ELL409 Assignment 2

Ritvik Gupta
2019MT10512

1 Part 1

In this part of assignment, we have implemented Support Vector Machines and used it to solve both binary and multiclass classification problems. We have trained the SVMs on the personalised data set provided and tested the performance on a separate part of the data set used for validation.

1.1 SVM using standard library and convex optimization package

We have implemented SVM using two methods: using a standard library (LIBSVM) and directly using a convex optimization package (CVX).

1.1.1 Using LIBSVM

LIBSVM has options to choose the kernel function and various hyper-parameters. The code for doing the same is given below.

```
params = f'-t {self.kernel} -c {self.c} -g {self.gamma} -q'
model = svm.svm_train(train_y, train_x, params)
pred_labels, pred_acc, pred_val = svm.svm_predict(valid_y, valid_x, model, options='-q')
```

Listing 1: For training and testing SVMs using LIBSVM

1.1.2 Using CVX

CVX is a convex optimization package, so we cannot directly train an SVM using CVX. We first have to solve the dual SVM problem using CVX and then use the dual solution to find the solution to the primal SVM problem.
We know that the dual SVM problem is given as:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)}y^{(j)} < x^{(i)}, x^{(j)} > -\sum_{i=1}^{m}\alpha_i \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^{m}\alpha_i y^{(i)} = 0 \end{aligned} \tag{1}$$

This is a quadratic optimization problem. To solve this problem using CVX, we need to re-write this problem in the CVX notation. The notation is given by:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T P\alpha + q^T\alpha \\ \text{s.t.} \quad & G\alpha \preceq H \text{ and } A\alpha = b \end{aligned} \tag{2}$$

On re-writing the dual SVM problem in this notation, we get:

$$P_{ij} = (y^{(i)}y^{(j)}) < x^{(i)}, x^{(j)} > \qquad A = y^T \qquad b = 0$$

$$q = \begin{bmatrix} -1 \\ -1 \\ -1 \\ \vdots \\ \vdots \\ -1 \\ -1 \\ -1 \end{bmatrix} \qquad G = \begin{bmatrix} -1 & 0 & \dots & 0 \\ 0 & -1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -1 \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \qquad h = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ C \\ C \\ \vdots \\ C \end{bmatrix}$$

We convert the training data given to us into this form using the code below:

```
def get_matrices(train_x, train_y)
    n = train_y.shape[0]
    P = np.outer(train_y, train_y) * get_xmat(train_x, train_x)
    q = -1.0 * np.ones(train_y.shape)
    G = np.concatenate((-1.0*np.eye(n), np.eye(n)))
    h = np.concatenate((np.zeros(train_y.shape), c*np.ones(train_y.shape)))
    A = train_y.reshape(1, n)
    b = 0.0
```

Listing 2: For converting the data into the CVX notation

After solving the dual problem using CVX, we get the optimal values of alpha. We use these to then find the value of b and the support vectors. The support vectors are those training examples for which the values of α are 0 or C. Since, the solution found by the optimization package is only a numerical approximation, we take $\epsilon = 10^{-10}$ as the tolerance for approximation errors.
After finding the support vectors, we find the value of b using the formula from CS229 Notes.

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T}\phi(x^{(i)}) + \min_{i:y^{(i)}=1} w^{*T}\phi(x^{(i)})}{2}$$

Also, $w^{*T}\phi(x) = \sum_{i=1}^{m} \alpha_i y^{(i)} < x^{(i)}, x >$
We use these results to find the value of b and the support vectors.
We can use the same results as above to find the predictions for validation set. The prediction can be simply given by:

$$pred(x) = \begin{cases} +1 & \sum_{i=1}^{m} \alpha_i y^{(i)} < x^{(i)}, x > +b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

```
EPS = 1e-10
def get_sv_b(train_x, train_y):
    sv_zero = np.where(alphas > EPS)
    sv_c = np.where(alphas <= c-EPS)
    sv_indices = np.intersect1d(sv_zero, sv_c)
    x_sv, y_sv = train_x[sv_indices], train_y[sv_indices]
    alphas = alphas[sv_indices]
    vals = (alphas * y_sv * get_xmat(x_sv, x_sv)).sum(0)
    pos_indices = np.where(y_sv == 1)[0]
    neg_indices = np.where(y_sv == -1)[0]
    M = max([vals[i] for i in neg_indices])
    m = min([vals[i] for i in pos_indices])
    b = -(M+m)/2
    return x_sv, y_sv, b
```

Listing 3: For finding the support vectors and the value of b

1.1.3 Comparing both the implementations

For comparing both the implementations, we first tune the parameters using the LIBSVM formulation and then use the same parameters on the CVX formulation. We compare the implementations on binary classification of classes 0 and 1.

Linear Kernel

For $C = 10$,

```
Using LIBSVM Model:
Time Taken: 0.010381221771240234s
Training Accuracy: 100.0
Validation Accuracy: 98.85057471264368
Using CVX Model:
Time Taken: 0.07333993911743164s
Training Accuracy: 96.03960396039604
Validation Accuracy: 97.1264367816092
```

From the above results, we can see that LIBSVM takes much less time as compared to CVX (more than 7 times lesser time). This is mainly due to the fact that LIBSVM is designed to specifically optimize SVM problems whereas in CVX, we used a generic Quadratic Solver. Also the training and validation set accuracy for the CVX implementaion is a bit lower. This can be due to numerical approximation errors in CVX.

Gaussian Kernel

For $C = 5, \gamma = 0.1$,

```
Using LIBSVM Model:
Time Taken: 0.018995046615600586s
Training Accuracy: 100.0
Validation Accuracy: 97.12643678160919
Using CVX Model:
Time Taken: 0.06254911422729492s
Training Accuracy: 100.0
Validation Accuracy: 97.1264367816092
```

In this case, we can clearly see that the results obtained are exactly the same. However, the CVX implementation still takes much more time than the LIBSVM Model.

1.2 Binary Classification

Here, we consider the just two classes out of the 10 classes and train SVM for binary classification. First, we consider the classes 1 and 2.

For (1,2):

For these classes, the classifier using a linear kernel gets 100% training and validation accuracy for $C = 5$. As, we are able to achieve complete classification using the linear kernel only, we need not use a more complex kernel unnecessarily. Thus, the data in this case is linearly separable.

For (6,9):

For these classes, the earlier model with linear kernel and $C = 5$, only achieves a training accuracy of 96.17% and a validation accuracy of 86.47%. For these classes, the best model with linear kernel is also only able to achieve about 90% accuracy. This is a case of underfitting.

So, we increase the complexity of the model by taking the Gaussian kernel. Using the Gaussian kernel, with $C = 5$ and $\gamma = 0.05$, the model gets an training and validation accuracy of 100% and 99.41% respectively.

For (7,8):

SVM classifier using a Gaussian kernel gets a training and validation accuracy of 100% and 99.41% respectively. This is a case of overfitting. An SVM model with linear kernel and $C = 5$ achieves complete classification on the same data set suggesting that the more complex Gaussian kernel model was overfitting the training data.

On decreasing the number of features considered, the accuracy decrease in all cases. The accuracy decreases from 100% to 98.94% in the first case, from 98.23% to 95.88% in the second case and finally from 100% to 98.88% in the third case. This is expected as the training data is reduced when the number of features are decreased.

The optimal hyperparameters and the kernel function is different for each pair of classes. For the first and third cases, even a linear kernel is sufficient while for the second case, a Gaussian kernel is required as the linear kernel model underfits the data.

This is can be explained by the fact that these examples are from handwritten digit data. Since these are low-dimensional representations of handwritten digits, some pairs like (6,9) are more difficult to separate in comparison to other cases. Pairs like (6,9) are very similar visually and hence, they cannot be separated by a linear decision boundary only. They require a different kernel in order to capture the more complex decision boundary involved.

1.3 Multiclass Classification

Here, instead of considering each of the classes separately in pairs, we consider the all the classes together and train a multiclass SVM classifier. Since, SVMs do not support multiclass classification natively, we employ some other strategies to build multiclass classifiers using SVMs. There are mainly two such strategies:

- One-versus-Rest: In this method, we train a classifier for each class considering the data from that class as positive and the remaining data as negative. Thus, there are in total n_c classifiers.
- One-versus-One: In this method, we train classifiers for each of the pairs of classes. There are in total $\binom{n_c}{2}$ classifiers in this case. The decision is taken by considering the majority results from the classifiers.

The standard library (LIBSVM) that we have used for this uses the one-versus-one method.

Results:
From the graph, we can see that the validation accuracy peaks at around $C = 0.1$. Thus, we keep $C = 0.1$.

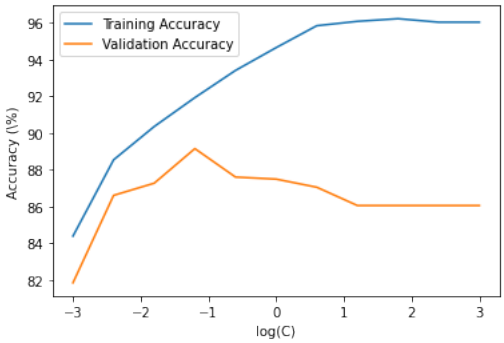


Figure 1: Accuracy v/s C for SVM with linear kernel

For SVM with linear kernel, $C = 0.1$,

```
Time Taken: 0.0989527702331543s
Training Accuracy: 92.12786259541986
Validation Accuracy: 88.93805309734513
```

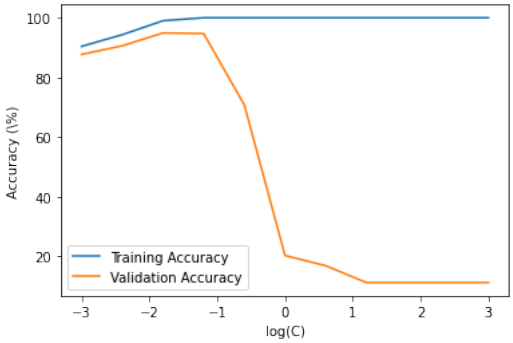


Figure 2: Accuracy v/s γ for SVM with Gaussian kernel ($C = 5$)

From the graph, we can see that the validation accuracy peaks at around $\log(C) = -1$. By slightly tweaking the value of γ near this peak, we find $\gamma = 0.025$ such that a validation accuracy of 95.02% is obtained in this setting.
For SVM with Gaussian kernel, $C = 5$, $\gamma = 0.025$,

```
Time Taken: 0.13340091705322266s
Training Accuracy: 99.76145038167938
Validation Accuracy: 95.02212389380531
```

The accuracy in the multiclass classification setting is much lower than the accuracy in the binary classification setting. This may be due to the fact that in the binary classification problem, we adjusted the hyperparameters for each class pair separately but in multiclass setting we keep a single set of hyperparameters for each of the one-versus-one classifiers. The hyperparameter setting is obviously different from the binary case as here we have to choose the setting which works well for all the classes together.
Again, the accuracy decreases on considering only the first 10 features instead of all the 25 features. The validation accuracy drops to 81.28% from 88.93% in the linear kernel model and to 88.93% from 95.02% in

the Gaussian kernel model. This shows that all the different features are useful and more features allow the model to classify the data on a much better level.

1.4 Simplified SMO Algorithm

In this part, we implement a simplified version of the Sequential Minimal Optimization algorithm. The code for the same can be seen below:

```
def solve(train_x, train_y, max_passes):
    n = train_x.shape[0]
    alphas, b = np.zeros(n), 0.0
    passes = 0
    while passes < max_passes:
        n_ch_alphas = 0
        for i in range(n):
            xi, yi = train_x[i], train_y[i]
            ei = f_x(xi, train_x, train_y) - yi
            if (yi*ei < -TOL and alphas[i] < c) or (yi*ei > TOL and alphas[i] > 0):
                j = random.choice(list(range(0, i)) + list(range(i+1, n)))
                xj, yj = train_x[j], train_y[j]
                ej = f_x(xj, train_x, train_y) - yj
                alpha_io, alpha_jo = alphas[i], alphas[j]
                L, H = get_lh(yi, yj, i, j)
                if L == H:
                    continue
                eta = get_eta(xi, xj)
                if eta >= 0:
                    continue
                alphas[j] -= (yj*(ei-ej)) / eta
                alphas[j] = min(H, max(alphas[j], L))
                if abs(alphas[j]-alpha_jo) < TOL:
                    continue
                alphas[i] += yi*yj*(alpha_jo-alphas[j])
                b = get_b(ei, ej, xi, xj, yi, yj, alpha_io, alpha_jo, i, j)
                n_ch_alphas += 1
        if n_ch_alphas == 0:
            passes += 1
        else:
            passes = 0
    return self.alphas
```

The above Python code follows the pseudocode for simplified SMO from the CS229 Notes.

Results:
For comparing the three implementations, we consider the classes (0,1) for binary classification.
For SVM with linear kernel, $C = 10$,

Model	Training Accuracy	Validation Accuracy	Time Taken
LIBSVM	100.0%	98.85%	0.0111s
CVX	96.04%	97.13%	0.0745s
SMO	100.0%	98.41%	2.4747s

As we can clearly see, the results obtained by our implementation of SMO are almost as good as the standard library implementation in this case. However, this implementation of SMO is much slower than the LIBSVM

model (almost 250 times slower).

For SVM with Gaussian kernel, $C = 5$, $\gamma = 0.1$,

Model	Training Accuracy	Validation Accuracy	Time Taken
LIBSVM	100.0%	97.13%	0.019s
CVX	100.0%	97.13%	0.06s
SMO	100.0%	100.0%	9.29s

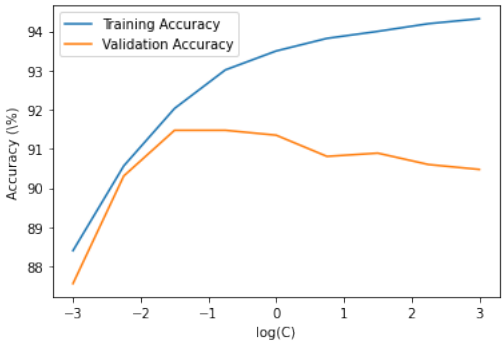
In this case, the results obtained from our implementation of SMO achieves complete classification but takes a lot of time (almost 500 times more time than the LIBSVM model).

2 Part 2

Here, we have to train a multiclass SVM model on the extended version of the data set from Part 1. This data set has 8000 instances of the same 25-dimensional data. Apart from this, we also have a test set for which we have to predict the class labels.
For identifying the optimal values of hyperparameters and the optimal kernel, we split the data set into training and validation sets and use the validation set accuracy to determine optimal hyperparameter settings.

Attempt 1: SVM with Linear Kernel

We start with training an SVM with linear kernel. We plot the graph of Accuracy v/s C in order to find the optimal value of C. From the figure we can easily see that the validation accuracy gets saturated after



$\log(C) = 1.5$. Therefore, the optimal value must be near about $C = 0.031$.
After tweaking the parameters around the obtained value, we obtain a validation accuracy of 91.51% at $C = 0.04$. Using this value of C, the model obtained a test accuracy of 90.75% on the unseen test data set. As this model has low training accuracy ans well as low validation accuracy, this is a case of underfitting. Hence, we have to increase the complexity of the model. Thus, we next consider SVM with Gaussian kernel.

Attempt 2: SVM with Gaussian Kernel

In this attempt, we train an SVM with Gaussian kernel.
We use grid search to find the optimal pair of hyperparameters (C, γ). We find the pair (C, γ) for which the validation accuracy is the highest.
The hyperparameters found by this method were $C = 4$, $\gamma = 0.05$.
Using these hyperparameters, an accuracy of **96.75%** was obtained on the unseen test data (on Kaggle).