In [1]:
```
import pandas as pd
```

Feature Engineering

Table of contents:

In [2]:
```
# load titanic dataset
url = 'https://raw.githubusercontent.com/um-perez-alvaro/Data-Science-Practic
e/master/Data/titanic.csv'
titanic = pd.read_csv(url ,index_col = 'PassengerId')
titanic.head()
```

Out[2]:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabi
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	Na
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C8
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	Na
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C12
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	Na

Dataframe Columns:

- survived: 0 = No; 1 = Yes
- Pclass: Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
- name: Name
- sex: Sex
- age: Age
- sibsp: Number of Siblings/Spouses Aboard
- parch: Number of Parents/Children Aboard
- ticket: - Ticket Number
- fare: Passenger Fare
- cabin: Cabin
- embarked: Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

Categorical features in the Titanic dataset: sex, ticket, cabin, embarked.

Numerical features in the Titanic dataset: Pclass, Age, SibSp, Parch, Fare

In [3]:
```
# missing values
titanic.isnull().sum()
```

Out[3]:
```
Survived      0
Pclass        0
Name          0
Sex           0
Age         177
SibSp         0
Parch         0
Ticket        0
Fare          0
Cabin       687
Embarked      2
dtype: int64
```

1. Enconding categorical features

- [Ordinal encoding]
- [One hot encoding]

Often features are not given as continuous values but categorical. For example a person could have features ["male", "female"], ["from Europe", "from US", "from Asia"], ["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]. Such features can be efficiently coded as integers. To convert categorical features to such integer codes, we can use the OrdinalEncoder.

1.1. Ordinal encoding

In [4]:
```
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder()
```

In [5]:
```
# ordingal encoding of the "Sex" feature
titanic.Sex.unique()
```

Out[5]:
```
array(['male', 'female'], dtype=object)
```

In [6]:
```
encoder.fit(titanic[['Sex']])
encoder.transform(titanic[['Sex']])[:10]
```

Out[6]:
```
array([[1.],
       [0.],
       [0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [0.],
       [0.]])
```

In [7]:
```
encoder.categories_
```

Out[7]:
```
[array(['female', 'male'], dtype=object)]
```

In [8]:
```
# ordinal encoding of the "Embarked" feature
titanic.Embarked.unique()
```

Out[8]:
```
array(['S', 'C', 'Q', nan], dtype=object)
```

In [10]:
```
# OrdinarEncoder does not work where there are missing values;
# for this example, we'll drop the 2 missing values in the "Embarked" column
titanic.dropna(subset=['Embarked'], how='any', axis=0, inplace=True)

encoder.fit(titanic[['Embarked']])
encoder.transform(titanic[['Embarked']])[:10]
```

Out[10]:
```
array([[2.],
       [0.],
       [2.],
       [2.],
       [2.],
       [1.],
       [2.],
       [2.],
       [2.],
       [0.]])
```

In [11]:
```
encoder.categories_
```

Out[11]:
```
[array(['C', 'Q', 'S'], dtype=object)]
```

In [11]:
```
# # ordinal encoding of the "Sex" and "Embarked" features
encoder.fit(titanic[['Sex','Embarked']])
encoder.transform(titanic[['Sex','Embarked']])
```

Out[11]:
```
array([[1., 2.],
       [0., 0.],
       [0., 2.],
       ...,
       [0., 2.],
       [1., 0.],
       [1., 1.]])
```

In [12]:
```
encoder.categories_
```

Out[12]:
```
[array(['female', 'male'], dtype=object), array(['C', 'Q', 'S'], dtype=objec
t)]
```

Such integer representation can, however, not be used directly with all scikit-learn models, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired

1.2. One hot encoding

Another possibility to convert categorical features to features that can be used with scikit-learn models is to use a one-of-K, also known as one-hot 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.

In [13]:
```
# one hot encoding of the "sex" feature
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False) # initializer the one hot enconder
```

In [14]:
```
titanic[['Sex']].head()
```

Out[14]:

	Sex
PassengerId	
1	male
2	female
3	female
4	female
5	male

In [15]:
```
encoder.fit(titanic[['Sex']])
encoder.transform(titanic[['Sex']])
```

Out[15]:
```
array([[0., 1.],
       [1., 0.],
       [1., 0.],
       ...,
       [1., 0.],
       [0., 1.],
       [0., 1.]])
```

In [16]:
```
encoder.categories_
```

Out[16]:
```
[array(['female', 'male'], dtype=object)]
```

In [17]:
```
# one hot encoding of the "Embarked" feature
titanic.Embarked.head(10)
```

Out[17]:
```
PassengerId
1     S
2     C
3     S
4     S
5     S
6     Q
7     S
8     S
9     S
10    C
Name: Embarked, dtype: object
```

In [18]:
```
encoder.fit(titanic[['Embarked']])
encoder.transform(titanic[['Embarked']])
```

Out[18]:
```
array([[0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       ...,
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

In [19]:
```
encoder.categories_
```

Out[19]:
```
[array(['C', 'Q', 'S'], dtype=object)]
```

In [20]:
```
# one hot encoding of the "Sex" and "Embarked" features
encoder.fit_transform(titanic[['Sex','Embarked']])
```

Out[20]:
```
array([[0., 1., 0., 0., 1.],
       [1., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1.],
       ...,
       [1., 0., 0., 0., 1.],
       [0., 1., 1., 0., 0.],
       [0., 1., 0., 1., 0.]])
```

In [21]:
```
encoder.categories_
```

Out[21]:
```
[array(['female', 'male'], dtype=object), array(['C', 'Q', 'S'], dtype=objec
t)]
```

2. Normalization and Standardization

- Normalization
- Standardization

In [22]:
```
X = titanic[['Pclass','Age','Fare']] # feature matrix
X
```

Out[22]:

PassengerId	Pclass	Age	Fare
1	3	22.0	7.2500
2	1	38.0	71.2833
3	3	26.0	7.9250
4	1	35.0	53.1000
5	3	35.0	8.0500
...	...	...	...
887	2	27.0	13.0000
888	1	19.0	30.0000
889	3	NaN	23.4500
890	1	26.0	30.0000
891	3	32.0	7.7500

889 rows × 3 columns

2.1. Normalization

Normalization is the process of converting an actual range of values which a numerical feature can take, into a standard range of values, typically in the interval [-1,1] or [0,1]. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively.

Normalizing the data is not a strict requirement. However, in practice, it can lead to an increased speed of training.

In [23]:
```
# minmaxscaler scales data to the [0, 1] range
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
scaler.transform(X)
```

Out[23]:
```
array([[1.        , 0.27117366, 0.01415106],
       [0.        , 0.4722292 , 0.13913574],
       [1.        , 0.32143755, 0.01546857],
       ...,
       [1.        ,        nan, 0.04577135],
       [0.        , 0.32143755, 0.0585561 ],
       [1.        , 0.39683338, 0.01512699]])
```

In [24]:
```
# maxabsscaler scales data to the [-1, 1] range
from sklearn.preprocessing import MaxAbsScaler
scaler = MaxAbsScaler()
scaler.fit(X)
scaler.transform(X)
```

Out[24]:
```
array([[1.        , 0.275     , 0.01415106],
       [0.33333333, 0.475     , 0.13913574],
       [1.        , 0.325     , 0.01546857],
       ...,
       [1.        ,        nan, 0.04577135],
       [0.33333333, 0.325     , 0.0585561 ],
       [1.        , 0.4       , 0.01512699]])
```

2.2. Standardization

Standardization (or mean removal and variance scaling) is the procedure during which the feature values are rescaled so that they have the properties of a standard normal distribution with mean 0 and standard deviation 1.

In [25]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X)
scaler.transform(X)
```

Out[25]:
```
array([[ 0.82520863, -0.52766856, -0.50023975],
       [-1.57221121,  0.57709388,  0.78894661],
       [ 0.82520863, -0.25147795, -0.48664993],
       ...,
       [ 0.82520863,         nan, -0.17408416],
       [-1.57221121, -0.25147795, -0.0422126 ],
       [ 0.82520863,  0.16280796, -0.49017322]])
```

3. Data Imputation

In [26]:
```
X.isnull().sum()
```

Out[26]:
```
Pclass      0
Age       177
Fare        0
dtype: int64
```

The typical approaches of dealing with missing values for a feature include:

- remove rows with missing features from the dataset (this can be done if your dataset is big enough)
- using a **data imputation** technique

The SimpleImputer class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located.

In [27]:
```
from sklearn.impute import SimpleImputer
```

In [28]:
```
titanic.Age.mean()
```

Out[28]:
```
29.64209269662921
```

In [29]:
```
imputer = SimpleImputer(strategy='mean')
imputer.fit(X)
imputed_X = imputer.transform(X)
imputed_X
```

Out[29]:
```
array([[ 3.        , 22.        ,  7.25      ],
       [ 1.        , 38.        , 71.2833    ],
       [ 3.        , 26.        ,  7.925     ],
       ...,
       [ 3.        , 29.6420927, 23.45      ],
       [ 1.        , 26.        , 30.       ],
       [ 3.        , 32.        ,  7.75      ]])
```

4. Polynomial features

Often it's useful to add complexity to the model by considering nonlinear features of the input data. A simple and common method to use is **polynomial features**, which can get features' high-order and interaction terms. It is implemented in PolynomialFeatures

In [30]:
```
from sklearn.preprocessing import PolynomialFeatures
```

In [31]:
```
poly = PolynomialFeatures(degree=2)
poly.fit(imputed_X)
poly.transform(imputed_X)
```

Out[31]:
```
array([[1.00000000e+00, 3.00000000e+00, 2.20000000e+01, ...,
        4.84000000e+02, 1.59500000e+02, 5.25625000e+01],
       [1.00000000e+00, 1.00000000e+00, 3.80000000e+01, ...,
        1.44400000e+03, 2.70876540e+03, 5.08130886e+03],
       [1.00000000e+00, 3.00000000e+00, 2.60000000e+01, ...,
        6.76000000e+02, 2.06050000e+02, 6.28056250e+01],
       ...,
       [1.00000000e+00, 3.00000000e+00, 2.96420927e+01, ...,
        8.78653659e+02, 6.95107074e+02, 5.49902500e+02],
       [1.00000000e+00, 1.00000000e+00, 2.60000000e+01, ...,
        6.76000000e+02, 7.80000000e+02, 9.00000000e+02],
       [1.00000000e+00, 3.00000000e+00, 3.20000000e+01, ...,
        1.02400000e+03, 2.48000000e+02, 6.00625000e+01]])
```

In [32]:
```
poly.get_feature_names(X.columns)
```

Out[32]:
```
['1',
 'Pclass',
 'Age',
 'Fare',
 'Pclass^2',
 'Pclass Age',
 'Pclass Fare',
 'Age^2',
 'Age Fare',
 'Fare^2']
```