6CS012 – Artificial Intelligence and Machine Learning.
Tutorial – 02
Softmax Regression for Multiclass Classification
and
ERM Framework.

Siman Giri {Module Leader – 6CS012}

1. Understanding ERM Framework.

1.1 What is ERM?

- **Empirical Risk Minimization (ERM)** is a fundamental framework in statistical learning theory and deep learning,
  - where a decision function (aka Models) is learned by minimizing the empirical risk, also known as the **cost function**.
- This approach seeks to **approximate the true risk**, which represents the expected loss over the entire data distribution,
  - by evaluating the model's performance on a **finite training dataset**.
- ERM serves as the foundation for many deep learning algorithms,
  - where complex models like **Neural Network** are optimized to generalize well to unseen data.
- Mathematically:

Empirical Risk Minimization (ERM)

Empirical Risk Minimization (ERM) is a fundamental framework in statistical learning theory where the optimal decision function f^* is learned by minimizing the empirical risk, an approximation of the expected risk over the training data.

$$\mathcal{R}(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(f(x), y)]$$

Since the data distribution $\mathcal{D}$ is unknown, the empirical risk is computed using a finite training dataset $S = \{(x_i, y_i)\}_{i=1}^{n}$:

$$\hat{\mathcal{R}}(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$$

where $\ell(f(x_i), y_i)$ is the loss function quantifying the error for each sample.

1.2 Components of an ERM Framework.

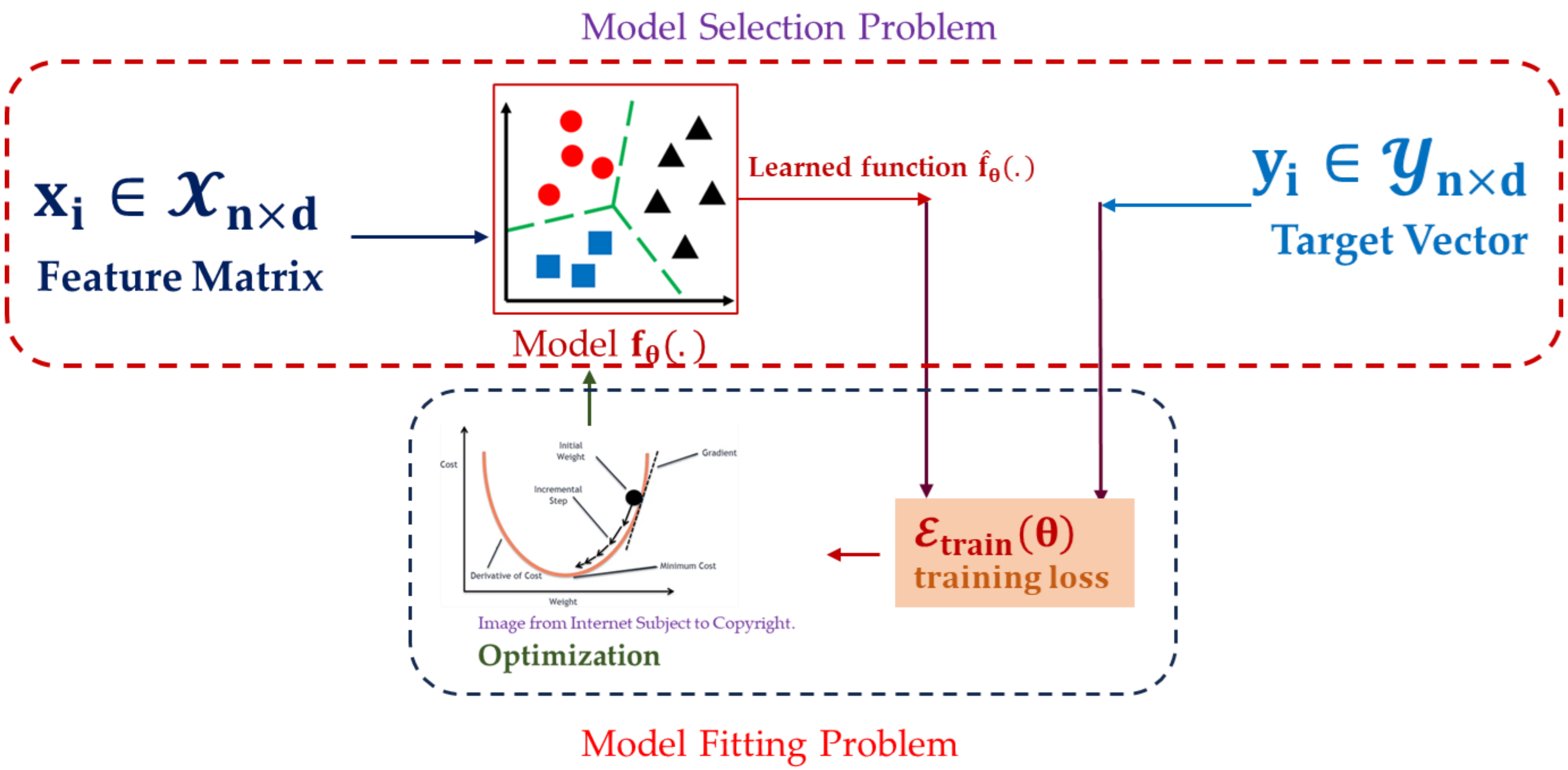


Fig: Framing Learning Problem within ERM framework.

1.3 Component – 1: A Decision Function.

1. Decision Function

A decision function (also known as a prediction function) takes an input $\mathbf{x} \in \mathcal{X}$ and produces an action $\mathbf{a} \in \mathcal{A}$, i.e.:

$$\mathbf{f} : \mathcal{X} \to \mathcal{A}$$

where:

$$\mathbf{x} \mapsto \mathbf{f}(\mathbf{x})$$

In the context of Supervised Machine Learning:

- The action $\mathbf{a} \in \mathcal{A}$ depends on the label $y \in \mathcal{Y}$.
- If $\mathcal{Y} \subset \mathbb{R}$, the action is to predict a continuous value $\to$ **Regression Task**.
- If $\mathcal{Y} \in \mathcal{C}$, where $\mathcal{C} = \{0, 1, \ldots, K\}$, the action is to assign a class to a label $\to$ **Classification Task**.

1.4 Component – 2: An Error Measure.

2. Loss Function

The loss function $\ell(\hat{f}(x), y)$ measures the discrepancy or difference between predicted $\hat{y}$ and actual values y.
Examples:

1. **For Regression:** Called Mean Square Error.

$$\ell(\hat{f}(x), y) = (\hat{f}(x) - y)^2$$

2. **For Classification:** Called Cross - Entropy Loss.

$$\ell(\hat{f}(x), y) = -y \log \hat{f}(x)$$

1.4.1 Component – 2: An Error Measure.

3. Cost Function / Empirical Risk

The cost function, or empirical risk $\hat{R}(f)$, is the average of the loss function over all training examples. It measures the overall discrepancy between the predicted values $\hat{f}(x_i)$ and the actual values y_i for all samples in the dataset.
For a dataset with n samples:

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n}\ell(\hat{f}(x_i), y_i)$$

Examples:

1. **For Regression:** The cost function is the Mean Squared Error (MSE), which is the average of squared differences between predicted and actual values.

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n}(\hat{f}(x_i) - y_i)^2$$

2. **For Classification:** The cost function is the Cross-Entropy Loss, which is the average of the negative log likelihood of the predicted class probabilities.

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n} -y_i \log \hat{f}(x_i)$$

1.5 Component – 3: Optimization Technique.

3. Optimization Technique

To find the optimal set of parameters w^*, b^* that yield the minimum value of the loss function $\ell(f(x_i), y_i)$, we solve:

$$(w^*, b^*) = \arg\min_{w,b} \mathcal{L}(x, y, w, b, f)$$

`Cautions!! The loss function itself isn't being minimized in an absolute sense; rather, we are finding the optimal parameters` w^*, b^* `that yield the minimum loss.`

How do we find such parameters?

The optimization process iteratively updates w and b using gradient-based methods:

$$w^{(t+1)} = w^{(t)} - \eta \nabla_w \hat{\mathcal{R}}(f_{w,b})$$

$$b^{(t+1)} = b^{(t)} - \eta \nabla_b \hat{\mathcal{R}}(f_{w,b})$$

where η is the learning rate and $\nabla_w \hat{\mathcal{R}}(f_{w,b})$, $\nabla_b \hat{\mathcal{R}}(f_{w,b})$ are the gradients of the empirical risk with respect to w and b.

Common Methods: Gradient Descent, Stochastic Gradient Descent (SGD), Adam.

ML in Practice: Applied ML

- A machine learning problem could be defined with:
  - Observed input $\mathbf{x} \in \mathcal{X}$ $\rightarrow$ **Input Space**.
  - Take action $\mathbf{a} \in \mathcal{A}$ $\rightarrow$ **Action Space{Regression or Classification}**
  - Observe Outcome $\mathbf{y} \in \mathcal{Y}$ $\rightarrow$ **Outcome Space**.
- Evaluate action in relation to the Outcome using **ERM framework** i.e.
  - Given a loss function
    - $\ell: \mathcal{A} \times \mathcal{Y} \rightarrow \mathbb{R}$,
  - Choose a hypothesis space $\mathcal{F}$ from $\mathcal{H} = \{\mathcal{F}_{(\text{model})} \in \mathcal{M}_{\text{class of Models}}\}$
  - Use an optimization method to find an empirical risk minimizer $\hat{f}_n \in \mathcal{F}$:
    - $\hat{f}_n = argmin_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$
    - (Or find a $\tilde{f}$ that comes close to $\hat{f}$)
- Your job as an ML practitioner:
  - Choose an appropriate Model $\mathcal{F}$ from class of model $\mathcal{M}_{class\ of\ Models}$ and fit model directly and hope model will perform appropriately – ***model fitting problem***;
  - Consider several model from model class and choose the best based on some performance metric – ***model selection problem***.

2. Softmax Regression within ERM Framework .

{For Multi – Class Classification Problem.}

2.1 Logistic Regression: Introduction.

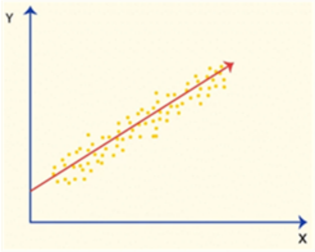


Image from Internet: Subject to copyright.

A Linear Model: $z = w_0 + \langle w^T x \rangle$

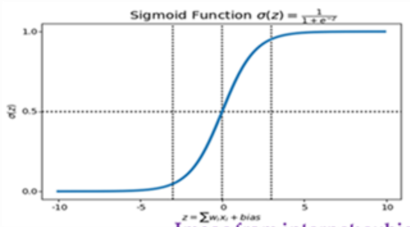


Image from internet: subject to copyright.

Fig: Sigmoid Function (σ).

A Sigmoid Function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Binary Classification

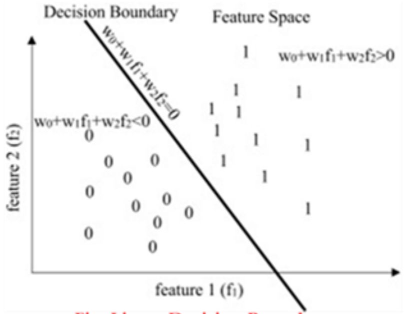


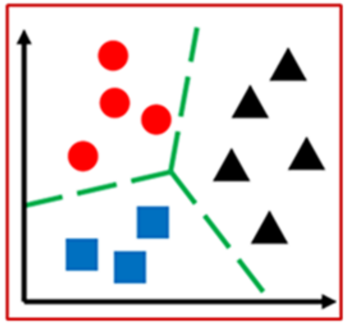
Fig: Linear Decision Boundary

Image by k. Worden et.al

Sigmoid Regression.

A Decision $\hat{y} = \begin{cases} 1 \text{ if } P(y=1|x) \geq 0.5 \\ 0, \text{if } P(y=1|x) < 0.5 \end{cases}$

Multiclass Classification



Softmax Regression.

A Decision $y_{pred} = \mathrm{argmax}_i P(Y = i|x, W, b)$

Fig: Logistic Regression for Binary Classification and Multiclass Classification Problem.

2.1 Logistic Regression: Introduction.

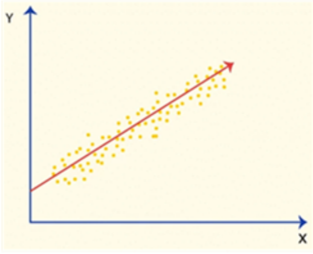


Image from Internet: Subject to copyright.

A Linear Model: $z = w_0 + \langle w^T x \rangle$

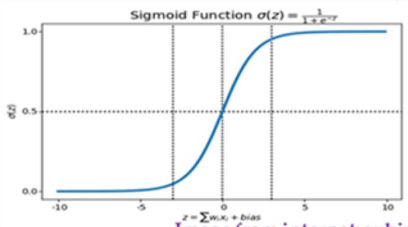


Image from internet: subject to copyright.

Fig: Sigmoid Function (σ).

A Sigmoid Function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Binary Classification

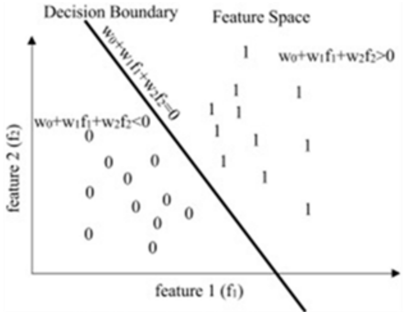


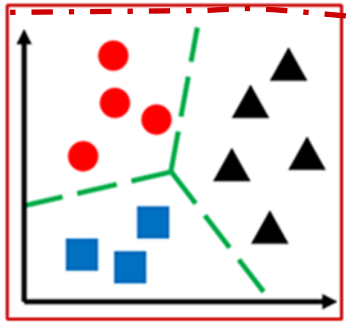
Fig: Linear Decision Boundary

Image by k. Worden et.al

Sigmoid Regression.

A Decision $\hat{y} = \begin{cases} 1 \text{ if } P(y = 1|x) \geq 0.5 \\ 0, \text{if } P(y = 1|x) < 0.5 \end{cases}$

Multiclass Classification



Softmax Regression.

A Decision $y_{pred} = \text{argmax}_i P(Y = i|x, W, b)$

Fig: Logistic Regression for Binary Classification and Multiclass Classification Problem.

Our Objective.

2.2 Understanding Data.

- For the Purpose of Demonstration, we will use **“iris.csv”** dataset.

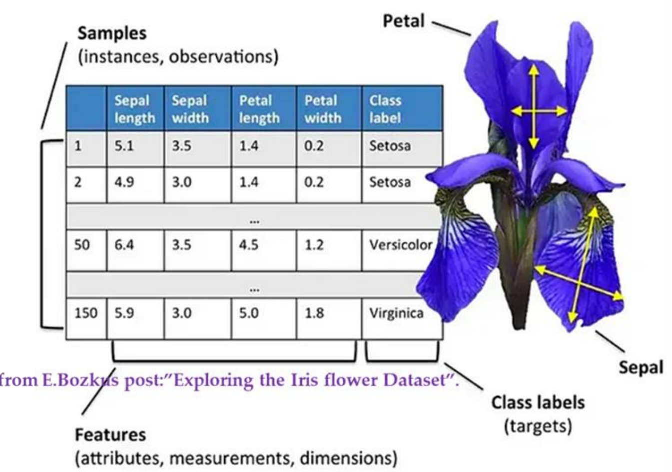


Image from E.Bozkus post:”Exploring the Iris flower Dataset”.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64
 5   target_name        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
None
```

Fig: A sample dataset: data – points $\mathbf{i = 150}$, dimension $\mathbf{d = 6}$, Number of Classes $\mathbf{= 3}$

About iris.csv dataset:
The Iris flower dataset is a classic dataset introduced by British statistician and biologist **Ronald Fisher** in his 1936 paper – “*The use of multiple measurements in taxonomic problems*.”

2.2.1 Data Pre – processing.

Loading Data:

```
# Necessary Imports
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
# Loading Dataframe:
df = pd.read_csv("Iris.csv") # changed to read_csv and the correct file name
# Step 2: Dataset Information
print("Dataset Preview:")
print(df.head()) # Show first 5 rows
print("\nDataset Information:")
print(df.info()) # Summary of dataset
```

2.2.1 Data Pre – processing.

Preparing Feature Matrix and Label Vector with Conversion to One Hot Encoding:

```
# Step 3: Extract features (X) and target labels (y)
X = df.iloc[:, 1:-1].values # All columns except the first and the last one (features) since the first
    column is an index
y = df.iloc[:, -1].values # Last column (target)
# Step 4: Convert categorical labels to numeric
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y) # Convert labels to integers (0,1,2)
# Step 5: One-Hot Encode the Labels
one_hot_encoder = OneHotEncoder(sparse_output=False) #changed sparse to sparse_output and set to False
y_one_hot = one_hot_encoder.fit_transform(y_encoded.reshape(-1, 1))
# Display results
print("\nUnique Classes:", np.unique(y))
print("Encoded Labels:", np.unique(y_encoded))
print("One-Hot Encoded Labels:\n", y_one_hot[:5]) # Show first 5
```

2.2.1 Data Pre – processing.

Train Test Split.

```
# Step 6: Split dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=42, stratify
    =y_one_hot)
# Output shapes
print("\nShapes:")
print("X_train:", X_train.shape, "y_train:", y_train.shape)
print("X_test:", X_test.shape, "y_test:", y_test.shape)
```

2.3 Softmax Regression as Learning Function.

- **Model**: Multiclass logistic regression or softmax regression uses **the softmax function** to compute class probabilities:
  - $\mathbf{f_{w,b} = \hat{y}_i = softmax(z_i);}$
  - where:
    - $\mathbf{z_i = w^T x_i + b,}$
  - and:
    - $\mathbf{softmax(z_i)_j = \frac{e^{z_{ij}}}{\sum_{k=1}^{C} e^{z_{ik}}}, \forall j \in \{1, \dots, C\}}.$
      - here, $\mathbf{C}$ is the number of classes.
- **Decision Function:**
  - The model's prediction $\mathbf{y_{pred}}$ is the class whose **probability is maximal** i.e.:
  - $\mathbf{y_{pred} = argmax_i P(Y = i | x, W, b)}$

2.3.1 Implementations.

Implementing Softmax Function:

```python
import numpy as np
def softmax(z):
    """
    Compute the softmax probabilities for a given input matrix.
    Parameters:
    z (numpy.ndarray): Logits (raw scores) of shape (m, n), where
                       - m is the number of samples.
                       - n is the number of classes.
    Returns:
    numpy.ndarray: Softmax probability matrix of shape (m, n), where
                   each row sums to 1 and represents the probability
                   distribution over classes.
    Notes:
    - The input to softmax is typically computed as: z = XW + b.
    - Uses numerical stabilization by subtracting the max value per row.
    """
    # Prevent numerical instability by normalizing input
    z_shifted = z - np.max(z, axis=1, keepdims=True)
    exp_z = np.exp(z_shifted)
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

2.4 Softmax Regression in ERM Framework: Loss and Cost Function.

- **Loss function – Categorical Cross Entropy:**
  - **Categorical Cross-Entropy Loss** measures how well the predicted probability distribution matches the true class labels in a classification task.
  - It is widely used in multiclass classification problems.
  - The goal of the loss is to maximize the probability of the correct class.
- **Formula**: For a single sample $\mathbf{x_i}$ with true label $\mathbf{y_i}$ (one hot encoded) and predicted probabilities $\mathbf{\hat{y}_i}$, the loss is:
  - $\ell(\mathbf{y_i}, \mathbf{\hat{y}_i}) = -\sum_{k=1}^{C} \mathbf{y_{ik}} \log(\mathbf{\hat{y}_{ik}})$
  - Where:
    - **C → is the number of classes.**
    - $\mathbf{y_{ik}} = \mathbf{1}$: → if the **k-th class** is the true class for sample i, otherwise $\mathbf{y_{ik}} = \mathbf{0}$ {one hot encoding}.
    - $\hat{y}_{ik}$ → is the predicted probability **for class k**.
  - For a dataset of n samples, the average loss (**empirical risk**) given by:
    - $\hat{\mathbf{R}} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{C} \mathbf{y_{ik}} \log(\mathbf{\hat{y}_{ik}})$ {**also called cost function**}

2.4.1 Implementations – Loss Function.

Implementing Categorical Cross Entropy Loss Function:

```
def loss_softmax(y_pred, y):
    """
    Compute the cross-entropy loss.
    Parameters:
    y_pred (numpy.ndarray): Predicted probabilities of shape (n, c), where n is the number of samples and
        c is the number of classes.
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    Returns:
    float: Cross-entropy loss.
    """
    epsilon = 1e-12 # To avoid log(0)
    y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon) # Prevent log(0) by clipping values
    n = y.shape[0] # Number of samples
    loss = -np.sum(y * np.log(y_pred)) / n
    return loss
```

2.4.2 Implementations – Cost Function.

Implementing Cost Function:

```
def cost_softmax(X, y, W, b):
    """
    Compute the softmax regression cost (cross-entropy loss).
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
        number of features.
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c), where c is the number of classes.
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    Returns:
    float: The softmax cost (cross-entropy loss).
    """
    n = X.shape[0] # Number of samples
    z = np.dot(X, W) + b
    y_pred = softmax(z)
    cost = loss_softmax(y_pred, y)
    return cost
```

2.5 Softmax Regression in ERM Framework: Model Fitting.

- **Softmax Regression Model Fitting Problem:**
  - **ERM Objective {Explicitly for Softmax Regression}:**
    - The objective is to minimize the average loss (empirical risk) over the training dataset:
      - $\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{C} y_{ik} \log(\hat{y}_{ik})$;
    - Substituting $\hat{y}_{ik} = \frac{\exp(z_i)}{\sum_{k=1}^{C}\exp(z_{ik})}$, the loss can be written as:
      - $\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{\exp(z_i)}{\sum_{k=1}^{C}\exp(z_{ik})}\right)$;
    - here $\mathbf{z_i} = \mathbf{W}^T\mathbf{x_i} + \mathbf{b}$; $\mathbf{W} \in \mathbb{R}^{d\times C}$ → is the weight matrix and $\mathbf{b} \in \mathbb{R}^{C}$ → is the bias vector.
  - **Formulating as an Optimization problem:**
    - For any parameter(s) → $\boldsymbol{\theta}^* = [\mathbf{w}, \mathbf{b}: \mathbf{w} \in \mathbb{R}^d, \mathbf{b} \in \mathbb{R}] \in \boldsymbol{\Theta}$:
      - $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}^*} \mathcal{L}(\mathbf{w}, \mathbf{b})$
    - This means **finding** the *weight vector* $\mathbf{w} \in \mathbb{R}^{d\times C}$ and *bias term* $\mathbf{b} \in \mathbb{R}^{C}$ that *minimize the average log loss* over the ***training data***.

Towards Gradient Descent!!

2.6 Softmax Regression in ERM Framework: Gradient Descent.

Algorithm 1 Gradient Descent for Softmax Regression

Input: Dataset $\mathcal{D}_n = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i = 1, 2, \ldots, n\}$, learning rate $\alpha > 0$, number of iterations T

Initialize: Parameters $\mathbf{W} \in \mathbb{R}^{d \times C}$ and $\mathbf{b} \in \mathbb{R}^{C}$ (e.g., set to 0 or small random values)

for t = 1 to T **do**

Compute logits for all samples:

$$\mathbf{z}_i = \mathbf{W}^\top \mathbf{x}_i + \mathbf{b} \quad \forall i \in \{1, 2, \ldots, n\}$$

Compute predictions using the softmax function:

$$\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{c=1}^{C} \exp(z_{ic})} \quad \forall i \in \{1, 2, \ldots, n\}$$

Compute gradients:

$$\frac{\partial \texttt{Categorical-Cross-Entropy Loss}}{\partial \mathbf{W}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{\mathbf{y}}_i - \mathbf{y}_i) \mathbf{x}_i^\top$$

$$\frac{\partial \texttt{Categorical-Cross-Entropy Loss}}{\partial \mathbf{b}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{\mathbf{y}}_i - \mathbf{y}_i)$$

Update parameters:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{\partial \texttt{Cross-Entropy Loss}}{\partial \mathbf{W}}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \frac{\partial \texttt{Cross-Entropy Loss}}{\partial \mathbf{b}}$$

end for

Output: Optimal parameters $\mathbf{W}^*$ and $\mathbf{b}^*$

2.6.1 Computing Gradient.

- For gradient descent we first compute a gradient of the loss function with respect to **w and b** which is given by:

Gradient against W

The gradients of the Categorical Cross-Entropy Loss are:

$$\frac{\partial \text{Categorical-Cross-Entropy Loss}}{\partial \mathbf{W}} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{\mathbf{y}}_i - \mathbf{y}_i\right)\mathbf{x}_i^\top,$$

where $\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{z}_i)$ and $\mathbf{z}_i = \mathbf{W}^\top\mathbf{x}_i + \mathbf{b}$.

Gradient against b

The gradients of the Categorical Cross-Entropy Loss are:

$$\frac{\partial \text{Categorical-Cross-Entropy Loss}}{\partial \mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{\mathbf{y}}_i - \mathbf{y}_i\right),$$

where $\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{z}_i)$ and $\mathbf{z}_i = \mathbf{W}^\top\mathbf{x}_i + \mathbf{b}$.

2.6.1.1 Implementations – Computing Gradients.

Computing the Gradients:

```python
def compute_gradient_softmax(X, y, W, b):
    """
    Compute the gradients of the cost function with respect to weights and biases.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d).
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    Returns:
    tuple: Gradients with respect to weights (d, c) and biases (c,).
    """
    n, d = X.shape
    z = np.dot(X, W) + b
    y_pred = softmax(z)
    grad_W = np.dot(X.T, (y_pred - y)) / n # Gradient with respect to weights
    grad_b = np.sum(y_pred - y, axis=0) / n # Gradient with respect to biases
    return grad_W, grad_b
```

2.6.2 Implementing Gradient Descent.

A Gradient Descent Algorithm:

```python
def gradient_descent_softmax(X, y, W, b, alpha, n_iter, show_cost=False):
"""
Perform gradient descent to optimize the weights and biases.
Parameters:
X (numpy.ndarray): Feature matrix of shape (n, d).
y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
W (numpy.ndarray): Weight matrix of shape (d, c).
b (numpy.ndarray): Bias vector of shape (c,).
alpha (float): Learning rate.
n_iter (int): Number of iterations.
show_cost (bool): Whether to display the cost at intervals.
Returns:
tuple: Optimized weights, biases, and cost history.
"""
cost_history = []
for i in range(n_iter):
    # Compute gradients
    grad_W, grad_b = compute_gradient_softmax(X, y, W, b)

    # Update weights and biases using the gradients
    W -= alpha * grad_W
    b -= alpha * grad_b

    # Compute and store cost
    cost = cost_softmax(X, y, W, b)
    cost_history.append(cost)

    # Print cost at regular intervals
    if show_cost and (i % 100 == 0 or i == n_iter - 1):
        print(f"Iteration {i}: Cost = {cost:.6f}")
return W, b, cost_history
```

2.7 Implementing a Decision Function.

Prediction Function to Assign a Class:

```python
def predict_softmax(X, W, b):
    """
    Predict the class labels for a set of samples using the trained softmax model.
    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
        number of features.
    W (numpy.ndarray): Weight matrix of shape (d, c), where c is the number of classes.
    b (numpy.ndarray): Bias vector of shape (c,).

    Returns:
    numpy.ndarray: Predicted class labels of shape (n,), where each value is the index of the predicted
        class.
    """
    z = np.dot(X, W) + b # Compute the scores (logits)
    y_pred = softmax(z) # Get the probabilities using the softmax function

    # Assign the class with the highest probability
    predicted_classes = np.argmax(y_pred, axis=1)
    return predicted_classes
```

2.8 Putting it all Together!!!

- **Training the Model.**

Training the Model:

```
# Initialize the weights and biases
d = X_train.shape[1] # Number of features
c = y_train.shape[1] # Number of classes
W = np.random.randn(d, c) * 0.01 # Small random weights
b = np.zeros(c) # Bias initialized to 0
# Set hyperparameters
alpha = 0.1 # Learning rate
n_iter = 1000 # Number of iterations
# Train the model using gradient descent
W_opt, b_opt, cost_history = gradient_descent_softmax(X_train, y_train, W, b, alpha, n_iter, show_cost=
    True)
# Plot the cost history to visualize the convergence
plt.plot(cost_history)
plt.title('Cost Function vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.grid(True)
plt.show()
```

2.7 Evaluation: Confusion Matrix.

- Error in classification problem can be broadly of two kind i.e.
  - True:
    - label $\{y\}$ is 0 predicted $\{\hat{y}\}$ is 0.
    - label $\{y\}$ is 1 predicted $\{\hat{y}\}$ is 1.
  - False:;
    - label $\{y\}$ is 0 predicted $\{\hat{y}\}$ is 1.
    - label $\{y\}$ is 1 predicted $\{\hat{y}\}$ is 0.
- We can extend this idea to build confusion Matrix for multi class problem.

predicted \ actual	0	1	2
0	True {T}	False {F}	False {F}
1	False {F}	True {T}	False {F}
2	False {F}	False {F}	True {T}

Confusion matrix with 3 class

2.7.1 Extending to: Precision and Recall

- In our example of email classification, we only have two class **spam and not a spam**.
- Now let's imagine there are three different kind of email tags namely:
    - **urgent, normal and spam**
- We built a Multinomial Logistic Regression or Softmax Regression we can determine precision and recall as:

	urgent	normal	spam	
urgent	8	10	1	$\textbf{precision}_u = \frac{8}{8+10+1}$
normal	5	60	50	$\textbf{precision}_n = \frac{60}{5+60+50}$
spam	3	30	200	$\textbf{precision}_s = \frac{200}{3+30+200}$
	$\textbf{recall}_u = \frac{8}{8+5+3}$	$\textbf{recall}_n = \frac{60}{10+60+30}$	$\textbf{recall}_s = \frac{200}{1+50+200}$	

2.7.2 Implementations – Evaluation Function.

Testing and Evalustion:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
# Evaluation Function
def evaluate_classification(y_true, y_pred):
    """
    Evaluate classification performance using confusion matrix, precision, recall, and F1-score.

    Parameters:
    y_true (numpy.ndarray): True labels
    y_pred (numpy.ndarray): Predicted labels

    Returns:
    tuple: Confusion matrix, precision, recall, F1 score
    """
    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Compute precision, recall, and F1-score
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return cm, precision, recall, f1
```

2.8 Final Result.

Final Evaluation:

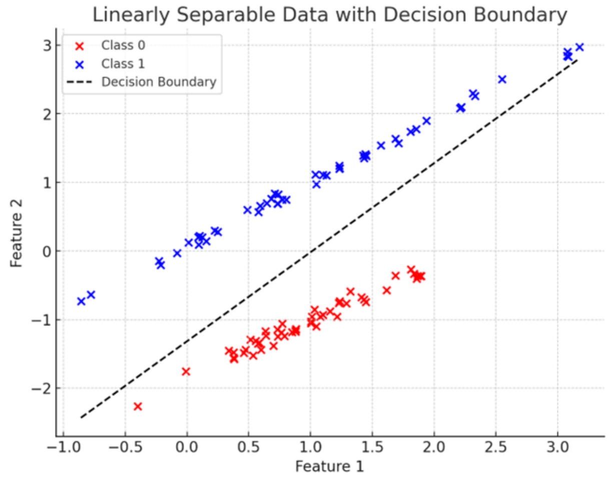
```python
# Predict on the test set
y_pred_test = predict_softmax(X_test, W_opt, b_opt)
# Evaluate accuracy
y_test_labels = np.argmax(y_test, axis=1) # True labels in numeric form
# Evaluate the model
cm, precision, recall, f1 = evaluate_classification(y_test_labels, y_pred_test)
# Print the evaluation metrics
print("\nConfusion Matrix:")
print(cm)
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

3. Limitations of Logistic Regression.

3.1 Linear Separability.

- “Logistic Regression are only suitable for Linearly Separable Data”.

- **Linear Separability** refers to the ability to **separate two classes of data** points in a feature space
  - using a single straight line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions).
- If there exists **a hyperplane** that can divide the data into two groups without any misclassification, then the dataset is said to be **linearly separable**.
  - Mathematically, a dataset with 2 classes $\mathbf{c_1}$ **and** $\mathbf{c_2}$ is linearly separable if there exists a hyperplane defined by:
    - $\mathbf{w \cdot x + b = 0}$
  - such that for every data point $\mathbf{x_i}$ **in class** $\mathbf{c_1}$, we have:
    - $\mathbf{w \cdot x_i + b > 0}$
  - and for every data point $\mathbf{x_i}$ **in class** $\mathbf{c_2}$, we have:
    - $\mathbf{w \cdot x_i + b < 0}$
  - where:
    - $\mathbf{x_i}$ is the feature vector
    - $\mathbf{w}$ is the weight vector
    - $\mathbf{b}$ is the bias term



The – End.