Question 1: Select ALL correct choices.

[**1.1**] Ans: **B**

[**1.2**] Ans: **A**, **B**

[**1.3**] Ans: **A**, **C**

[**1.4**] Ans: **A**, **B**, **D**

[**1.5**] Ans: **A**, **B**, **C**

Question 2: A linear ML model can be written as:

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T \mathbf{x}$$

The loss function can be written as:

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{n} \left[f(\mathbf{x}^{(i)}, \mathbf{w}) - y^{(i)} \right]^2$$

[**2.1**] Show analytically that the optimal weight vector that minimizes the cost function $J(\mathbf{w})$ is:

$$\mathbf{w}^* = \left(\mathbf{X}^T \mathbf{X} \right)^{-1} \mathbf{X}^T \mathbf{y}$$

Solution.

Since our model is linear, we can write the cost function as:

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{n} \left[\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)} \right]^2 \tag{1}$$

The product $\mathbf{w}^T \mathbf{x}^{(i)} = \mathbf{X}\mathbf{w}, \forall i \in \{1, 2, \cdots, n\}$ since the LHS implies the matrix mulplitcation of the RHS, as $\mathbf{X}$ is the matrix of all input entries $\mathbf{x}^{(i)}$. So, Eq. 1 can be written as:

$$J(\mathbf{w}) = \frac{1}{m} \left[\mathbf{X}\mathbf{w} - \mathbf{y} \right]^2 \tag{2}$$

The inside of the brackets is just a vector, and the square of a vector is the norm of a vector, so we can reduce Eq. 2:

$$\begin{aligned}
J(\mathbf{w}) &= \frac{1}{m} \|\mathbf{X}\mathbf{w} - \mathbf{y}\| && (3) \\
&= \frac{1}{m} (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) && (4) \\
&= \frac{1}{m} (\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{w}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\mathbf{w} + \mathbf{y}^T \mathbf{y}) && (5)
\end{aligned}$$

Now to optimize the cost with respect to weights, we can take the gradient of $J(\mathbf{w})$ w.r.t. the weights $\mathbf{w}^{(i)}$.

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m}\nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\mathbf{w} + \mathbf{y}^T\mathbf{y}) \tag{6}$$

$$= \frac{1}{m}\nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\mathbf{w}) \tag{7}$$

$$= \frac{1}{m}[\nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}) - \nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{y}) - \nabla_{\mathbf{w}}(\mathbf{y}^T\mathbf{X}\mathbf{w}))] \tag{8}$$

The gradients $\nabla_{\mathbf{w}}$ are simply derivatives of each matrix function w.r.t. $\mathbf{w}$, which can be computed using equations (69) and (81) from *The Matrix Cookbook* [2].

$$= \frac{1}{m}\left[\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}) - \frac{\partial}{\partial \mathbf{w}}(\mathbf{w}^T\mathbf{X}^T\mathbf{y}) - \frac{\partial}{\partial \mathbf{w}}(\mathbf{y}^T\mathbf{X}\mathbf{w}))\right] \tag{9}$$

$$= \frac{1}{m}((\mathbf{X}^T\mathbf{X} + (\mathbf{X}^T\mathbf{X})^T)\mathbf{w} - \mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}) \tag{10}$$

$$= \frac{1}{m}((\mathbf{X}^T\mathbf{X} + \mathbf{X}^T(\mathbf{X}^T)^T)\mathbf{w} - 2\mathbf{X}^T\mathbf{y}) \tag{11}$$

$$= \frac{1}{m}((\mathbf{X}^T\mathbf{X} + \mathbf{X}^T\mathbf{X})\mathbf{w} - 2\mathbf{X}^T\mathbf{y}) \tag{12}$$

$$= \frac{1}{m}(2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{X}^T\mathbf{y}) \tag{13}$$

$$= \frac{2}{m}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}) \tag{14}$$

We find the minimum when $\nabla_{\mathbf{w}} J(\mathbf{w}) = 0$,

$$\frac{2}{m}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}) = 0 \tag{15}$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} = 0 \tag{16}$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y} \tag{17}$$

$$(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{X}\mathbf{w}) = (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y}) \tag{18}$$

$$\mathbf{I}\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{19}$$

$$\boxed{\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{w}^*} \tag{20}$$

Without advanced methods, multiplying two $(n \times n)$ matrices is of $O(n^3)$ complexity [1]. For very large data sets, computing $\mathbf{w}$ would be enormously computationally expensive.

[2.2] Develop a pseudo-codes for implementing the batch gradient descent, stochastic gradient descent, and mini-batch gradient descent algorithms to train the above linear model.

Solution.

Batch Gradient Descent

```
a = learning rate
N = size of dataset

# Psuedocode for Batch Gradient Descent
REPEAT UNTIL CONVERGENCE
{
for all j:
    temp_j = w_j - (a / N) * sum( i=1..N, gradient_w(J(x_i, w_j) )

w_j = temp_j
}
```

Stochastic Gradient Descent

```
a = learning rate

# Pseudocode for Stochastic Gradient Descent
REPEAT UNTIL CONVERGENCE
{
for all i:
    w_j = w_j - a * gradient_w(J(x_i, w_j))
}
```

Mini-Batch Gradient Descent

```
a = learning rate
b = batch size
m = rows in training set

# Psuedocode for Mini-Batch Gradient Descent
REPEAT UNTIL CONVERGENCE
{
# i indexes will be multiples of batch size
for i = (1*b + 1, 2*b + 1, ... , (m-1)*b + 1):
    w_j = w_j - (a / b) * sum(k=i..i+(b-1), gradient_w(J(x_k, w_j)))
}
```

[**2.3**] Discuss the performance versus computational complexity of each of the above algorithms.

Solution.

Batch-Gradient Descent (BDG) is the most compuationally complex since the algorithm has to sum over the entire dataset *every single update* of w_j. Stochastic Gradient Descent (SDG) is much faster than BDG since each update of w_j only looks over a single data row in the X matrix, instead of all of them. Mini-Batch Gradient Descent (MBDG) is inbetween BDG and SDG in terms of computational complexity, since MBDG updates w_j after summing the gradients of a specified batch size $b \in (1, \cdots, N)$.

A ranking of each algorithm from least compuationally complex to least is as follows:

(1) Stochastic
(2) Mini-Batch
(3) Batch

Question 3: Housing data preprocessing.

Solution.

```
# Chase Lotito - SIUC Fall 2024 - ECE469: Intro to Machine Learning
# HW1 - Question 3

import numpy as np
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder    # For encoding categorical features
from sklearn.impute import SimpleImputer            # For adding missing values
from sklearn.preprocessing import StandardScaler    # For standardizing data

# (A) Get housing data
RAW_DATA = 'https://github.com/ageron/data/raw/main/housing/housing.csv'
housing = pd.read_csv(RAW_DATA)

# (B) Choose input features and output features (saved into numpy.ndarray type)
X = housing[
        ['longitude',
        'latitude',
        'housing_median_age',
        'total_rooms',
        'total_bedrooms',
        'population',
        'households',
        'median_income',
        'ocean_proximity']
    ].values
Y = housing[['median_house_value']].values

# (C) Ocean Proximity is a categorical feature. Drop it or transform into numerical values (encode).

```

```
# Isolate the ocean_proximity data in input data X
ocean_proximity = X[:,8].reshape(-1,1)    # reshape(-1,1) to make 2D array for Ordinal
# Initalize the ordinal encoder
ordinal_encoder = OrdinalEncoder()
# Encode the ocean_proximity strings into numerical data
encoded_ocean = ordinal_encoder.fit_transform(ocean_proximity)
# Put the encoded version of ocean_proximity into input data X
X[:,8] = encoded_ocean.flatten()          # flatten to add 1D version of array back into X

# (D) Clean the dasta by either dropping or replacing missing values

# Initialized SimpleImputer, will use the median to add missing entries
simple_imputer = SimpleImputer(strategy='median')

# Change X np ndarray into a Pandas Dataframe to use SimpleImputer
dX = pd.DataFrame(X)
dY = pd.DataFrame(Y)

# Perform SimpleImputer transformation, for both inputs X and outputs Y
imputed_data = simple_imputer.fit_transform(dX)
X = imputed_data
imputed_data = simple_imputer.fit_transform(dY)
Y = imputed_data

# (E) Carry out feature scaling either via normalization or standardization.
std_scaler = StandardScaler()
scaled_data = std_scaler.fit_transform(X)
X = scaled_data
scaled_data = std_scaler.fit_transform(Y)
Y = scaled_data

# (F) Create a training dataset and testing dataset
def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

```

# first, recombine X and Y into augmented matrix
housing_data = np.hstack((X,Y))

# split into testing and training set (both outputted as
    pd.DataFrames)
housing_training, housing_testing =
    shuffle_and_split_data(pd.DataFrame(housing_data),
    0.2)
print('TRAINING:')
print(housing_training)
print('TESTING:')
print(housing_testing)
```

References

[1] Andy He and Evan Williams. Computational complexity of matrix multiplication. https://www.cs.cornell.edu/courses/cs6810/2023fa/Matrix.pdf, Fall 2023. Accessed: 2024-09-10.

[2] Kaare Brandt Petersen and Michael Syskind Pederson. The matrix cookbook. Distributed by University of Waterloo, November 2012.