SVM model

December 30, 2023

1 Support vector machine (SVM)

1.1 1. Import necessary libraries

[ ]:
```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

import matplotlib.pyplot as plt
import seaborn as sns
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')

import sys
sys.executable
```

[ ]:
```
'/home/dagngyen5462/miniconda3/envs/min_ds-env/bin/python'
```

1.2 2. Insert and preprocess data

1.2.1 2.1. Load data

Load our data from path `../Data/processed_data.csv`.

[ ]:
```
# Load our data from path `../Data/processed_data.csv`
courses_df = pd.read_csv('../Data/processed_data.csv', sep=',',engine='python',␣
    ↪encoding='utf-8')
courses_df.sample(5)
```

```
[ ]:                                                  name  \
5562                                       Web of Data
5422     Applied Public History: Places, People, Stories
3044   Introduction to Particle Accelerators (NPAP MOOC)
1107   Connect Your Services with Microsoft Azure Ser…
2056   Data Processing and Feature Engineering with M…

                                 general              specify  enrollment  \
5562                        Data Science        Data Analysis        5918
5422                 Arts and Humanities              History        5234
3044   Physical Science and Engineering      Research Methods        6200
1107                    Computer Science  Software Development        2612
2056                        Data Science        Data Analysis       14316

     language  rating         level  duration  \
5562  English     4.1  Intermediate        18
5422  English     4.8      Beginner        26
3044  English     4.7  Intermediate        11
1107  English     4.6  Intermediate        10
2056  English     4.7  Intermediate        20

                                              instructor  instructor_rate  \
5562   Fabien Gandon, Catherine Faron Zucker, Olivier…             3.9
5422                                    Catherine Clarke               4.9
3044   Mats Lindroos, Sverker Werin, Erik Adli, Franc…             4.6
1107                                           Microsoft               4.7
2056   Amanda  Wang, Matt Rich, Cris LaPierre, Adam F…             4.7

                 offered by
5562           EIT Digital
5422   University of London
3044        Lund University
1107              Microsoft
2056              MathWorks
```

1.2.2 2.2. Preprocess data

- The `name` and `instructor` features needs to be removed because it is not useful in training the model.

```
[ ]: data_ = courses_df.copy().drop(columns=['name', 'instructor'])
     data_.sample(5)
```

```
[ ]:              general                   specify  enrollment language  \
761    Computer Science      Software Development        3299   English
3404           Business  Leadership and Management        3483   English
1213   Computer Science        Design and Product        9176   Spanish
```

```
5540  Computer Science                 Algorithms       20741  English
1967          Business           Entrepreneurship        7246   French

      rating         level  duration  instructor_rate  \
761      4.7      Beginner        13              4.8
3404     4.9      Beginner         5              4.7
1213     4.9      Beginner        20              4.9
5540     4.7  Intermediate        24              4.4
1967     4.6      Beginner        11              4.6

                                    offered by
761                    Institut Mines-Télécom
3404                    Politecnico di Milano
1213                                   Google
5540  University of Illinois at Urbana-Champaign
1967                    ESSEC Business School
```

- Explore missing values in variables:
  - View summary of dataset.

```
[ ]: data_.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5702 entries, 0 to 5717
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   general          5702 non-null   object
 1   specify          5702 non-null   object
 2   enrollment       5702 non-null   int64
 3   language         5702 non-null   object
 4   rating           5702 non-null   float64
 5   level            5702 non-null   object
 6   duration         5702 non-null   int64
 7   instructor_rate  5702 non-null   float64
 8   offered by       5702 non-null   object
dtypes: float64(2), int64(2), object(5)
memory usage: 445.5+ KB
```

```
[ ]: print('Number missing values in each column:\n',data_.isnull().sum())
```

```
Number missing values in each column:
 general            0
specify            0
enrollment         0
language           0
rating             0
level              0
```

```
duration           0
instructor_rate    0
offered by         0
dtype: int64
```

- Explore missing values in variables:
  - Drop rows with missing values: Because the number of missing values each variable is insignificant, we will remove rows containing missing data.

```
[ ]: # Drop rows with missing values
data_.dropna(inplace=True)
print('Number missing values in each column:\n',data_.isnull().sum())
```

```
Number missing values in each column:
 general            0
specify            0
enrollment         0
language           0
rating             0
level              0
duration           0
instructor_rate    0
offered by         0
dtype: int64
```

1.3 3. Prepare for training model

1.3.1 3.1. Define kind of features

- Define selection and target features to prepare data for training model.

```
[ ]: # Define selection and target features to prepare data for training model
target = ['rating']
specificities = list(set(data_.columns) - set(target))
```

- Define numerical and categorical features to transformer.

```
[ ]: # Define numerical and categorical features
numeracy_ = list(set(specificities) - set(courses_df.
 ↪select_dtypes(include=['object']).columns))
category_ = list(set(specificities) - set(courses_df.
 ↪select_dtypes(exclude=['object']).columns))
print('Numerical features:', numeracy_)
print('Categorical features:', category_)
```

```
Numerical features: ['duration', 'enrollment', 'instructor_rate']
Categorical features: ['offered by', 'level', 'general', 'language', 'specify']
```

1.3.2 3.2. Split data

Split data into 3 datasets: `Training dataset`, `Validation dataset` and `Testing dataset`. We'll perform splitting on the following ratio 80-20.

[ ]:
```
# Define the constant variable random_state
random_state = 2112

# Select features and target variable
X = data_[specificities]
y = data_[target]

# Split data on the following ratio 80-20
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
↪3,random_state=random_state)
```

1.3.3 3.3. Initialize transformer

[ ]:
```
# Create transformer for numerical and categorical features by using Pipeline
num_transformer = Pipeline(steps=[('scaler', StandardScaler())])
cat_transformer = Pipeline(steps=[('onehot', OneHotEncoder(sparse=False,␣
↪handle_unknown='ignore'))])

# Create a preprocessor using ColumnTransformer
preprocessor = ColumnTransformer(transformers=[('num', num_transformer,␣
↪numeracy_),
                                               ('cat', cat_transformer,␣
↪category_)])
```

1.3.4 3.4. Search hyperparameter for available data fitting model

- When tuning the hyperparameter, I use `GridSearchCV`.
- While setting up the `Pipeline` for `GridSearchCV`, I set the variable value `cv=5`.
  - This means that `GridSearchCV` will divide the training dataset into 5 sub-datasets.
  - Each sub-dataset will be used as a ***validation dataset*** while the others will be used for ***training*** and ***hyperparameter-tuning***.
- To calculate the **Mean Squared Error** (MSE) score, `GridSearchCV` will take the average of splitting the training dataset 5 times.

We take into account some essential hyperparameters for fine-tuning SVMs: - **`C`**: The regularization parameter that controls the trade-off between the margin and the number of training errors. A larger value of C penalizes training errors more heavily, resulting in a smaller margin but potentially better generalization performance. A smaller value of C allows for more training errors but may lead to overfitting. - **`Kernel`**: The kernel function that defines the similarity between data points. Different kernels can capture different relationships between data points, and the choice of kernel can significantly impact the performance of the SVM. Common kernels include linear, polynomial, radial basis function (RBF), and sigmoid. - **`Gamma`**: The parameter that controls the influence of support vectors on the decision boundary. A larger value of gamma indicates that nearby support vectors have a stronger influence, while a smaller value indicates that distant support vectors have a weaker influence. The choice of gamma is particularly important for RBF kernels.

[ ]:
```python
# Use SVR for regression
pipe_svm = Pipeline([('preprocessor', preprocessor),
                     ('regressor', SVR())])

# Fine-tuning hyperparameters
param_kernel = ['linear', 'rbf', 'sigmoid', 'poly']
param_C = [0.01, 0.1, 1.0, 10.0]
param_gamma = [0.001, 0.01, 0.1, 1.0]
param_grid = [{'regressor__C': param_C,
               'regressor__kernel': param_kernel,
               'regressor__gamma': param_gamma}]

# Use a regression-specific scoring metric
reg_gs = GridSearchCV(
    estimator=pipe_svm,
    param_grid=param_grid,
    scoring="neg_mean_squared_error",
    return_train_score=True,
    cv=5)

reg_gs = reg_gs.fit(X_train, y_train)
```

[ ]:
```python
# Get scoring values of ref_gs
recording_df = pd.DataFrame({'kernel':np.ma.getdata(reg_gs.
 ↪cv_results_['param_regressor__kernel']),
                             'C':np.ma.getdata(reg_gs.
 ↪cv_results_['param_regressor__C']),
                             'gamma':np.ma.getdata(reg_gs.
 ↪cv_results_['param_regressor__gamma']),
                             'mean_test':['{:f}'.format(item) for item in␣
 ↪-reg_gs.cv_results_['mean_test_score'].round(6)],
                             'mean_train':['{:f}'.format(item) for item in␣
 ↪-reg_gs.cv_results_['mean_train_score'].round(6)],
                             'ranking':reg_gs.cv_results_['rank_test_score']},
                            index=reg_gs.cv_results_['params'])\
                                .sort_values(by=['kernel', 'C', 'gamma'],␣
 ↪ascending=[True, True, True])
recording_df.to_csv('record_hyperparameters_svm.csv', encoding='utf-8-sig')
```

Since hyperparameter-tuning is time-consuming, I have saved the table that records the **Mean Squared Error** (**MSE**) data results to a `record_hyperparameters_svm.csv` file. By doing so, I can retrieve the saved data later on to explore the results, saving my time in case of interruptions during the project. This way, I can avoid redoing the hyperparameter tuning process.

```
[ ]: # Read recording data from record_hyperparameters_svm.csv
recording_save_df = pd.read_csv('record_hyperparameters_svm.csv', sep=',',␣
 ↪engine='python', encoding='utf-8', index_col=[0])
recording_save_df
```

```
[ ]:                                                     kernel      C  gamma  \
{'regressor__C': 0.01, 'regressor__gamma': 0.00…   linear   0.01  0.001
{'regressor__C': 0.01, 'regressor__gamma': 0.01…   linear   0.01  0.010
{'regressor__C': 0.01, 'regressor__gamma': 0.1,…   linear   0.01  0.100
{'regressor__C': 0.01, 'regressor__gamma': 1.0,…   linear   0.01  1.000
{'regressor__C': 0.1, 'regressor__gamma': 0.001…   linear   0.10  0.001
…                                                     …      …     …
{'regressor__C': 1.0, 'regressor__gamma': 1.0, …  sigmoid   1.00  1.000
{'regressor__C': 10.0, 'regressor__gamma': 0.00…  sigmoid  10.00  0.001
{'regressor__C': 10.0, 'regressor__gamma': 0.01…  sigmoid  10.00  0.010
{'regressor__C': 10.0, 'regressor__gamma': 0.1,…  sigmoid  10.00  0.100
{'regressor__C': 10.0, 'regressor__gamma': 1.0,…  sigmoid  10.00  1.000

                                                     mean_test  \
{'regressor__C': 0.01, 'regressor__gamma': 0.00…  3.433600e-02
{'regressor__C': 0.01, 'regressor__gamma': 0.01…  3.433600e-02
{'regressor__C': 0.01, 'regressor__gamma': 0.1,…  3.433600e-02
{'regressor__C': 0.01, 'regressor__gamma': 1.0,…  3.433600e-02
{'regressor__C': 0.1, 'regressor__gamma': 0.001…  3.418000e-02
…                                                          …
{'regressor__C': 1.0, 'regressor__gamma': 1.0, …  2.418177e+04
{'regressor__C': 10.0, 'regressor__gamma': 0.00…  3.433300e-02
{'regressor__C': 10.0, 'regressor__gamma': 0.01…  4.741948e+00
{'regressor__C': 10.0, 'regressor__gamma': 0.1,…  6.242521e+04
{'regressor__C': 10.0, 'regressor__gamma': 1.0,…  2.398410e+06

                                                    mean_train  ranking
{'regressor__C': 0.01, 'regressor__gamma': 0.00…  3.288000e-02       10
{'regressor__C': 0.01, 'regressor__gamma': 0.01…  3.288000e-02       10
{'regressor__C': 0.01, 'regressor__gamma': 0.1,…  3.288000e-02       10
{'regressor__C': 0.01, 'regressor__gamma': 1.0,…  3.288000e-02       10
{'regressor__C': 0.1, 'regressor__gamma': 0.001…  3.055900e-02        4
…                                                          …        …
{'regressor__C': 1.0, 'regressor__gamma': 1.0, …  2.503934e+04       62
{'regressor__C': 10.0, 'regressor__gamma': 0.00…  3.288200e-02        9
{'regressor__C': 10.0, 'regressor__gamma': 0.01…  4.718597e+00       58
{'regressor__C': 10.0, 'regressor__gamma': 0.1,…  6.056644e+04       63
{'regressor__C': 10.0, 'regressor__gamma': 1.0,…  2.503694e+06       64

[64 rows x 6 columns]
```

Let's examine the MSE value of each kernel through the relationship between `C` and `gamma`.

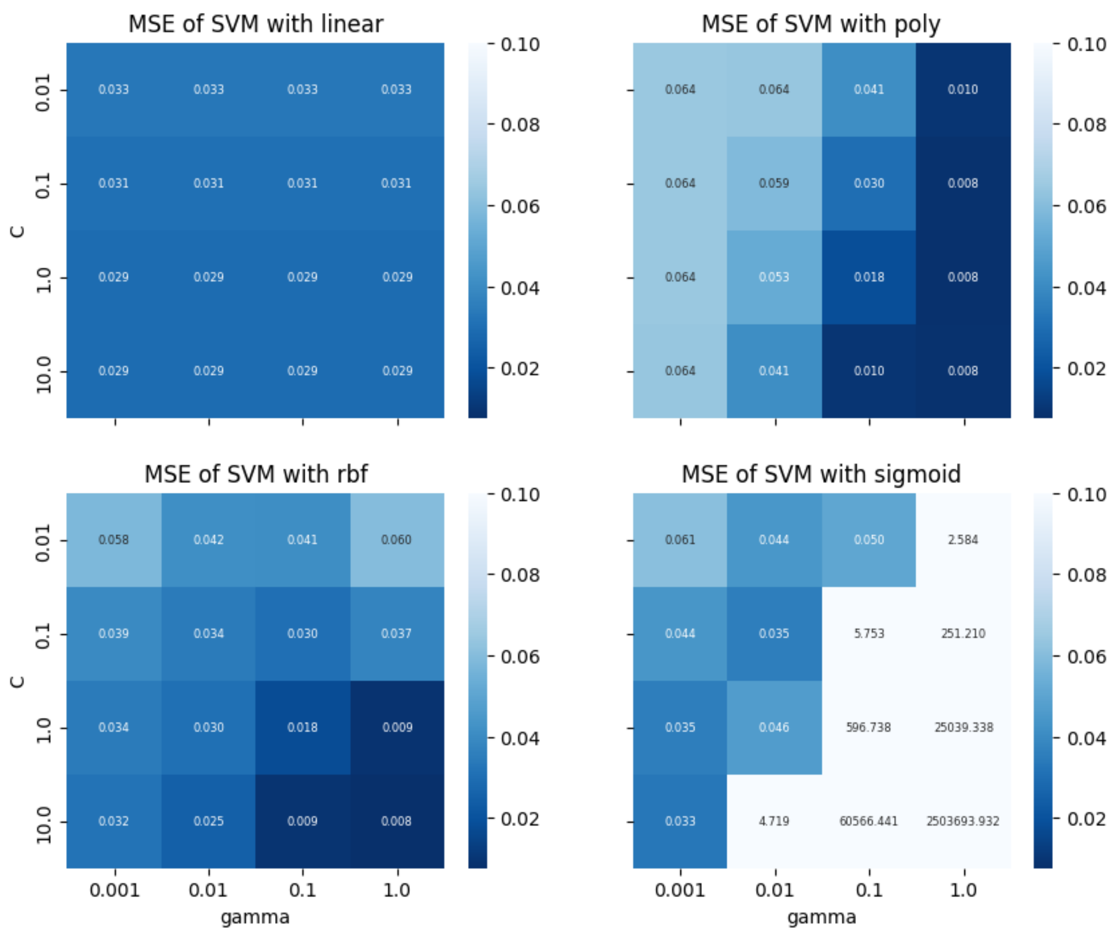
```
[ ]: heatmap_df = recording_save_df.copy().reset_index().drop(columns=['index',␣
     ↪'mean_test', 'ranking']).set_index(['kernel'])
     min_range = heatmap_df['mean_train'].min()

     # Set up the matplotlib figure
     fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 8), sharex=True,␣
     ↪sharey=True)

     # Plot heatmap each kernel
     for index, kernel in enumerate(heatmap_df.index.unique()):
         ax = axes[index // 2, index % 2]
         data_plot = heatmap_df.loc[kernel].copy()\
             .reset_index().drop(columns=['kernel'])\
             .set_index(['C', 'gamma'])
         data_plot = data_plot['mean_train'].unstack('gamma').rename_axis(None,␣
     ↪axis=1)
         data_plot.index.name = None
         sns.heatmap(data_plot, annot=True, ax=ax,
                     vmin=min_range, vmax=0.1, fmt='.3f',
                     cmap='Blues_r', annot_kws={"size": 6})
         ax.set_title('MSE of SVM with ' + kernel)
         if index % 2 == 0:
             ax.set_ylabel('C')

         if index // 2 == 1:
             ax.set_xlabel('gamma')

     plt.show()
```



It appears that the `sigmoid` kernel have lighter colors compared to the `linear`, `rbf` and `poly` kernels. When using the `sigmoid` kernel, the MSE tends to increase as the hyperparameter gets larger. This indicates that when the penalty coefficient `C` is higher, the SVM with sigmoid tends to suffer from ***underfitting***.

Let's check if ***overfitting*** occurs with the kernels of the SVM model.

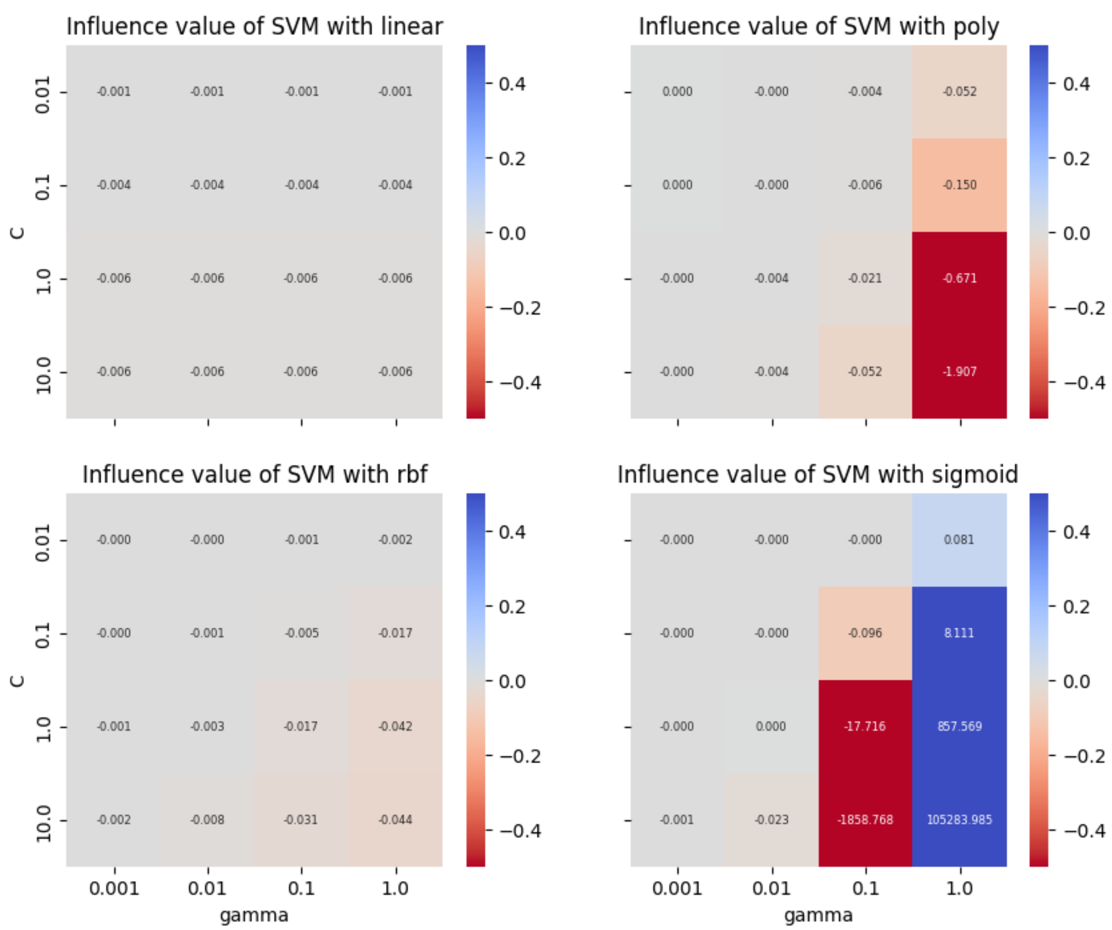
[ ]:
```
bar_df = pd.DataFrame({'kernel': recording_save_df['kernel'],
                       'C': recording_save_df['C'],
                       'gamma': recording_save_df['gamma'],
                       'influence': recording_save_df['mean_train'] -␣
↪recording_save_df['mean_test']},
                       index=recording_save_df.index)
bar_df = bar_df.reset_index().drop(columns=['index']).set_index(['kernel'])

# Set up the matplotlib figure
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 8), sharex=True,␣
↪sharey=True)
```

```python
# Plot heatmap each kernel
for index, kernel in enumerate(bar_df.index.unique()):
    ax = axes[index // 2, index % 2]
    data_plot = bar_df.loc[kernel].copy()\
        .reset_index().drop(columns=['kernel'])\
        .set_index(['C', 'gamma'])
    data_plot = data_plot['influence'].unstack('gamma').rename_axis(None, axis=1)
    data_plot.index.name = None
    sns.heatmap(data_plot, annot=True, ax=ax,
                vmin=-0.5, vmax=0.5, fmt='.3f',
                cmap='coolwarm_r', annot_kws={"size": 6})
    ax.set_title('Influence value of SVM with ' + kernel)
    if index % 2 == 0:
        ax.set_ylabel('C')

    if index // 2 == 1:
        ax.set_xlabel('gamma')

plt.show()
```



The **influence** value has a specific meaning: - When the value is positive (in blue), it indicates that the MSE of the training dataset is higher than the MSE of the validation dataset. It means that the model performs better on the validation dataset compared to the training dataset. - When the value is negative (in red), it indicates that the MSE of the training dataset is lower than the MSE of the validation dataset. It means that the model performs better on the training dataset compared to the validation dataset.

It has been observed that the difference between the MSE of the training dataset and validation dataset is larger for SVM with `poly` and `sigmoid` kernels when the hyperparameter value is higher.

Besides, for SVM with `poly` kernel, the `influence` value tends to be redder, which indicates that as the value of the penalty coefficient `C` increases, the model performs better on the training dataset but not on the validation dataset. This suggests that SVM with `poly` kernel is prone to ***overfitting***.

After conducting `cross-validation` and `hyperparameter-tuning`, `GridSearchCV()` gave the best hyperparameter on my dataset:

[ ]:
```
# # If you ran Grid Search with cv=5, you can print the best parameters and the␣
↪best score by code below
# print('SVR: Grid Search with cv=5')
# print('- Kernel: ', reg_gs.best_params_['regressor__kernel'])
# print('- C: ', reg_gs.best_params_['regressor__C'])
# print('- Gamma: ', reg_gs.best_params_['regressor__gamma'])
# print('- Validation MSE: %.6f' % -reg_gs.best_score_)

# Otherwise, you can print the best parameters and the best score by code below
print('SVR: Grid Search with cv=5')
print('- Kernel: ', recording_save_df[recording_save_df['ranking'] ==␣
↪1]['kernel'].values[0])
print('- C: ', recording_save_df[recording_save_df['ranking'] == 1]['C'].
↪values[0])
print('- Gamma: ', recording_save_df[recording_save_df['ranking'] ==␣
↪1]['gamma'].values[0])
print('- Validation MSE: %.6f' % recording_save_df[recording_save_df['ranking']␣
↪== 1]['mean_test'].values[0])
```

```
SVR: Grid Search with cv=5
- Kernel:  rbf
- C:  10.0
- Gamma:  0.01
- Validation MSE: 0.033245
```

1.4 4. Evaluate SVM model

[ ]:
```
# SVM model with best parameters on testing dataset
kernel_best = recording_save_df[recording_save_df['ranking'] == 1]['kernel'].
 ↪values[0]
C_best = recording_save_df[recording_save_df['ranking'] == 1]['C'].values[0]
gamma_best = recording_save_df[recording_save_df['ranking'] == 1]['gamma'].
 ↪values[0]

# Create the SVM model
modelSVM = SVR(kernel=kernel_best, C=C_best, gamma=gamma_best)
pipe_svm = Pipeline([('preprocessor', preprocessor),
                     ('regressor', modelSVM)])

# Training the model on the training dataset
pipe_svm.fit(X_train, y_train)

# Evaluate the model on the testing dataset
print('MSE of SVM: %.6f' % mean_squared_error(y_test, pipe_svm.predict(X_test)))
```

```
MSE of SVM: 0.032113
```