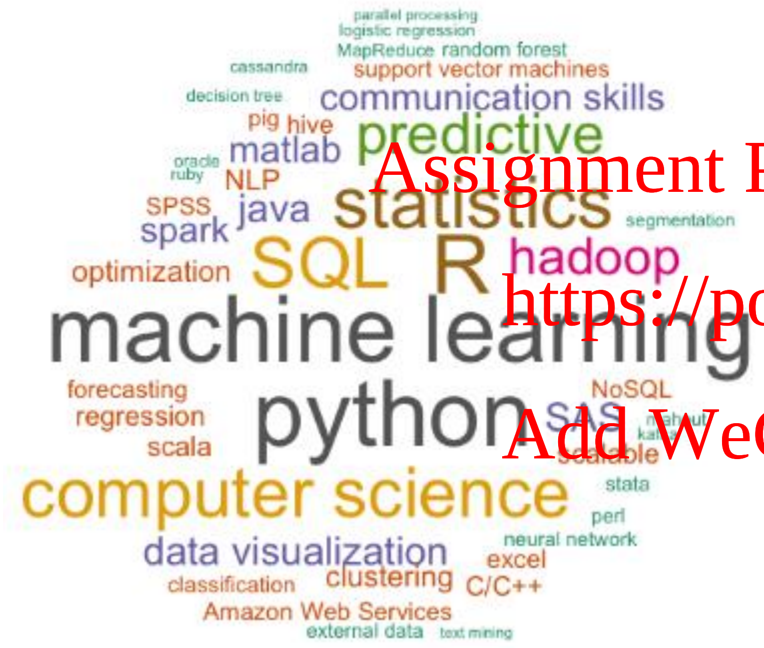
REGRESSION - CONCEPTS (PART 2)



Contents

- Support Vector Machine (SVM)
- Support Vector Regression (SVR)
- Hyperparameter Optimization
- K-fold Cross Validation

Support Vector Machine (SVM)

Support Vector Machine (SVM)

SVM is a powerful and versatile ML model, capable of performing linear or non-linear classification, regression, and even outlier detection. It is one of the more complex but accurate family of models making it one of most popular models in ML despite being a black box technique. SVMs are particularly well suited for classification of complex and small- or medium-size datasets.

Advantages

- Effective in high dimensional spaces
- Still effective in cases where number of features is greater than the number of samples
- Uses a subset of the training dataset in the decision functions (called support vectors), so it is also memory efficient
- Versatile as different kernel functions, including customised kernel functions, can be specified for the decision function

Disadvantages

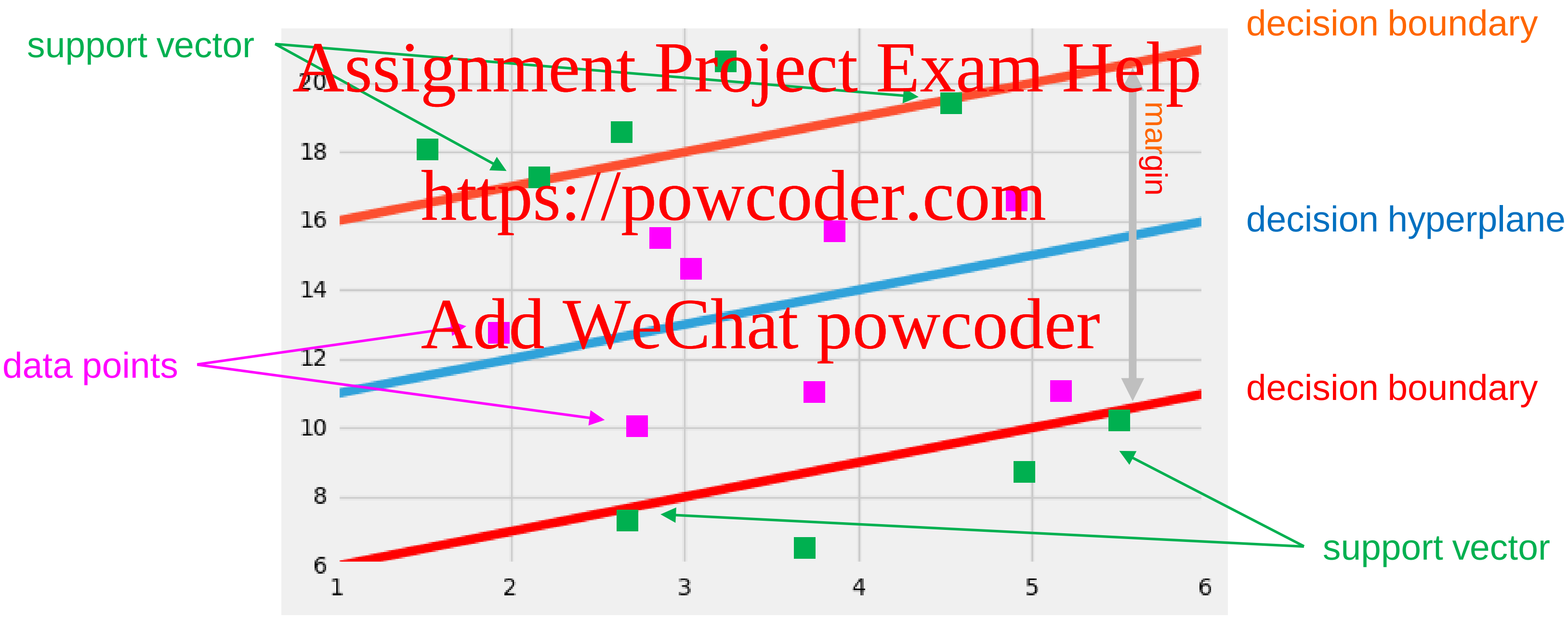
- If the number of features is much greater than the number of samples, avoid over-fitting in choosing kernel functions and regularization term is crucial
- SVMs do not directly provide probability estimates

Support Vector Regression (SVR)

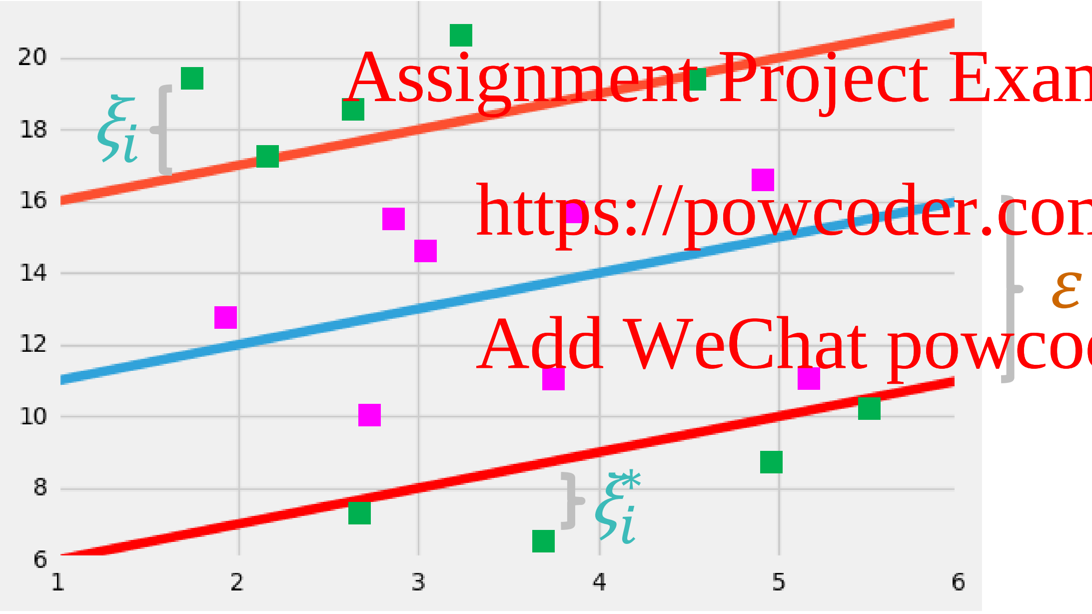
Support Vector Regression (SVR)

Similar to regression, SVR's goal is to discover a hyperplane that minimizes error and obtain a minimum margin interval which contains the maximum number of data points. The key difference is with the cost function. The cost function of regression considers all data points in the dataset and uses regularization to introduce bias and constrain complexity. Whereas the cost function of SVR considers only a subset of the training dataset – the data points that fall into the margin are not included in the calculation of the cost.

SVR finds the best hyperplane with the maximum number of data points captured within the margin



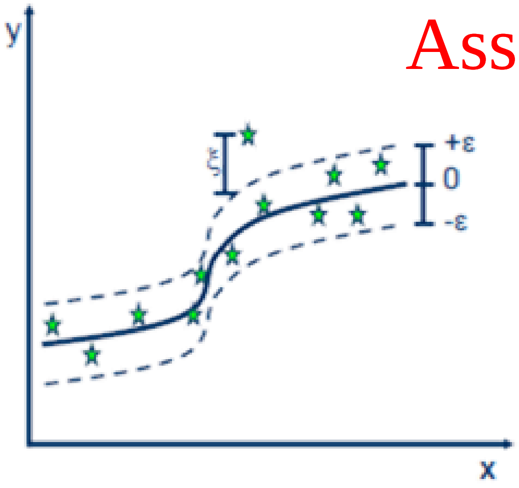
The cost covers only data points beyond the decision boundaries at $+\varepsilon$ and $-\varepsilon$ distance from the hyperplane



The decision boundaries reflect the tolerance level are at ε distance from the decision hyperplane

Error (e.g. ξ_i and ξ_i^*) is measured only for data points outside of the margin

Kernel functions transform data points into a higher dimensional feature space to make them linearly separable



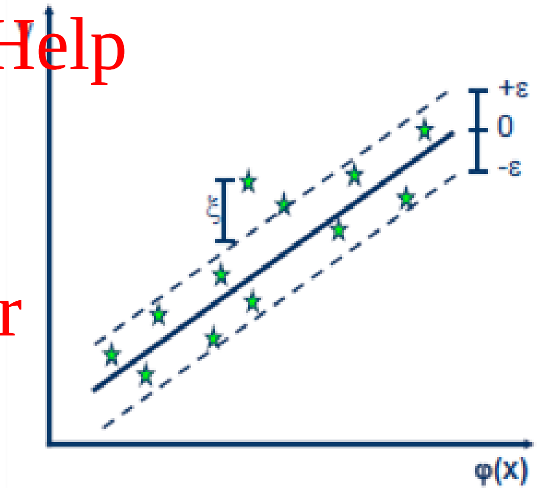
Radial Basis Function (RBF)

$$\emptyset_\gamma(x, l) = \exp(-\gamma\|x - l\|^2)$$

where

$\|x - l\|$ *is the distance of point x from l*

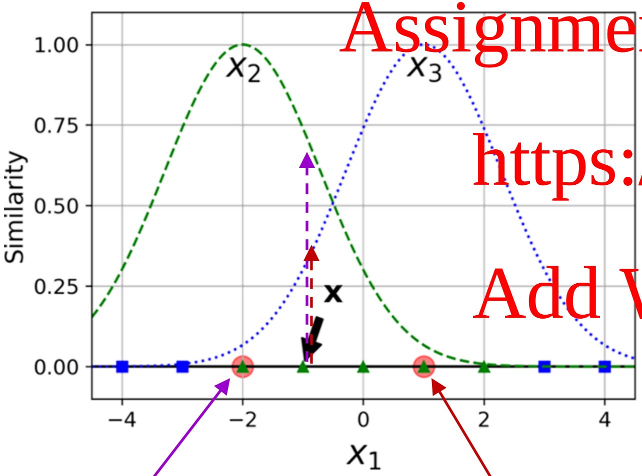
$\gamma = \frac{1}{2\sigma^2}$ *is a shaping parameter*



Radial Basis Function (RBF) introduces a new feature having values in (0,1)

x_2 is a new feature obtained by applying $\emptyset_\gamma(x, l_1)$ over the existing data points

x_3 is a new feature obtained by applying $\emptyset_\gamma(x, l_2)$ over the existing data points



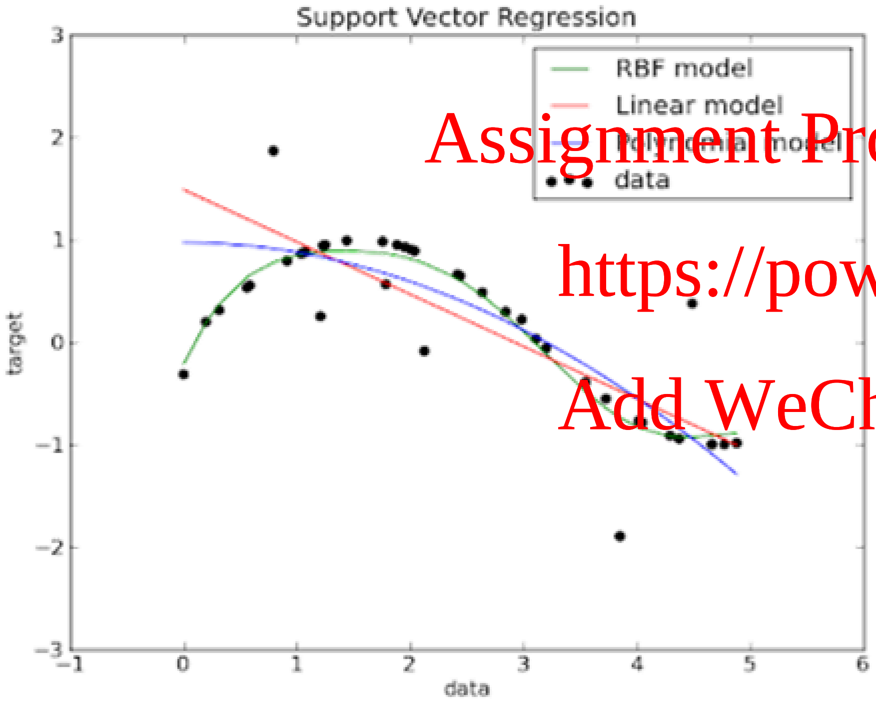
landmark l_1

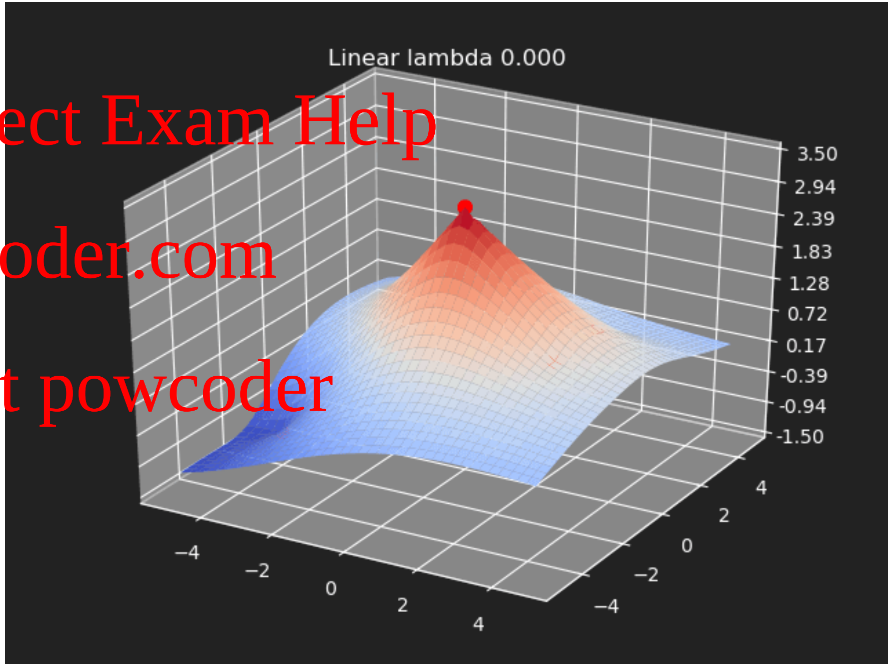
landmark l_2

input x_1 has a 1D feature space

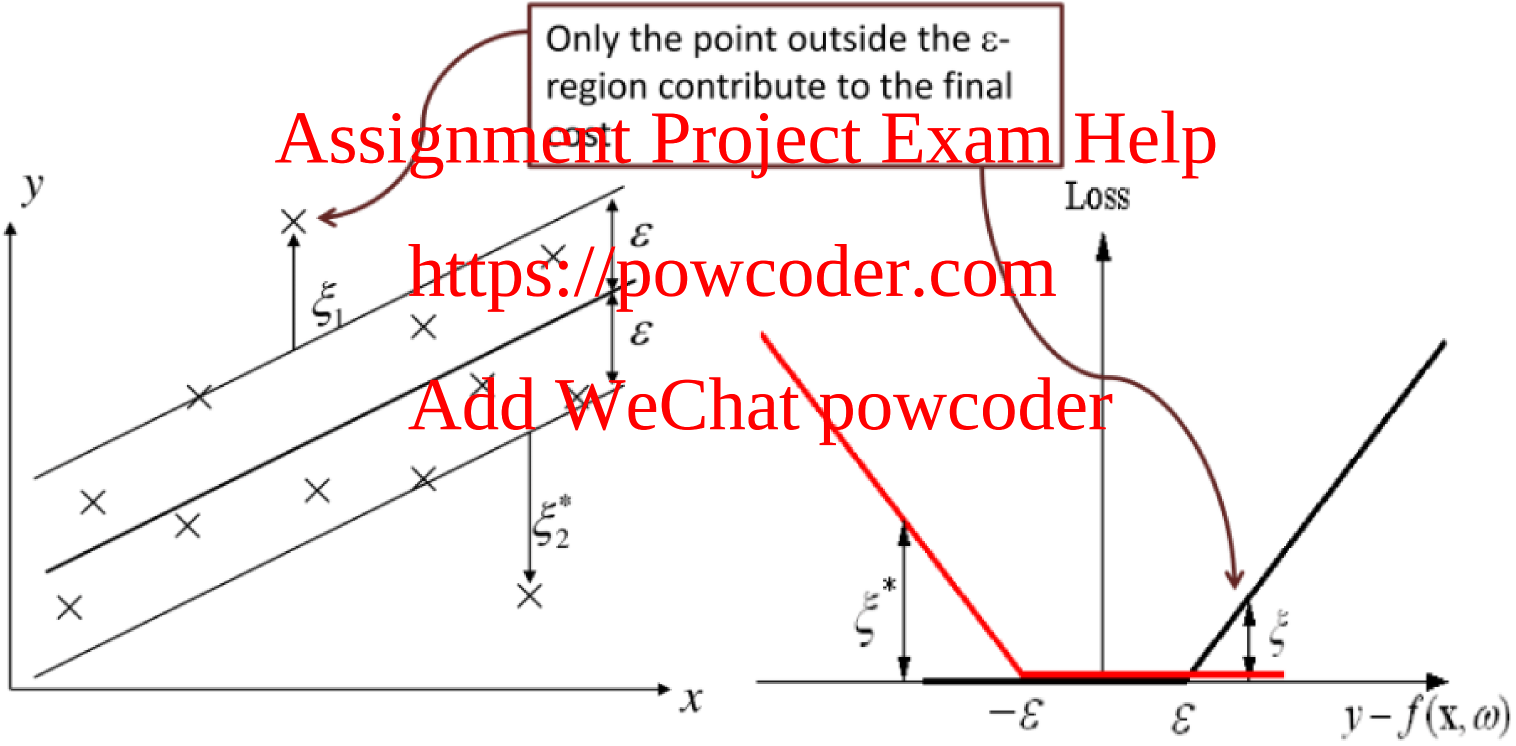
- The RBF is a bell-shaped function measuring the similarity between a landmark point (i.e. l) and any existing data point (e.g. x)
  - $\emptyset_\gamma(x, l) = 0$ indicates the data point x is far from the landmark point l
  - $\emptyset_\gamma(x, l) = 1$ indicates the data point x is at the landmark point l
- γ is a hyperparameter and can be perceived as the inverse of the radius of influence of data points selected by the model as support vectors
  - It can be perceived as deciding how much curvature we want in a decision boundary (i.e. high γ means more curvature)

Some data points will end up outside the decision boundaries introduced by RBF

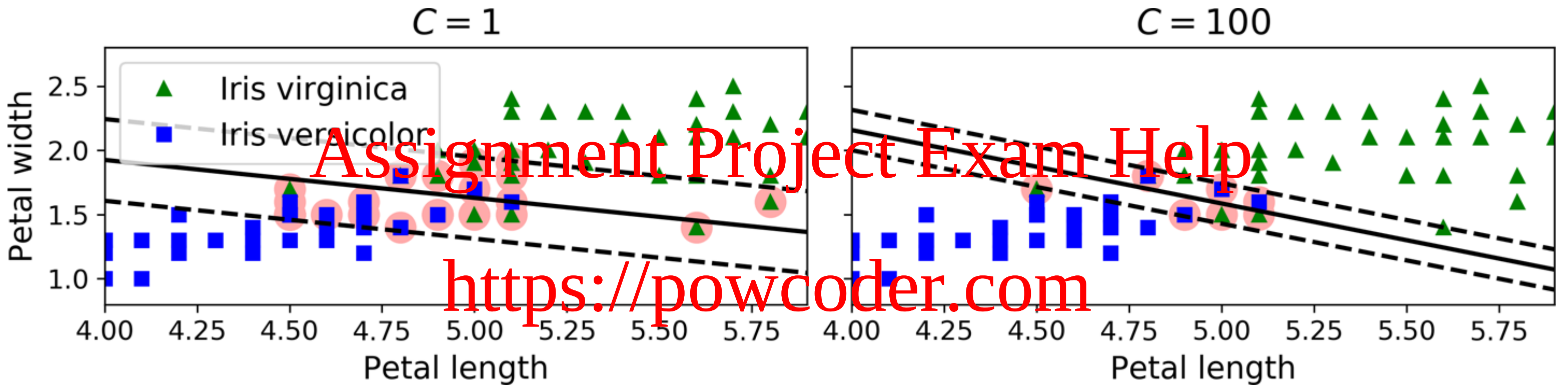




Samples falling between the boundary lines incur no cost (i.e. loss is 0)

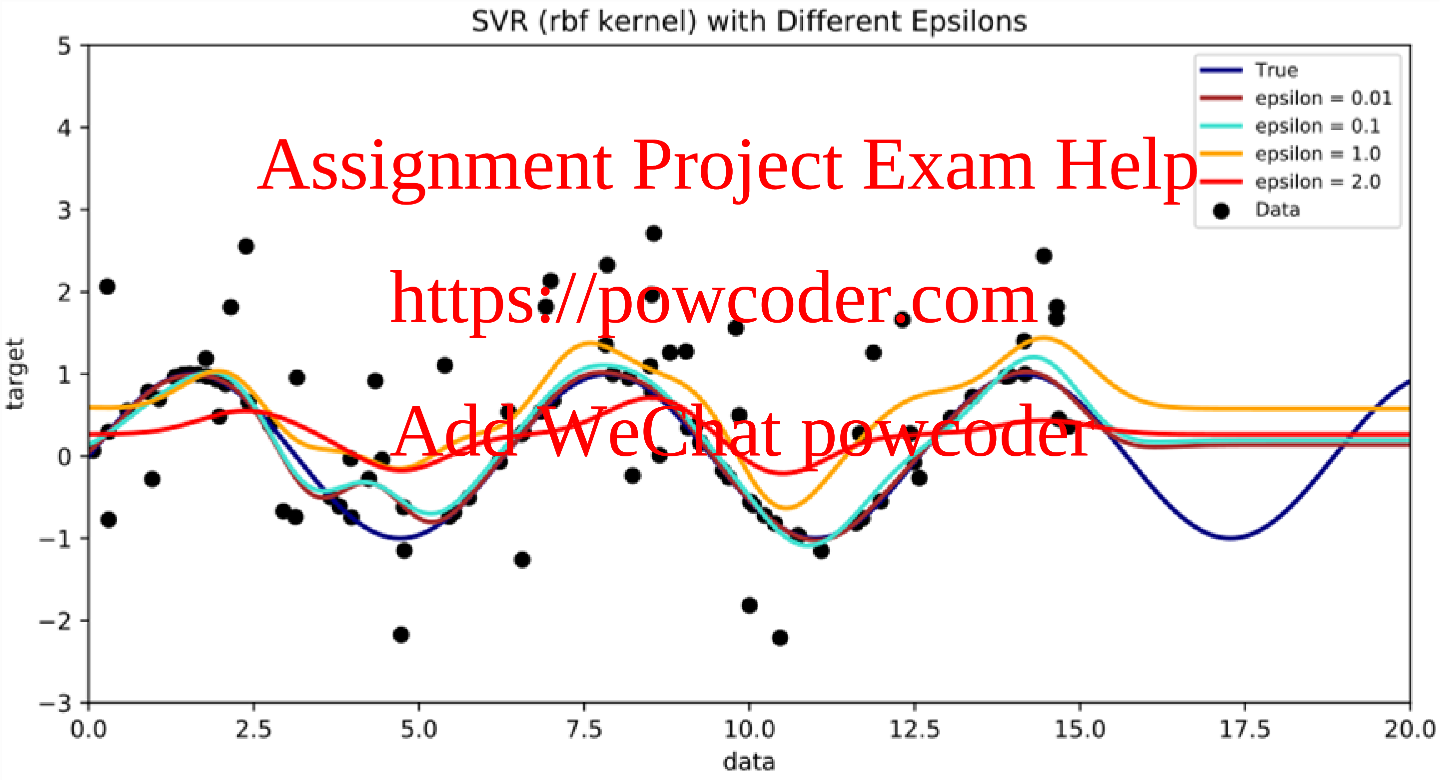


The strength of the regularization is inversely proportional to the regularization hyperparameter C



- C is a hyperparameter for SVR
  - A low value might end up having less error and less predictive power
  - A high value might get more error but better predictive power
- Reducing C can regularize the model to avoid overfitting

Epsilon ε specifies the epsilon-tube within which no penalty is associated in the training loss function



SVR with Python (1)

```
# Import the relevant libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score

# Generate sample data
X = np.sort(3 * np.random.rand(60, 1), axis=0)
y = np.sin(X).ravel()

# Add noise to targets
y[::3] += 3 * (0.6 - np.random.rand(20))
```

SVR with Python (2)

Create regression models

```
svr_rbf1 = SVR(kernel='rbf', C=0.1, gamma=0.1, epsilon=0.1)
svr_rbf2 = SVR(kernel='rbf', C=1, gamma=0.1, epsilon=0.1)
svr_rbf3 = SVR(kernel='rbf', C=10, gamma=0.1, epsilon=0.1)
```

Specify parameters to use for visualization

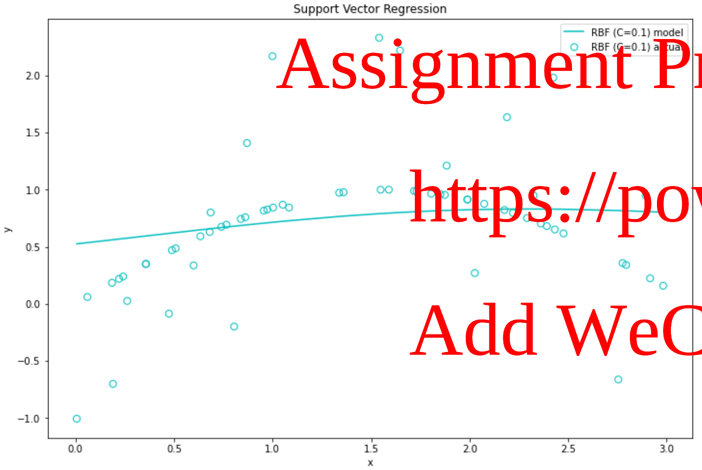
```
svrs = [svr_rbf1, svr_rbf2, svr_rbf3]
kernel_label = ['RBF (C=0.1)', 'RBF (C=1)', 'RBF (C=10)']
model_color = ['c', 'g', 'm']
```

SVR with Python (3)

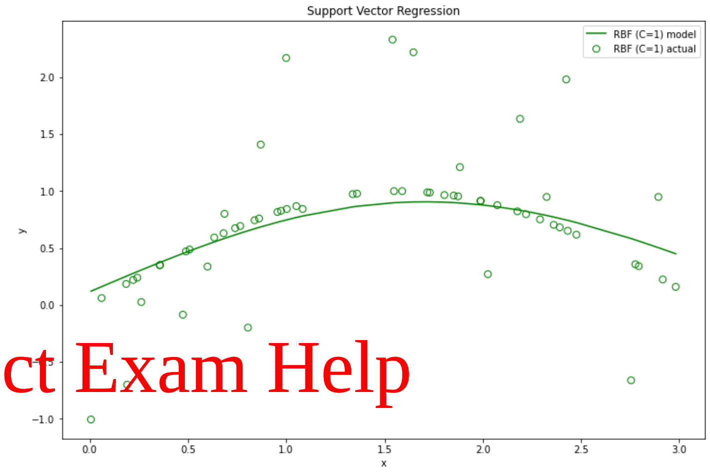
Display 3 models, one after another, with the MSE

```
for ix, svr in enumerate(svrs):
    plt.figure(figsize=(12,8))
    plt.ylabel('y')
    plt.xlabel('x')
    plt.title('Support Vector Regression')

    plt.plot(X, svr.fit(X, y).predict(X), color=model_color[ix],
             label='{} model'.format(kernel_label[ix]))
    plt.scatter(X, y, facecolor="none",
                edgecolor=model_color[ix], s=50,
                label='{} actual'.format(kernel_label[ix]))

    plt.legend()

    plt.show()
    print ('MSE %0.3f' % mean_squared_error(y, svr.fit(X,y).predict(X)))
```

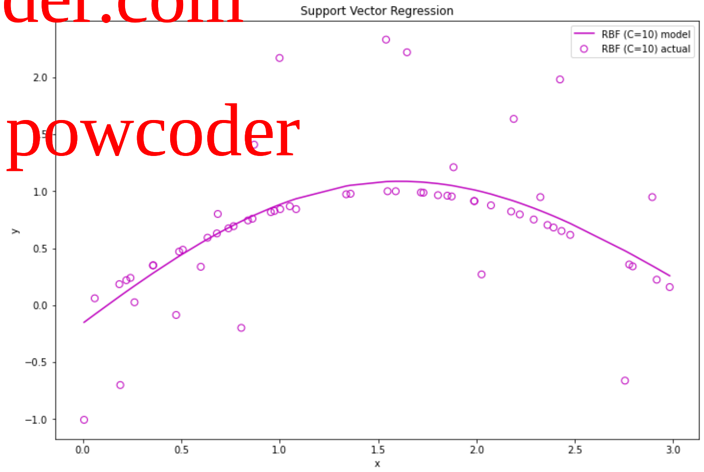

SVR with Python (4)



MSE 0.330



MSE 0.242



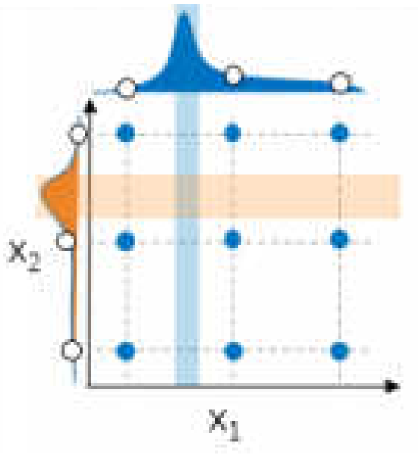
MSE 0.200

Hyperparameter Optimization

Hyperparameter Optimization

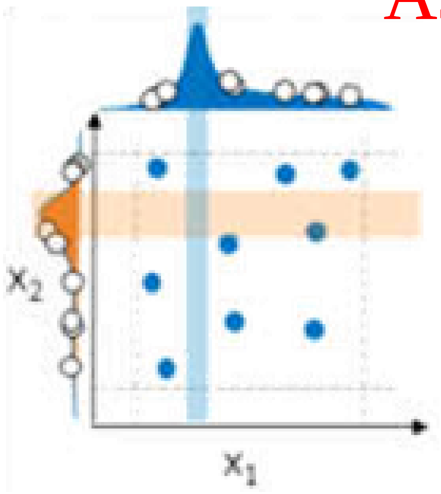
The technique of identifying an ideal set of parameters for a prediction algorithm (e.g. coefficient values for regression problems), which provides the optimum performance. The algorithm learns which parameter values provide us with better performance by iteratively working on a pre-defined set of parameters.

Grid Search



- Take each hyperparameter of interest and select a set of values
  - e.g. the epsilon ε hyperparameter in an SVR model having a value of 0.1, 0.3, 0.5 and the gamma γ hyperparameter having value 0.001 and 0.0001
- Train models using the combination of potential hyperparameter values
- Identify the best performing model and the corresponding hyperparameter values
- Computationally costly for a grid of fine granularity
- Might miss optimal hyperparameter values

Random Search



- Train models using the combination of potential hyperparameter values chosen at random
- May try more values per hyperparameter than the case with grid search
- Identify the best performing model and the corresponding hyperparameter values
- May find the optimal combination of hyperparameter values by chance or may miss the optimal points altogether
- Often preferable when the hyperparameter search space is large

Random search turns out to be a surprisingly effective technique

The reason random search turns out to work so well is due to two key properties

- It turns out that the hyperparameter space has a low effective dimensionality
  - Some parameters matter much more than others when it comes to finding good settings
- The optimal combination of hyperparameter values varies according to the dataset
  - One cannot just find the two most important hyperparameters for some model architecture and then always optimize based on just those

Grid Search with Python (1)

Import the relevant libraries

```
import numpy as np
import pandas as pd

from sklearn import datasets
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
```


Partition the dataset into training and testing datasets
Testing dataset being the last 40 samples

```
data = datasets.load_diabetes()
num_test = 40
X_train = data.data[:-num_test, :]
y_train = data.target[:-num_test]
X_test = data.data[-num_test:, :]
y_test = data.target[-num_test:]
```

Grid Search with Python (2)

```
# Create a regression model
model = SVR()

# Specify hyperparameter values to perform the search with
param_grid = [
    {'kernel': ['rbf'], 'C': [0.1, 1, 10], 'gamma': [0.001, 0.0001], 'epsilon': [0.1]},
    {'kernel': ['rbf'], 'C': [10, 100], 'gamma': [0.001, 0.0001], 'epsilon': [0.5]}
]
```


"param_grid" tells the algorithm to first evaluate 3x2=6 (representing the potential values for "C" and "gamma") combinations of hyperparameter values. The algorithm will then try another 2x2=4 combinations. Altogether grid search will search using 10 combinations.

Grid Search with Python (3)

Set up grid search and use cross-validation

```
grid_search = GridSearchCV(model, param_grid, cv=5)
```


Perform grid search

```
grid_search.fit(X_train, y_train)
```


Show the over results

```
grid_search.cv_results_
```

Grid Search with Python (4)

```
{'mean_fit_time': array([0.0044014 , 0.00379667, 0.00338044, 0.00299153, 0.00319176,
       0.00339122, 0.00339112, 0.0031918 , 0.00339031, 0.00319123]),
 'std_fit_time': array([8.00060603e-04, 4.03429559e-04, 4.77072791e-04, 1.23977661e-06,
       4.00638722e-04, 4.89434272e-04, 5.02297546e-04, 3.99074604e-04,
       5.04200653e-04, 3.82878177e-04]),
 'mean_score_time': array([0.00078816, 0.00060005, 0.00079789, 0.00099759, 0.0009973 ,
       0.00060472, 0.00079784, 0.0008049 , 0.00071359, 0.00099111]),
 'std_score_time': array([3.94430157e-04, 4.89[illegible]e-04, [illegible]e-04, [illegible]e-07,
       8.79244276e-07, 4.93885490e-04, 3.994[illegible]e-04, [illegible]e-04,
       6.17738002e-04, 1.23812492e-05]),
 'param_C': masked_array(data=[0.1, 0.1, 1, 1, 10, 10, 10, 10, 100, 100],
             mask=[False, False, False, False, False, False, False, False,
                   False, False],
       fill_value='?',
            dtype=object),
 'param_epsilon': masked_array(data=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5],
             mask=[False, False, False, False, False, False, False, False,
                   False, False],
       fill_value='?',
            dtype=object),
 'param_gamma': masked_array(data=[0.001, 0.0001, 0.001, 0.[illegible], [illegible], [illegible], 0.001,
                   0.0001, 0.001, 0.0001],
             mask=[False, False, False, False, False, False, False, False,
                   False, False],
       fill_value='?',
            dtype=object),
 'param_kernel': masked_array(data=['rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf',
                   'rbf', 'rbf'],
             mask=[False, False, False, False, False, False, False, False,
                   False, False],
       fill_value='?',
            dtype=object),
 'params': [{'C': 0.1, 'epsilon': 0.1, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 0.1, 'epsilon': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'},
  {'C': 1, 'epsilon': 0.1, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 1, 'epsilon': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'},
  {'C': 10, 'epsilon': 0.1, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 10, 'epsilon': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'},
  {'C': 10, 'epsilon': 0.5, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 1[illegible], 'epsilon': 0.5, 'gamma': [illegible], 'kernel': 'rbf'},
  {'C': 100, 'epsilon': 0.5, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 100, 'epsilon': 0.5, 'gamma': 0.0001, 'kernel': 'rbf'}],
 'split0_test_score': array([-0.00240799, -0.00241125, -0.00237562, -0.00240799, -0.0020518 ,
        -0.00237562, -0.00264776, -0.00297245,  0.00058728, -0.00264773]),
 'split1_test_score': array([-0.03681614, -0.03681957, -0.03678179, -0.03681614, -0.03643847,
        -0.03678181, -0.03461382, -0.03492082, -0.03155588, -0.03461385]),
 'split2_test_score': array([-0.03099271, -0.03099757, -0.03094407, -0.0309927 , -0.03045783,
        [illegible], [illegible], -0.03094405, -0.02561176, -0.03045766]),
 'split3_test_score': array([-0.01789224, -0.01789643, -0.01785031, -0.01789223, -0.0174312 ,
        -0.01785031, -0.01872298, -0.01919757, -0.01399301, -0.01872286]),
 'split4_test_score': array([-0.10273943, -0.10274387, -0.102695  , -0.10273943, -0.1022508 ,
        -0.102695  , -0.10218742, -0.10268868, -0.09718791, -0.10218763]),
 'mean_test_score': array([[illegible] , -0.03817374, -0.03812936, -0.0381697 , -0.03772602,
        [illegible], [illegible], -0.03814471, -0.03355226, -0.03772595]),
 'std_test_score': array([0.03438798, 0.03438825, 0.03438528, 0.03438798, 0.03435829,
        0.03438528, 0.03408632, 0.0341299 , 0.03365889, 0.03408642]),
 'rank_test_score': array([ 9, 10,  6,  8,  4,  5,  3,  7,  1,  2])}
```

Grid Search with Python (5)

Show the hyperparameter values for the best performing model

```
grid_search.best_params_
```



```
{'C': 100, 'epsilon': 0.5, 'gamma': 0.001, 'kernel': 'rbf'}
```

Get hold of the best performing model

```
best_model = grid_search.best_estimator_
```


Use the best performing model to make predictions for the testing dataset

```
pred = best_model.predict(X_test)
```

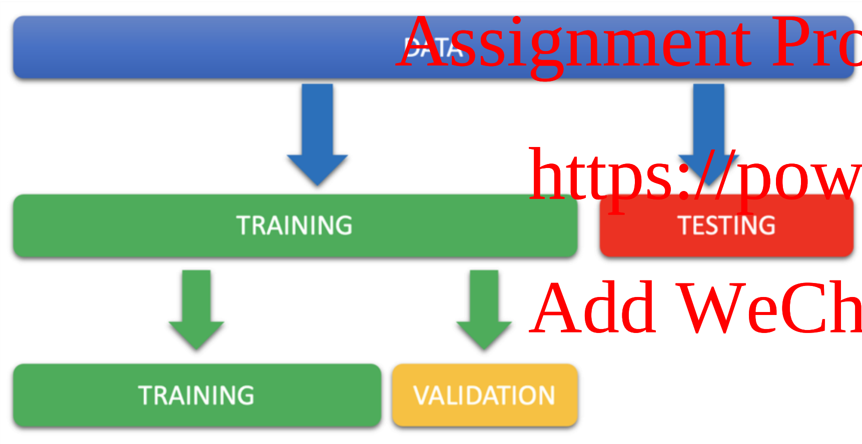

Show the RMSE and R^2 score of the prediction

```
print ('RMSE: %0.3f' % mean_squared_error(y_test, pred, squared=False))
print ('R^2 Score: %0.3f' % r2_score(y_test, pred))
```

```
RMSE: 73.865
R^2 Score: 0.046
```

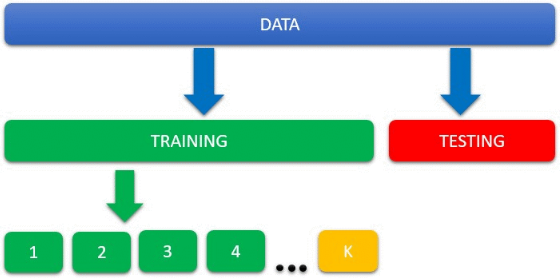
K-fold Cross Validation

Machine Learning Validation



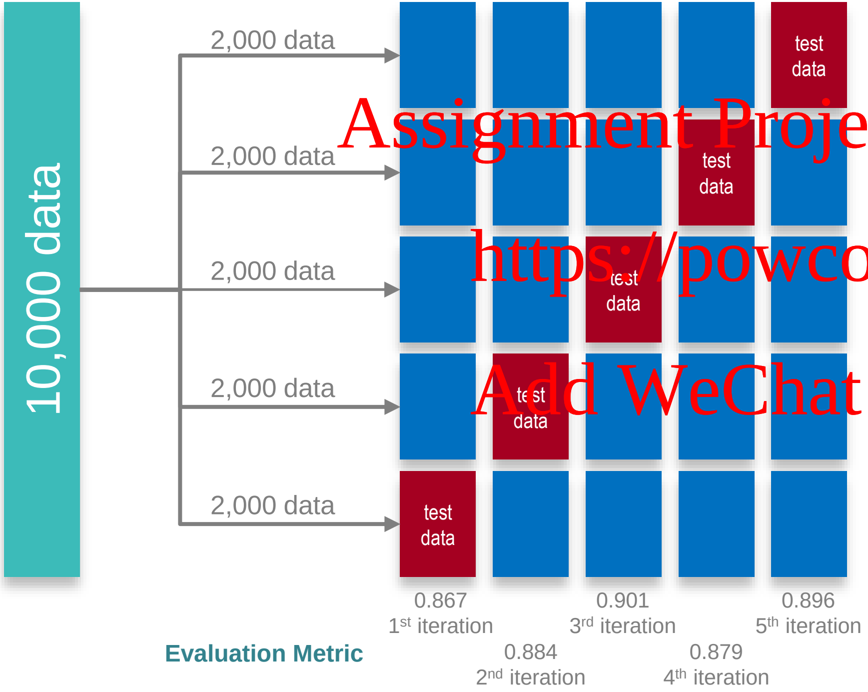
- Validation is the process of making sure that the model generalizes well
- Generalization is when model is built using one set of data and it performs well on a completely different set of data
- Validation dataset is used to fine tune hyperparameters and serves also as an intermediary testing dataset
- Sometimes referred to as the hold-out validation set

K-fold Cross Validation



- Evaluates the data across the entire training set
- Divides the training set into K folds and then training the model K times
- Each time leaving a different fold out of the training data and using it instead as a validation dataset
- The performance metric is averaged across all K tests
- Once the best parameter combination has been found, the model is retrained on the full dataset

All data in the hold-out dataset can be used for both training and testing through k-fold cross-validation



- Free of selection bias
- Generalizes well
- Matters less how the data gets divided
- However, it has higher computational cost as training has to be done k times

K-fold Cross Validation (1)

\# Import the relevant libraries

```
import numpy as np
import pandas as pd

from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import cross_validate

import plotly.graph_objects as go
import plotly.express as px
```


\# Load data
\# https://www.kaggle.com/quantbruce/real-estate-price-prediction?select=Real+estate.csv

```
data = pd.read_csv('Real estate.csv', encoding='utf-8')
```

K-fold Cross Validation (2)

```
# Identify features and target to use
X = data['X3 distance to the nearest MRT station'].values.reshape(-1,1)
y = data['Y house price of unit area'].values

# Create an SVR
C = 100
epsilon = 0.1
gamma = 0.001
svr = SVR(kernel='rbf', C=C, epsilon=epsilon, gamma=gamma)

# Cross-validate the SVR
scores = cross_validate(svr, X, y, cv=5,
                        scoring=('r2', 'neg_mean_squared_error'),
                        return_train_score=True)
```

K-fold Cross Validation (3)

Show scores collected during cross-validation

```
scores
```



```
{'fit_time': array([0.01[illegible]449, 0.00[illegible]22, 0.01[illegible], 0.00[illegible], 0.01496029]),
 'score_time': array([0.00099778, 0.00199676, 0.00199533, 0.0009985 , 0.00099921]),
 'test_r2': array([0.61550123, 0.62433129, 0.57562613, 0.44666894, 0.59794955]),
 'train_r2': array([0.69178463, 0.67666219, 0.67961301, 0.74562687, 0.68458942]),
 'test_neg_mean_squared_error': array([ -66.[illegible], -69.59168879,  -72.62012163, -133.09466993,
        -59.35752982]),
 'train_neg_mean_squared_error': array([-57.80203218, -59.479319  , -60.08431974, -43.32734085,
        -61.10299231])}
```

Conclusion

Support Vector Machine (SVM) Models in a Nutshell

	Property	Description
1	Feature Data Types	Requires the feature scaling of the data.
2	Target Data Types	Numeric values
3	Key Principles	Introduce a margin on either side of the regression plane such that data points falling within the margin will not contribute to the loss function calculation. The goal is to minimise the margin and loss while fitting as many data points into the margin as possible.
4	Hyperparameters	With linear or polynomial kernel, the C hyperparameter (cost of misclassification) is needed but gamma (curvature weight of the decision boundary) is not needed. With the Gaussian RBF kernel, both Gamma and C are needed.
5	Data Assumptions	No data distributional requirement.
6	Performance	Fairly robust against overfitting, especially in higher dimensional space. Handles nonlinear relationships quite well, with many kernels to choose from. Can be inefficient to train and memory-intensive to run and tune. Does not perform well with large datasets.
7	Accuracy	Generally, performs better than linear, polynomial regressions.
8	Explainability	Black box technique

References

References

“Hands-On Machine Learning with Scikit-Learn and TensorFlow”, Aurelien Geron, O'Reilly Media, Inc., 2017

References

- "Support Vector Regression (SVR) Model: A Regression-Based Machine Learning Approach", Abhilash Singh, 2020 (https://medium.com/analytics-vidhya/support-vector-regression-svr-model-a-regression-based-machine-learning-approach-f4641670c5bb)
- "Support Vector Regression Tutorial for Machine Learning", Alakh Sethi, 2020 (https://www.analyticsvidhya.com/blog/2020/03/support-vector-regression-tutorial-for-machine-learning/)
- "Support Vector Regression: SVR", Indresh Bhattacharyya, 2018 (https://medium.com/coinmonks/support-vector-regression-or-svr-8eb3acf6d0ff)
- "Support Vector Machine – Regression (SVR)" (http://www.saedsayad.com/support_vector_machine_reg.htm)
- "Support Vector Regression (SVR) – One of the Most Flexible Yet Robust Prediction Algorithms", Saul Dobilas, 2020 (https://towardsdatascience.com/support-vector-regression-svr-one-of-the-most-flexible-yet-robust-prediction-algorithms-4d25fbdaca60)

THANK YOU