1 Soft-Margin SVM Dual Derivation

HW #2

1.

a) (i) $y^{(i)}(\bar{\theta} \cdot \bar{x}^{(i)} + b) \geq 1 - \xi_i$ s.t. $\xi_i \geq 0 \quad \forall i \in \{1 \dots n\}$

$y^{(i)}(\bar{\theta} \cdot \bar{x}^{(i)} + b) - 1 + \xi_i \geq 0$

$$L(\bar{\theta}, \bar{\alpha}, b, \bar{\xi}) = \frac{\|\bar{\theta}\|^2}{2} + C\sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \alpha_i \left(y^{(i)}(\bar{\theta} \cdot x^{(i)} + b) - 1 + \xi_i \right)$$

(ii) $\frac{\partial}{\partial \theta}\left(L(\bar{\theta}, \bar{\alpha}, b, \bar{\xi})\right) = 0$ will help to find the point where the Lagrangian is minimized.

$$\frac{\partial}{\partial \theta}\left[\frac{\|\bar{\theta}\|^2}{2} + C\sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n} \alpha_i \xi_i - \sum_{i=1}^{n} \alpha_i y^{(i)} b - \sum_{i=1}^{n} \alpha_i y^{(i)} (\bar{\theta} \cdot x^{(i)}) \right] = 0$$

$$\Rightarrow \bar{\theta} - \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} = 0$$

$$\theta^* = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \quad \text{// optimal value of } \bar{\theta}$$

Now we take the partial derivatives w.r.t. to b and ξ_i

$$\partial/\partial b \left(L(\bar{\theta}, \bar{\alpha}, b, \bar{\xi}) \right) \Rightarrow \sum_{i=1}^{n} \alpha_i y^{(i)} = 0$$

α_i and $y^{(i)}$ are the only ones multiplied to b in the last summation, hence they remain. The rest are constants w.r.t. to b and equal 0.

$$\partial/\partial \xi \left(L(\bar{\theta}, \bar{\alpha}, b, \bar{\xi}) \right) \Rightarrow C - \alpha_i = 0$$

$\partial/\partial \xi \left(C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \xi_i \right)$ simplifies to the answer above. Rest of the equation $= 0$.

Hence, $0 \leq \alpha_i \leq C$.

b) If $\alpha_i = 0$, it implies that $y^{(i)}(\theta \cdot x^{(i)} + b) > 1$ because α_i returns the max. value at $\alpha_i = 0$ when $y^{(i)}(\theta \cdot x^{(i)} + b)$ since $0 \geq 1 - \xi_i - y^{(i)}(\theta \cdot x^{(i)} + b)$. Hence all points with $\alpha_i = 0$ are non-support vectors.
$i = 2, 4, 6 \Rightarrow$ support vectors

c) $\theta^* = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} = 0 + 0.322(1)[-1.9, 0] + 0$
$- 0.497(1)[0.5, 0.75] + 0$
$+ 0.175(1)[1.6, -2.5]$
$= [-0.6118, 0] - [0.2485, 0.3728] + [0.28, -0.438]$
$= [-0.5803, -0.8108]$

$b = \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 + 0.322 + 0 - 0.497 + 0 + 0.175$
$= 0 \rightarrow$ implies that it is indeed optimal

$[-4, 1]$: Positively Classified
$\bar{\theta} \cdot \bar{x}^{(i)} + b = [-0.5803, -0.8108] \cdot [-4, 1] = 1.514 \geq 0$

$[2, 5]$: Negatively Classified
$\bar{\theta} \cdot \bar{x}^{(i)} + b = [-0.5803, -0.8108] \cdot [2, 5] = -5.2146 < 0$

Q2

2.1 Comparing Optimization Algorithms

a) generate_polynomial_features(X, M)

```python
def generate_polynomial_features(X, M):
    """
    Create a polynomial feature mapping from input examples. Each element x
    in X is mapped to an (M+1)-dimensional polynomial feature vector
    i.e. [1, x, x^2, ...,x^M].

    Args:
        X: np.array, shape (n, 1). Each row is one instance.
        M: a non-negative integer

    Returns:
        Phi: np.array, shape (n, M+1)
    """
    # TODO: Implement this function
    n = X.shape[0]
    Phi = np.zeros((n, M+1))
    #X_arr = []
    for i in range(X.shape[0]):
        for j in range(M+1):
            Phi[i][j] = X[i]**j
    #Phi.append(X_arr)
    return Phi
```

b)

(i)

ls _gradient-descent

```python
def ls_gradient_descent(X, y, learning_rate=0):
    """
    Implements the Gradient Descent (GD) algorithm for least squares regression.
    Note:
        - Please use the stopping criteria: number of iterations >= 1e6 or |new_loss - prev_loss| <= 1e-10

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)

    Returns:
        theta: np.array, shape (d,)
    """
    # TODO: Implement this function
    d = len(X[0])
    theta = np.zeros((d,))
    prev_loss = calculate_empirical_risk(X, y, theta)
    new_loss = 0
    #step_size = [1e-4, 1e-3, 1e-2, 1e-1]
    num_iterations = 0
    while (num_iterations <= 1e6) and (abs(new_loss - prev_loss) >= 1e-10):
        grad_sum = 0
        for i in range(X.shape[0]):
            grad_sum += (y[i] - np.dot(theta, X[i])) * (-X[i])
        theta = theta - learning_rate * grad_sum/X.shape[0]
        prev_loss = new_loss
        new_loss = calculate_empirical_risk(X, y, theta)
        num_iterations += 1
    print("iterations = ", num_iterations)
    return theta
```

ls_stochastic gradient descent

```
def is_stochastic_gradient_descent(X, y, learning_rate=0):
    """
    Implements the Stochastic Gradient Descent (SGD) algorithm for least squares regression.
    Note:
        - Please do not shuffle your data points.
        - Please use the stopping criteria: number of iterations >= 1e6 or |new_loss - prev_loss| <= 1e-10

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)

    Returns:
        theta: np.array, shape (d,)
    """
    # TODO: Implement this function
    d = len(X[0])
    theta = np.zeros((d,))
    prev_loss = calculate_empirical_risk(X, y, theta)
    new_loss = 0
    step_size = [1e-4, 1e-3, 1e-2, 1e-1]
    num_iterations = 0

    i = 0 #used for accessing current i in X and y matrices during SGD

    while (num_iterations <= 1e6) and (abs(new_loss - prev_loss) >= 1e-10):
        for i in range(X.shape[0]):
            curr_emp = (y[i] - np.dot(theta, X[i])) * (-X[i])  #empirical risk
            theta = theta - learning_rate * curr_emp
            num_iterations += 1
        prev_loss = new_loss
        new_loss = calculate_empirical_risk(X,y, theta)
    print("num_it =", num_iterations)
    return theta
```

Closed_form_optimization

```python
def closed_form_optimization(X, y, reg_param=0):
    """
    Implements the closed form solution for least squares regression.

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        `reg_param`: float, an optional regularization parameter

    Returns:
        theta: np.array, shape (d,)
    """
    # TODO: Implement this function
    d = len(X[0])
    theta = np.zeros((d,))
    #(X transposed x X)^-1
    X_transpose = np.transpose(X)
    Xsquare = np.matmul(X_transpose, X)
    XInvX = np.linalg.inv(Xsquare)

    #X transposed x y
    Xy = np.matmul(X_transpose, y)

    #Theta optimal
    theta = np.matmul(XInvX, Xy)
    return theta
```

(ii)

Algorithm	η	θ_0	θ_1	#iterations	Runtime (s)
GD	10^{-4}	0.2970493	-0.10938172	459712	25.723377242000424
GD	10^{-3}	0.30288373	-0.11917784	65168	3.546101773999908
GD	10^{-2}	0.30472896	-0.12227601	8435	0.4578238510002848
GD	10^{-1}	0.3053143	-0.1232588	1034	0.06125162799980899

SGD	10^{-4}	0.30364222	-0.12053361	709100	2.239500088 9991934
SGD	10^{-3}	0.30465554	-0.12298073	88920	0.289306979 0006848
SGD	10^{-2}	0.30154789	-0.1254963	7960	0.034824602 00029891
SGD	10^{-1}	0.26740424	-0.16397638	2300	0.008231843 99996535
Closed Form	-	0.30558175	-0.12370784	-	0.017103795 000366517

c) The number of iterations on SGD is greater than GD, however its runtimes are significantly lower. The greatest difference in the runtimes is when eta = 10^{-4} . As eta gets larger, we notice that the runtimes of SGD and GD get closer, though SGD continues to run faster. Evidently, this is because GD uses the entire training set whereas SGD just iterates through X's data points. Since we aren't using shuffle, the data points for SGD aren't randomly chosen so we have to make a for loop to iterate through X. As for the convergence criteria, both SGD and GD converged in less than 1000000 iterations because their (new loss - prev loss) was good enough to pass our 2nd criteria. This also shows that our data was linearly separable since we converged even before our limit of a million iterations. The coefficients at convergence of GD were fairly similar, though GD's coefficients would be slightly more accurate. The intercept of the hyperplane would be around 0.3 and the slope would be around -0.12. Also, in the last eta of SGD where eta = 0.1, we can see the values of SGD fluctuate more when compared to smaller eta values. As eta grew larger, we can see that the SGD hyperplane shifted by a larger amount in each update hence resulting in the larger disparity. For smaller eta values, the difference is not as noticeable for SGD.

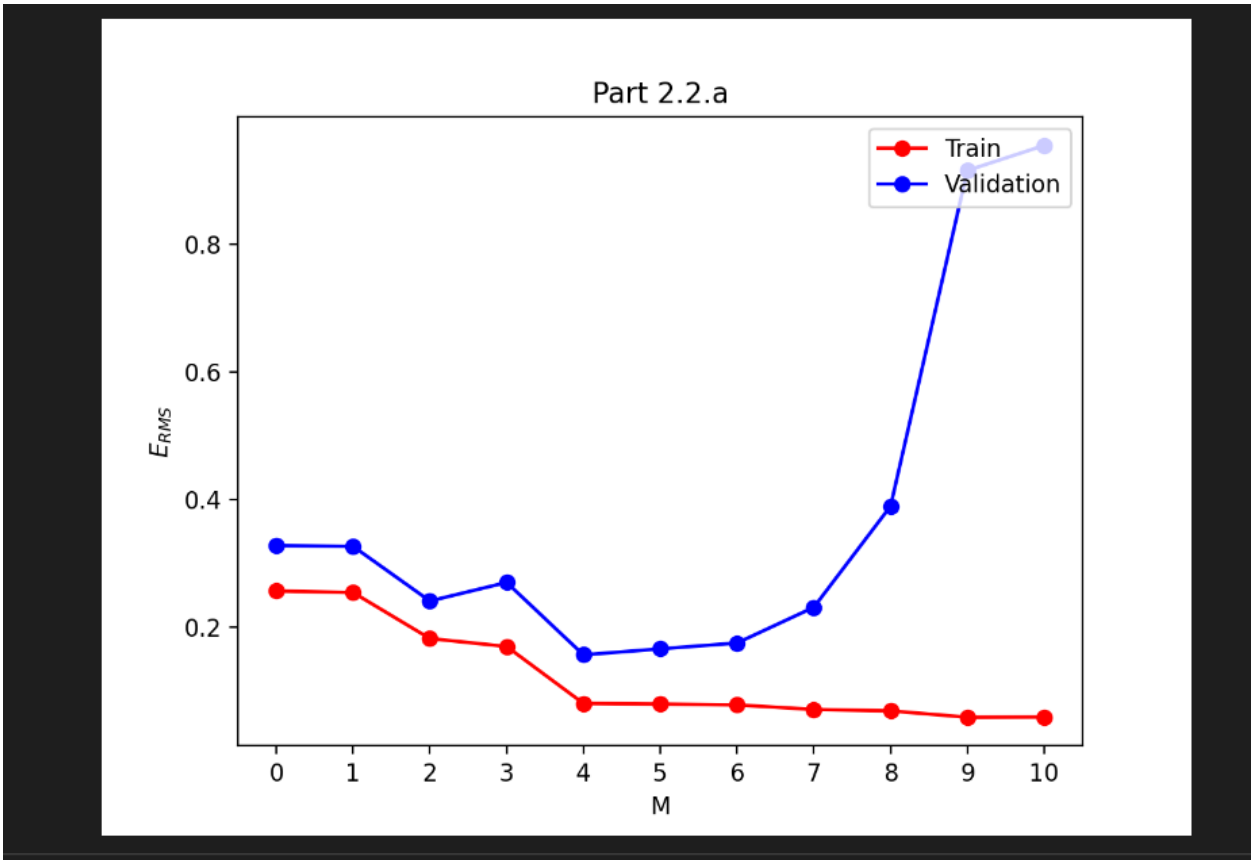
d) The runtime of the closed form optimization algorithm runs faster than SGD at all eta values except for eta = 0.1. Since smaller values of eta make less of a shift in the direction of the hyperplane, the iterations required are naturally more. However, those also result in better hyperplanes. We can see that the closed form solution performs better than SGD in all other cases and its solution is also reasonable relative to our hyperplanes from GD and SGD. Overall, it would be better to use either a smaller eta value to get a more accurate SGD or just use the closed form optimization depending on whether you wish to save runtime. If runtime is a large issue, then SGD with larger eta values will be a better solution since they may run faster than our closed form solution despite not being very accurate. Furthermore, if we have very large d-dimension matrices, taking the inverse will take time and may not be the best solution. In conclusion, it's a matter of efficiency for which route you wish to take.

2.2 Overfitting and Regularization

a) (i) P.S. I added an extra parameter M to my Erms function to iterate through the different M's (0 to 10) because I use generate_polynomial-features inside Erms and that needs an M parameter which I had initially set to 1, but since it needs to change values, I just added M as a parameter.

```
def calculate_RMS_Error(X, y, theta, M = 0):
    """
    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        theta: np.array, shape (d,). Specifies an (d-1)^th degree polynomial

    Returns:
        E_rms: float. The root mean square error as defined in the assignment.
    """
    # TODO: Implement this function
    import math as mth
    E_rms = 0
    E_theta = 0
    sum = 0
    PhiX = generate_polynomial_features(X, M)
    for i in range(len(X)):
        sum = np.dot(theta, PhiX[i])
        sum = (sum - y[i]) ** 2
        E_theta += sum
    E_rms = mth.sqrt(E_theta/len(X))
    return E_rms
```



(ii)

The RMS error at M = 4 has the lowest error and therefore a 4th degree polynomial best fits the data. After that, we can see how M = 5 and onwards in fact increase the RMS error due to overfitting because they are too sensitive to data points. We know this because the RMS tells you how concentrated the data is around our decision boundary, and points that are a bit farther off tend to change higher Ms more easily. Hence, M = 4 is the best polynomial feature mapping. We can see this in how our training error decreases steadily as M increases, however with new validation data, it drastically increases the RMS error. M = 1,2,3 are likely underfitting the data because they are not sensitive enough to fit the training data well and therefore have higher RMS errors.

b) (i)

```
    d = len(X[0])
    theta = np.zeros((d,))

    #create identity matrix and multiply reg_param(lambda) to it
    #X^T times X is a d x d matrix since X = n x d matrix
    id_matrix = np.identity(d)
    lambda_matrix = reg_param * id_matrix

    #Multiples X^T to X
    X_transpose = np.transpose(X)
    Xsquare = np.matmul(X_transpose, X)

    #X transposed times y
    Xy = np.matmul(X_transpose, y)

    #Creates added matrix and Inverses added matrix
    X_add_lambda = lambda_matrix + Xsquare
    InvX = np.linalg.inv(X_add_lambda)

    #Multiplies ivnersed matrix with X^T*y
    theta = np.matmul(InvX, Xy)

    return theta
```

(ii) **Got an error while trying to plot**
Code:

```python
# (b) REGULARIZATION

errors_train = np.zeros((10,))
errors_validation = np.zeros((10,))
L = np.append([0], 10.0 ** np.arange(-8, 1))
# Add your code here

PhiX = generate_polynomial_features(X_train, M = 10)

lambda_arr = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

for i in lambda_arr:
    theta = closed_form_optimization(PhiX, y_train, reg_param = i)
    errors_train[i] = calculate_RMS_Error(X_train, y_train, theta, M = i)
    errors_validation[i] = calculate_RMS_Error(X_validation, y_validation, theta, M = i)

plt.figure()
plt.plot(L, errors_train, '-or', label='Train')
plt.plot(L, errors_validation, '-ob', label='Validation')
plt.xscale('symlog', linthresh=1e-8)
plt.xlabel('$\lambda$')
plt.ylabel('$E_{RMS}$')
plt.title('Part 2.2.b')
plt.legend(loc=2)
plt.savefig('q2_2_b.png', dpi=200)
```

OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

Terminal Error:

```
chaitanya@chaitanya:/mnt/c/Users/chait/OneDrive/Desktop/HW2/Skeleton_Code$ python3 q2_linear_regression.py
========== Part 2.1 ==========
closed form optimization = [ 0.30558175 -0.12370784]
Closed Form Optimization Time:  0.03135070899952552
Done!
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no a
rgument.
========== Part 2.2 ==========
Traceback (most recent call last):
  File "q2_linear_regression.py", line 297, in <module>
    main("dataset/q2_train.csv", "dataset/q2_validation.csv")
  File "q2_linear_regression.py", line 294, in main
    part_2_2(fname_train, fname_validation)
  File "q2_linear_regression.py", line 275, in part_2_2
    errors_train[i] = calculate_RMS_Error(X_train, y_train, theta, M = i)
  File "q2_linear_regression.py", line 62, in calculate_RMS_Error
    sum = np.dot(theta, PhiX[i])
  File "<__array_function__ internals>", line 180, in dot
ValueError: shapes (11,) and (1,) not aligned: 11 (dim 0) != 1 (dim 0)
chaitanya@chaitanya:/mnt/c/Users/chait/OneDrive/Desktop/HW2/Skeleton_Code$
```

(iii) The lambda of value 10^{-5} is the best value for lambda because even though the lambda value of 10^{-3} also gives a similarly small RMS error, the lambda value is the regularization penalty, and we can see that the model rises in RMS error for lambda values greater than 10^{-3} due to overfitting. Since 10^{-3} is on the border of overfitting, it is better to use the 10^{-5} value because even though setting a higher penalty value might make our model more accurate, we have to account for overfitting.

3 Decision Trees

3. a) If a data point is duplicated, the entropy H(Y), will be affected because now you have an extra point adding to your H(Y) as the algorithm goes through your dataset even though it is just a duplicate label.

If every single data point was duplicated, both k and the sum would double, hence resulting in an identical decision tree.

However, duplicating a single data point may not necessarily give $D1 \equiv D2$.

b) Information Gain helps to describe and classify the mixture of classes in a node. Starting out with a node with less than max. 16 will lead to less homogenous nodes being created down the line. This can also cause efficiency problems since you may need more nodes to describe and classify a given class.

Hence, you won't always find a tree that fits the data exactly although it is possible depending on the dataset.

We can see an example of this from 3c) where even though Altitude provides the most information gain, if we use Temperature as our feature, we can still build a decision tree that fits our data exactly, only that we would need more nodes to do so.

c) (i) It will use the node with the maximum info. gain.
i.e. $IG(X, Y) = H(Y) - H(Y|X)$
We need to test which of the 3 features returns the greatest IG value to choose that as our root feature vector.

Suppose $H(D)$ = Dinosaur

$H(D) = \frac{-5}{9}\log\left(\frac{5}{9}\right) - \frac{4}{9}\log\left(\frac{4}{9}\right) = 0.9911$

Humidity = X:

$H(D|X=\text{Low}) = \frac{-1}{2}\log\left(\frac{1}{2}\right) - \frac{1}{2}\log\left(\frac{1}{2}\right) = 1$

$H(D|X=\text{Med}) = 0$ $\quad 1+1=2$

$H(D|X=\text{High}) = \frac{-2}{4}\log\left(\frac{2}{4}\right) - \frac{2}{4}\log\left(\frac{2}{4}\right) = 1$

Temperature = X:

$H(D|X=\text{Low}) = 0$

$H(D|X=\text{Med}) = -2/4\log(2/4) - 2/4\log(2/4) = 1$ $\quad 1+0.9183 = 1.9183$

$H(D|X=\text{High}) = -2/3\log(2/3) - 1/3\log(1/3) = 0.9183$

Altitude = X:

$H(D|X=\text{Low}) = -3/4\log(3/4) - 1/4\log(1/4) = 0.81125$ $\quad 0.81125 + 0.9183 = 1.7296$

$H(D|X=\text{Med}) = -2/3\log(2/3) - 1/3\log(1/3) = 0.9183$

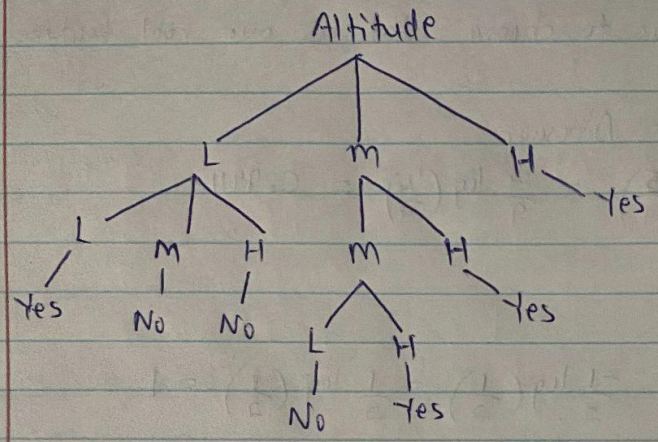
$H(D|X=\text{High}) = 0$

Altitude	< Temperature	< Humidity	$\Rightarrow H(D) - H(D\|X=\text{Alt.})$
1.7296	1.9183	2	= max.

Hence, Altitude should be the feature for the root node followed by Temperature as the 2nd feature and finally Humidity.

(ii) As stated in part (i), the order is going to
Altitude → Temperature → Humidity

// L = Low
// M = Medium
// H = High



Since we went in the order of the most info. gained to the least info. gained, the decision tree above should have the minimal # leaf nodes with 0 error.