Introduction

One of the preprocessing steps in machine learning is feature encoding. It is the process of turning categorical data in a dataset into numerical data. It is important that we perform feature encoding because most machine learning algorithms only handle numerical data and not data in text form.

We will learn the difference between nominal variables and ordinal variables. In addition, we will explore how OneHotEncoder and OrdinalEncoder can be used to transform these variables as part of a machine learning pipeline.

We will use this pipeline to predict the mean test score of different students. This is a regression problem in machine learning.

Import libraries

In [48]:

```
# Data wrangling
import pandas as pd
import numpy  as np

# Data visualisation
import seaborn           as sns
import matplotlib.pyplot as plt

# Machine learning
from sklearn.preprocessing   import OneHotEncoder
from sklearn.compose         import make_column_transformer
from sklearn.pipeline        import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.linear_model    import LinearRegression
from sklearn.ensemble        import GradientBoostingRegressor
from sklearn.metrics         import mean_absolute_error, mean_squared_error

# Remove warnings
import warnings
warnings.filterwarnings('ignore')
```

Import and read data

In [2]:

```
data1 = pd.read_csv('/Users/lucasmassaroppe/Documents/rhi_magnesita/md_raw_d
data2 = pd.read_csv('/Users/lucasmassaroppe/Documents/rhi_magnesita/md_targe
```

Missing values, merge dataframes and data types

In [3]:

```
# making the 'Unamed: 0' column (from 'md_raw_dataset.csv' dataset) as one c
# the second key to merge both datasets is the column 'groups'
data1['index'] = data1['Unnamed: 0']
```

```
data = pd.merge(data1, data2, on=['index', 'groups'])

# droping unwanted columns
data = data.drop(columns=['crystal_type', 'crystal_supergroup', 'Cycle', 'Un
                          'Unnamed: 17', 'etherium_before_start', 'expected_
                          'start_process', 'start_subprocess1', 'start_criti
                          'predicted_process_end', 'process_end', 'subproces
                          'reported_on_tower', 'opened', 'raw_kryptonite', '

# handling missing values by interpolation
data = data.interpolate(method='cubicspline')
```

In [4]:
```
data
```

Out[4]:

	tracking	place	tracking_times	human_behavior_report	human_measure	crystal_weig
0	84941	1	1	4	700	350.6301
1	84951	1	1	4	800	347.4298
2	84971	1	1	3	700	333.1576
3	84981	1	1	3	700	362.3764
4	84991	1	1	3	720	349.7962
...	...	...	...	...	...	
8539	546381	2	1	4	530	356.2083
8540	546411	2	1	3	610	372.5217
8541	546421	2	1	3	560	363.0748
8542	546421	2	2	3	550	344.7791
8543	44471	2	1	3	580	348.8972

In [5]:
```
data.columns
```

Out[5]:
```
Index(['tracking', 'place', 'tracking_times', 'human_behavior_report',
       'human_measure', 'crystal_weight', 'expected_factor_x',
       'previous_factor_x', 'first_factor_x', 'expected_final_factor_x',
       'final_factor_x', 'previous_adamantium', 'chemical_x', 'argon',
       'pure_seastone', 'groups', 'target'],
      dtype='object')
```

Exploratory data analysis (EDA)

Exploratory data analysis is the process of analysing and visualising the variables in a dataset.

Predictor variables

The predictor variables in the dataset are:

- super_hero_group;
- tracking;
- place;
- tracking_times;

- human_behavior_report;
- human_measure;
- crystal_weight;
- expected_factor_x;
- previous_factor_x;
- first_factor_x;
- expected_final_factor_x;
- final_factor_x;
- previous_adamantium;
- chemical_x;
- argon;
- pure_seastone;
- groups;

In this section, we will explore how these different features influence the outcome of the 'target' test score.

Correlation Matrix

In [6]:
```
data.corr()
```

Out[6]:

	tracking	place	tracking_times	human_behavior_report	huma
tracking	1.000000	0.021053	0.011412	-0.016823	
place	0.021053	1.000000	0.017774	0.002657	
tracking_times	0.011412	0.017774	1.000000	0.026406	
human_behavior_report	-0.016823	0.002657	0.026406	1.000000	
human_measure	-0.038424	-0.188259	-0.029212	-0.015553	
...	...	...	...	...	
chemical_x	0.006936	0.000489	0.071702	-0.084843	
argon	-0.015680	-0.047466	0.112584	-0.162244	
pure_seastone	0.044682	0.025704	0.037670	-0.146317	
groups	0.039181	-0.278115	-0.004061	0.006640	
target	0.032006	-0.054274	-0.014535	0.005232	

As we can see from the correlation structure, the predictors have a dependency on each other and on the target column, making it difficult to reduce the dimensionality of this dataset.

Build machine learning pipeline

A pipeline chains together multiple steps in the machine learning process where the output of each step is used as input to the next step. It is typically used to chain data preprocessing procedures together with modelling into one cohesive workflow.

Here, we will build two pipelines that share the same column transformer that we have created above but with a different machine learning model, one using linear regression and the other using gradient boosting.

We will then compare the accuracy of the prediction results using mean absolute error (MAE) as well as root mean squared error (RMSE). The model with a lower prediction error is deemed more accurate than the other.

In [71]:
```
# Train test split

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size = 0.3)

print("X_train shape: ", X_train.shape)
print("Y_train shape: ", Y_train.shape)
print("X_test shape: ", X_test.shape)
print("Y_test shape: ", Y_test.shape)
```

```
X_train shape:  (5980, 16)
Y_train shape:  (5980,)
X_test shape:  (2564, 16)
Y_test shape:  (2564,)
```

In [72]:
```
# Instantiate pipeline with linear regression

lm = LinearRegression()
lm_pipeline = make_pipeline(lm)
```

In [73]:
```
# Instantiate pipeline with gradient boosting

gbm = GradientBoostingRegressor()
gbm_pipeline = make_pipeline(gbm)
```

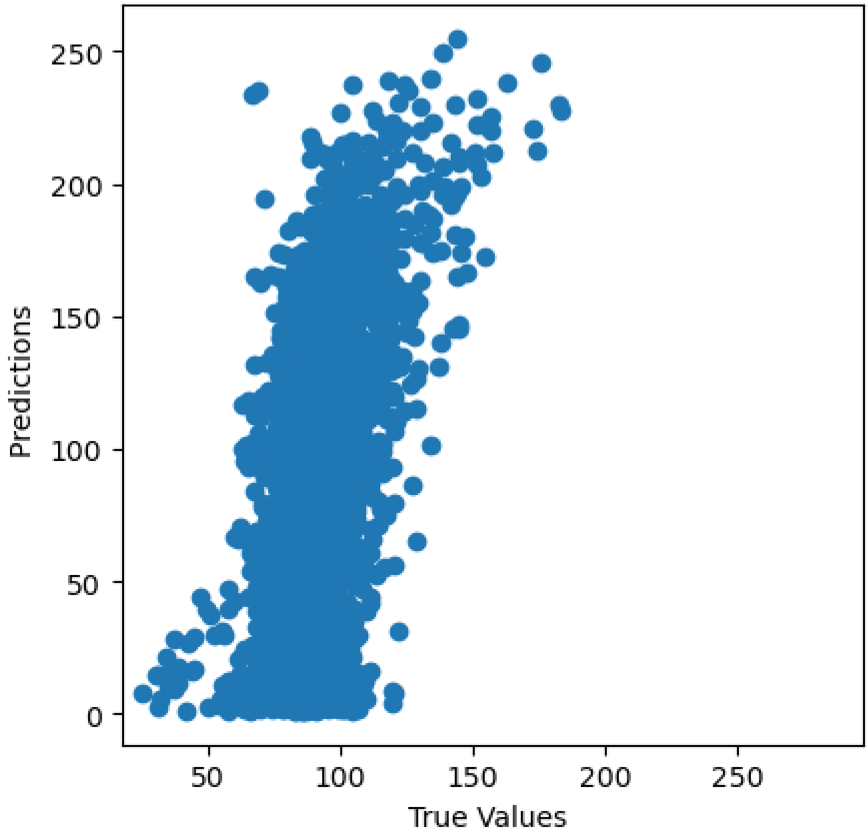
In [74]:
```
# Fit pipeline to training set and make predictions on test set

lm_pipeline.fit(X_train, Y_train)
lm_predictions = lm_pipeline.predict(X_test)

gbm_pipeline.fit(X_train, Y_train)
gbm_predictions = gbm_pipeline.predict(X_test)
```

In [75]:
```
# Calculate mean square error and root mean squared error

lm_mae = mean_absolute_error(lm_predictions, Y_test)
lm_rmse = np.sqrt(mean_squared_error(lm_predictions, Y_test))
print("LM MAE: {:.2f}".format(round(lm_mae, 2)))
print("LM RMSE: {:.2f}".format(round(lm_rmse, 2)))

gbm_mae = mean_absolute_error(gbm_predictions, Y_test)
gbm_rmse = np.sqrt(mean_squared_error(gbm_predictions, Y_test))
print("GBM MAE: {:.2f}".format(round(gbm_mae, 2)))
print("GBM RMSE: {:.2f}".format(round(gbm_rmse, 2)))
```

```
LM MAE: 47.45
LM RMSE: 56.24
GBM MAE: 42.92
GBM RMSE: 51.15
```

As we can see, the gradient boosting regression method performs better than the linear regression method.

In [82]:
```
plt.scatter(gbm_predictions, Y_test)
plt.xlabel('True Values ')
plt.ylabel('Predictions ')
plt.axis('equal')
plt.axis('square')
plt.show()
```



In [ ]:

In [ ]: