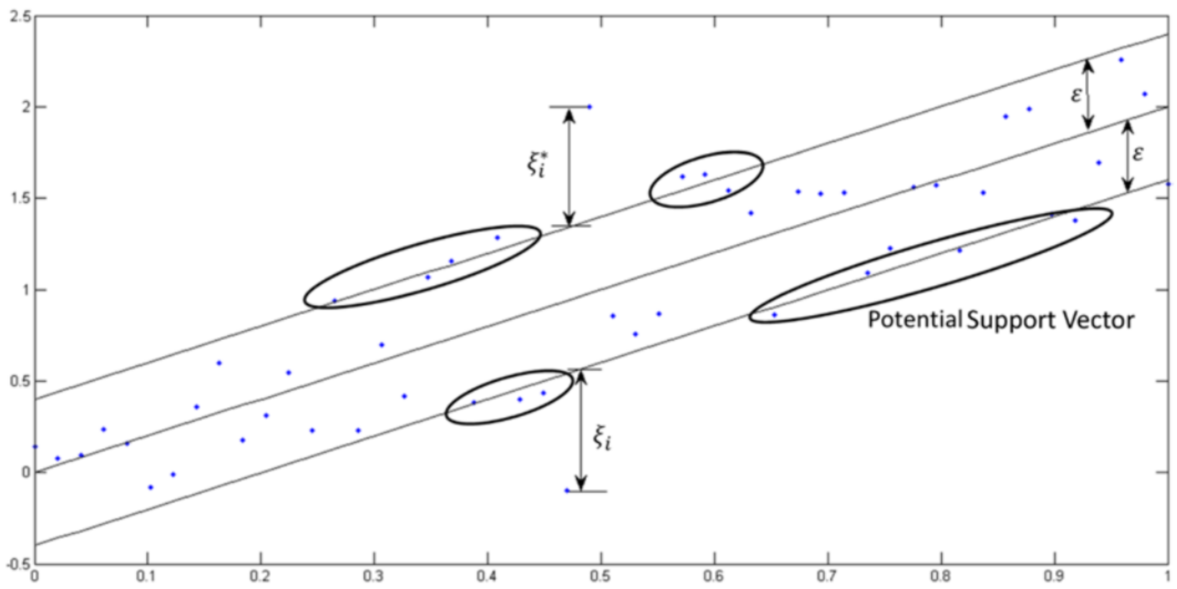
Support Vector Regression

Support Vector Regression (SVR) is a regression function that is generalized by Support Vector Machines - a machine learning model used for data classification on continuous data.

Introduction support vectors

Unlike in the Ordinary Least Squares, the SVR model sets a threshold error allowance ϵ, around the regression line such that all the data points within ϵ are not penalized for their error.



Therefore, the bound is error insensitive and is called ϵ - insensitive tube or simply ϵ - tube.

As we stated earlier, the distance between the regression line and the upper or lower bound is usually ϵ units. The ϵ-tube is thus 2ϵ, 2 units wide and symmetrical about the regression line.

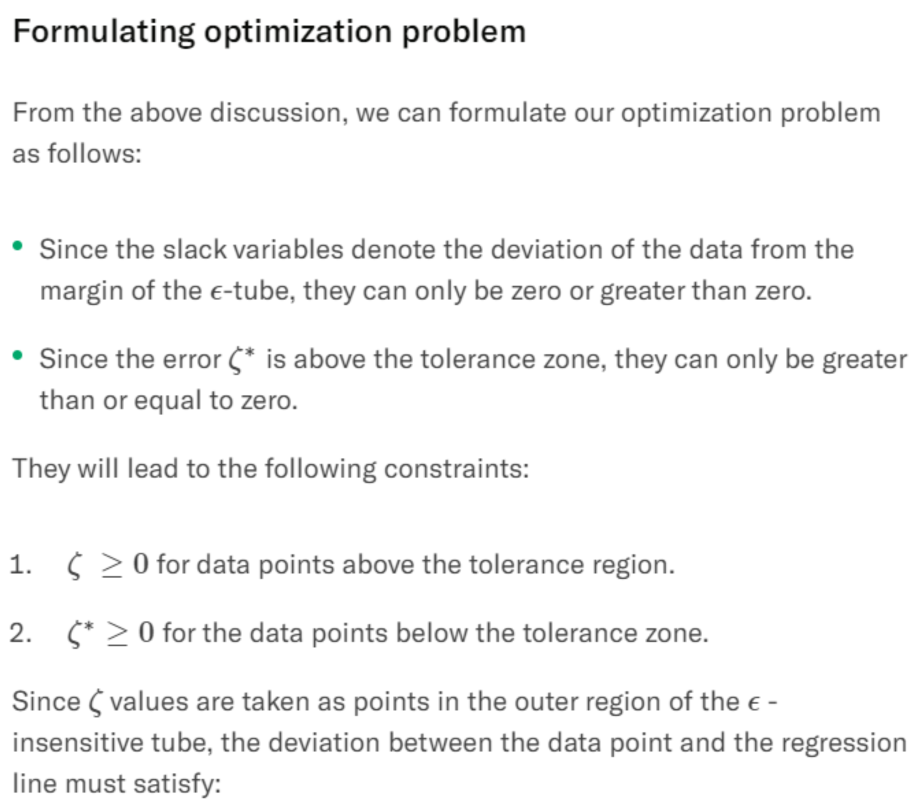
Data points that fall outside the ϵ-tube are penalized for their error:

- The error associated with the data point above ϵ-tube is computed as the vertical distance between the datapoint and ϵ-tube's margin.
- If the data is below the tube, an error is the vertical distance between the ϵ-tube's lower bound and the datapoint.

We need to note that the error is taken from the tube's margin and not the regression line.

Whenever a data point is above the tube's margin, the deviation is denoted as ζ and $\zeta\ast\ast$ when it is below. Both the ζ and $\zeta\ast$ are called Slack Variables.

These points are the ones that dictate how the ϵ�-tube is created. They are thus called the *Support Vector*.



Formulating optimization problem

From the above discussion, we can formulate our optimization problem as follows:

- Since the slack variables denote the deviation of the data from the margin of the ϵ-tube, they can only be zero or greater than zero.
- Since the error ζ^* is above the tolerance zone, they can only be greater than or equal to zero.

They will lead to the following constraints:

1. $\zeta \geq 0$ for data points above the tolerance region.
2. $\zeta^* \geq 0$ for the data points below the tolerance zone.

Since ζ values are taken as points in the outer region of the ϵ-insensitive tube, the deviation between the data point and the regression line must satisfy:

$y_i - (W^T x + b) \leq \epsilon + \zeta_i$, if the point is above the tube and,

$(W^T x + b) - y_i \leq \epsilon + \zeta_i^*$, if the point is below the tube.

Primal problem

Our objective is to ensure that $\sum(\zeta + \zeta_i^*)$, is minimal.

Therefore, our primal problem is:

$\frac{1}{2}||W||^2 + C\sum_{n=1}^{m}(\zeta + \zeta_i^*) \rightarrow$ minimize

Where C is some constant that give weight on minimizing $\sum(\zeta + \zeta_i^*)$.

Advantages of Support Vector Regression:

1. SVR works relatively well when there is a clear margin of separation between classes.
2. Effective in High Dimensional Spaces.
3. Works well on non-linear problems.
4. Not biased by outliers.

Disadvantages of Support Vector Regression:

1. Not suitable for large datasets since it takes comparatively more time to train.
2. Quiet prone to noise.

3. SVR underperforms if data is not well separated or is overlapping.

4. Interpretability and Explainability is difficult.

Implementing SVR in Python

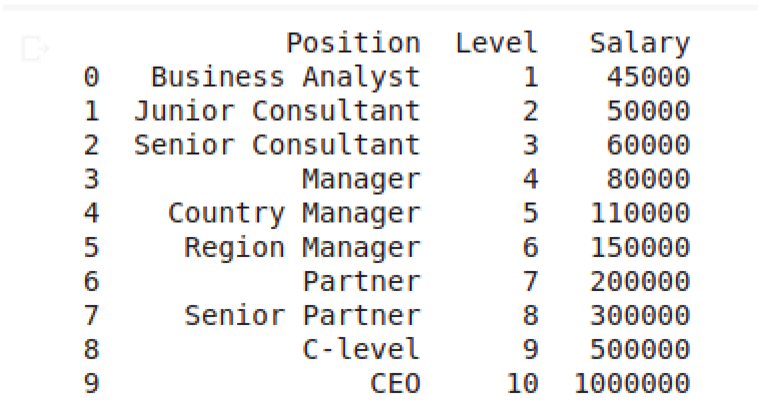
Data preprocessing

As in any other implementation, first, we get the necessary libraries in place. The code below imports these libraries:

```
# get the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

The dataset used in this session can be downloaded here.

```
# get the dataset
dataset = pd.read_csv('Position_Salaries.csv')
# our dataset in this implementation is small, and thus we can
print it all instead of viewing only the end
print(dataset)
```



```
            Position  Level   Salary
0   Business Analyst      1    45000
1  Junior Consultant      2    50000
2  Senior Consultant      3    60000
3            Manager      4    80000
4    Country Manager      5   110000
5     Region Manager      6   150000
6            Partner      7   200000
7     Senior Partner      8   300000
8            C-level      9   500000
9                CEO     10  1000000
```

Output:

The above dataset contains ten instances. The significant feature in this dataset is the `Level` column. The `Position` column is just a description of the `Level` column, and therefore, it adds no value to our analysis.

Therefore, we will separate the dataset into a set of features and study variables.

As discussed above, we only have one feature in this dataset. Therefore, we carry out our feature-study variable separation as shown in the code below:

```
# split the data into featutes and target variable seperately
X_l = dataset.iloc[:, 1:-1].values # features set
y_p = dataset.iloc[:, -1].values # set of study variable
```

We can look at our feature set using the `print()` function.

```
print(X_l)
```

Output:

```
[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]]
```

From this output, it’s clear that the `X_l` variable is a 2D array. Similarly, we can have a look at the `y_p` variable:

```
print(y_p)
```

Output:

```
 [  45000   50000   60000   80000  110000  150000  200000  300000  500000
  1000000]
```

It's seen from the output above that the `y_p` variable is a vector, i.e., a 1D array.

We need to note that the values of `y_p` are huge compared to `x_1`.

Therefore, if we implement a model on this data, the study variable will dominate the feature variable, such that its contribution to the model will be neglected.

Due to this, we will have to scale this study variable to the same range as the scaled study variable.

The challenge here is that the `StandardScaler`, the class we use to scale the data, takes in a 2D array; otherwise, it returns an error.

Due to this, we have to reshape our `y_p` variable from 1D to 2D. The code below does this for us:

```
y_p = y_p.reshape(-1,1)
```

Output:

```
[[  45000]
 [  50000]
 [  60000]
 [  80000]
 [ 110000]
 [ 150000]
 [ 200000]
 [ 300000]
 [ 500000]
 [1000000]]
```

From the above output, `y_p` was successfully reshaped into a 2D array.

Now, import the `StandardScalar` class and scale up the `X_1` and `y_p` variables separately as shown:

```
from sklearn.preprocessing import StandardScaler
StdS_X = StandardScaler()
```

```
StdS_y = StandardScaler()
X_l = StdS_X.fit_transform(X_l)
y_p = StdS_y.fit_transform(y_p)
```

Let’s simultaneously print and check if our two variables were scaled.

```
print("Scaled X_l:")
print(X_l)
print("Scaled y_p:")
print(y_p)
```

Output:

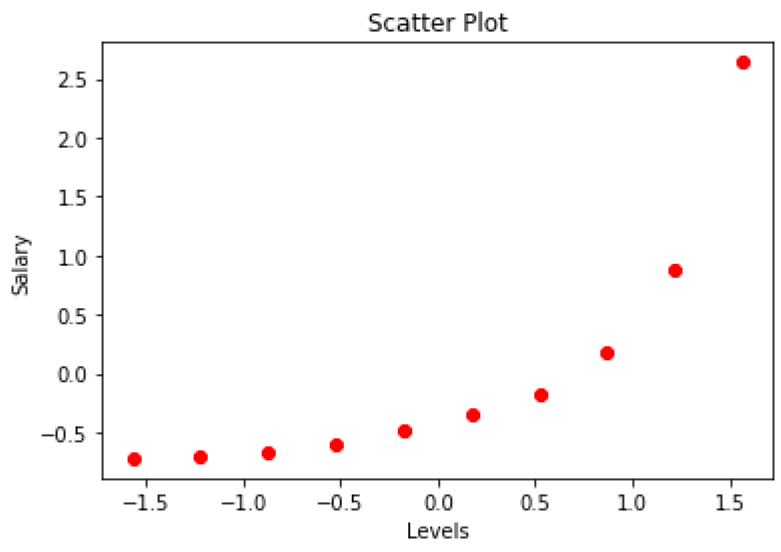
```
Scaled X_l:
[[-1.5666989 ]
 [-1.21854359]
 [-0.87038828]
 [-0.52223297]
 [-0.17407766]
 [ 0.17407766]
 [ 0.52223297]
 [ 0.87038828]
 [ 1.21854359]
 [ 1.5666989 ]]
Scaled y_p:
[[-0.72004253]
 [-0.70243757]
 [-0.66722767]
 [-0.59680786]
 [-0.49117815]
 [-0.35033854]
 [-0.17428902]
 [ 0.17781001]
 [ 0.88200808]
 [ 2.64250325]]
```

As we can see from the obtained output, both variables were scaled within the range `-3` and `+3`.

Our data is now ready to implement our SVR model.

However, before we can do so, we will first visualize the data to know the nature of the SVR model that best fits it. So, let us create a scatter plot of our two variables.

```
plt.scatter(X_l, y_p, color = 'red') # plotting the training
set
plt.title('Scatter Plot') # adding a tittle to our plot
plt.xlabel('Levels') # adds a label to the x-axis
plt.ylabel('Salary') # adds a label to the y-axis
plt.show() # prints
```



The plot shows a non-linear relationship between the `Levels` and `Salary`.

Due to this, we cannot use the linear SVR to model this data. Therefore, to capture this relationship better, we will use the SVR with the kernel functions.

Linear SVR

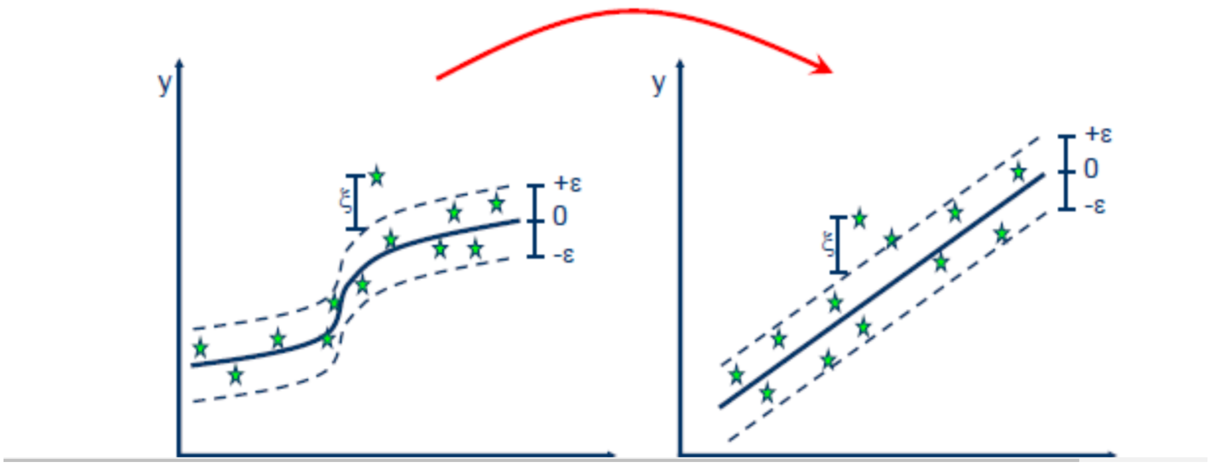
$$y = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot \langle x_i, x \rangle + b$$

Non-linear SVR

The kernel functions transform the data into a higher dimensional feature space to make it possible to perform the linear separation.

$$y = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot \langle \varphi(x_i), \varphi(x) \rangle + b$$

$$y = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot K(x_i, x) + b$$



Kernel functions

Polynomial

$$k(\mathrm{x}_i, \mathrm{x}_j) = (\mathrm{x}_i . \mathrm{x}_j)^d$$

Gaussian Radial Basis function

$$k(\mathrm{x}_i, \mathrm{x}_j) = \exp\left(- \frac{\left\| \mathrm{x}_i - \mathrm{x}_j \right\|^2}{2\sigma^2} \right)$$

Implementing SVR

To implement our model, first, we need to import it from the scikit-learn and create an object to itself.

Since we declared our data to be non-linear, we will pass it to a kernel called the Radial Basis function (RBF) kernel.

After declaring the kernel function, we will fit our data on the object. The following program performs these rules:

```
# import the model
from sklearn.svm import SVR
# create the model object
regressor = SVR(kernel = 'rbf')
# fit the model on the data
regressor.fit(X_l, y_p)
```

Since the model is now ready, we can use it and make predictions as shown:

```
A=regressor.predict(StdS_X.transform([[6.5]]))
print(A)
```

Output:

```
array([-0.27861589])
```

As we can see, the model prediction values are for the scaled study variable. But, the required value for the business is the output of the unscaled data. So, we need to get back to the real scale of the study variable.

To go back to the real study variable, we will write a program whose objective is to take the predicted values on the scaled range and transform them to the actual scale.

We do so by taking an inverse of the transformation on the study variable.

Note that the predicted values are returned in a 1D array.

However, as we can recall, we had reshaped our study variable from 1D to 2D array since the `StandarScaler` method takes in only 2D arrays.

So, for any predicted value to fit within such a new dimension of the study variable, it must be transformed from 1D to 2D; otherwise, we will get an error.

So, let's implement these commands and get the required value:

```
# Convert A to 2D
A = A.reshape(-1,1)
print(A)
```

Output:

```
array([[-0.27861589]])
```

It is clear from the output above is a 2D array. Using the `inverse_transform()` function, we can convert it to an unscaled value in the original dataset as shown:

```
# Taking the inverse of the scaled value
A_pred = StdS_y.inverse_transform(A)
print(A_pred)
```

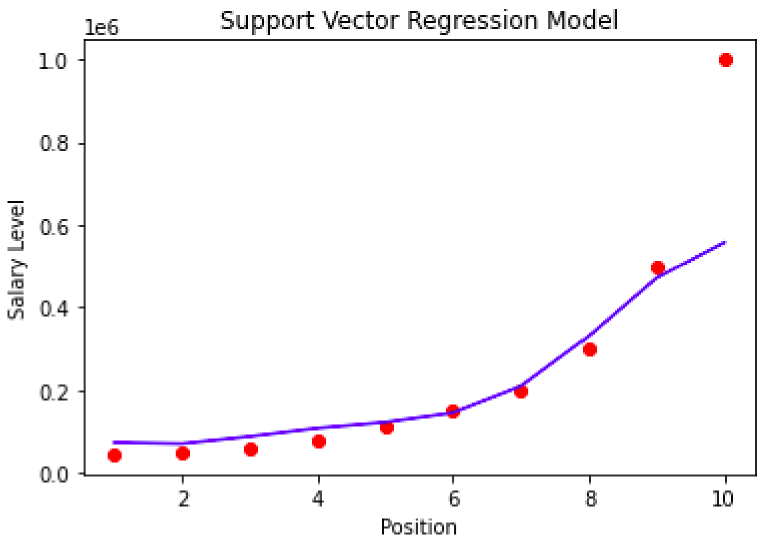
Output:

```
array([[170370.0204065]])
```

Here is the result, and it falls within the expected range.

However, if we were to run a polynomial regression on this data and predict the same values, we would have obtained the predicted values as `158862.45265155`, which is only fixed on the curve.

With the Support Vector regression, this is not the case. So there is that allowance given to the model to make the best prediction.


Support Vector Regression Model
1e6
1.0
0.8
0.6
0.4
0.2
0.0
Salary Level
2
4
6
8
10
Position