1. SVM in sklearn

- Gamma (γ) parameter in Radial Basis Function (RBF) kernel behaves as the inverse of standard deviation (σ) in a Gaussian kernel.
- SVM regularization parameter (C) controls the penalty for misclassified training examples ($\propto \frac{1}{\lambda}$).

First, the iris dataset is observed by using a plot function in the *matplotlib* library. Only the first two features ('Sepal length' and 'Sepal width') were taken as we are plotting on a 2d plane.

```
iris = load_iris()
X = iris.data[:, :2]
y = iris.target
```

Then, a comparison between different values of gamma and C has been done in order to observe the influence of each parameter prior moving to the grid search.

```
C = 1.0  # SVM regularization parameter
models = (svm.SVC(kernel='rbf', gamma=0.7, C=1),
          svm.SVC(kernel='rbf', gamma=0.7, C=10),
          svm.SVC(kernel='rbf', gamma=0.7, C=100),
          svm.SVC(kernel='rbf', gamma=0.7, C=1000))
models = (clf.fit(X, y) for clf in models)
# title for the plots
titles = ('C=1', 'C=10', 'C=100', 'C=1000')
```

Similarly, the other figure is obtained by making C a constant and vary the value of γ.

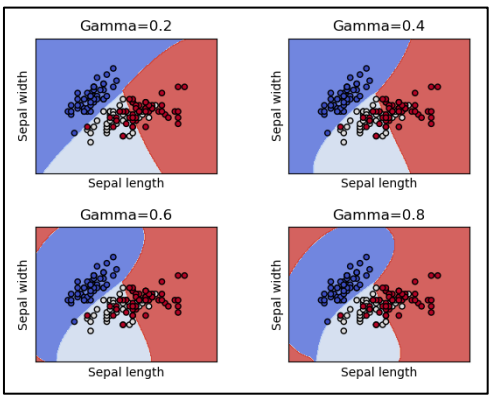

Figure 1.1: Behavior of SVM Classification w.r.t. Gamma (keeping C as a constant)

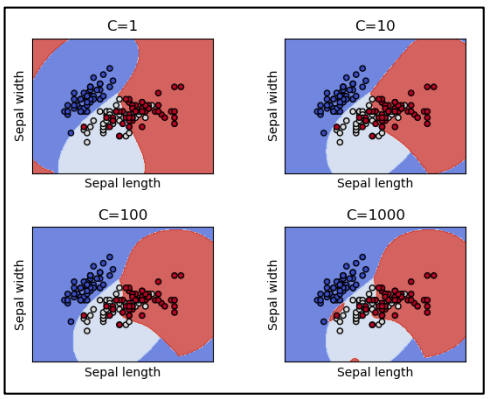

Figure 1.2: Behavior of SVM Classification w.r.t. C (keeping Gamma as a constant)

Observations:

Figure 1.1: For lower gamma values, the decision boundaries look more linear (cannot identify the complexity of 'shape' of the data) whereas it gets more non-linear for larger gamma values.

Figure 1.2: The decision boundary classifies more training examples correctly as the SVM regularization parameter is increased.

Then, the dataset is split into training and test data by getting a random 20% portion as the test data. Also, the datasets are normalized prior sending to the grid search.

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0, shuffle=True)
```

Followed by, a grid search is performed using *sklearn.model_selection.GridSearchCV()* function to get the parameters producing the highest score.

```
C_range = np.array([0.1, 1, 10, 100, 1000])
gamma_range = np.array([0.001, 0.01, 0.1, 0.3, 0.5, 0.8, 1])

param_grid = dict(gamma=gamma_range, C=C_range)
cv = StratifiedShuffleSplit(test_size=0.2, random_state=42)
grid = GridSearchCV(estimator=SVC(), param_grid=param_grid, cv=cv)
grid.fit(X_train, y_train)
print("The best parameters from Training are %s with a score of %0.4f"
      % (clf.best_params_, clf.best_score_))
```

The best parameters got from the grid search was $C = 10$ and $\gamma = 0.01$.

Training Accuracy = 85.42%

Test Accuracy = 73.33%

Further, two heatmaps were generated to observe the accuracy variation with γ and C.

```
train_scores = np.zeros((5,7))
test_scores = np.zeros((5,7))
for c in range(len(C_range)):
    for gamma in range(len(gamma_range)):
        svc = SVC(C=C_range[c], gamma=gamma_range[gamma], kernel='rbf')
        svc.fit(X_train, y_train)
        train_scores[c, gamma] = svc.score(X_train, y_train)
        test_scores[c, gamma] = svc.score(X_test, y_test)


class MidpointNormalize(Normalize):

    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))


# Heatmap for Training Accuracy
plt.figure(figsize=(8, 6))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(train_scores, interpolation='nearest', cmap=plt.cm.hot,
           norm=MidpointNormalize(vmin=0.2, midpoint=0.732))
plt.xlabel('Gamma')
plt.ylabel('SVM Regularization Parameter (C)')
plt.colorbar()
plt.xticks(np.arange(len(gamma_range)), gamma_range, rotation=45)
plt.yticks(np.arange(len(C_range)), C_range)
plt.title('Training Accuracy')
plt.show()
```

Code snippets for Midpoint normalization and heatmap are referred from https://scikit-learn.org/stable/model_selection.html#model-selection.

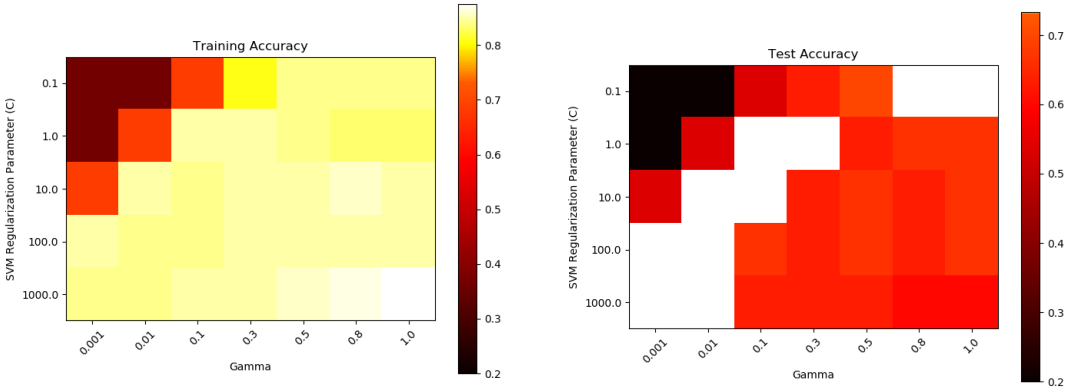

Figure 1.3: Heatmaps for training and test accuracies

There are lot of parameter tunings which give high training accuracies around the bottom-right corner of the Training accuracy heatmap but, they give much less test accuracies (second heatmap). Higher training accuracies lead to overfit the model to the dataset.

So, as per the best parameter tuning received from the grid search and also by observing, we can accept the aforementioned values.

2. CIFAR-10 using sklearn

Going through the code:

i. Loading the dataset and splitting to training and test data.

ii. Getting key points and descriptors of 1000 images using *cv2.xfeatures2d.SIFT_create().detectAndCompute(grayscale_image, None)*.
Then, all SIFT features and descriptors are taken to a single numpy array using *np.concatenate()*.

```
N = 1000
sift_features = np.empty(shape=(0, 128))
for img in X_train[:N, :, :, :]:
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    extractor = cv2.xfeatures2d.SIFT_create()
    kp, des = extractor.detectAndCompute(gray_img, None) #Keypoints &
Descriptors
    if des is None:
        continue
    assert des.shape[1] == 128, 'wrong shape'
    sift_features = np.concatenate([sift_features, des], axis=0)
```

iii. Clustering them to 50 cluster centers using *cluster.KMeans().fit()*.

```
# kmeans clustering
k_means = KMeans(n_clusters=50, random_state=0).fit(sift_features)
```

iv. Generating feature vectors for training and test data separately. Only first part of the code is below.

```
feature_vectors = np.zeros(shape=(N, 50))
for i, img in enumerate(X_train[:N, :, :, :]):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    extractor = cv2.xfeatures2d.SIFT_create()
    kp, des = extractor.detectAndCompute(gray, None)
    if des is None:
        continue
    hist = k_means.predict(des)
    for val in hist:
        feature_vectors[i][val] += 1
```

v. Creating a SVM model and fit feature vectors and labels. The training and test accuracies gained from the scores of the above model is as follows.

```
model = svm.SVC(
                kernel='rbf',
                C=10,
                gamma=0.01
            )
model = model.fit(feature_vectors, labels)

train_accuracy = model.score(feature_vectors, labels)
test_accuracy = model.score(test_vectors, test_labels)
```

Vocabulary Size	=	50
Training Accuracy	=	57.2%
Test Accuracy	=	21.4%

3. Our SVM using cvxopt

Function *qp* is an interface for quadratic programs. ***cvxopt.solvers.qp()*** solves the pair of primal and dual convex quadratic programs.

$$minimize\ \frac{1}{2}x^T Px\ +\ q^T x$$

$$subject\ to\ Gx\ \leq\ h\ \ and$$

$$Ax\ =\ b$$

The 6 parameters of ***cvxopt.solvers.qp()*** have been defined in the following manner.

```
# Gram matrix
k_mat = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        k_mat[i, j] = self.kernel(x[i], x[j], self.param)

P = cvxopt.matrix(np.outer(y, y) * k_mat)
q = cvxopt.matrix(np.ones(n_samples) * -1)
A = cvxopt.matrix(y, (1, n_samples), tc='d')
b = cvxopt.matrix(0.0)

if self.C is None:
    G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
    h = cvxopt.matrix(np.zeros(n_samples))
else:
    tmp1 = - np.identity(n_samples)
    tmp2 = np.identity(n_samples)
    G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
    tmp1 = np.zeros(n_samples)
    tmp2 = np.ones(n_samples) * self.C
    h = cvxopt.matrix(np.hstack((tmp1, tmp2)))

# Solving
solution = cvxopt.solvers.qp(P, q, G, h, A, b)
```

The results gained from the Convex Optimization (cvxopt) solver for the best parameter tuning from question 1 are reported below.

Dataset: iris

Training Accuracy = 60.83%
Test Accuracy = 40%

In question 1 we got a training accuracy of 0.8542 and a test accuracy of 0.7333 for the same dataset. Above results from convex optimization (cvxopt) are considerably lesser than previous results. SVM uses a one-vs-one classifier whereas cvxopt uses a one-vs-rest classifier. This is a main reason for the deviation of results.

The following training (top value) and test (bottom value) accuracies were resulted for other parameters of C and γ grid. Used the same random_state as in question 1 to split training and test data.

C \ γ	***0.001***	***0.01***	***0.1***	***0.3***	***0.5***	***0.8***	***1***
0.1	Tr:0.325 Te:0.367	0.5 0.4	0.617 0.467	0.367 0.2	0.367 0.2	0.367 0.2	0.367 0.2
1	0.475 0.4	0.683 0.533	0.325 0.367	0.325 0.367	0.367 0.2	0.367 0.2	0.367 0.2
10	0.767 0.533	0.608 0.4	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367
100	0.833 0.633	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367
1000	0.833 0.733	0.328 0.367	0.683 0.567	0.325 0.367	0.325 0.367	0.325 0.367	0.325 0.367

As we can see in the table of results, C = 1000 and γ = 0.001 tuning gives the best scores in cvxopt solver (highlighted in orange).

Solutions to the problems faced:

- ✓ *numpy+mkl* package was needed.
- ✓ *numpy* should be imported before importing cvxopt.

4. Two-Layer NN on TensorFlow

Studying the given code:

i. The code is written using the *keras* neural-network Python library.
ii. Dataset:
    a. Consist of 50000 RGB images with size 32×32.
    Log output:
    ```
    x_train shape: (50000, 32, 32, 3)
    50000 train samples
    10000 test samples
    ```
    CIFAR-10 data is split into training and test datasets as 50000 and 10000 examples respectively. *Log output:*
    ```
    Train on 50000 samples, validate on 10000 samples
    ```
iii. 32 number of examples were taken as the batch; classified into 10 classes/outputs.
iv. The 'RMSprop' optimizer in *keras* is used which is usually a good choice for recurrent NNs. Learning Rate is tuned as 0.0001 where it's decay constant is set as 1×10^{-6}.
v. Loss function: 'categorical_crossentropy' loss function have used ('mean_squared_error' is more popular).
vi. Input data is normalized by dividing 255 before fitting it to the model.
vii. Layers:
    a. 'RELU' non-linearity activation function is used in the first layer of NN.
    b. The final output is taken through a 'SOFTMAX' activation layer.
viii. Then, 10 epochs were run for all 50000 training samples.

Results:

- Training:
  According to the log outputs, after 10 epochs, a loss of 1.3592 was there.
  Training Accuracy = 52.07%

- Testing:
  Log output:
  ```
  Test Loss: 1.4418268886566161
  Test Accuracy: 0.4882
  ```
  Test Accuracy = 48.82%