = preprocessing.OrdinalEncoder()
enc.fit_transform(df[['Make','Fuel','Gearbox','Colour','Claimed']])
```

```
Out[65]: array([[2., 1., 0., 3., 1.],
                [0., 1., 1., 0., 1.],
                [2., 0., 0., 4., 0.],
                [1., 0., 1., 4., 0.],
                [1., 0., 1., 0., 0.],
                [0., 0., 1., 2., 0.],
                [2., 1., 1., 0., 1.],
                [1., 0., 1., 2., 1.],
                [2., 0., 0., 1., 0.],
                [1., 1., 1., 0., 1.]])
```

30) Another possibility to convert categorical features to features that can be used with scikit-learn library is to use a one-hot or dummy encoding. This type of encoding can be obtained with the **OneHotEncoder**, which transforms each categorical feature with n_categories possible values into n_categories binary features, with one of them 1, and all others 0. It is also possible to encode each column into n_categories - 1

columns instead of n_categories columns by using the drop parameter. This parameter allows the user to specify a category for each feature to be dropped.

```
In [72]:
enc = preprocessing.OneHotEncoder(drop='first')
enc.fit_transform(df[['Make','Fuel','Gearbox','Colour','Claimed']]).toarray()
```

```
Out[72]: array([[0., 1., 1., 0., 0., 0., 1., 0., 1.],
                [0., 0., 1., 1., 0., 0., 0., 0., 1.],
                [0., 1., 0., 0., 0., 0., 0., 1., 0.],
                [1., 0., 0., 1., 0., 0., 0., 1., 0.],
                [1., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0., 1., 0., 0., 0.],
                [0., 1., 1., 1., 0., 0., 0., 0., 1.],
                [1., 0., 0., 1., 0., 1., 0., 0., 1.],
                [0., 1., 0., 0., 1., 0., 0., 0., 0.],
                [1., 0., 1., 1., 0., 0., 0., 0., 1.]])
```

```
In [73]:
enc.categories_
```

```
Out[73]: [array(['Ford', 'Nissan', 'Toyota'], dtype=object),
          array(['D', 'P'], dtype=object),
          array(['A', 'M'], dtype=object),
          array(['Black', 'Blue', 'Green', 'Red', 'White'], dtype=object),
          array(['No', 'Yes'], dtype=object)]
```

31) To convert all variables in the dataset we can run the following code:

```
In [28]:
# All together

# target variable, it just uses the ordinal encoder
enc = preprocessing.OrdinalEncoder()
df['Claimed']=enc.fit_transform(df['Claimed'].values.reshape(-1,1))

# input variables one hot encoding
enc = preprocessing.OneHotEncoder(drop='first')
onehots = enc.fit_transform(df[['Make','Fuel','Gearbox','Colour']]).toarray()
# creating the new df
cols = []
for i in enc.categories_:
    i = np.delete(i,0)
    cols.extend(i)
df = df.join(pd.DataFrame(onehots, columns=cols))

df = df.drop(['Make','Fuel','Gearbox','Colour'], axis=1)
df
```

Out[28]:

	Age	Mileage	Claimed	Nissan	Toyota	P	M	Blue	Green	Red	White
0	2.000000	27000.000000	1.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
1	4.000000	30500.000000	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
2	15.000000	120000.000000	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0
3	13.000000	53000.000000	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
4	2.000000	64722.222222	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
5	8.000000	73000.000000	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
6	9.333333	138000.000000	1.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0
7	20.000000	38000.000000	1.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
8	13.000000	67000.000000	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
9	7.000000	36000.000000	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0

Standardization

Now we have a dataset without any null value containing numerical variables. Yet the variables are not in the same interval which can cause many problems in machine learning algorithms.

32) The **preprocessing** module provides the **StandardScaler** utility class, which is a quick and easy way to perform the following operation on an array-like dataset and make them normalized.

In [174]:
```
from sklearn import preprocessing

X = df.iloc[:,:-1].values
Y = df.iloc[:,-1].values
```

In [177]:
```
scaler = preprocessing.StandardScaler().fit(X)
```

In [178]:
```
scaler.mean_
```

Out[178]:
```
array([9.33333333e+00, 6.47222222e+04, 5.00000000e-01, 4.00000000e-01,
       4.00000000e-01, 4.00000000e-01, 7.00000000e-01, 1.00000000e-01,
       2.00000000e-01, 1.00000000e-01])
```

In [179]:
```
scaler.scale_
```

Out[179]:
```
array([5.62138773e+00, 3.57345149e+04, 5.00000000e-01, 4.89897949e-01,
       4.89897949e-01, 4.89897949e-01, 4.58257569e-01, 3.00000000e-01,
       4.00000000e-01, 3.00000000e-01])
```

In [180]:
```
X_scaled = scaler.transform(X)
```

```
In [182]: 1 X_scaled
```

```
Out[182]: array([[-1.30454146e+00, -1.05562430e+00,  1.00000000e+00,
                  -8.16496581e-01,  1.22474487e+00,  1.22474487e+00,
                  -1.52752523e+00, -3.33333333e-01, -5.00000000e-01,
                   3.00000000e+00],
                 [-9.48757423e-01, -9.57679776e-01,  1.00000000e+00,
                  -8.16496581e-01, -8.16496581e-01,  1.22474487e+00,
                   6.54653671e-01, -3.33333333e-01, -5.00000000e-01,
                  -3.33333333e-01],
                 [ 1.00805476e+00,  1.54690159e+00, -1.00000000e+00,
                  -8.16496581e-01,  1.22474487e+00, -8.16496581e-01,
                  -1.52752523e+00, -3.33333333e-01, -5.00000000e-01,
                  -3.33333333e-01],
                 [ 6.52270728e-01, -3.28036417e-01, -1.00000000e+00,
                   1.22474487e+00, -8.16496581e-01, -8.16496581e-01,
                   6.54653671e-01, -3.33333333e-01, -5.00000000e-01,
                  -3.33333333e-01],
                 [-1.30454146e+00, -2.03611484e-16, -1.00000000e+00,
                   1.22474487e+00, -8.16496581e-01, -8.16496581e-01,
                   6.54653671e-01, -3.33333333e-01, -5.00000000e-01,
                  -3.33333333e-01],
                 [-2.37189356e-01,  2.31646569e-01, -1.00000000e+00,
                  -8.16496581e-01, -8.16496581e-01, -8.16496581e-01,
                   6.54653671e-01, -3.33333333e-01,  2.00000000e+00,
                  -3.33333333e-01],
                 [ 0.00000000e+00,  2.05061627e+00,  1.00000000e+00,
                  -8.16496581e-01,  1.22474487e+00,  1.22474487e+00,
                   6.54653671e-01, -3.33333333e-01, -5.00000000e-01,
                  -3.33333333e-01],
                 [ 1.89751485e+00, -7.47798656e-01,  1.00000000e+00,
                   1.22474487e+00, -8.16496581e-01, -8.16496581e-01,
                   6.54653671e-01, -3.33333333e-01,  2.00000000e+00,
                  -3.33333333e-01],
```

33) An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using **MinMaxScaler**.

```
In [183]: 1 min_max_scaler = preprocessing.MinMaxScaler()
          2 X_minmax = min_max_scaler.fit_transform(X)
          3 X_minmax
```

```
Out[183]: array([[0.        , 0.        , 1.        , 0.        , 1.        ,
                  1.        , 0.        , 0.        , 0.        , 1.        ],
                 [0.11111111, 0.03153153, 1.        , 0.        , 0.        ,
                  1.        , 1.        , 0.        , 0.        , 0.        ],
                 [0.72222222, 0.83783784, 0.        , 0.        , 1.        ,
                  0.        , 0.        , 0.        , 0.        , 0.        ],
                 [0.61111111, 0.23423423, 0.        , 1.        , 0.        ,
                  0.        , 1.        , 0.        , 0.        , 0.        ],
                 [0.        , 0.33983984, 0.        , 1.        , 0.        ,
                  0.        , 1.        , 0.        , 0.        , 0.        ],
                 [0.33333333, 0.41441441, 0.        , 0.        , 0.        ,
                  0.        , 1.        , 0.        , 1.        , 0.        ],
                 [0.40740741, 1.        , 1.        , 0.        , 1.        ,
                  1.        , 1.        , 0.        , 0.        , 0.        ],
                 [1.        , 0.0990991 , 1.        , 1.        , 0.        ,
                  0.        , 1.        , 0.        , 1.        , 0.        ],
                 [0.61111111, 0.36036036, 0.        , 0.        , 1.        ,
                  0.        , 0.        , 1.        , 0.        , 0.        ],
                 [0.27777778, 0.08108108, 1.        , 1.        , 0.        ,
                  1.        , 1.        , 0.        , 0.        , 0.        ]])
```

Now we have a cleaned dataset to some extent.

34) Save your notebook, shut down the notebook and close the tab.