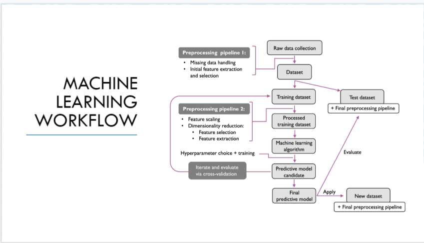


Estimator

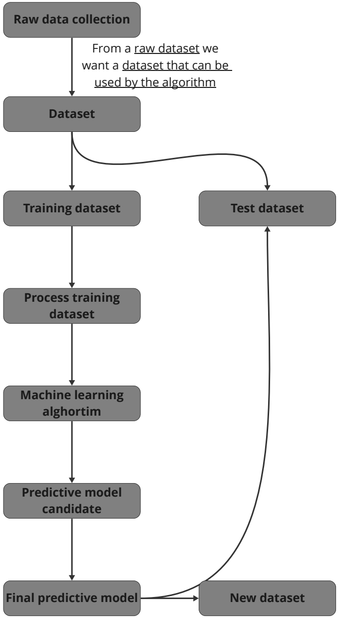
- An object which manages the estimation and decoding of a model
- Must provide the methods:
  `fit(data, target)` or
  `fit(data)`
- The model is estimated as a deterministic function of:
  - Data passed to the most recent call to `fit()`, or `partial_fit()`
  - Parameters provided in object construction = hyperparameters or through the method `set_params()`
  - A random state `numpy.random.RandomState`

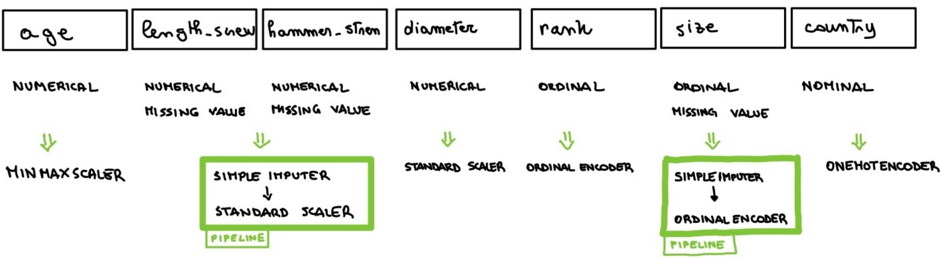
Predictor

- An estimator supporting
  - `predict()`
  - `fit_predict()`

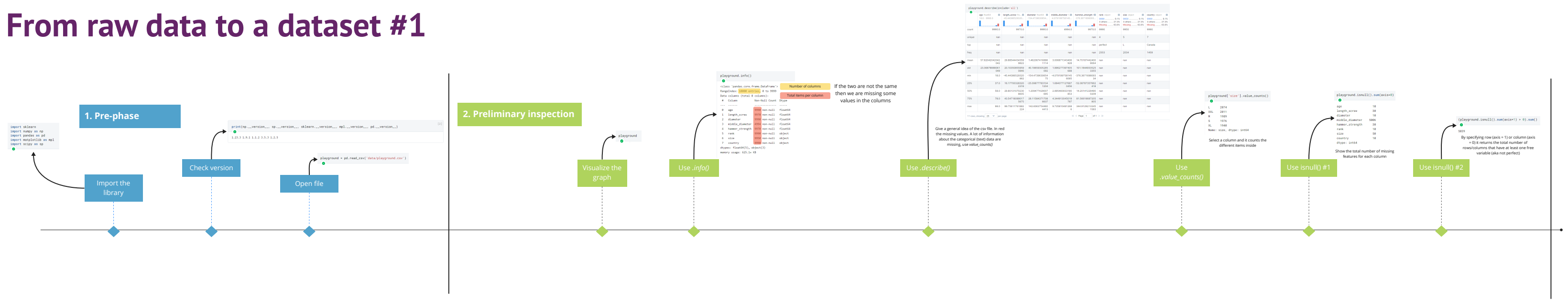
Transformer

- An estimator supporting
  - `transform()`: transforms the input into some transformed space. If the transformer was not already fitted, calling this method raise an exception.
  - `fit_transform()`





From raw data to a dataset #1



From raw data to a dataset #2

3. Dealing with missing data

Two type of data
- Float (number)
- Categorical (text)

Remove all the columns below a treshold

```
playground = playground.dropna(thresh=6000,axis=1).dropna(axis=0,how='all')
```

Remove all the columns that have less than (ex 6000) items.
Some columns will still have some missing values

```
# PRE
playground.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             9990 non-null   float64
 1   length_screw    9970 non-null   float64
 2   diameter        9990 non-null   float64
 3   middle_diameter 1992 non-null   float64
 4   hammer_strength 9970 non-null   float64
 5   rank            9990 non-null   object
 6   size            9950 non-null   object
 7   country         9990 non-null   object
dtypes: float64(5), object(3)
memory usage: 625.1+ KB
```

```
# POST
playground.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9990 entries, 0 to 9999
Data columns (total 7 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             9990 non-null   float64
 1   length_screw    9970 non-null   float64
 2   diameter        9990 non-null   float64
 3   hammer_strength 9970 non-null   float64
 4   rank            9990 non-null   object
 5   size            9950 non-null   object
 6   country         9990 non-null   object
dtypes: float64(4), object(3)
memory usage: 624.4+ KB
```

Use .simpleImputer()

```
from sklearn.impute import SimpleImputer
```

```
si = SimpleImputer(strategy= 'most_frequent')                  # or strategy = 'mean'
playground['size'] = si.fit_transform(playground[['size']])      # remember double [[  ]] !!!

# or use the mean strategy

si_mean = SimpleImputer(strategy = 'mean')
playground[['length_screw', 'hammer_strength']] = si_mean.fit_transform(playground[['length_screw', 'hammer_strength']])      # remember double [[  ]] !!!
```

Import and use *SimpleImputer*. Replace missing value with the **mode** (most frequent value) or with the **mean** of the feature column.
Useful for categorical data.

Now the dataset is completely filled

4 Handling categorical data

- Two type of categorical data
- **Ordinal** (have an order. T-shirt size, grade[bad, ok, nice, good], ...) - **Ordinal encoder**
- **Nominal** (no order. Nations, book title...) - **OneHotEncoding**

Or skip this if there are no categorical data (very rare)

4A. Ordinal data

Repeat for all ordinal data column

Import *OrdinalEncoder*

```
from sklearn.preprocessing import OrdinalEncoder
```

Choose the order and the column

```
oe_size = OrdinalEncoder(categories = [['S', 'M', 'L', 'XL', 'XXL']])
oe_size.fit_transform(playground[['size']])
```

```
array([[2.],
       [4.],
       [2.],
       ...,
       [2.],
       [1.],
       [4.]])
```

Double check

```
playground['size'][:4]
```

```
0      L
1    XXL
2      L
3     XL
Name: size, dtype: object
```

4B. Nominal data

Repeat for all nominal data column

marvel
dc comics
bao

One-hot encoding

marvel_col	dccomics_col	Bao_col
1	0	0
0	1	0
0	0	1

Import *OneHotEncoder*

```
from sklearn.preprocessing import OneHotEncoder
```

Start and create *temp*

```
ohe_country = OneHotEncoder()
temp = ohe_country.fit_transform(playground[['country']]).toarray()     # remember double [[  ]] !!!
```

Create *names*

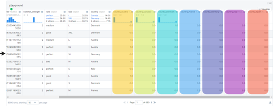
```
names = ohe_country.get_feature_names_out()
```

Merge all

```
playground[list(names)] = temp
```

Double check

Double check by using *playground* or *playground.info()* check that the table have new columns with 0 or 1 (one for each country) for the column selected



From raw data to a dataset #3

5 Handling numerical data

Majority of ML algorithms work better if features are on the same scale. It avoids

- Dominance of some features in the loss computation
- Dominance of some features in Euclidean distance

Most common approaches

- Normalization (or min-max scaling)
- Standardization

5A. Normalization or min-max scaling

Apply the min-max scaling to numerical features:

$$x^{i}_{norm} = \frac{x^{i} - x_{min}}{x_{max} - x_{min}}$$

where x_{max} and x_{min} are the largest and smallest values in the feature column we are scaling.

- SKLearn provides the `preprocessing.MinMaxScaler` class

Import *MinMaxScaler*

```
from sklearn.preprocessing import MinMaxScaler
```

Apply MinMaxScaler

```
mm_scalar= MinMaxScaler()
mm_scalar.fit_transform(playground[["age"]])
```

```
array([[0.3625],
       [0.025 ],
       [0.675 ],
       ...,
       [0.1625],
       [0.1875],
       [0.725 ]])
```

5B. Standardization

More practical for some algorithms with weights initialization centered around 0

Apply the formula

$$x^{i}_{std} = \frac{x^{i} - \mu_x}{\sigma_x}$$

Where μ_x and σ_x are the sample mean and the standard deviation of the feature column

- SKLearn implements standardization by `preprocessing.StandardScaler` class

Import *StandardScaler*

```
from sklearn.preprocessing import StandardScaler
```

Apply StandardScaler

```
sscaler = StandardScaler()
sscaler.fit_transform(playground[["length_screw", "diameter", "hammer_strength"]])
```

```
array([[ 0.        ,  0.38789898,  0.        ],
       [ 0.        , -1.71031482,  0.        ],
       [ 0.        , -0.56908024,  0.        ],
       ...,
       [ 0.        , -0.8439075 ,  0.        ],
       [ 0.        , -0.95371625,  0.        ],
       [ 0.        ,  0.51831827,  0.        ]])
```

6. Pipeline

Import *Pipeline*

```
from sklearn.pipeline import Pipeline
```

Define the pipeline

Choose a name (imp_scaler) and the composition of the pipeline

```
imp_scaler = Pipeline(
    [
        ('pipe1_si', SimpleImputer(strategy='mean')),
        ('pipe1_scalar', StandardScaler())
    ]
)

imp_scaler.fit_transform(playground[['length_screw', 'hammer_strength']])
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       ...,
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

Data loading + confusion matrix #1

1. Call the columns

Load the data and extract the output column

```
heart = pd.read_csv('data/heart.csv')
labels = heart['output'].values
heart.drop(['output'],axis=1, inplace=True)
```

In this way we separate the dataset and the labels

Run column transformer

To get the data that we will use for comparison

2. Split the data

We need to split the data between
- Training set
- Test set (used for later)

Import *train_test_split*

```
from sklearn.model_selection import train_test_split
```

To separate the data

Use train_test_split

No need to express both test_size and train_size, is ok if we have just one (the other can be deducted)

```
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size = 0.2, train_size = 0.8,
                                                    random_state = 42, shuffle = True, stratify = labels)
```

Test_size + train_size = 1

Train_test_split() uses 3 parameters
- **test_size**: % of observation to put in the test (usually 0.2/0.3)
- **Stratify**: balance between positive and negative instances of a specified column in both train and test size
- **Random state**: shuffle the rows. Select portion for test and portion for train

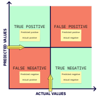
Check splitting

```
sum(y_train == 1)/len(y_train), sum(y_test == 1)/len(y_test)
```

(0.5454545454545454, 0.5409836065573771)

Check the average of positive classes in train and test size. **Less the difference better it is**

2. Confusion matrix



The first diagonal \ express how many items are correctly classified
The second diagonal / express how many items are wrongly classifed

Fit the perceptron

```
perceptron = Perceptron(eta0 = 0.001)

perceptron.fit(X_train, y_train)
```

Perceptron(eta0=0.001)

Import

(Can be expressed in one row only)

```
from sklearn.metrics import precision_score, accuracy_score
from sklearn.metrics import recall_score, f1_score, matthews_corrcoef, confusion_matrix
```

Check on train

Check on test

```
y_test_predicted = perceptron.predict(X_test)

confusion_matrix(y_test, y_test_predicted)
```

Accuracy

```
accuracy_score(y_train, y_train_predicted)
```

More accurate => closer to 1

Also there is *precision_score()*, *recall_score*, *f1_score* and *matthews_corrcoef*