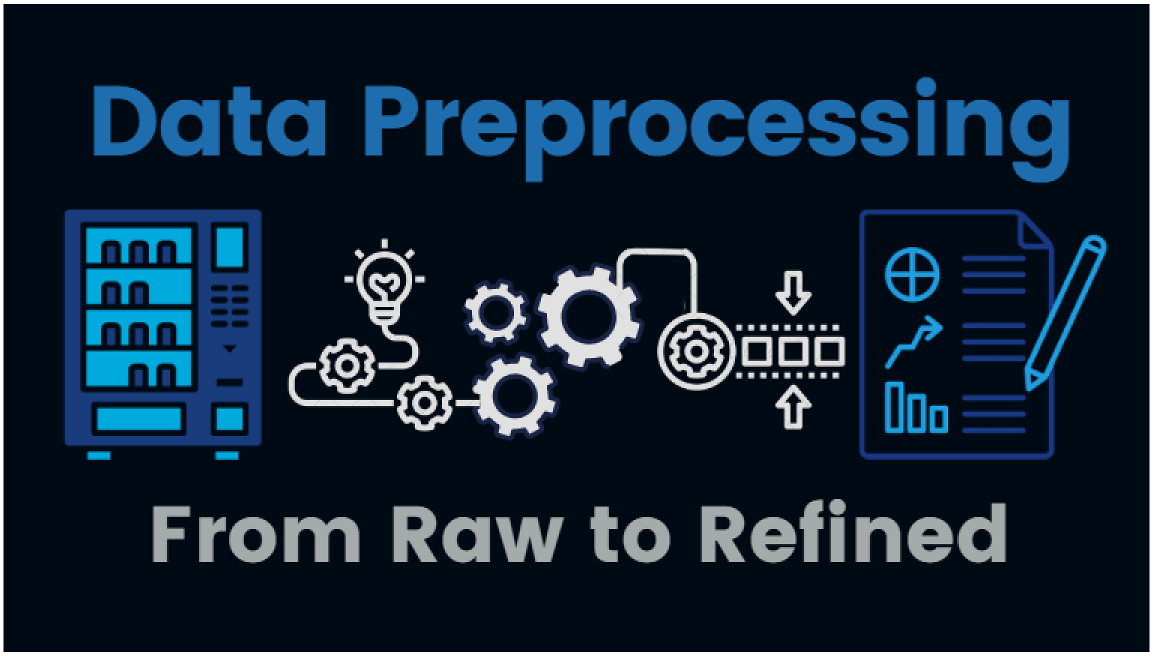
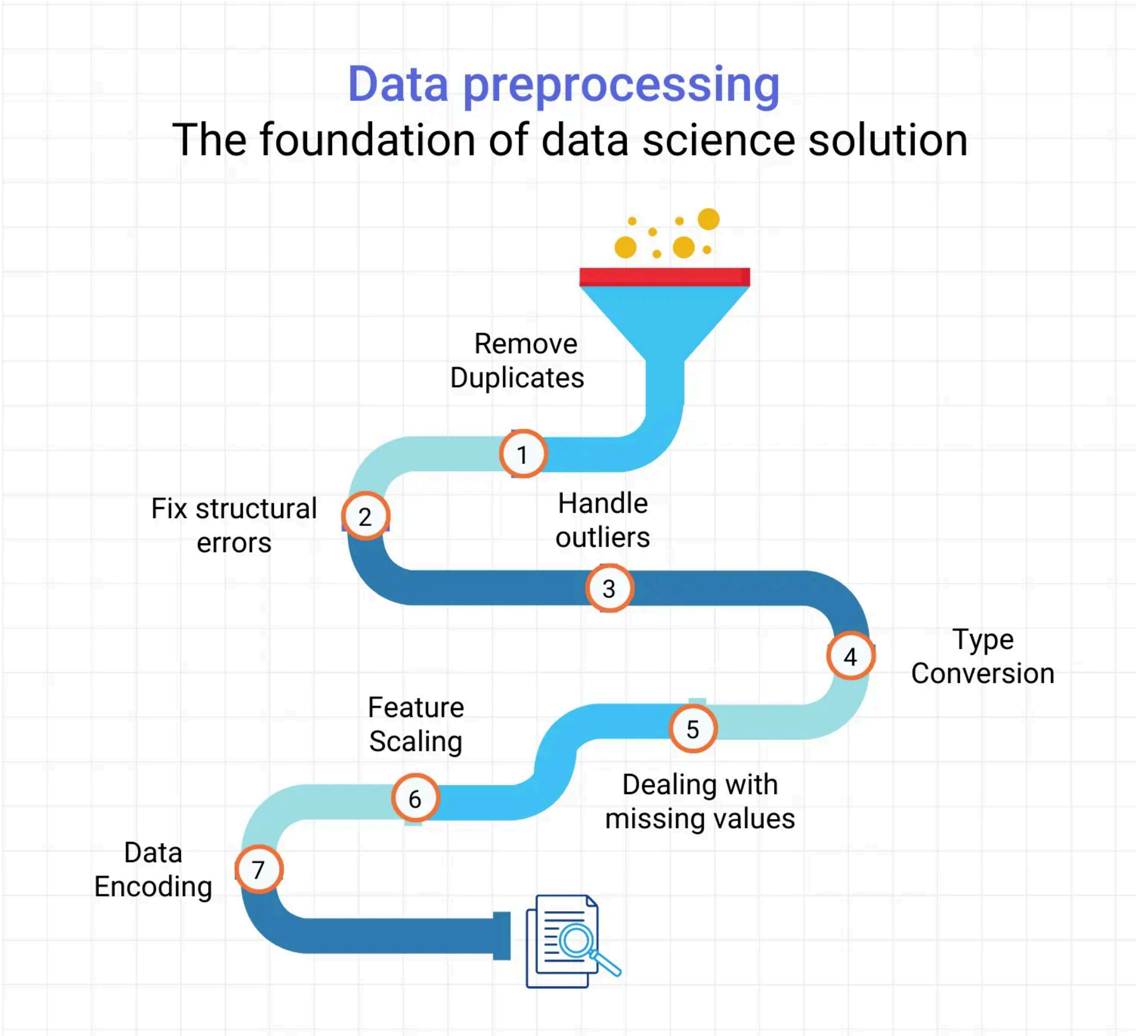
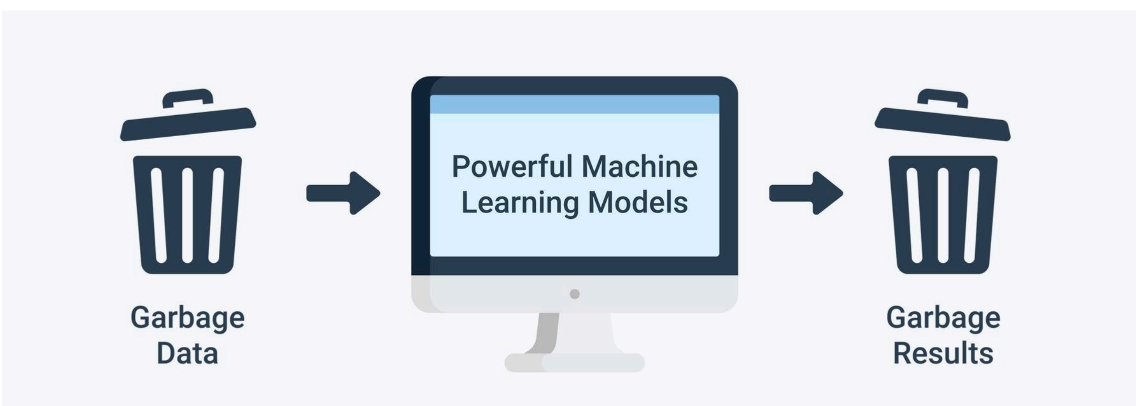


Scikit-learn is the most useful and robust library for machine learning in Python.

- It provides a selection of efficient tools for machine learning and statistical modeling including:
  - classification,
  - regression,
  - clustering
  - dimensionality reduction
  - **Preprocessing**
- This library is built upon NumPy, SciPy and Matplotlib.





Data preprocessing steps:

1. Remove not relevant or constant features such as ID, Mobile Number,...
2. Verify data types
3. Remove duplicates
4. Define feature matrix and Target vector
5. Split the data to train and test
6. Handle missing values

7. Encode categorical data
8. Feature scaling

In [17]:
```python
# def family_size(children):
#     if children > 5:
#         return "Large"
#     elif children > 2:
#         return 'Mid'
#     else:
#         return "Small"

# df['familySize'] = df['children'].apply(lambda x: family_size(x))
```

In [18]:
```python
import pandas as pd
df = pd.read_csv("galton_handson.csv")

df
```

Out[18]:

	family	father	mother	midparentHeight	children	childNum	gender	childHeight
0	1	78.5	67.0	75.43	4	1.0	male	73.2
1	1	78.5	67.0	75.43	4	2.0	female	69.2
2	1	78.5	67.0	75.43	4	3.0	female	69.0
3	1	78.5	67.0	75.43	4	4.0	female	69.0
4	2	75.5	66.5	73.66	4	1.0	male	73.5
...	...	...	...	...	...	...	...	...
929	203	62.0	66.0	66.64	3	1.0	male	64.0
930	203	62.0	66.0	66.64	3	2.0	female	62.0
931	203	62.0	66.0	66.64	3	3.0	female	61.0
932	204	62.5	63.0	65.27	2	1.0	male	66.5
933	204	62.5	63.0	65.27	2	2.0	female	57.0

934 rows × 9 columns

Drop not informative features

In [19]:
```python
df = df.drop(columns='family')
df
```

Out[19]:

	father	mother	midparentHeight	children	childNum	gender	childHeight	familySiz
0	78.5	67.0	75.43	4	1.0	male	73.2	Mi
1	78.5	67.0	75.43	4	2.0	female	69.2	Mi
2	78.5	67.0	75.43	4	3.0	female	69.0	Mi
3	78.5	67.0	75.43	4	4.0	female	69.0	Mi
4	75.5	66.5	73.66	4	1.0	male	73.5	Mi
...	...	...	...	...	...	...	...	
929	62.0	66.0	66.64	3	1.0	male	64.0	Mi
930	62.0	66.0	66.64	3	2.0	female	62.0	Mi
931	62.0	66.0	66.64	3	3.0	female	61.0	Mi
932	62.5	63.0	65.27	2	1.0	male	66.5	Sma
933	62.5	63.0	65.27	2	2.0	female	57.0	Sma

934 rows × 8 columns

Check data types

In [20]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 934 entries, 0 to 933
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   father           912 non-null    float64
 1   mother           899 non-null    float64
 2   midparentHeight  927 non-null    object 
 3   children         934 non-null    int64  
 4   childNum         929 non-null    float64
 5   gender           915 non-null    object 
 6   childHeight      934 non-null    float64
 7   familySize       934 non-null    object 
dtypes: float64(4), int64(1), object(3)
memory usage: 58.5+ KB
```

In [21]:

```
df['midparentHeight'] = df['midparentHeight'].str.replace(",", '.')
df['midparentHeight'] =  df['midparentHeight'].astype(float)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 934 entries, 0 to 933
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   father          912 non-null    float64
 1   mother          899 non-null    float64
 2   midparentHeight 927 non-null    float64
 3   children        934 non-null    int64  
 4   childNum        929 non-null    float64
 5   gender          915 non-null    object 
 6   childHeight     934 non-null    float64
 7   familySize      934 non-null    object 
dtypes: float64(5), int64(1), object(2)
memory usage: 58.5+ KB
```

These steps sequence is a workaround to convert a column with NaN values into integers — which normally isn't allowed directly.

In [22]:
```python
import numpy as np

df['childNum'] = df['childNum'].fillna(-1)
df['childNum'] = df['childNum'].astype(int)
df['childNum'] = df['childNum'].replace(-1, np.nan)
df.info()

#If you're using pandas >= 1.0, a cleaner way is:
# df['childNum'] = df['childNum'].astype('Int64')  # Nullable integer
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 934 entries, 0 to 933
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   father          912 non-null    float64
 1   mother          899 non-null    float64
 2   midparentHeight 927 non-null    float64
 3   children        934 non-null    int64  
 4   childNum        929 non-null    float64
 5   gender          915 non-null    object 
 6   childHeight     934 non-null    float64
 7   familySize      934 non-null    object 
dtypes: float64(5), int64(1), object(2)
memory usage: 58.5+ KB
```

Check duplicates

In [23]:
```python
df.duplicated().sum()
```

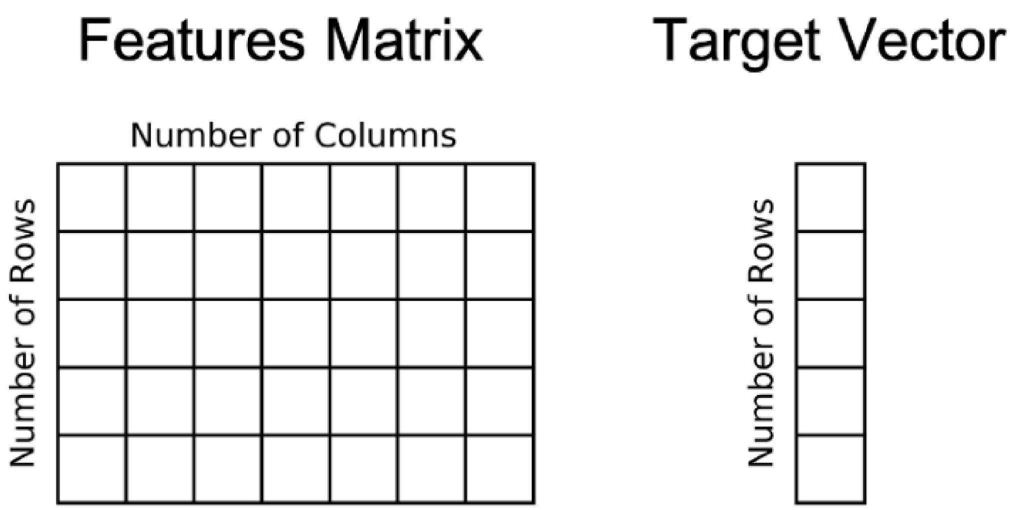
Out[23]:
```
np.int64(1)
```

In [24]:
```python
df = df.drop_duplicates()
df.duplicated().sum()
```

Out[24]: np.int64(0)

Create feature matrix and target vector

Feature matrix and Target vector



- Target y
- Features X

In [25]:
```
# The target we are trying to predict
y = df['childHeight']

# The features we will use to make the prediction
X = df.drop(columns = 'childHeight')
X
```

Out[25]:

	father	mother	midparentHeight	children	childNum	gender	familySize
0	78.5	67.0	75.43	4	1.0	male	Mid
1	78.5	67.0	75.43	4	2.0	female	Mid
2	78.5	67.0	75.43	4	3.0	female	Mid
3	78.5	67.0	75.43	4	4.0	female	Mid
4	75.5	66.5	73.66	4	1.0	male	Mid
...	...	...	...	...	...	...	...
929	62.0	66.0	66.64	3	1.0	male	Mid
930	62.0	66.0	66.64	3	2.0	female	Mid
931	62.0	66.0	66.64	3	3.0	female	Mid
932	62.5	63.0	65.27	2	1.0	male	Small
933	62.5	63.0	65.27	2	2.0	female	Small

933 rows × 7 columns

Split data to train and test

In [26]:

```
#pip install scikit-learn
```

In [27]:

```
# Import the TTS from sklearn
from sklearn.model_selection import train_test_split

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1020)

print(f"X.shape = {X.shape}")
print(f"X_train.shape = {X_train.shape}")
print(f"X_test.shape = {X_test.shape}")
print(f"y.shape = {y.shape}")
print(f"y_train.shape = {y_train.shape}")
print(f"y_test.shape = {y_test.shape}")

y_test
```

```
X.shape = (933, 7)
X_train.shape = (699, 7)
X_test.shape = (234, 7)
y.shape = (933,)
y_train.shape = (699,)
y_test.shape = (234,)
```

Out[27]:
```
226    70.5
271    61.0
11     68.5
353    63.2
704    66.5
       ...
236    71.0
300    72.5
72     69.0
87     65.0
631    71.2
Name: childHeight, Length: 234, dtype: float64
```

Rule: Any change to the data should be **justified**

Handle missing values

In [28]:
```
df.isna().sum()
```

Out[28]:
```
father             22
mother             35
midparentHeight     7
children            0
childNum            5
gender             19
childHeight         0
familySize          0
dtype: int64
```

For the sake of illustration, assume we fill the missing values as following:

- Categorical fetures, replace missing with 'NA'
- Numeric features, replace by median

In [29]:
```
# Define list of categorical features
cat_cols = X_train.select_dtypes("object").columns

# Define the list of numerical features
num_cols = X_train.select_dtypes("number").columns

print(cat_cols)
num_cols
```

```
Index(['gender', 'familySize'], dtype='object')
```

Out[29]:
```
Index(['father', 'mother', 'midparentHeight', 'children', 'childNum'], dtype='obje
ct')
```

In [30]:
```
from sklearn.impute import SimpleImputer

# Instantiate the imputer with the desired strategy
impute_na = SimpleImputer(strategy='constant', fill_value='NA')

# Instantiate the imputer object from the SimpleImputer class with strategy 'median
impute_median = SimpleImputer(strategy='median')
```

In [31]:
```
# Fit the imputer object on the training data with .fit
impute_na.fit(X_train[cat_cols])

# Fit the imputer object on the numeric training data with .fit()
impute_median.fit(X_train[num_cols])
```

Out[31]:
```
SimpleImputer

SimpleImputer(strategy='median')
```

In [32]:
```
from sklearn import set_config
set_config(transform_output='pandas') #To generate dataframes instead of numpy arra

# Transform the categorical training data
X_train_cat_imputed = impute_na.transform(X_train[cat_cols])
# Transform the categorical testing data
X_test_cat_imputed = impute_na.transform(X_test[cat_cols])
```

```
# Transform the training data
X_train_num_imputed = impute_median.transform(X_train[num_cols])
# Transfrom the testing data
X_test_num_imputed = impute_median.transform(X_test[num_cols])
X_test_cat_imputed
```

Out[32]:

	gender	familySize
226	male	Large
271	female	Mid
11	male	Mid
353	female	Large
704	male	Mid
...	...	...
236	male	Mid
300	male	Mid
72	female	Large
87	female	Large
631	male	Mid

234 rows × 2 columns

Data encoding

In order for features to be interpreted by a machine learning algorithm, the data must be in a numeric form (integers or floats).

1. Ordinal features
2. Nominal features

Ordinal encoding

Used for converting categorical data into numeric values that preserve their inherent ordering

In [33]:

```
from sklearn.preprocessing import OrdinalEncoder
# define a list of columns to encode as ordinal
ordinal_cols = ['familySize']

print(X_test_cat_imputed['familySize'].value_counts())

# Specifying the order of categories in quality/condition columns
```

```python
fam_size_order = ["Small", "Mid", "Large"]

# Making the list of order lists for OrdinalEncoder
ordinal_category_orders = [fam_size_order]

# Instantiate the encoder and include the list of ordered values as an argument
ord_encoder = OrdinalEncoder(categories=ordinal_category_orders)

# Fit the encoder on the training data
ord_encoder.fit(X_train_cat_imputed[ordinal_cols])

# Transform the training data
X_train_ordinal_enc = ord_encoder.transform(X_train_cat_imputed[ordinal_cols])
# Transform the test data
X_test_ordinal_enc = ord_encoder.transform(X_test_cat_imputed[ordinal_cols])

# # Value counts after transformation
X_test_ordinal_enc['familySize'].value_counts()
```

```
familySize
Large    129
Mid       85
Small     20
Name: count, dtype: int64
```

Out[33]:
```
familySize
2.0    129
1.0     85
0.0     20
Name: count, dtype: int64
```

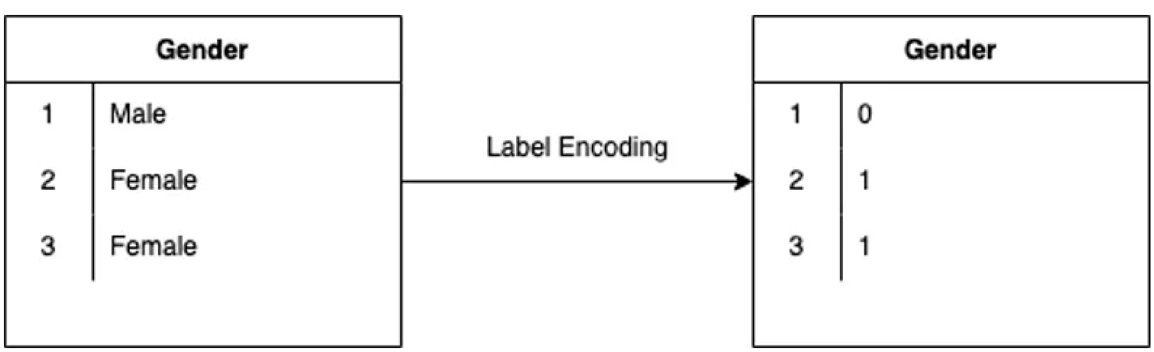
Nominal features

In [34]:
```python
df['gender'].value_counts()
```

Out[34]:
```
gender
male      467
female    447
Name: count, dtype: int64
```

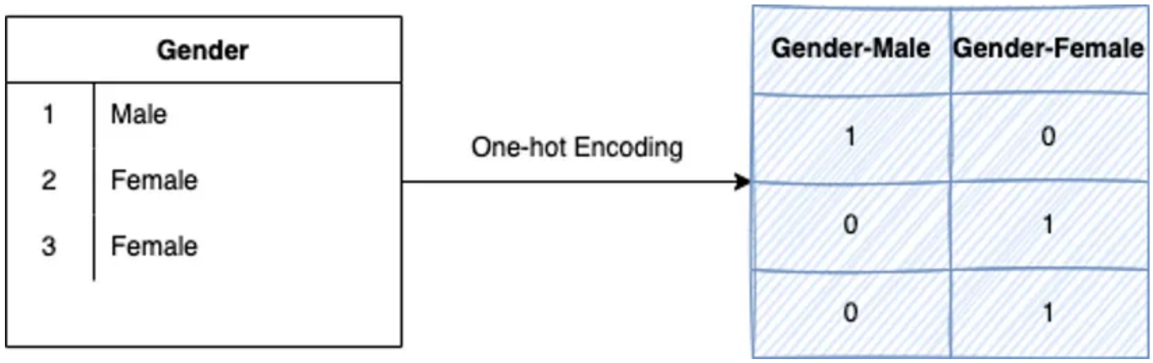
How can represent these values in numbers ?

Can we replace every category with numeric value such as "Male" -> 0, "Female" -> 1?



NO! The ML algorithms might interpret Female labelled data to be having higher weight than others since 1 > 0

One-hot encoding: It creates a binary column for each class in the column.



In [35]:
```python
from sklearn.preprocessing import OneHotEncoder

# saving list of categorical features to one-hot-encode
ohe_cols = X_train_cat_imputed.drop(columns=ordinal_cols).columns
print(ohe_cols)

# Instantiate one hot encoder
''' handle_unknown='ignore': prevents errors on unseen categories
    sparse_output=False: returns a dense NumPy array or DataFrame '''
ohe_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
print(ohe_encoder)
```

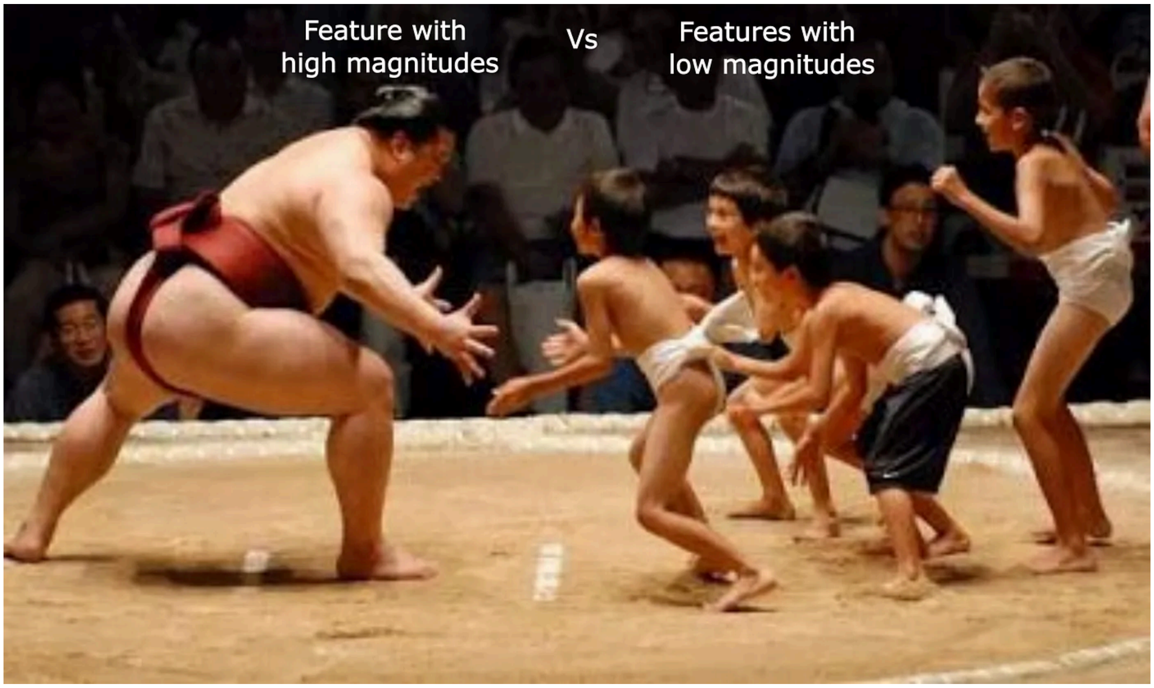
```
Index(['gender'], dtype='object')
OneHotEncoder(handle_unknown='ignore', sparse_output=False)
```

In [300…
```python
# Fit the OneHotEncoder on the training data
ohe_encoder.fit(X_train_cat_imputed[ohe_cols])

# Transform the training data
X_train_cat_ohe = ohe_encoder.transform(X_train_cat_imputed[ohe_cols])

# Transform the testing data

X_test_cat_ohe = ohe_encoder.transform(X_test_cat_imputed[ohe_cols])
X_test_cat_ohe.head(5)
```

Out[300…

	gender_NA	gender_female	gender_male
226	0.0	0.0	1.0
271	0.0	1.0	0.0
11	0.0	0.0	1.0
353	0.0	1.0	0.0
704	0.0	0.0	1.0

Feature Scaling/Normalization



Feature scaling is the process of making sure that all the values in a dataset are within a certain range.

Why scaling ?

- Many machine learning algorithms require scaled or normalized data in order to work properly.
- Scaled and normalized data is often easier to work with
- Scaled and normalized data can be helpful when comparing different datasets
- Reduce the impact of outliers
- Reduce the complexity of the mathematics performed by our models to speed up our models.

Scaling methods:

1. **MinMax scaling**: Values are shifted and rescaled so that they end up ranging between 0 and 1.

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

2. **Standardization (z-score)**

- Scaling the values so that the distribution has a standard deviation **sd** of **1** with a **mean** of **0**.
- It outputs something very close to a normal distribution.
- **Z-score** is the number of standard deviations above and below the mean that the value falls. For example, a Z-score of 2 indicates that an observation is two standard deviations above the average while a Z-score of -2 signifies it is two standard deviations below the mean.
- **Z-score = (feature - mean_of_feature) / std_dev_of_feature**

3. . | .

- | -

$$Z = \frac{X - \mu}{\sigma}$$ where:

- X are the values in the feature being scaled.
- μ is the mean of the feature.
- σ is the standard deviation of the feature.

|

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

σ = population standard deviation

N = the size of the population

x_i = each value from the population

μ = the population mean

Quiz:

The following features need to be scaled (Yes, No):

- Ordinal features

- Nominal features
- Target feature

Scaling in python

In [308…

```python
# Obtain summary statistics for training data before scaling
X_train_num_imputed.describe().round(2)
```

Out[308…

	father	mother	midparentHeight	children	childNum
count	699.00	699.00	699.00	699.00	699.00
mean	69.26	64.11	69.18	6.21	3.62
std	2.49	2.30	1.79	2.77	2.40
min	62.00	58.00	64.40	1.00	1.00
25%	68.00	63.00	68.14	4.00	2.00
50%	69.00	64.00	69.18	6.00	3.00
75%	71.00	66.00	70.14	8.00	5.00
max	78.50	70.50	75.43	15.00	15.00

In [309…

```python
# New import for scaler
from sklearn.preprocessing import StandardScaler

# instantiate scaler
scaler = StandardScaler()

# fit scaler on training data only
scaler.fit(X_train_num_imputed)

# transform training data
X_train_num_scaled = scaler.transform(X_train_num_imputed)
# transform testing data
X_test_num_scaled = scaler.transform(X_test_num_imputed)

# Obtain summary statistics for training data
X_train_num_scaled
```

Out[309…

	father	mother	midparentHeight	children	childNum
728	-0.908051	0.818909	-0.019875	0.645798	0.991754
699	-0.505815	-1.786145	-1.549751	0.284379	0.574895
462	0.097538	0.818909	0.678061	0.284379	-0.675681
40	1.907597	-0.917794	0.728312	0.645798	1.825471
510	-0.103580	-0.266530	-0.215297	0.284379	0.574895
...	...	...	...	...	...
229	0.700891	-0.917794	-0.109211	0.284379	0.158036
397	0.298655	-1.351970	-0.689894	-1.522718	-0.675681
857	-1.310286	-2.220321	-2.409608	3.175734	4.326622
167	0.700891	-0.049443	0.493806	0.645798	-1.092539
831	-1.109168	-0.700706	-1.214742	0.284379	-1.092539

699 rows × 5 columns

Scikit-learn Pipelines

What is a Pipeline in Machine Learning?

- A pipeline contains multiple transformers (or even models!) and performs operations on data IN SEQUENCE.
- When a pipeline is fit on data, all of the transformers inside it are fit

Why?

- Pipelines use less code than doing each transformer individually. Call **fit** and **transform** once
- Pipelines make your preprocessing workflow easier to understand.
- Pipelines are easy to use in production models. and to make sure the same preprocessing happens
- Pipelines can prevent data leakage. **Fit only training data**

Build pipeline for numerical features

In [312…

```
from sklearn.pipeline import make_pipeline

# instantiate preprocessors
impute_median = SimpleImputer(strategy='median')
scaler = StandardScaler()

# Make a numeric preprocessing pipeline
```

```
num_pipe = make_pipeline(impute_median, scaler)

# Fit the pipeline on the numeric training data
num_pipe.fit(X_train[num_cols])

# Transform the training data
X_train_num_tf = num_pipe.transform(X_train[num_cols])
# Transform the testing data
X_test_num_tf = num_pipe.transform(X_test[num_cols])
X_test_num_tf
```

Out[312…

	father	mother	midparentHeight	children	childNum
226	0.700891	-0.917794	-0.109211	0.284379	-1.092539
271	0.499773	0.384733	0.655727	-0.438460	0.158036
11	2.309832	-0.049443	1.610503	-0.438460	-0.675681
353	0.298655	-0.049443	0.124179	0.645798	0.574895
704	-0.505815	-1.786145	-1.549751	-0.799879	-0.258822
...	...	...	...	...	...
236	0.298655	2.121436	1.722173	-0.799879	-1.092539
300	0.298655	0.384733	0.516140	-1.161299	-0.675681
72	1.384691	2.121436	2.475944	0.645798	0.574895
87	1.103126	0.384733	1.074488	0.284379	0.991754
631	-0.505815	-0.049443	-0.343718	-0.438460	-1.092539

234 rows × 5 columns

Build pipeline for Ordinal features

In [314…

```
impute_na_ord = SimpleImputer(strategy='constant', fill_value='NA')

# Specifying the order of categories in quality/condition columns
fam_size_order = ["Small", "Mid", "Large"]
# Making the list of order lists for OrdinalEncoder
ordinal_category_orders = [fam_size_order]
# Instantiate the encoder and include the list of ordered values as an argument
ord_encoder = OrdinalEncoder(categories=ordinal_category_orders)

# Making a final scaler to scale category #'s
scaler_ord = StandardScaler()

ord_pipe = make_pipeline(impute_na_ord, ord_encoder, scaler_ord)

# Fit the encoder on the training data
ord_pipe.fit(X_train[ordinal_cols])
```

```python
# Transform the training data
X_train_ordinal_tf = ord_pipe.transform(X_train[ordinal_cols])
# Transform the test data
X_test_ordinal_tf = ord_pipe.transform(X_test[ordinal_cols])

# Value counts after transformation
X_test_ordinal_tf
```

Out[314…

	familySize
226	0.802811
271	-0.743097
11	-0.743097
353	0.802811
704	-0.743097
...	...
236	-0.743097
300	-0.743097
72	0.802811
87	0.802811
631	-0.743097

234 rows × 1 columns

Build pipeline for Nominal features

In [316…

```python
impute_na = SimpleImputer(strategy='constant', fill_value = "NA")

# Instantiate one hot encoder
ohe_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')

# Make pipeline with imputer and encoder
ohe_pipe = make_pipeline(impute_na, ohe_encoder)

ohe_pipe.fit(X_train[ohe_cols])

# Transform the training data
X_train_ohe_tf = ohe_pipe.transform(X_train[ohe_cols])
# Transform the test data
X_test_ohe_tf = ohe_pipe.transform(X_test[ohe_cols])

# Value counts after transformation
X_test_ohe_tf
```

Out[316...

	gender_NA	gender_female	gender_male
226	0.0	0.0	1.0
271	0.0	1.0	0.0
11	0.0	0.0	1.0
353	0.0	1.0	0.0
704	0.0	0.0	1.0
...	...	...	...
236	0.0	0.0	1.0
300	0.0	0.0	1.0
72	0.0	1.0	0.0
87	0.0	1.0	0.0
631	0.0	0.0	1.0

234 rows × 3 columns

Using Column Transformer

In [318...

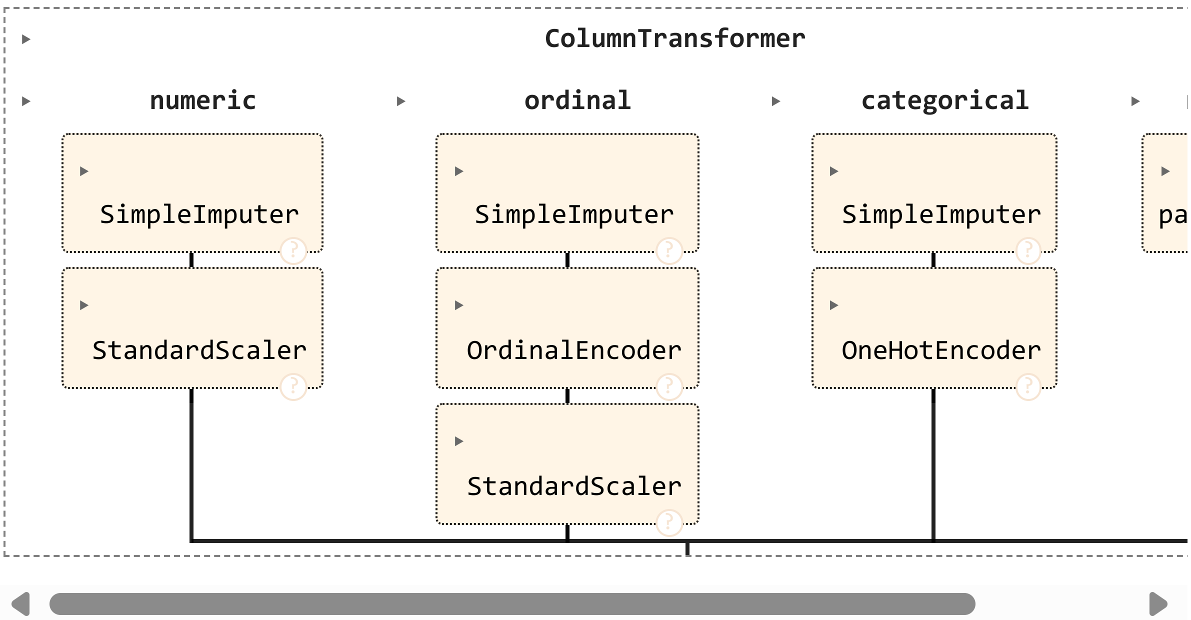
```
from sklearn.compose import ColumnTransformer

########### Numerical pipeline
# instantiate preprocessors
impute_median = SimpleImputer(strategy='median')
scaler = StandardScaler()

# Make a numeric preprocessing pipeline
num_pipe = make_pipeline(impute_median, scaler)

########### Ordinal pipeline
impute_na_ord = SimpleImputer(strategy='constant', fill_value='NA')

# Specifying the order of categories in quality/condition columns
fam_size_order = ["Small", "Mid", "Large"]
# Making the list of order lists for OrdinalEncoder
ordinal_category_orders = [fam_size_order]
# Instantiate the encoder and include the list of ordered values as an argument
ord_encoder = OrdinalEncoder(categories=ordinal_category_orders)

# Making a final scaler to scale category #'s
scaler_ord = StandardScaler()

ord_pipe = make_pipeline(impute_na_ord, ord_encoder, scaler_ord)

######### Nominal pipeline
impute_na = SimpleImputer(strategy='constant', fill_value = "NA")
```

```python
# Instantiate one hot encoder
ohe_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')

# Make pipeline with imputer and encoder
ohe_pipe = make_pipeline(impute_na, ohe_encoder)

######### Build pipelines tuples
num_tuple = ('numeric', num_pipe, num_cols)
ord_tuple = ('ordinal', ord_pipe, ordinal_cols)
ohe_tuple = ('categorical', ohe_pipe, ohe_cols)
```

In [319…

```python
######### Create ColumnTransformer
# remainder: Keep unlisted columns unchanged
# Instantiate with verbose_feature_names_out=False
'''False => Keeps clean column names (e.g., 'gender_female'),
   True => Adds transformer name as prefix (e.g., 'categorical__gender_female')'''
col_transformer = ColumnTransformer([num_tuple, ord_tuple, ohe_tuple],
                                    remainder='passthrough',
                                    verbose_feature_names_out=False)
col_transformer
```

Out[319…



In [320…

```python
# Fit on training data
col_transformer.fit(X_train)
# Transform the training data
X_train_processed = col_transformer.transform(X_train)
# Transform the testing data
X_test_processed = col_transformer.transform(X_test)
# View the processed training data
X_train_processed.head()
```

Out[320...

	father	mother	midparentHeight	children	childNum	familySize	gender_NA
728	-0.908051	0.818909	-0.019875	0.645798	0.991754	0.802811	0.0
699	-0.505815	-1.786145	-1.549751	0.284379	0.574895	0.802811	0.0
462	0.097538	0.818909	0.678061	0.284379	-0.675681	0.802811	0.0
40	1.907597	-0.917794	0.728312	0.645798	1.825471	0.802811	0.0
510	-0.103580	-0.266530	-0.215297	0.284379	0.574895	0.802811	0.0

Column Transformer

Transform heterogeneous data types at once. It lets you apply different types of transformers to different columns in your data