Homework 2

Name: Edward Ajayi

Andrew ID: eaajayi

Program: MS EAI

Institution: Carnegie Mellon University Africa

Semester: Spring 2025

Unit: 18-661 - Introduction to Machine Learning for Engineers

Faculty: Prof. Carlee Joe-Wong and Prof. Gauri Joshi

Date: February 21, 2025

Question 1

Given a dataset $X \in \mathbb{R}^{N \times 4}$ consisting of N data points, each with 4 features, and its label y. The features take discrete values:

$$X_1 \in \{1, 2\}$$

$$X_2 \in \{1, 2, 3\}$$

$$X_3 \in \{1, 2, 3, 4\}$$

$$X_4 \in \{1, 2, 3, 4, 5\}$$

Also, there are 4 possible labels: $Y \in \{1, 2, 3, 4\}$.

(a) Number of Free Parameters in Naïve Bayes

We need to estimate:

1. **Class Priors** $P(Y)$, denoted as π.
2. **Conditional Probabilities** $P(X_j|Y)$, denoted as θ.

Computing Class Priors $P(Y)$

Since there are 4 possible class labels:

$$\pi_c = P(Y = c) \quad \text{for } c \in \{1, 2, 3, 4\}$$

Since probabilities must sum to 1, we only need to estimate 3 independent parameters:

$$\pi_4 = 1 - (\pi_1 + \pi_2 + \pi_3)$$

The number of free parameters for class priors is:

$$\text{Total class prior parameters} = 3$$

Computing Conditional Probabilities $P(X_j|Y)$

Each feature X_j takes a certain number of values, and for each class Y, we need to estimate probabilities:

$$\theta_{j,v,c} = P(X_j = v|Y = c)$$

Since we estimate these for 4 classes:

Feature X_j	Possible Values $\lvert X_j \rvert$	Needed Parameters per Class
X_1	2	$2 - 1 = 1$
X_2	3	$3 - 1 = 2$
X_3	4	$4 - 1 = 3$
X_4	5	$5 - 1 = 4$

Table 1: Feature Value Counts and Required Parameters

$$(1 + 2 + 3 + 4) \times 4 = 10 \times 4 = 40$$

Total Free Parameters in Naïve Bayes

Summing class priors and conditional probabilities:

$$\text{Total parameters} = 3(\pi) + 40(\theta) = \mathbf{43}$$

(b) Number of Free Parameters Without Independence Assumption

If we do **not** assume feature independence, we must estimate the full joint probability:

$$P(X_1, X_2, X_3, X_4|Y)$$

For each class, the number of possible feature combinations is:

$$|X_1| \times |X_2| \times |X_3| \times |X_4| = 2 \times 3 \times 4 \times 5 = 120$$

Since probabilities must sum to 1 for each class, we estimate:

$$120 - 1 = 119 \quad \text{parameters per class}$$

For 4 classes, the total number of free parameters is:

$$(119 \times 4) + 3 = \mathbf{479}$$

(c) Advantage of Assuming Conditional Independence

Comparing the two approaches:

- **Naïve Bayes**: 43 parameters
- **Full Joint Distribution**: 479 parameters

Advantages of assuming conditional independence:

Less overfitting: It reduces complexity, therefore leading to better generalization.

Question 2

(a) Prior Probabilities

$$P(y=1) = \frac{4}{8} = 0.5,$$
$$P(y=2) = \frac{2}{8} = 0.25,$$
$$P(y=3) = \frac{2}{8} = 0.25.$$

(b) Feature Vectors for Positive Reviews

Each review is represented as a feature vector $x = [x_1, x_2, \ldots, x_{10}]$, where x_i is the number of times word V_i appears.

$$x_1 = [0, 0, 1, 1, 0, 0, 1, 0, 0, 0],$$
$$x_2 = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0],$$
$$x_3 = [0, 1, 1, 1, 0, 0, 0, 0, 1, 0],$$
$$x_4 = [0, 1, 0, 2, 0, 0, 0, 0, 1, 0].$$

(c) Maximum Likelihood Estimates of θ

The likelihood parameter is given by:

$$\theta_{c,k} = \frac{\text{count of word } k \text{ in class } c}{\text{total words in class } c}.$$

Total word counts:

$$\text{Total words in } y = 1 : 13, \quad \text{Total words in } y = 2 : 6, \quad \text{Total words in } y = 3 : 5.$$

Computing required values:

$$\theta_{1,4} = \frac{4}{13}, \qquad \theta_{1,7} = \frac{2}{13},$$
$$\theta_{2,4} = \frac{1}{6}, \qquad \theta_{2,7} = \frac{1}{6},$$
$$\theta_{3,4} = \frac{1}{5}, \qquad \theta_{3,7} = \frac{1}{5}.$$

(d) Classify "amazing movie"

$$P(x|y=1) = 0.5 \times \frac{4}{13} \times \frac{2}{13} = \frac{4}{169} \approx 0.0237,$$
$$P(x|y=2) = 0.25 \times \frac{1}{6} \times \frac{1}{6} = \frac{1}{144} \approx 0.0069,$$
$$P(x|y=3) = 0.25 \times \frac{1}{5} \times \frac{1}{5} = \frac{1}{100} = 0.01.$$

Since $P(y=1)$ is the highest, "amazing movie" is classified as **Positive**.

(e) Classify "decent movie" Using Laplacian Smoothing

Using $\alpha = 1$:

$$\theta_{1,6} = \frac{1}{23}, \qquad \theta_{1,7} = \frac{3}{23},$$
$$\theta_{2,6} = \frac{2}{16}, \qquad \theta_{2,7} = \frac{2}{16},$$
$$\theta_{3,6} = \frac{1}{15}, \qquad \theta_{3,7} = \frac{2}{15}.$$

Computing probabilities:

$$P(x|y=1) = 0.5 \times \frac{1}{23} \times \frac{3}{23} = \frac{3}{1058} \approx 0.0028,$$
$$P(x|y=2) = 0.25 \times \frac{2}{16} \times \frac{2}{16} = \frac{1}{256} = 0.0039,$$
$$P(x|y=3) = 0.25 \times \frac{1}{15} \times \frac{2}{15} = \frac{2}{900} \approx 0.0022.$$

Since $P(y=2)$ is the highest, "decent movie" is classified as **Neutral**.

Why is Laplacian Smoothing Necessary? Without smoothing, words absent in a class would have zero probability, making classification impossible.

Question 3

Given a training set $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ is the feature vector and $y^{(i)} \in \{0, 1\}$ is the binary label, we define the probability of class $y = 1$ as:

$$p(y = 1 \mid \mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}.$$

The conditional log-likelihood of the training set is given by:

$$L(\mathbf{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}) + (1 - y^{(i)}) \log(1 - p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w})) \right].$$

The gradient of the log-likelihood is:

$$\nabla L(\mathbf{w}) = \sum_{i=1}^{n} \left(y^{(i)} - p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}) \right) \mathbf{x}^{(i)}.$$

(a) Closed-form Solution

Logistic regression does not have a closed-form solution due to the non-linearity of the sigmoid function. Instead, we use gradient ascent:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha \sum_{i=1}^{n} \left(y^{(i)} - p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}) \right) \mathbf{x}^{(i)},$$

where α is the learning rate.

Gradient ascent iteratively adjusts w to move toward the parameters that maximize the log-likelihood. The learning rate α controls how large each step is. If α is too small, convergence is slow; if α is too large, the algorithm may overshoot the optimal solution.

(b) Decision Boundary

The decision boundary occurs where $P(y = 1 \mid \mathbf{x}, \mathbf{w}) = P(y = 0 \mid \mathbf{x}, \mathbf{w})$, which simplifies to:

$$\begin{aligned}
\frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} &= 0.5 \\
1 + e^{-\mathbf{w}^T\mathbf{x}} &= 2 \\
e^{-\mathbf{w}^T\mathbf{x}} &= 1 \\
-\mathbf{w}^T\mathbf{x} &= 0 \\
\mathbf{w}^T\mathbf{x} &= 0.
\end{aligned}$$

This defines a linear decision boundary.

(c) Is Logistic Regression a Linear Classifier?

Yes, logistic regression is a linear classifier because the decision boundary is a linear equation $\mathbf{w}^T\mathbf{x} = 0$.

(d) Adjusting for Misclassification Costs

To reduce false negatives, we can lower the decision threshold τ from 0.5:

$$P(y = 1|\mathbf{x}) \geq \tau, \quad \text{where } \tau < 0.5.$$

Alternatively, we can modify the loss function by assigning different weights w_1 and w_0:

$$L(\mathbf{w}) = -\sum_i \left[w_1 y^{(i)} \log P(y^{(i)}|\mathbf{x}^{(i)}) + w_0(1 - y^{(i)}) \log(1 - P(y^{(i)}|\mathbf{x}^{(i)})) \right].$$

where $w_1 > w_0$ increases the penalty for misclassifying class 1.

These adjustments help optimize the model based on specific misclassification costs.

Question 4

(a) Showing that the Negative Log-Likelihood Equals the Cross-Entropy Loss

The likelihood function for logistic regression is:

$$p(y|X, w) = \prod_{i=1}^{n} p(y_i|x_i, w)$$

where

$$p(y_i = 1|x_i, w) = \frac{1}{1 + \exp(-w^\top x_i)}$$

$$p(y_i = 0|x_i, w) = 1 - \frac{1}{1 + \exp(-w^\top x_i)} = \frac{\exp(-w^\top x_i)}{1 + \exp(-w^\top x_i)}$$

Thus, the likelihood function becomes:

$$\prod_{i=1}^{n} \left(\frac{1}{1 + \exp(-w^\top x_i)} \right)^{y_i} \left(\frac{\exp(-w^\top x_i)}{1 + \exp(-w^\top x_i)} \right)^{1-y_i}$$

Taking the log-likelihood:

$$\ell(w) = \sum_{i=1}^{n} \left[y_i \log \sigma(w^\top x_i) + (1 - y_i) \log(1 - \sigma(w^\top x_i)) \right]$$

Expanding the sigmoid function:

$$\ell(w) = \sum_{i=1}^{n} \left[y_i \log \frac{1}{1 + e^{-w^\top x_i}} + (1 - y_i) \log \frac{e^{-w^\top x_i}}{1 + e^{-w^\top x_i}} \right]$$

Expanding the logarithm:

$$\ell(w) = \sum_{i=1}^{n} \left[-y_i \log(1 + e^{-w^\top x_i}) - (1 - y_i) w^\top x_i - (1 - y_i) \log(1 + e^{-w^\top x_i}) \right]$$

Factor out $\log(1 + e^{-w^\top x_i})$

$$\sum_{i=1}^{n} \left[-(1 - y_i) w^\top x_i - \log(1 + e^{-w^\top x_i}) \right]$$

Note that $-\log(1 + e^{-w^\top x_i}) = \log(1 + e^{w^\top x_1}) - w^\top x_i$

Rearranging:

$$\ell(w) = \sum_{i=1}^{n} \left[y_i w^\top x_i - \log(1 + e^{w^\top x_i}) \right]$$

Taking the negative log-likelihood:

$$-\ell(w) = -\sum_{i=1}^{n} \left[y_i w^\top x_i - \log(1 + e^{w^\top x_i}) \right]$$

This is equal to the cross-entropy loss function.

(b) Proving the Convexity of the Negative Log-Likelihood

A function $f(w)$ is convex if its Hessian matrix $\nabla^2 f(w)$ is positive semi-definite.

We begin by computing the gradient of the loss function:

$$L(w) = \sum_{i=1}^{n} \left[-y_i w^\top x_i + \log(1 + e^{w^\top x_i}) \right]$$

Taking the derivative of the first term:

$$\nabla_w (-y_i w^\top x_i) = -y_i x_i$$

For the second term, using the chain rule:

$$\nabla_w \log(1 + e^{w^\top x_i}) = \frac{e^{w^\top x_i} x_i}{1 + e^{w^\top x_i}}$$

Recognizing that:

$$\sigma(w^\top x_i) = \frac{1}{1 + e^{-w^\top x_i}} = \frac{e^{w^\top x_i}}{1 + e^{w^\top x_i}}$$

we substitute:

$$\nabla_w \log(1 + e^{w^\top x_i}) = \sigma(w^\top x_i) x_i$$

Thus, the full gradient is:

$$\nabla_w L(w) = \sum_{i=1}^{n} \left(\sigma(w^\top x_i) - y_i \right) x_i$$

Now, we compute the second derivative to confirm convexity: Since:

$$\nabla_w \sigma(w^\top x_i) = \sigma(w^\top x_i)(1 - \sigma(w^\top x_i)) x_i$$

The Hessian is:

$$\nabla^2 L(w) = \sum_{i=1}^{n} \sigma(w^\top x_i)(1 - \sigma(w^\top x_i)) x_i x_i^\top$$

Since $\sigma(w^\top x_i)(1-\sigma(w^\top x_i)) \geq 0$ for all i, the Hessian is a sum of positive semi-definite matrices, making it positive semi-definite.

Thus, the loss function $L(w)$ is convex. a sum of positive semi-definite matrices, which makes it positive semi-definite.

(c) Proving the Iterative Weighted Least Squares Update Rule

The Newton-Raphson update formula is:

$$w_{k+1} = w_k - (\nabla^2 L(w_k))^{-1} \nabla L(w_k)$$

From the problem statement:

$$\nabla L(w_k) = -X^\top (y - p_k)$$

$$\nabla^2 L(w_k) = X^\top W_k X$$

where: - W_k is the diagonal matrix with i-th diagonal entry $\sigma(w_k^\top x_i)(1-\sigma(w_k^\top x_i))$ - p_k is the vector with i-th entry $\sigma(w_k^\top x_i)$

Using the Newton-Raphson update formula:

$$w_{k+1} = w_k - (X^\top W_k X)^{-1}(-X^\top (y - p_k))$$

$$= w_k + (X^\top W_k X)^{-1} X^\top (y - p_k)$$

Defining:

$$z_k = X w_k + W_k^{-1}(y - p_k)$$

we obtain the update equation:

$$w_{k+1} = (X^\top W_k X)^{-1} X^\top W_k z_k$$

which proves the iterative weighted least squares update rule.

Question 5

5a).

The hinge loss function for a support vector machine (SVM) is given by:

$$\ell((x, y), w) = \max[0, 1 - yw^T x]$$

Correctly Classified Example

A sample is correctly classified if $y = \text{sgn}(w^T x)$. The hinge loss behaves as follows:

- If $yw^T x \geq 1$, then $1 - yw^T x \leq 0$, and thus the hinge loss is 0.
- If $0 < yw^T x < 1$, then $1 - yw^T x > 0$, and the hinge loss is positive.

Hence, the hinge loss for correct classifications is in the range $[0, 1)$.

Incorrectly Classified Example

A sample is incorrectly classified if $y \neq \text{sgn}(w^T x)$, meaning $yw^T x < 0$. The hinge loss behaves as follows:

- If $yw^T x < 0$, then $1 - yw^T x > 1$, leading to a large hinge loss.
- The hinge loss for incorrect classifications is in the range $[1, +\infty)$.

5b). Bounding the Number of Mistakes

We want to show the bound:

$$\frac{1}{N} M(w) \leq \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i w^T x_i)$$

Since $M(w)$ represents the number of misclassified samples, i.e., instances where $y_i \neq \text{sgn}(w^T x_i)$, which occurs when $y_i w^T x_i < 0$.

If $y_i w^T x_i \geq 1$, the hinge loss is 0, meaning the sample is correctly classified. If $y_i w^T x_i < 1$, then the hinge loss is: $1 - y_i w^T x_i \geq 1$, which is at least 1 for

misclassified samples.

Summing across the dataset, we obtain:

$$M(w) \leq \sum_{i=1}^{N} \max(0, 1 - y_i w^T x_i).$$

Dividing by N gives the required bound.

5c). Effect of Increasing λ

If the dataset is separable but some points are still misclassified, decreasing λ allows the model to prioritize minimizing classification errors over maximizing margin while increasing λ forces a larger margin at the cost of potential classification errors.

Question 6

(a) Why might the hard-SVM formulation fail in real-world datasets? The hard-SVM formulation assumes perfect linear separability, which often fails in real-world datasets due to noise, mislabeled points, and overlapping classes. If strict separation is impossible, the optimization problem becomes infeasible.

(b) How do slack variables ξ_i resolve these issues?

Slack variables ξ_i introduce flexibility by allowing some data points to be within the margin or even misclassified. The constraint is modified to: $y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$. and this ensures that the optimization problem remains feasible even when perfect separation is not possible. The term $C \sum \xi_i$ in the objective function penalizes large violations, balancing margin maximization with classification errors and by adjusting C, we control how much misclassification is tolerated, allowing the SVM to generalize better to real-world datasets.

(c) Number of variables in primal and dual SVM formulations

In the primal formulation, we optimize $d + 1 + n$ variables: $w \in \mathbb{R}^d$, $b \in \mathbb{R}$, and ξ_i for each of the n data points. In the dual formulation, we optimize n Lagrange multipliers α_i, leading to fewer variables. When using different kernels such as the radial basis function (RBF) or polynomial kernel, the number of dual variables remains n, but the kernel replaces the dot product $x_i^T x_j$, enabling non-linear decision boundaries without explicitly increasing dimensionality.

(d) Why prefer the dual formulation over the primal formulation? The dual formulation is often preferred because it enables the use of kernel methods, allowing SVMs to classify non-linearly separable data efficiently. It depends only on dot products between data points, making it computationally advantageous when the number of samples is smaller than the number of features. Additionally, solving the dual problem can be more efficient in high-dimensional spaces where direct optimization of the primal formulation is difficult.

Question 8

8.1a). Basis Function $\phi(x)$

The given regression model is:

$$y_i = w_0 + w_1 \sin(x_i) + w_2 \cos(x_i) + w_3 \sin(2x_i) + w_4 \cos(2x_i) + \cdots + w_{2k-1} \sin(kx_i) + w_{2k} \cos(kx_i)$$

From this, we define the basis function transformation:

$$\phi(x) = \Big[1, \sin(x), \cos(x), \sin(2x), \cos(2x), \ldots, \sin(kx), \cos(kx)\Big]^T$$

which maps input x into a $(2k+1)$-dimensional feature space.

8.1b). Residual Sum of Squares Error

The residual sum of squares (RSS) error is given by:

$$E(w) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

where the predicted output $\hat{y}_i$ is:

$$\hat{y}_i = w^T \phi(x_i)$$

Substituting this in the error equation:

$$E(w) = \sum_{i=1}^{N} \left(y_i - w^T \phi(x_i)\right)^2$$

In matrix notation, let: - Y be the $N \times 1$ vector of target values, - Φ be the $N \times (2k+1)$ design matrix where each row is $\phi(x_i)$, - w be the $(2k+1) \times 1$ parameter vector.

Then, we rewrite RSS as:

$$E(w) = \|Y - \Phi w\|^2$$

8.1c). Optimal Parameter Values

To minimize $E(w)$, we differentiate it with respect to w:

$$\frac{\partial}{\partial w} E(w) = -2\Phi^T(Y - \Phi w)$$

Setting the derivative to zero:

$$\Phi^T \Phi w = \Phi^T Y$$

Solving for w:

$$w = (\Phi^T \Phi)^{-1} \Phi^T Y$$

This is the normal equation used to compute the optimal regression weights.

Question 7

```
# !pip install ucimlrepo
# !pip install cvxpy
# !pip install clarabel

from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
import cvxpy as cp
from matplotlib import pyplot as plt

# !! DO NOT MODIFY THIS CELL !!

# Download and preprocess the dataset.
# fetch dataset
heart_disease = fetch_ucirepo(id=45)
X = heart_disease.data.features
# Convert categorical features into one-hot encode
categorical_features = ['cp','thal','slope','restecg']
X = pd.get_dummies(X, columns=categorical_features)

y = heart_disease.data.targets
print(f"Number of samples in all full dataset is: {len(X)}.")

# Check if our train set has missing value
na_in_features = X.isna().any(axis=1).sum()
na_in_trainY = y.isna().sum()
print(f"Number of rows with missing values in features:
{na_in_features}")

# Drop the rows with missing values.
indices_with_nan = X.index[X.isna().any(axis=1)]
X = X.drop(indices_with_nan)
y = y.drop(indices_with_nan)

# Divide train/test
np.random.seed(6464)
msk = np.random.rand(len(X)) < 0.75
X_train = X[msk]
X_test = X[~msk]
y_train = y[msk]
y_test = y[~msk]

# Convert problem to binary problem
X_train = np.array(X_train,dtype='float')
X_test = np.array(X_test,dtype='float')
y_train = np.array([-1 if i==0 else 1 for i in
y_train.values],dtype='float')
y_test = np.array([-1 if i==0 else 1 for i in
```

```
y_test.values],dtype='float')

print(f"Shapes: X_train: {X_train.shape}, y_train: {y_train.shape},
X_test: {X_test.shape}, y_test: {y_test.shape}")
```

```
Number of samples in all full dataset is: 303.
Number of rows with missing values in features: 4
Shapes: X_train: (216, 22), y_train: (216,), X_test: (83, 22), y_test:
(83,)
```

Question 7.1 - Normalization of X_train and X_test

7.1.1 - Implementing data normalization

```
# Normalize X_train and X_test using the statistics of X_train.
# 1. Compute the mean and standard deviation for each feature in
X_train
# 2. Subtract the mean from each feature and divide by the standard
deviation
#    for both X_train and X_test.

#YOUR CODE HERE!
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)

X_train_normalized = (X_train - mean) / std
X_test_normalized = (X_test - mean) / std

s1 = X_train_normalized.shape == X_train.shape
s2 = X_test_normalized.shape == X_test.shape

print("X_train shape assertion:", s1)
print("X_test shape assertion:", s2)
```

```
X_train shape assertion: True
X_test shape assertion: True
```

7.1.2 - Why normalize with mean and standard deviation?

The mean and standard deviation are computed from the training data because the test set is meant to simulate unseen data and should not influence the preprocessing step. Using statistics from the training data ensures that the model generalizes well, preventing data leakage and making the test data truly representative of real-world scenarios.

```
# Print the mean and standard deviation of the first and last feature.

# YOUR CODE HERE!
print(f"Mean of first feature: {mean[0]}")
print(f"Standard deviation of first feature: {std[0]}")
```

```
print(f"Mean of last feature: {mean[-1]}")
print(f"Standard deviation of  last feature: {std[-1]}")
```

```
Mean of first feature: 54.99074074074074
Standard deviation of first feature: 9.077847854724455
Mean of last feature: 0.5046296296296297
Standard deviation of  last feature: 0.49997856607007907
```

Q 7.1.3

The mean and standard deviation of the first feature are 54.99 and 9.08, while for the last feature, they are 0.5046 and 0.49998, respectively. These values are computed from the training data to ensure consistent normalization for both training and test datasets, preventing data leakage.

```
# Train SVM

# Complete the `trainSVM` function to find the optimal w and b that
minimize
# the primal SVM objective given in the write-up.
# The function takes three inputs:
# - trainX: the normalized train features with shape (#train_samples,
#features)
# - trainY: train labels with shape (#train_samples,)
# - C: C parameter of the minimization problem
# The function should return a three-tuple with:
# - w: the weight vector with shape (#features,)
# - b: the bias. A scalar with shape (1,)
# - xi: the slack variables with shape (#train_samples,)

# You can use cvxpy that we imported as cp
# You may find cp.Variable, cp.Minimize, cp.Problem useful
# For the problem solver, prefer the default, cp.CLARABEL

def trainSVM(trainX, trainY, C):

    # YOUR CODE HERE!
    n, m = trainX.shape
    w = cp.Variable(m)
    b = cp.Variable()
    xi = cp.Variable(n)

    # loss = cp.sum(cp.pos(1 - cp.multiply(trainY, trainX @ w + b)) +
C * xi)
    # objective = cp.Minimize(loss)
    objective = cp.Minimize(0.5 * cp.sum_squares(w) + C * cp.sum(xi))
    # constraints = [xi >= 0]
    constraints = [cp.multiply(trainY, trainX @ w + b) >= 1 - xi, xi
>= 0]
```

```
    prob = cp.Problem(objective, constraints)
    prob.solve(solver = cp.CLARABEL)

    return w.value, b.value, xi.value
```

7.2 SVM Training

7.2.1 - Implemeting trainSVM

```
# Solve SVM with C = 1 and print the first three weights, b and the
first
# three slack variables as instructed in the write-up

# YOUR CODE HERE!
w, b, xi = trainSVM(X_train_normalized, y_train, 1)
print(f"First three weights: {w[:3]}")
print(f"Bias: {b}")
print(f"First three slack variables: {xi[:3]}")

w_1 = w
b_1 = b
xi_1 = xi
```

```
First three weights: [-0.01280085  0.51706872  0.27813637]
Bias: 0.08109278680708337
First three slack variables: [-1.08487196e-10 -1.07320090e-10 -
1.09846811e-10]
```

```
# Solve SVM with C = 0 and print the first three weights, b and the
first
# three slack variables as instructed in the write-up

# YOUR CODE HERE!
w, b, xi = trainSVM(X_train_normalized, y_train, 0)
print(f"First three weights: {w[:3]}")
print(f"Bias: {b}")
print(f"First three slack variables: {xi[:3]}")
```

```
First three weights: [3.90035347e-11 2.92283697e-11 1.90604453e-11]
Bias: -13.492472743509481
First three slack variables: [504.1019176  497.5850476  495.44091299]
```

7.2.2

For C = 1, the first three weights are [-0.0128, 0.5171, 0.2781], the bias is 0.0811, and the first three slack variables are nearly zero, indicating a well-regularized model. For C = 0, the weights are extremely small, the bias is -13.4925, and the high slack variables (over 495) show that many points violate the margin constraints, leading to poor classification.

7.2.3 - Difference between the slack variables with C = 0 and C = 1?

Yes, there is a significant difference. When $C=1$, the slack variables are nearly zero, meaning most points are correctly classified with minimal margin violations, while for $C=0$, the slack variables are large, indicating that the model is allowing many misclassifications.

This explains why we introduce the $C\sum\xi_i$ term in Soft-SVM, it penalizes misclassification, ensuring a balance between maximizing the margin and minimizing classification errors.

7.3 SVM Evaluation

7.3.1 - evalSVM implementation

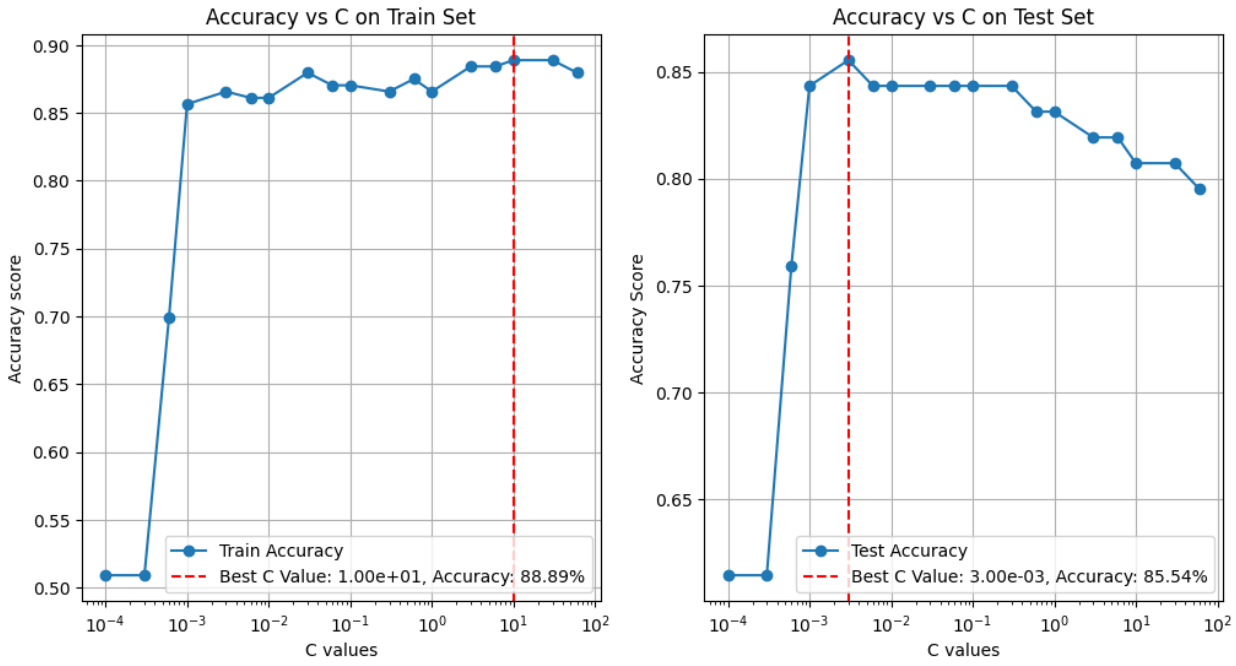
```
# Eval SVM

# Write a function to evaluate the SVM model given its `w` and `b`
parameters
# on evaluation data `X_eval` and true labels `y_eval`.
#  1. Estimate the labels of `X_eval`.
#  2. Return the ratio of accurately estimated labels by comparing
with `y_eval`.


def evalSVM(X_eval, y_eval, w, b):

    # YOUR CODE HERE!
    y_pred = np.sign(X_eval @ w + b)
    accuracy = np.mean(y_pred == y_eval)
    return accuracy


# evalSVM(X_test_normalized, y_test, w_1, b_1)

train_accuracies = []
test_accuracies = []
c_values = [a * 10 ** q for a in [1,3,6] for q in [-4, -3, -2, -1, 0,
1]]
c_values = sorted(c_values)

print(c_values)

# c_values
# For each C value given in the homework, find optimal w, b
# values using the normalized train set. calculate the accuracy
# on train and test sets using found w and b.
# Save those values as we will plot them

# YOUR CODE HERE!
for c in c_values:
    w, b, _ = trainSVM(X_train_normalized, y_train, c)
    train_accuracies.append(evalSVM(X_train_normalized, y_train, w,
```

```
b))
    test_accuracies.append(evalSVM(X_test_normalized, y_test, w, b))

print(test_accuracies)

check_val_train = zip(c_values, train_accuracies)
for val in check_val_train:
    if val[1] == max(train_accuracies):
        print(f"Best C value for train set: {val[0]}, accuracy:
{val[1]}")
        break

check_val_test = zip(c_values, test_accuracies)
for val in check_val_test:
    if val[1] == max(test_accuracies):
        print(f"Best C value for test set: {val[0]}, accuracy:
{val[1]}")
        break
```

```
[0.0001, 0.00030000000000000003, 0.0006000000000000001, 0.001, 0.003,
0.006, 0.01, 0.03, 0.06, 0.1, 0.30000000000000004, 0.6000000000000001,
1, 3, 6, 10, 30, 60]
[0.6144578313253012, 0.6144578313253012, 0.7590361445783133,
0.8433734939759037, 0.8554216867469879, 0.8433734939759037,
0.8433734939759037, 0.8433734939759037, 0.8433734939759037,
0.8433734939759037, 0.8433734939759037, 0.8313253012048193,
0.8313253012048193, 0.8192771084337349, 0.8192771084337349,
0.8072289156626506, 0.8072289156626506, 0.7951807228915663]
Best C value for train set: 10, accuracy: 0.8888888888888888
Best C value for test set: 0.003, accuracy: 0.8554216867469879
```

```
# Plotting and reporting the desired values

# sort the values of c
c_values_sorted = sorted(c_values)
train_accuracies_sorted = [train_accuracies[i] for i in
np.argsort(c_values)]

fig, ax = plt.subplots(1, 2, figsize=(12, 6))
ax[0].plot(c_values_sorted, train_accuracies_sorted, label='Train
Accuracy', marker='o')
# ax[0].axvline(x= c_values_sorted[np.argmax(train_accuracies)],
color='r', linestyle='--', label='Best C Value:
{:.2e}'.format(c_values_sorted[np.argmax(train_accuracies)]))
ax[0].axvline(x=c_values_sorted[np.argmax(train_accuracies)],
color='r', linestyle='--',
              label='Best C Value: {:.2e}, Accuracy: {:.2f}%'.format(
                  c_values_sorted[np.argmax(train_accuracies)],
                  train_accuracies[np.argmax(train_accuracies)] *
100))  # Convert to percentage
```

```
# ax[0].plot(c_values, train_accuracies)
ax[0].set_xscale('log')
ax[0].set_xlabel('C values')
ax[0].set_ylabel('Accuracy score')
ax[0].legend()
ax[0].grid()
ax[0].set_title('Accuracy vs C on Train Set')

test_accuracies_sorted = [test_accuracies[i] for i in
np.argsort(c_values)]
ax[1].plot(c_values, test_accuracies, label='Test Accuracy',
marker='o')
# ax[1].axvline(x= c_values_sorted[np.argmax(test_accuracies)],
color='r', linestyle='--', label='Best C Value:
{:.2e}'.format(c_values_sorted[np.argmax(test_accuracies)]))
ax[1].axvline(x=c_values_sorted[np.argmax(test_accuracies)],
color='r', linestyle='--',
              label='Best C Value: {:.2e}, Accuracy: {:.2f}%'.format(
                  c_values_sorted[np.argmax(test_accuracies)],
                  test_accuracies[np.argmax(test_accuracies)] * 100))
# Convert to percentage
ax[1].set_xscale('log')
ax[1].set_xlabel('C values')
ax[1].set_ylabel('Accuracy Score')
ax[1].legend()
ax[1].grid()
ax[1].set_title('Accuracy vs C on Test Set')

plt.show()
```



SVM Evaluation

On the train data, the C value of 10 achieved the best accuracy of 88.9% while the best C value on the test data is 0.003 with an accuracy of 85.5%

Question 8.2

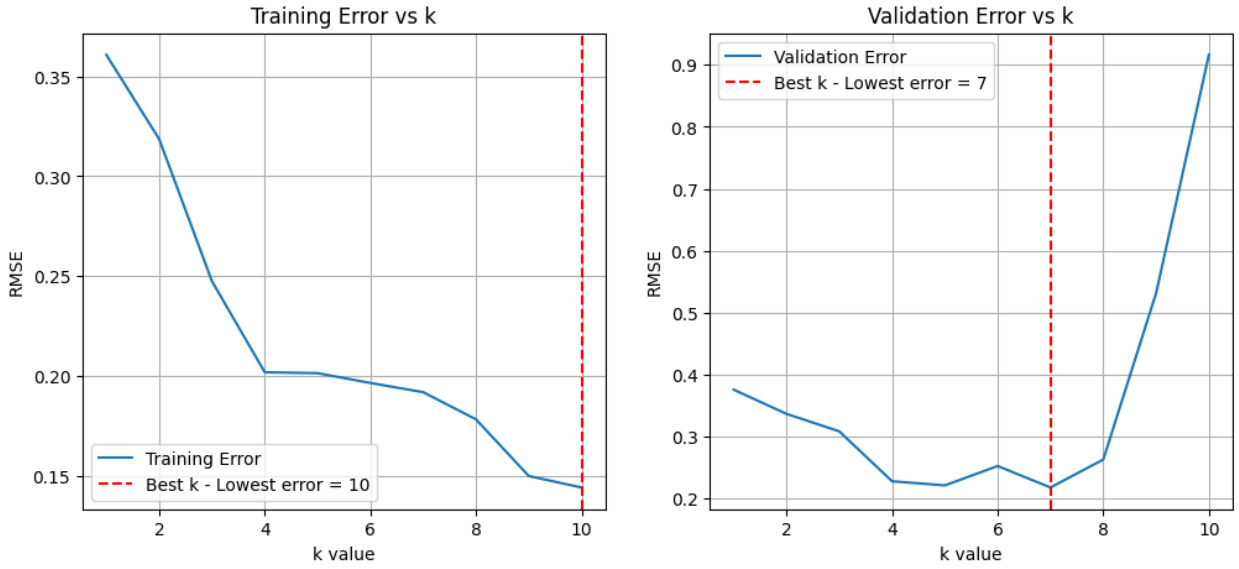
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import inv, pinv
from sklearn.model_selection import train_test_split

class SinusoidalRegressor:
    def __init__(self):
        self.k = None
        self.weights = None

    def phi(self, x):
        # The basis function for a general 2k
        x = np.array(x).reshape(-1, 1)
        features = [np.ones_like(x)]  # w0 (bias term)
        for i in range(1, self.k + 1):
            features.append(np.sin(i * x))
            features.append(np.cos(i * x))
        return np.hstack(features)


    def fit(self, X_train, Y_train, k):
        self.k = k
        # Construct the design matrix Phi for all data points in
X_train
        Phi = self.phi(X_train)
        # Solve for the weights using the normal equation with a
pseudo-inverse
        # Make sure the shapes align: Phi.T @ Phi should be a square
matrix and Phi.T @ Y_train should be a vector
        self.weights = pinv(Phi.T @ Phi) @ Phi.T @ Y_train

    def predict(self, X):
        # Check if the model is fitted
        if self.weights is None:
            raise ValueError("Model is not fitted yet.")
        # Apply the learned model
        return self.phi(X) @ self.weights

    def rmse(self, X_val, Y_val):
        # Predict the values for X_val
        Y_pred = self.predict(X_val)
        # Calculate the RMSE
        return np.sqrt(np.mean((Y_pred - Y_val) ** 2))

np.random.seed(61)
csv_file = 'nonlinear-regression-data.csv'
data = pd.read_csv(csv_file)
```

```
x = np.array(data['X'])
y = np.array(data['Noisy_y'])

print(x.shape, y.shape)
### Evaluation Part 0
##########################################################################
###########

# Split the data

X_train, X_val, Y_train, Y_val = train_test_split(x, y, train_size=45,
test_size=16, random_state=61)

### Evaluation Part 1 and 2
##########################################################################
###########

# Initialize the model
model = SinusoidalRegressor()
# Vary k from 1 to 10 and obtain RMSE error on the training set and
validation set
train_errors = []
val_errors = []
for k in range(1, 11):
    model.fit(X_train, Y_train, k)
    train_errors.append(model.rmse(X_train, Y_train))
    val_errors.append(model.rmse(X_val, Y_val))
    # print(f"Training RMSE for k={k}: {model.rmse(X_train,
Y_train)}")
    # print(f"Validation RMSE for k={k}: {model.rmse(X_val, Y_val)}")


(61,) (61,)
```

Question 8.3: 1 and 2

```
# Plotting the training error versus k
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(range(1, 11), train_errors, label='Training Error')
plt.axvline(x=np.argmin(train_errors) + 1, color='r', linestyle='--',
label='Best k - Lowest error = {}'.format(np.argmin(train_errors) +
1))
plt.xlabel('k value')
plt.ylabel('RMSE')
plt.title('Training Error vs k')
plt.legend()
plt.grid()
```

```
# Plotting the validation error versus k

plt.subplot(1, 2, 2)
plt.plot(range(1, 11), val_errors, label='Validation Error')
plt.axvline(x=np.argmin(val_errors) + 1, color='r', linestyle='--',
label='Best k - Lowest error = {}'.format(np.argmin(val_errors) + 1))
plt.xlabel('k value')
plt.ylabel('RMSE')
plt.title('Validation Error vs k')
plt.legend()
plt.grid()

plt.show()
```



Question 8.3: 3 - Optimal k comparism

The optimal k for training error is 10, while for validation error, it is 7, indicating that the model fits the training data better with higher k but does not generalize well. The difference arises because increasing k reduces training error but leads to overfitting, causing validation error to increase beyond $k = 7$.

```
### Evaluation Part 4
##########################################################################
###########
# You will reate separate plots for each k you can use plt.subplots
function
# Scatter plots with fitted lines for k = 1, 3, 5, 10
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
k_values_to_plot = [1, 3, 5, 10]

for ax, k in zip(axes.ravel(), k_values_to_plot):
```

```
    model.fit(X_train, Y_train, k)
    x_plot = np.linspace(min(X_val), max(X_val), 100)
    y_plot = model.predict(x_plot)
    ax.scatter(X_val, Y_val, color='red', label='Validation Data')
    ax.plot(x_plot, y_plot, color='blue', label=f'Fitted Curve
(k={k})')
    ax.set_title(f'k = {k}')
    ax.legend()
    ax.grid()

plt.tight_layout()
plt.show()
```