MO432 - Exercício 2

June 13, 2021

1 MO432 - Exercício 2

1.0.1 Alunos

1. Giovanna Vendramini - RA: 173304
2. Luiz Eduardo Cartolano - RA: 183012

1.1 1. Leitura dos Dados

Para realizar o trabalho iremos usar os dados obtidos do arquivo *Bias_correction_ucl.csv*. A leitura foi feita usando o método `pd.read_csv` da biblioteca Pandas.

As colunas *Next_Tmin* e *Next_tmax* são os atributos de saída, porém, para este projeto foi utilizado apenas *Next_tmax* como o rótulo que queremos prever.

Assim, foram desconsiderados e removidos o atributo de saída *Next_Tmin*, o atributo de entrada *Date*, e as linhas que apresentaram valores faltantes. Por esse motivo, a tabela lida que apresentava 7752 linhas e 25 colunas, passou a apresentar 7588 linhas e 23 colunas após o tratamento.

A seguir apresentamos uma descrição mais detalhada de cada um dos atributos.

Attribute Information:

1. station - used weather station number: 1 to 25
2. Date - Present day: yyyy-mm-dd (‘2013-06-30’ to ‘2017-08-30’)
3. Present_Tmax - Maximum air temperature between 0 and 21 h on the present day (Â°C): 20 to 37.6
4. Present_Tmin - Minimum air temperature between 0 and 21 h on the present day (Â°C): 11.3 to 29.9
5. LDAPS_RHmin - LDAPS model forecast of next-day minimum relative humidity (%): 19.8 to 98.5
6. LDAPS_RHmax - LDAPS model forecast of next-day maximum relative humidity (%): 58.9 to 100
7. LDAPS_Tmax_lapse - LDAPS model forecast of next-day maximum air temperature applied lapse rate (Â°C): 17.6 to 38.5
8. LDAPS_Tmin_lapse - LDAPS model forecast of next-day minimum air temperature applied lapse rate (Â°C): 14.3 to 29.6
9. LDAPS_WS - LDAPS model forecast of next-day average wind speed (m/s): 2.9 to 21.9
10. LDAPS_LH - LDAPS model forecast of next-day average latent heat flux (W/m2): -13.6 to 213.4

11. LDAPS_CC1 - LDAPS model forecast of next-day 1st 6-hour split average cloud cover (0-5 h) (%): 0 to 0.97
12. LDAPS_CC2 - LDAPS model forecast of next-day 2nd 6-hour split average cloud cover (6-11 h) (%): 0 to 0.97
13. LDAPS_CC3 - LDAPS model forecast of next-day 3rd 6-hour split average cloud cover (12-17 h) (%): 0 to 0.98
14. LDAPS_CC4 - LDAPS model forecast of next-day 4th 6-hour split average cloud cover (18-23 h) (%): 0 to 0.97
15. LDAPS_PPT1 - LDAPS model forecast of next-day 1st 6-hour split average precipitation (0-5 h) (%): 0 to 23.7
16. LDAPS_PPT2 - LDAPS model forecast of next-day 2nd 6-hour split average precipitation (6-11 h) (%): 0 to 21.6
17. LDAPS_PPT3 - LDAPS model forecast of next-day 3rd 6-hour split average precipitation (12-17 h) (%): 0 to 15.8
18. LDAPS_PPT4 - LDAPS model forecast of next-day 4th 6-hour split average precipitation (18-23 h) (%): 0 to 16.7
19. lat - Latitude (Â°): 37.456 to 37.645
20. lon - Longitude (Â°): 126.826 to 127.135
21. DEM - Elevation (m): 12.4 to 212.3
22. Slope - Slope (Â°): 0.1 to 5.2
23. Solar radiation - Daily incoming solar radiation (wh/m2): 4329.5 to 5992.9
24. Next_Tmax - The next-day maximum air temperature (Â°C): 17.4 to 38.9
25. Next_Tmin - The next-day minimum air temperature (Â°C): 11.3 to 29.8

A matrix de dados lidos é denominada ***data***.

[ ]:
```
import pandas as pd

import warnings
warnings.filterwarnings("ignore")
```

1.1.1 Leitura do Dataset

[ ]:
```
data = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪00514/Bias_correction_ucl.csv")
print(f'Volume de dados inicial: {data.shape}')
```

```
Volume de dados inicial: (7752, 25)
```

1.1.2 Remoção das colunas e valores nulos

[ ]:
```
# remocao das colunas Next_Tmin e Date
data = data.drop(columns=['Next_Tmin', 'Date'], axis=1)
# remocao dos valores nulos
data = data.dropna()
print(f'Volume de dados após retirar valores nulos: {data.shape}\n')
data.head()
```

```
Volume de dados após retirar valores nulos: (7588, 23)
```

[ ]:
```
   station  Present_Tmax  Present_Tmin  LDAPS_RHmin  LDAPS_RHmax  \
0      1.0          28.7          21.4    58.255688    91.116364
1      2.0          31.9          21.6    52.263397    90.604721
2      3.0          31.6          23.3    48.690479    83.973587
3      4.0          32.0          23.4    58.239788    96.483688
4      5.0          31.4          21.9    56.174095    90.155128

   LDAPS_Tmax_lapse  LDAPS_Tmin_lapse  LDAPS_WS    LDAPS_LH  LDAPS_CC1  …  \
0         28.074101         23.006936  6.818887   69.451805   0.233947  …
1         29.850689         24.035009  5.691890   51.937448   0.225508  …
2         30.091292         24.565633  6.138224   20.573050   0.209344  …
3         29.704629         23.326177  5.650050   65.727144   0.216372  …
4         29.113934         23.486480  5.735004  107.965535   0.151407  …

   LDAPS_PPT1  LDAPS_PPT2  LDAPS_PPT3  LDAPS_PPT4      lat      lon       DEM  \
0         0.0         0.0         0.0         0.0  37.6046  126.991  212.3350
1         0.0         0.0         0.0         0.0  37.6046  127.032   44.7624
2         0.0         0.0         0.0         0.0  37.5776  127.058   33.3068
3         0.0         0.0         0.0         0.0  37.6450  127.022   45.7160
4         0.0         0.0         0.0         0.0  37.5507  127.135   35.0380

    Slope  Solar radiation  Next_Tmax
0  2.7850      5992.895996       29.1
1  0.5141      5869.312500       30.5
2  0.2661      5863.555664       31.1
3  2.5348      5856.964844       31.7
4  0.5055      5859.552246       31.2

[5 rows x 23 columns]
```

1.1.3 Separação do X/y

Foram separados os atributos de entrada do atributo de saída. O numpy array y passou a representar os rótulos, enquanto que os atributos de entrada foi armazenado na variável x.

[ ]:
```
y = data['Next_Tmax'].to_numpy()
x = data.drop(['Next_Tmax'], axis=1).to_numpy()
```

1.1.4 Normalização e Centralização dos valores de X

Para centralizar uma variável, o valor médio é subtraído de todos os valores. Como resultado da centralização, os atributos tem uma média 0. Da mesma forma, para dimensionar os dados, cada valor da variável é dividido por seu desvio padrão. O escalonamento dos dados força os valores a terem um desvio padrão comum de 1.

Para o projeto usou-se a função `StandardScaler` do sklearn.

[ ]:
```
from sklearn.preprocessing import StandardScaler
```

[ ]:
```
x_after_norm = StandardScaler().fit_transform(x)
```

[ ]:
```
print(f'Médias dos valores após a normalização: {x_after_norm.mean()}')
print(f'Desvio dos valores após a normalização: {x_after_norm.std()}')
```

```
Médias dos valores após a normalização: -3.4728640255030726e-15
Desvio dos valores após a normalização: 0.9999999999999999
```

1.2 Definição da função de erro

Como função de custo foi adotada a métrica RSME, por meio do método `make_scorer` do Sklearn, utilizando como argumento a função *MSE* e declarando *squared* como False. Sendo assim, é gerada a função de custo *neg_root_mean_squared_error*.

[ ]:
```
from sklearn.metrics import make_scorer, mean_squared_error
```

[ ]:
```
rmse_cost = make_scorer(score_func=mean_squared_error, squared=False,␣
↪greater_is_better=False)
```

1.3 2. Regressores

Para cada um dos regressores, foi definido o valor da função de custo utilizando os hiperparâmetros padrões do método e, em seguida, foram calculados os melhores hiperparâmetros e a função de custo ao utilizá-los.

1.3.1 2.1 Regressão Linear

A regressão linear é uma equação para se estimar uma variável y, dados os valores de atributos x, determinando a relação entre as variáveis. Para isso, foi adotado o método `LinearRegression` do Sklearn.

[ ]:
```
from sklearn.linear_model import LinearRegression
```

[ ]:
```
estimator = LinearRegression()
```

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.4742306701826382
```

1.3.2 2.2 Regressão Linear com regularização L2

Para a regressão linear com regularização L2, foi adotado o método `Ridge` do Sklearn.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.linear_model import Ridge
from scipy.stats import loguniform
```

[ ]:
```
estimator = Ridge()
```

1.3.3 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.4742265105173766
```

1.3.4 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'alpha': loguniform(10**(-3), 10**3)
    }

random_search_l2 = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

l2_solution = random_search_l2.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {l2_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*l2_solution.best_score_}')
```

```
Melhor hiperparametro: {'alpha': 180.13404791374109}
Melhor custo encontrado: 1.572225627976686
```

1.3.5 2.3 Regressão Linear com regularização L1

Para a regressão linear com regularização L1, foi adotado o método `Lasso` do Sklearn.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.linear_model import Lasso
from scipy.stats import loguniform
```

[ ]:
```
estimator = Lasso()
```

1.3.6 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.972809801012016
```

1.3.7 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'alpha': loguniform(10**(-3), 10**3)
    }

random_search_l1 = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

l1_solution = random_search_l1.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {l1_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*l1_solution.best_score_}')
```

```
Melhor hiperparametro: {'alpha': 0.04320715110139978}
Melhor custo encontrado: 1.5686038460069853
```

1.3.8 2.4 SVM Linear

Para o SVM linear, foi adotado o método `LinearSVR` do Sklearn.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import LinearSVR
from scipy.stats import loguniform
```

[ ]:
```
estimator = LinearSVR()
```

1.3.9 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.476351732292993
```

1.3.10 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'C': loguniform(2**-5.0, 2**15),
    'epsilon': [0.1,0.3]
    }

random_search_svrl = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

svrl_solution = random_search_svrl.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {svrl_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*svrl_solution.best_score_}')
```

```
Melhor hiperparametro: {'C': 0.4445411902913089, 'epsilon': 0.1}
Melhor custo encontrado: 1.5635241581040258
```

1.3.11 2.5 SVM com kernel RBF

Para o SVM linear com kernel RBF, foi adotado o método `LinearSVR` do Sklearn, passando como argumento `kernel='rbf'`.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVR
from scipy.stats import loguniform
```

[ ]:
```
estimator = SVR(kernel='rbf')
```

1.3.12 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.1588424577728607
```

1.3.13 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'C': loguniform(2**-5.0, 2**15),
    'epsilon': [0.1,0.3],
    'gamma': loguniform(2**-9, 2**3)
    }

random_search_svr = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

svr_solution = random_search_svr.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {svr_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*svr_solution.best_score_}')
```

```
Melhor hiperparametro: {'C': 1.3677950805540304, 'epsilon': 0.1, 'gamma':
```

```
0.010182425776863237}
Melhor custo encontrado: 1.5786088389396586
```

1.3.14 2.6 KNN

Para o KNN, foi adotado o método `KNeighborsRegressor` do Sklearn.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.neighbors import KNeighborsRegressor
from scipy.stats import randint
```

[ ]:
```
estimator = KNeighborsRegressor()
```

1.3.15 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.2486732510260647
```

1.3.16 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {'n_neighbors': randint(1, 1000)}

random_search_knn = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

knn_solution = random_search_knn.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {knn_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*knn_solution.best_score_}')
```

```
Melhor hiperparametro: {'n_neighbors': 54}
Melhor custo encontrado: 1.8525761690653044
```

1.3.17 2.7 MLP

Para o MLP, foi adotado o método `MLPRegressor` do Sklearn.

[ ]:
```
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import RandomizedSearchCV
from numpy import arange
```

[ ]:
```
estimator = MLPRegressor()
```

1.3.18 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.1691131221142252
```

1.3.19 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {'hidden_layer_sizes': arange(5, 21, 3).tolist()}

random_search_mlp = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

mlp_solution = random_search_mlp.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {mlp_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*mlp_solution.best_score_}')
```

```
Melhor hiperparametro: {'hidden_layer_sizes': 17}
Melhor custo encontrado: 2.3570743373569414
```

1.3.20 2.8 Árvore de Decisão

Para a Árvore de Decisão, foi adotado o método `DecisionTreeRegressor` do Sklearn.

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeRegressor
from scipy.stats import uniform
```

[ ]:
```
estimator = DecisionTreeRegressor()
```

1.3.21 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.463442683773399
```

1.3.22 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {'ccp_alpha': uniform(0, 0.04)}

random_search_dt = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

dt_solution = random_search_dt.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {dt_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*dt_solution.best_score_}')
```

```
Melhor hiperparametro: {'ccp_alpha': 0.03503730538968379}
Melhor custo encontrado: 1.8479908832557044
```

1.3.23 2.9 Random Forest

Para a Random Forest, foi adotado o método `RandomForestRegressor` do Sklearn.

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV
```

[ ]:
```
estimator = RandomForestRegressor()
```

1.3.24 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 0.999757775055358
```

1.3.25 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'n_estimators': [10,100,1000],
    'max_features': [5, 10, 22]
    }

random_search_rf = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

rf_solution = random_search_rf.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {rf_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*rf_solution.best_score_}')
```

```
Melhor hiperparametro: {'n_estimators': 1000, 'max_features': 10}
Melhor custo encontrado: 1.6247149864331647
```

1.3.26 2.10 GBM

Para o GBM, foi adotado o método `GradientBoostingRegressor` do Sklearn.

[ ]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint
```

[ ]:
```
estimator = GradientBoostingRegressor()
```

1.3.27 Usando valores default

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate

cv_split = ShuffleSplit(n_splits=5, random_state=1234)

default_sol = cross_validate(estimator,
                             x_after_norm, y, cv=cv_split,
                             scoring=rmse_cost)

print(f'Custo encontrado: {-1 * default_sol["test_score"].mean()}')
```

```
Custo encontrado: 1.220501150443596
```

1.3.28 Buscando os melhores Hiperparâmetros

[ ]:
```
param_dist = {
    'n_estimators': randint(5,100),
    'learning_rate': uniform(0.01,0.3),
    'max_depth': [2, 3]
    }

random_search_gbm = RandomizedSearchCV(
    estimator=estimator,
    param_distributions=param_dist,
    n_iter=10,
    cv=5,
    scoring=rmse_cost,
    random_state=1234
)

gbm_solution = random_search_gbm.fit(x_after_norm, y)

print(f'Melhor hiperparametro: {gbm_solution.best_params_}')
print(f'Melhor custo encontrado: {-1*gbm_solution.best_score_}')
```

```
Melhor hiperparametro: {'learning_rate': 0.11946579517041167, 'max_depth': 2,
'n_estimators': 80}
Melhor custo encontrado: 1.5928044800644765
```

1.4 3. Tabela Final

Foi gerada uma tabela sumarizando os principais resultados encontrados. Nela, foram organizados os valores de custo para cada um do regressores, utilizando os hiperparâmetros padrões do método (`Default Cost`) e adotando os melhores hiperparâmetros (`Best Cost`).

[3]:
```
import pandas as pd
```

```
columns = ['Default Cost', 'Best Cost']
index = ['Linear', 'Linear_L2reg', 'Linear_L1reg', 'SVM_Linear', 'SVM_RBF',␣
↪'KNN', 'MLP', 'Decision_Tree', 'Random_Forest', 'GBM']
final_results = pd.DataFrame(index=index, columns=columns).fillna('0')
```

[11]:
```
final_results['Default Cost']['Linear'] = 1.4742306701826382
final_results['Best Cost']['Linear'] = '-'
```

[12]:
```
final_results['Default Cost']['Linear_L2reg'] = 1.4742265105173766
final_results['Best Cost']['Linear_L2reg'] = 1.572225627976686
```

[13]:
```
final_results['Default Cost']['Linear_L1reg'] = 1.972809801012016
final_results['Best Cost']['Linear_L1reg'] = 1.5686038460069853
```

[14]:
```
final_results['Default Cost']['SVM_Linear'] = 1.476351732292993
final_results['Best Cost']['SVM_Linear'] = 1.5635241581040258
```

[15]:
```
final_results['Default Cost']['SVM_RBF'] = 1.1588424577728607
final_results['Best Cost']['SVM_RBF'] = 1.5786088389396586
```

[16]:
```
final_results['Default Cost']['KNN'] = 1.2486732510260647
final_results['Best Cost']['KNN'] = 1.8525761690653044
```

[17]:
```
final_results['Default Cost']['MLP'] = 1.1691131221142252
final_results['Best Cost']['MLP'] = 2.3570743373569414
```

[18]:
```
final_results['Default Cost']['Decision_Tree'] = 1.463442683773399
final_results['Best Cost']['Decision_Tree'] = 1.8479908832557044
```

[19]:
```
final_results['Default Cost']['Random_Forest'] = 0.999757775055358
final_results['Best Cost']['Random_Forest'] = 1.6247149864331647
```

[20]:
```
final_results['Default Cost']['GBM'] = 1.220501150443596
final_results['Best Cost']['GBM'] = 1.5928044800644765
```

[21]:
```
final_results
```

[21]:

	Default Cost	Best Cost
Linear	1.474231	-
Linear_L2reg	1.474227	1.572226
Linear_L1reg	1.97281	1.568604
SVM_Linear	1.476352	1.563524
SVM_RBF	1.158842	1.578609
KNN	1.248673	1.852576
MLP	1.169113	2.357074
Decision_Tree	1.463443	1.847991
Random_Forest	0.999758	1.624715

```
GBM                1.220501  1.592804
```

Created in Deepnote