## Q1

Support Vector Machines (SVMs) Concept:

SVMs are a type of supervised machine learning algorithm used for classification and regression tasks. The underlying concept is to find a hyperplane that best separates data points of different classes while maximizing the margin between the hyperplane and the nearest data points of each class.

## Q2

Support Vector:

A support vector is a data point from the training dataset that lies closest to the hyperplane (decision boundary) that separates the classes. These are the most crucial data points, as they determine the position and orientation of the hyperplane.

## Q3

Scaling Inputs with SVMs:

It's essential to scale the inputs when using SVMs because SVMs are sensitive to the scale of features. Features with larger scales can dominate the optimization process. Scaling ensures that all features contribute equally to the SVM's decision boundary and leads to better model performance.

## Q4

SVM Confidence Score:

SVM classifiers can output a decision function that gives the signed distance of a sample from the decision boundary. This signed distance can be used as a confidence score. A positive value indicates one class, while a negative value indicates the other. The absolute value of the score can be used as a measure of confidence. However, SVMs don't directly provide a percentage chance or probability estimate like some other classifiers.

## Q5

Primal or Dual Form for Large Datasets:

For training on large datasets with millions of instances and hundreds of features, it's generally more efficient to use the primal form of the SVM problem. The primal form is well-suited for linear SVMs in high-dimensional spaces. The dual form is useful when the number of features is greater than the number of instances.

## Q6

Raising or Lowering Gamma and C for Underfitting:

To address underfitting in an SVM with an RBF kernel, you can try:

Increasing Gamma: This makes the decision boundary more flexible, which can help capture complex patterns in the data.

Decreasing C: Reducing the regularization parameter C allows for more training errors, which can help when the model is too constrained.

## Q7

Setting QP Parameters for Soft Margin Linear SVM:

To solve the soft margin linear SVM classifier problem with a Quadratic Programming (QP) solver, the QP parameters should be set as follows:

H: The Hessian matrix based on the training data.

f: The vector representing the linear coefficients of the decision function.

A: The matrix that defines equality constraints.

b: The vector of equality constraint values.

## Q8

# Import necessary libraries

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.svm import LinearSVC, SVC

from sklearn.linear\_model import SGDClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Create a linearly separable dataset

X, y = make\_classification(n\_samples=100, n\_features=2, n\_informative=2, n\_redundant=0, random\_state=42)

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train LinearSVC

linear\_svc = LinearSVC()

linear\_svc.fit(X\_train, y\_train)

# Train SVC with linear kernel

svc = SVC(kernel='linear')

svc.fit(X\_train, y\_train)

# Train SGDClassifier with SVM loss

sgd\_classifier = SGDClassifier(loss='hinge', alpha=0.01, max\_iter=1000, random\_state=42)

sgd\_classifier.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred\_linear\_svc = linear\_svc.predict(X\_test)

y\_pred\_svc = svc.predict(X\_test)

y\_pred\_sgd\_classifier = sgd\_classifier.predict(X\_test)

# Calculate accuracies

accuracy\_linear\_svc = accuracy\_score(y\_test, y\_pred\_linear\_svc)

accuracy\_svc = accuracy\_score(y\_test, y\_pred\_svc)

accuracy\_sgd\_classifier = accuracy\_score(y\_test, y\_pred\_sgd\_classifier)

# Print accuracies

print("Accuracy of LinearSVC:", accuracy\_linear\_svc)

print("Accuracy of SVC (Linear Kernel):", accuracy\_svc)

print("Accuracy of SGDClassifier (SVM Loss):", accuracy\_sgd\_classifier)

These classifiers should perform similarly on the linearly separable dataset, as they aim to find a linear decision boundary. The accuracies of all three classifiers ARE high, indicating successful separation of classes.

## Q9

from sklearn.datasets import fetch\_openml

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.svm import SVC

from sklearn.metrics import precision\_score

# Load the MNIST dataset

mnist = fetch\_openml('mnist\_784')

X, y = mnist.data, mnist.target.astype(int)

# Split the dataset into a training set and a smaller validation set

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.1, random\_state=42)

# Define the SVM classifier with OvR strategy

svm\_classifier = SVC(decision\_function\_shape='ovr')

# Define hyperparameter grid for tuning

param\_grid = {

'C': [0.1, 1, 10], # Regularization parameter

'kernel': ['linear', 'rbf'], # Kernel type

# Add other hyperparameters as needed

}

# Perform hyperparameter tuning using grid search

grid\_search = GridSearchCV(svm\_classifier, param\_grid, cv=3, n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

# Get the best hyperparameters

best\_params = grid\_search.best\_params\_

# Train the SVM classifier with the best hyperparameters

best\_svm\_classifier = SVC(\*\*best\_params, decision\_function\_shape='ovr')

best\_svm\_classifier.fit(X\_train, y\_train)

# Make predictions on the validation set

y\_val\_pred = best\_svm\_classifier.predict(X\_val)

# Calculate precision on the validation set

precision = precision\_score(y\_val, y\_val\_pred, average='weighted')

print("Best Hyperparameters:", best\_params)

print("Validation Precision:", precision)

GOT Precision in the range of 95-99%.

## Q10

import numpy as np

from sklearn.datasets import fetch\_california\_housing

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.svm import SVR

from sklearn.metrics import mean\_squared\_error

# Load the California housing dataset

california\_housing = fetch\_california\_housing()

X, y = california\_housing.data, california\_housing.target

# Split the dataset into a training set and a test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the SVR regressor

svr\_regressor = SVR()

# Define hyperparameter grid for tuning

param\_grid = {

'C': [0.1, 1, 10], # Regularization parameter

'kernel': ['linear', 'rbf'], # Kernel type

# Add other hyperparameters as needed

}

# Perform hyperparameter tuning using grid search

grid\_search = GridSearchCV(svr\_regressor, param\_grid, cv=3, n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

# Get the best hyperparameters

best\_params = grid\_search.best\_params\_

# Train the SVR regressor with the best hyperparameters

best\_svr\_regressor = SVR(\*\*best\_params)

best\_svr\_regressor.fit(X\_train, y\_train)

# Make predictions on the test set

y\_test\_pred = best\_svr\_regressor.predict(X\_test)

# Calculate the mean squared error (MSE) on the test set

mse = mean\_squared\_error(y\_test, y\_test\_pred)

print("Best Hyperparameters:", best\_params)

print("Test MSE:", mse)