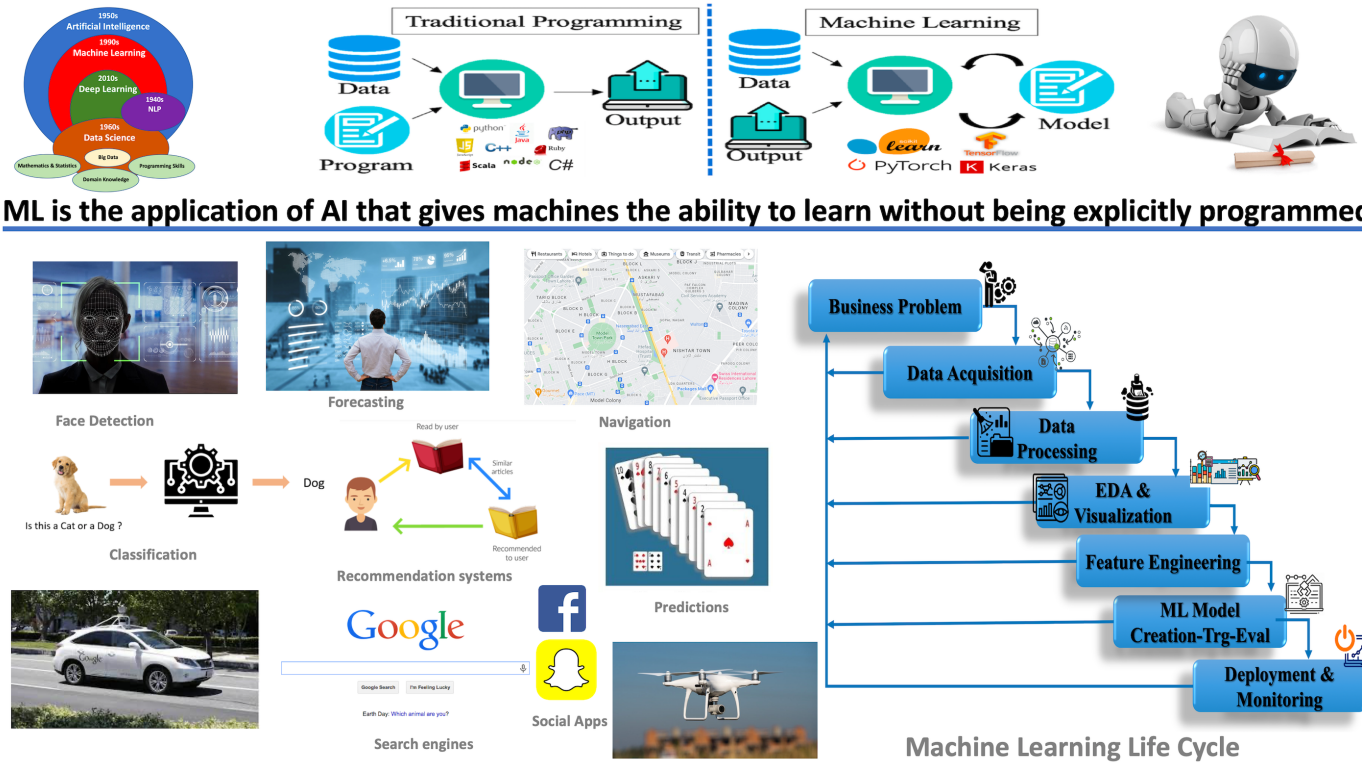
Department of Data Science

Course: Tools and Techniques for Data Science

Instructor: Muhammad Arif Butt, Ph.D.

Lecture 6.15 (Hyperparameter Tuning using Grid and Random Search)

Open in Colab
(https://colab.research.google.com/github/arifpucit/data-science/blob/master/Section-4-Mathematics-for-Data-Science/Lec-4.1(Descriptive-Statistics).ipynb)



Learning agenda of this notebook

A Recap of What we have done so far?

1. **Data Acquisition**

    - Libraries Built-in Datasets
        - Seaborn: (iris, titanic, tips, flights, panguins, car_crashes)
        - Scikit-learn: (iris, digits, diabetes, Bostan housing)
        - NLTK: (movie-reviews, product_reviews, twitter_samples, gutenerg, genesis, timeit, voice, wordnet, sentiword)
    - Use Public Dataset Repositories:
        - https://www.kaggle.com/ (https://www.kaggle.com/)
        - https://data.gov/ (https://data.gov/)
        - https://archive.ics.uci.edu/ml/index.php (https://archive.ics.uci.edu/ml/index.php)
        - https://github.com/ (https://github.com/)
    - Use Company's Database: (SQL, NOSQL, Data warehouse, Data lake)
    - Generate your own Datasets:
        - Use Web scraping or Web API
        - IoT Devices
        - Crowd Sourcing (Amazon Mechanical Turk, Lionbridge AI)
        - Data Augmentation

2. **Data Preprocessing and Feature Engineering**

    - Detecting and handling outliers
    - Missing values Imputation
    - Encoding Categorical Features
    - Feature Scaling
    - Extracting Information
    - Combining Information

3. **EDA and Visualization**

    - Matplotlib
    - Seaborn
    - Plotly

4. **Machine Learning Model Creation, Training and Evaluation**

    - Choosing the right estimator
    - Fit or train the model on training data
    - Do predictions on the test data
    - Evaluate the Model using simple train-test split and using Cross Validation Techniques

Today's Agenda

- **Improve the Baseline Model**

    - **From a `Data Perspective:`**
        - Could we collect more data samples?
        - Could we improve our data?
            - Add more relevent features (if possible)
            - Drop irrelevant features (if any)

    - **From a `Model Perspective:`**
        - Choose some other estimator or model
        - Perform Hyperparameter Tuning of the current model

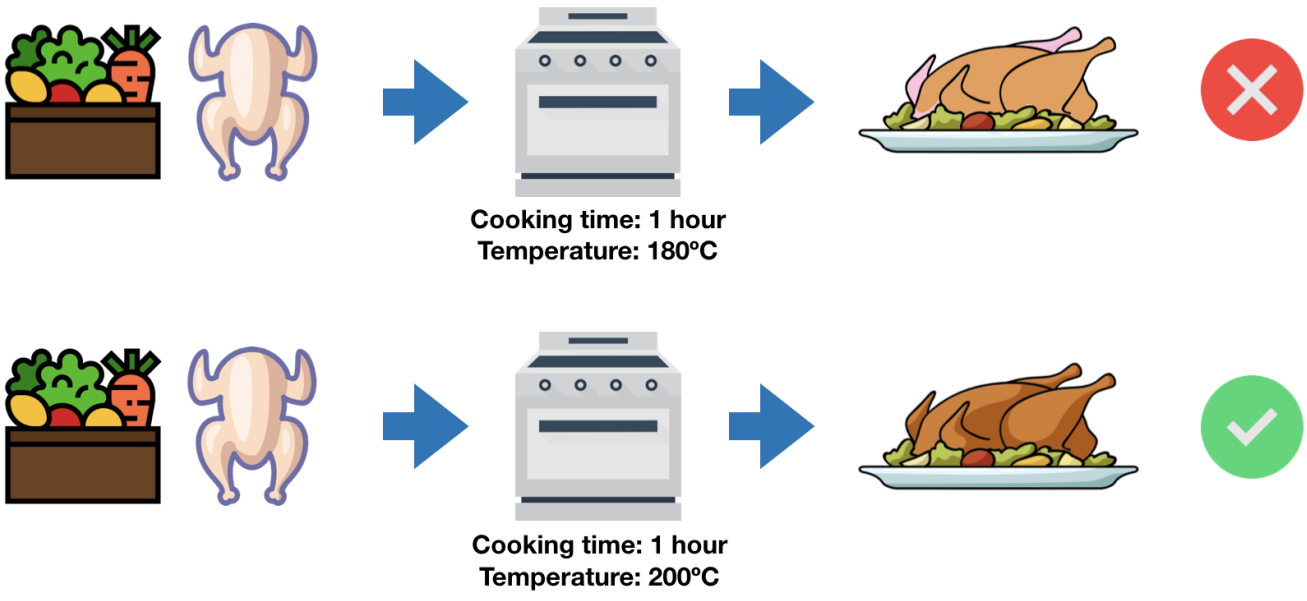
- **Model Parameters vs Hyperparameters**
- **Hyperparameters of Ridge Regression**
- **Ways to find the best hyperparameters for your model**
  - By hand manually using a loop
    - Using simple Train-Test split

1. Overview of Hyperparameters

a. Model Parameters vs Hyperparameters

Ser	Hyperparameters	Model Parameters
1	Hyperparameters are set manually by ML engineer/practitioner prior to the start of the model's training.	Model parameters are learnt by the learning algorithm during the training phase.
2	Hyperparameters are used to optimize machine learning model.	Model's parameters are later used for prediction.
3	They are internal to the model.	Thy are external to the model.
4	Examples: Value of K in KNN, learning rate for training a neural network, number of trees in RandomForrest.	Examples: Coefficients in a Linear or Logistic regression, support vectors in a support vector machine, and weights in an artificial neural network.

b. An Intuition of Hyperparameter Tuning



c. How to find Hyperparameters of a Model?

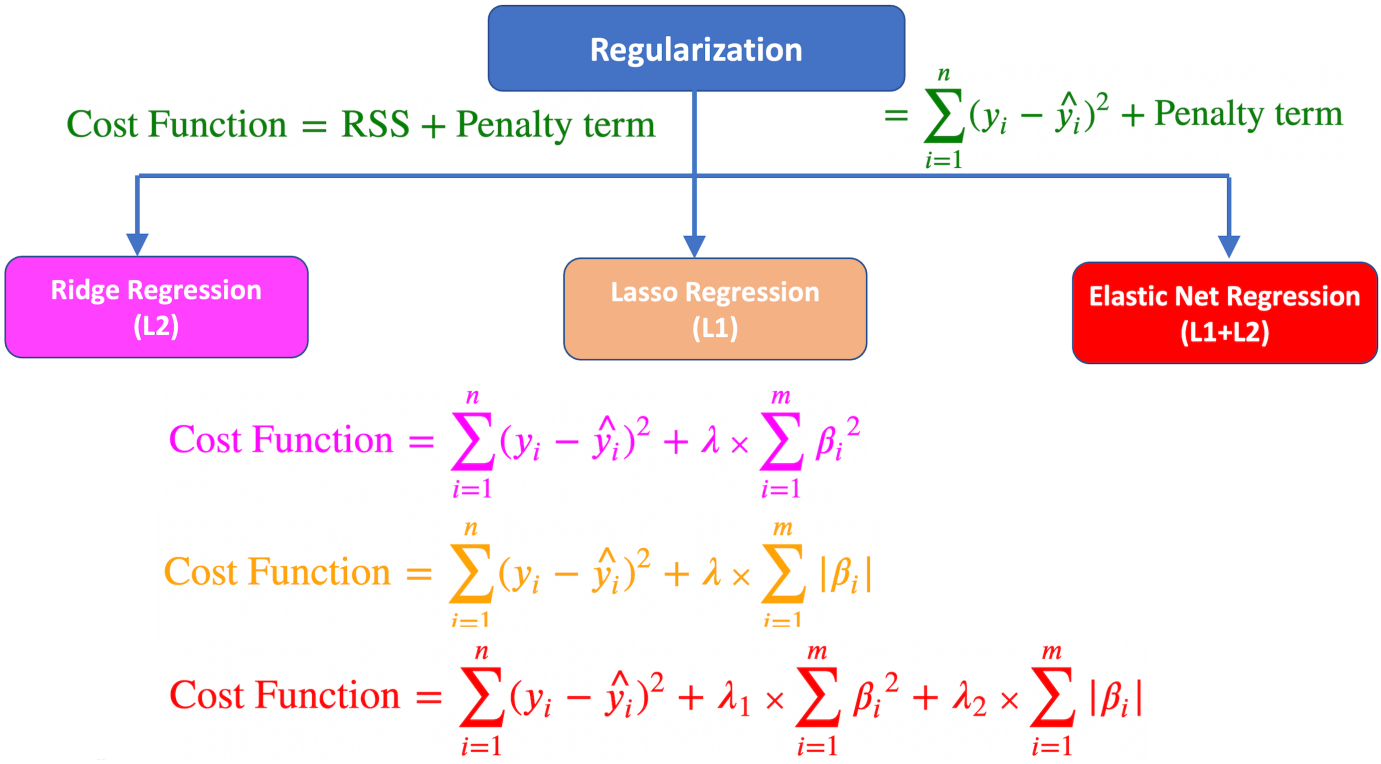
- Visit: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html)
- Research Problem: Understand and implement and compare different `solvers` and try designing a new solver.

In [1]:

```
from sklearn.linear_model import Ridge
model = Ridge()
model.get_params()
```

Out[1]:

```
{'alpha': 1.0,
 'copy_X': True,
 'fit_intercept': True,
 'max_iter': None,
 'normalize': 'deprecated',
 'positive': False,
 'random_state': None,
 'solver': 'auto',
 'tol': 0.001}
```



d. How to find best hyperparameters for our model

Hyperparameter Tuning Techniques:

- `By Hand:` Select the hyperparameters values based on intuition/experience/guessing, train the model with the hyperparameters, and score on the validation data. Repeat process until you run out of patience or are satisfied with the results.
- `GridSearchCV:` Set up a grid of hyperparameter values and for each combination, train a model and score on the validation data. In this approach, every single combination of hyperparameters values is tried which can be very inefficient!
- `RandomizedSearchCV:` Set up a grid of hyperparameter values and select random combinations to train the model and score. The number of search iterations is set based on time/resources.

Hypertuning Steps

- Make a list of different hyperparameters based on the problem in hand. If there are more than one hyperparameter then make grid with different combination of parameters
- Fit all of them separately to the model.

- Note down the model performance
- Choose the best performing one
- Always use cross validation technique for hyperparameter tuning to avoid the model overfitting on test data.

3. Find Optimized Hyperparameter By Hand

a. Find Optimized Hyperparameter by Trial and Error

- Approach 1: Use train_test_split and manually tune parameters by trial and error

Load the Advertising Dataset:

In [2]:

```
import pandas as pd
import numpy as np
df = pd.read_csv("datasets/advertising4D.csv")
df
```

Out[2]:

	TV	radio	newspaper	sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9
...	...	...	...	...
195	38.2	3.7	13.8	7.6
196	94.2	4.9	8.1	9.7
197	177.0	9.3	6.4	12.8
198	283.6	42.0	66.2	25.5
199	232.1	8.6	8.7	13.4

200 rows × 4 columns

Do a Train-Test Split:

In [3]:

```
from sklearn.model_selection import train_test_split

X = df.drop('sales', axis=1)
y = df['sales']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
```

Scale the Data:

In [4]:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Instentiate the model `Ridge()` , Train and Evaluate

In [5]:

```
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score

# Instentiate a model and fit it to training data with default hyperparameters
model = Ridge()
model.fit(X_train, y_train)

# Evaluate
r2 = r2_score(y_test, model.predict(X_test))
r2 = model.score(X_test, y_test)
print("R2 Score: ", r2)
```

```
R2 Score:  0.8721902529930639
```

Adjust Parameters `Ridge(alpha=1.0, solver='auto')` , Train and Re-evaluate

In [6]:

```
# Adjust values of hyperparameters
model = Ridge(alpha=1.0, solver='auto')
# Retrain
model.fit(X_train, y_train)
# Re-evaluate
r2 = r2_score(y_test, model.predict(X_test))
r2 = model.score(X_test, y_test)
print("R2 Score: ", r2)
```

```
R2 Score:  0.8721902529930639
```

Adjust Parameters `Ridge(alpha=1000, solver='auto')` , Train and Re-evaluate

In [7]:

```
# Adjust values of hyperparameters
model = Ridge(alpha=1000, solver='auto')
# Retrain
model.fit(X_train, y_train)
# Re-evaluate
r2 = r2_score(y_test, model.predict(X_test))
r2 = model.score(X_test, y_test)
print("R2 Score: ", r2)
```

```
R2 Score:  0.2585536265498327
```

Adjust Parameters `Ridge(alpha=100, solver='lsqr')` , Train and Re-evaluate

In [8]:

```
# Adjust values of hyperparameters
model = Ridge(alpha=100, solver='lsqr')
# Retrain
model.fit(X_train, y_train)

# Re-evaluate
r2 = r2_score(y_test, model.predict(X_test))
r2 = model.score(X_test, y_test)
print("R2 Score: ", r2)
```

```
R2 Score:  0.7770218673478944
```

- Above approach is tiresome and very manual.
- Moreover, you get a different score each time you train the model on a different split

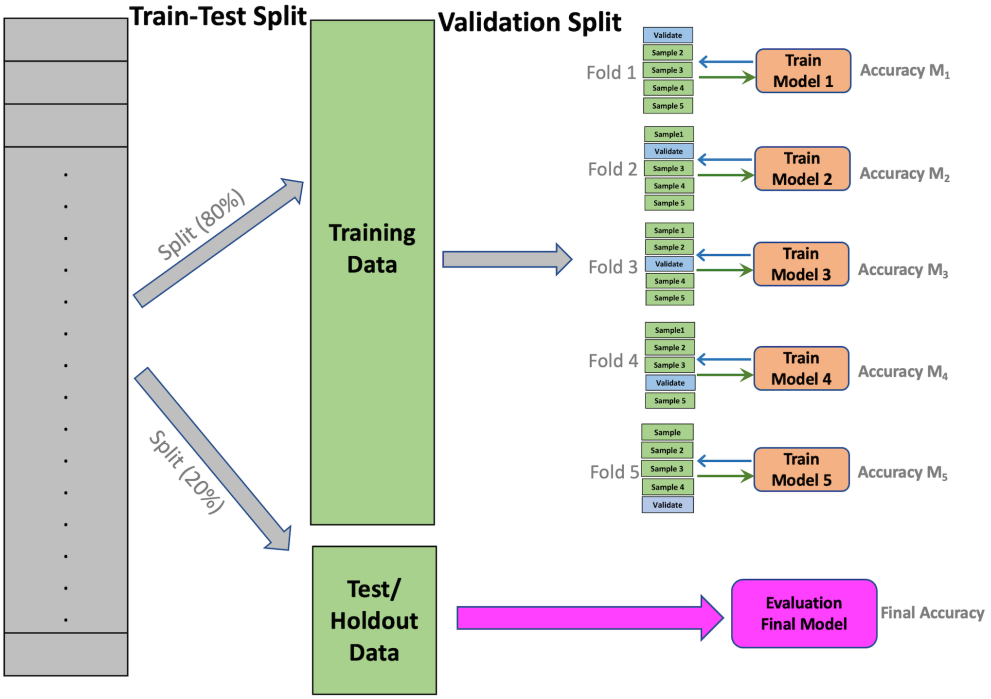
In [9]:

```
df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']

# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru

# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Specify values of hyperparameters
model = Ridge(alpha=100, solver='lsqr')

# Train
model.fit(X_train, y_train)

# Evaluate
r2 = r2_score(y_test, model.predict(X_test))
r2 = model.score(X_test, y_test)
print("R2 Score: ", r2)
```

```
R2 Score:  0.764051601636525
```

- Each time you execute the above code cell, you get different R2 scores. None of these R2 scores is the true representation of the entire dataset. This is because of the train-test split. The limitation is each time the model is tested on different 20% of the test set. The solution is to use some Cross-Validation technique.

Use of `cross_val_score()` Method (KFold)



- Regularization parameters
- Value of K in KNN
- Learning Rate in optimization algos like Gradient Descent
- Number of iterations or epochs
- Number of branches in DecisionTree
- Number of clusters in clustering task
- Choice of activation function in NN
- Number of hidden layers in Neural Network

In [10]:

```
from sklearn.model_selection import cross_val_score
df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']
# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)


#cross_val_score() is a cross validation method that trains and tests a model over m
cv_scores = cross_val_score(
                            estimator = Ridge(alpha=100, solver='lsqr'),
                            X = X_train,
                            y = y_train,
                            scoring = 'r2',
                            cv = 5 #Instead of integer value you can pass KFold obec
                           )
print("R2 scores for all the folds: ", cv_scores)
print("Mean R2 score: ", np.mean(cv_scores))

```

```
R2 scores for all the folds:  [0.70994653 0.75849807 0.81715825 0.66597145
0.71846425]
Mean R2 score:  0.7340077066411028
```

b. Find Optimized Hyperparameter using For Loops

(i) Use Simple Train-Test-Split

In [11]:

```
df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']
# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)


alpha_list = [1000, 100, 10, 5, 1, 0.8, 0.5, 0.2]
solver_list = ['lsqr', 'svd']

for a in alpha_list:
    for s in solver_list:
        model = Ridge(alpha=a, solver=s)# Instentiate model each time with different
        model.fit(X_train, y_train)     # Fit the model to training data
        r2 = model.score(X_test, y_test)  # Evaluate by calculating R2 Score
        print(a, s,":",r2)
```

```
1000 lsqr : 0.2575040659257757
1000 svd : 0.2575257813818227
100 lsqr : 0.7986674628544999
100 svd : 0.7986674628544999
10 lsqr : 0.8937868349191451
10 svd : 0.8937868349191451
5 lsqr : 0.8929082989595012
5 svd : 0.8929082989595012
1 lsqr : 0.8910509252507594
1 svd : 0.8910509252507595
0.8 lsqr : 0.8909281604323819
0.8 svd : 0.890928160432382
0.5 lsqr : 0.8907383931830093
0.5 svd : 0.8907383931830093
0.2 lsqr : 0.8905418191554166
0.2 svd : 0.8905418191554166
```

Limitation:

- Model evaluation is done on just 20% of the data.
- After finalizing the best combination of hyperparameters, we are not left with any unseen data on which we can do the final evaluation of the model with the best hyperparameters

(ii) Use Cross Validation

In [12]:

```
df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']
# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)


alpha_list = [1000, 100, 10, 5, 1, 0.8, 0.5, 0.2]
solver_list = ['lsqr', 'svd']

for a in alpha_list:
    for s in solver_list:
        cv_scores = cross_val_score(Ridge(alpha=a, solver=s ), X_train, y_train, sco
        print(a, s,":",np.mean(cv_scores))
```

```
1000 lsqr : 0.11472999499702023
1000 svd : 0.11472749867031158
100 lsqr : 0.697575096256928
100 svd : 0.697575096256928
10 lsqr : 0.8742327018136052
10 svd : 0.8742327018136052
5 lsqr : 0.8773353814464739
5 svd : 0.8773353814464739
1 lsqr : 0.8779427642965087
1 svd : 0.877942764296509
0.8 lsqr : 0.8779217675681077
0.8 svd : 0.8779217675681077
0.5 lsqr : 0.8778803392675361
0.5 svd : 0.8778803392675361
0.2 lsqr : 0.8778268171366636
0.2 svd : 0.8778268171366636
```

4. Find Optimized Hyperparameters using `GridSearchCV()`

- Grid search is a method for hyperparameter optimization that involves specifying a list of values for each hyperparameter that you want to optimize, and then training a model for each combination of these values.
- Basically, we divide the domain of the hyperparameters into a discrete grid.
- Then, we try every combination of values of this grid, calculating some performance metrics using cross-validation.
- The point of the grid that maximizes the average value in cross-validation, is the optimal combination of values for the hyperparameters.
- Additionally, it is recommended to use cross-validation when performing hyperparameter optimization. This can provide a more accurate estimate of the model's performance and help to avoid overfitting.

In [13]:

```
from sklearn.model_selection import GridSearchCV

df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']
# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

#Dictionary with parameters names (`str`) as keys and lists of parameter settings to
params = { 'alpha': [1000, 100, 10, 1, 0.5],
           'solver': ['lsqr', 'svd'] }

gs = GridSearchCV(estimator=Ridge(),
                  param_grid=params,
                  scoring='r2',
                  cv=5,
                  n_jobs=-1)

gs.fit(X_train, y_train)

print("Best Score: ", gs.best_score_)
print("Best Score: ", gs.best_params_)
```

```
Best Score:  0.8934622964085636
Best Score:  {'alpha': 0.5, 'solver': 'lsqr'}
```

In [14]:

```
#Check out the attributes and methods of this trained gs object
print(dir(gs))
```

```
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstat
e__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__l
t__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclas
shook__', '__weakref__', '_abc_impl', '_check_feature_names', '_check_n_fea
tures', '_check_refit_for_multimetric', '_estimator_type', '_format_result
s', '_get_param_names', '_get_tags', '_more_tags', '_pairwise', '_repr_html
_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_run_
search', '_select_best_index', '_validate_data', 'best_estimator_', 'best_i
ndex_', 'best_params_', 'best_score_', 'classes_', 'cv', 'cv_results_', 'de
cision_function', 'error_score', 'estimator', 'fit', 'get_params', 'inverse
_transform', 'multimetric_', 'n_features_in_', 'n_jobs', 'n_splits_', 'para
m_grid', 'pre_dispatch', 'predict', 'predict_log_proba', 'predict_proba',
'refit', 'refit_time_', 'return_train_score', 'score', 'score_samples', 'sc
orer_', 'scoring', 'set_params', 'transform', 'verbose']
```

In [15]:

```
# cv_results_ attribute of gs is a dictionary object containing following informatio
gs.cv_results_
```

Out[15]:

```
{'mean_fit_time': array([9.69123840e-04, 1.56237230e+00, 6.07395172e-04, 1.
59132328e+00,
        4.90617752e-04, 3.52191925e-04, 6.51884079e-04, 1.15833282e-03,
        7.34519958e-04, 9.11951065e-04]),
 'std_fit_time': array([7.05208787e-05, 2.29147087e-02, 8.98225788e-05, 2.3
0979328e-02,
        5.49639170e-05, 2.91200357e-05, 3.05959039e-04, 4.60511138e-04,
        2.57739566e-04, 5.58453656e-04]),
 'mean_score_time': array([0.00029221, 0.00043378, 0.00023098, 0.00049858,
0.00019703,
        0.00018539, 0.00021052, 0.00039701, 0.00021396, 0.00068874]),
 'std_score_time': array([1.71857298e-05, 4.22674116e-05, 1.96699807e-05,
1.84395426e-04,
        6.80491901e-06, 5.33887291e-06, 2.41996619e-05, 1.81005363e-04,
        5.66333212e-06, 9.03057966e-04]),
 'param_alpha': masked_array(data=[1000, 1000, 100, 100, 10, 10, 1, 1, 0.5,
0.5],
              mask=[False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_solver': masked_array(data=['lsqr', 'svd', 'lsqr', 'svd', 'lsqr',
'svd', 'lsqr',
                    'svd', 'lsqr', 'svd'],
              mask=[False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 1000, 'solver': 'lsqr'},
  {'alpha': 1000, 'solver': 'svd'},
  {'alpha': 100, 'solver': 'lsqr'},
  {'alpha': 100, 'solver': 'svd'},
  {'alpha': 10, 'solver': 'lsqr'},
  {'alpha': 10, 'solver': 'svd'},
  {'alpha': 1, 'solver': 'lsqr'},
  {'alpha': 1, 'solver': 'svd'},
  {'alpha': 0.5, 'solver': 'lsqr'},
  {'alpha': 0.5, 'solver': 'svd'}],
 'split0_test_score': array([0.1757953 , 0.17577465, 0.70221764, 0.7022176
4, 0.84379969,
        0.84379969, 0.8445015 , 0.8445015 , 0.84431631, 0.84431631]),
 'split1_test_score': array([0.10282152, 0.10284861, 0.69006285, 0.6900628
5, 0.8820991 ,
        0.8820991 , 0.89026034, 0.89026034, 0.89049015, 0.89049015]),
 'split2_test_score': array([0.19564963, 0.19564403, 0.70063813, 0.7006381
3, 0.85671034,
        0.85671034, 0.86168909, 0.86168909, 0.86176479, 0.86176479]),
 'split3_test_score': array([0.19833316, 0.19834319, 0.77226971, 0.7722697
1, 0.92533988,
        0.92533988, 0.92412466, 0.92412466, 0.92377372, 0.92377372]),
 'split4_test_score': array([0.21636479, 0.21636366, 0.76928643, 0.7692864
3, 0.93884092,
        0.93884092, 0.94669901, 0.94669901, 0.94696651, 0.94696651]),
 'mean_test_score': array([0.17779288, 0.17779483, 0.72689495, 0.72689495,
0.88935799,
        0.88935799, 0.89345492, 0.89345492, 0.8934623 , 0.8934623 ]),
 'std_test_score': array([0.03963144, 0.03962171, 0.03608573, 0.03608573,
0.03724883,
        0.03724883, 0.03792434, 0.03792434, 0.03797467, 0.03797467]),
 'rank_test_score': array([10,  9,  7,  7,  5,  5,  3,  3,  1,  1], dtype=i
nt32)}
```

In [16]:

```
# For better readability let us display the dictionary object as a dataframe
df = pd.DataFrame(gs.cv_results_)
df
```

Out[16]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	param_solver	params	s
0	0.000969	0.000071	0.000292	0.000017	1000	lsqr	{'alpha': 1000, 'solver': 'lsqr'}	
1	1.562372	0.022915	0.000434	0.000042	1000	svd	{'alpha': 1000, 'solver': 'svd'}	
2	0.000607	0.000090	0.000231	0.000020	100	lsqr	{'alpha': 100, 'solver': 'lsqr'}	
3	1.591323	0.023098	0.000499	0.000184	100	svd	{'alpha': 100, 'solver': 'svd'}	
4	0.000491	0.000055	0.000197	0.000007	10	lsqr	{'alpha': 10, 'solver': 'lsqr'}	
5	0.000352	0.000029	0.000185	0.000005	10	svd	{'alpha': 10, 'solver': 'svd'}	
6	0.000652	0.000306	0.000211	0.000024	1	lsqr	{'alpha': 1, 'solver': 'lsqr'}	
7	0.001158	0.000461	0.000397	0.000181	1	svd	{'alpha': 1, 'solver': 'svd'}	
8	0.000735	0.000258	0.000214	0.000006	0.5	lsqr	{'alpha': 0.5, 'solver': 'lsqr'}	
9	0.000912	0.000558	0.000689	0.000903	0.5	svd	{'alpha': 0.5, 'solver': 'svd'}	

In [17]:

```
df.loc[:, ['param_alpha', 'param_solver', 'mean_test_score']]
```

Out[17]:

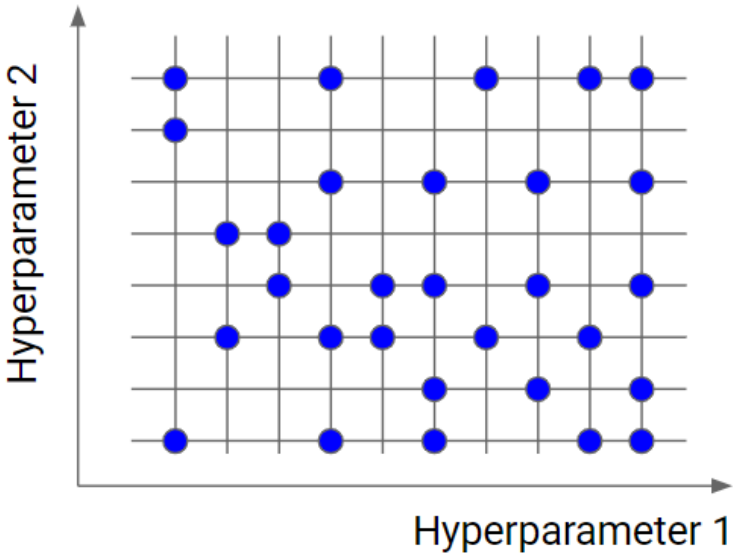
	param_alpha	param_solver	mean_test_score
0	1000	lsqr	0.177793
1	1000	svd	0.177795
2	100	lsqr	0.726895
3	100	svd	0.726895
4	10	lsqr	0.889358
5	10	svd	0.889358
6	1	lsqr	0.893455
7	1	svd	0.893455
8	0.5	lsqr	0.893462
9	0.5	svd	0.893462

Limitations of GridSearchCV:

- Grid search is an exhaustive algorithm that spans all the combinations, so it can actually find the best point in the domain.
- For that it trains a separate model for every combination of hyperparameter values.
- Suppose you have million of data points in your dataset and a bundle of hyperparameters and their values to tune. In that case the grid will be multidimensional and the algorithm will become computationally expensive as well as time consuming.

5. Find Optimized Hyperparameters using `RandomizedSearchCV()`

- Random search is similar to grid search, but instead of using all the points in the grid, it tests only a randomly selected subset of these points.
- The smaller this subset, the faster but less accurate the optimization. The larger this dataset, the more accurate the optimization but the closer to a grid search.
- Random search is a very useful option when you have several hyperparameters with a fine-grained grid of values.



In [18]:

```
from sklearn.model_selection import RandomizedSearchCV

df = pd.read_csv("datasets/advertising4D.csv")
X = df.drop('sales', axis=1)
y = df['sales']
# Do a train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Tru
# SCALE DATA
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

#Dictionary with parameters names (`str`) as keys and lists of parameter settings to
params = { 'alpha': [1000, 100, 10, 1, 0.5],
           'solver': ['lsqr', 'svd'] }

rs = RandomizedSearchCV(estimator=Ridge(),
                  param_distributions=params,
                  n_iter=6, #Number of parameter combinations to try.
                  scoring='r2',
                  cv=5,
                  n_jobs=-1)

rs.fit(X_train, y_train)

print("Best Score: ", rs.best_score_)
print("Best Score: ", rs.best_params_)
```

```
Best Score:  0.8871952696785751
Best Score:  {'solver': 'svd', 'alpha': 1}
```