Homework 3

SHREYA CHETAN PAWASKAR
pawaskas@uci.edu
12041645

Question 1:

a) The likelihood function can be simplified:

$$p(x \mid \theta) = \prod_{n=1}^{N} p\left(x_n \mid \theta\right) = \prod_{n=1}^{N} \theta^{-1} \mathbb{I}_{0,\theta}\left(x_n\right) = \theta^{-N} \prod_{n=1}^{N} \mathbb{I}_{0,\theta}\left(x_n\right) = \theta^{-N} \mathbb{I}_{0,\theta}\left(\max\left\{x_1, \ldots, x_N\right\}\right)$$

- If $\max\left\{x_1, \ldots, x_N\right\} \leq \theta$, then $x_n \leq \theta$ for all $n = 1, \ldots, N$.
- If any single observation x_n falls outside the interval $[0, \theta]$, the entire dataset x is assigned likelihood zero. The constraint is $\hat{\theta} \geq \max\left\{x_1, \ldots x_N\right\}$ or else the likelihood will be zero.
- Given a $\hat{\theta}$ that the constraint is met, the data has likelihood $\hat{\theta}^{-N}$.
- This likelihood **decreases** as $\hat{\theta}$ **increases**.
- The likelihood function decreases as the constraint increases, so to maximize it, we need to minimize $\hat{\theta}$, resulting in the maximum value among the observations
  Answer:

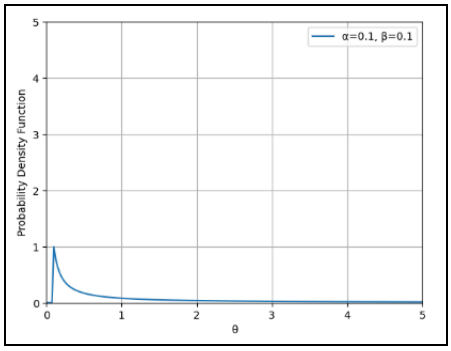
$$\hat{\theta} = \max\left\{x_1, \ldots x_N\right\}$$

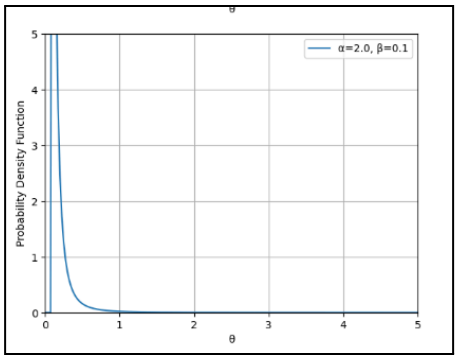
Code:

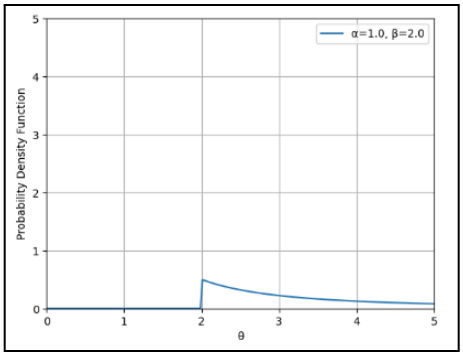
```
import numpy as np
import matplotlib.pyplot as plt

def pareto_pdf(theta, alpha, beta):
   if theta >= beta:
       return alpha * beta**alpha / theta**(alpha + 1)
   else:
       return 0

params = [(0.1, 0.1), (2.0, 0.1), (1.0, 2.0)]
theta_values = np.linspace(0, 5, 200)

for i, (alpha, beta) in enumerate(params, start=1):
```

```
pdf_values = [pareto_pdf(theta, alpha, beta) for theta in theta_values]
plt.plot(theta_values, pdf_values, label=f'α={alpha}, β={beta}')
plt.xlabel('θ')
plt.ylabel('Probability Density Function')
plt.legend()
plt.grid(True)
plt.axis([0, 5, 0, 5])
plt.show()
```

Output:







b)

- β determines the support of the distribution: $\theta < \beta$ has zero probability.
- β also determines the mode of the distribution.
- α influences the shape of the distribution.
- Larger α produces sharper peaks at the mode.
- Smaller α produces a more spread out and uniform distribution.

---

c) Let $\bar{x} \triangleq \max\{x_1, \ldots, x_N\}$.
Consider the simplified distribution

$$
\begin{aligned}
p(\theta \mid x) &\propto p(\theta)p(x \mid \theta) \\
&\propto \theta^{-\alpha-1}\mathbb{I}_{\beta,\infty}(\theta) \times \theta^{-N}\mathbb{I}_{0,\theta}(\bar{x}) \\
&\propto \theta^{-\alpha-N-1}\mathbb{I}_{\beta,\infty}(\theta)\mathbb{I}_{\bar{x},\infty}(\theta) \\
&\propto \theta^{-\alpha-N-1}\mathbb{I}_{\max\{\beta,\bar{x}\},\infty}(\theta) \propto \text{Pareto}(\alpha+N, \max\{\beta,\bar{x}\})
\end{aligned}
$$

- Here, for positive $\bar{x}$ and θ, the constraint that $0 \leq \bar{x} \leq \theta$ is equivalent to $\bar{x} \leq \theta < \infty$.
- We can see that the posterior distribution is again Pareto, but with the modified parameters.
- So, the Pareto prior is conjugate to the uniform likelihood.

---

d)

- The mode of a Pareto distribution is found at the lowest value within its range where the probability density is non-zero.

- So the mode occurs at the left boundary of its non-zero support, β

So,

$$\hat{\theta}_{MAP} = \max\{x_1, \ldots, x_N, \beta\} = \max\{\bar{x}, \beta\} = \max\left\{\hat{\theta}_{ML}, \beta\right\}$$

- In cases where at least one observation is larger than the prior lower-bound β, Maximum A Posteriori **(MAP) estimator and the Maximum Likelihood (ML) estimators are equal**.
- Else the **MAP estimator is** β**.**

---

e)

The MMSE estimator minimizing the quadratic loss is the posterior mean.

$$\mathbb{E}[\theta] = \int_0^\infty \theta\alpha\beta^\alpha\theta^{-\alpha-1}\mathbb{I}_{\beta,\infty}(\theta)d\theta = \int_\beta^\infty \alpha\beta^\alpha\theta^{-\alpha}d\theta$$

If $0 < \alpha \leq 1$, this integral diverges and the mean is undefined.
For $\alpha > 1$

$$\mathbb{E}[\theta] = \int_\beta^\infty \alpha\beta^\alpha\theta^{-\alpha}d\theta = \frac{\alpha\beta^\alpha}{1-\alpha}\left(0 - \beta^{1-\alpha}\right) = \frac{\alpha\beta}{\alpha-1}$$

To derive the actual form of the MMSE estimator, we evaluate this expression for the Pareto posterior.
Given at least one observation, $\alpha_{\text{post}} = \alpha_{\text{prior}} + N > 1$, and the MMSE estimate becomes

$$\mathbb{E}[\theta \mid x] = \frac{\alpha+N}{\alpha+N-1}\max\{x_1, \ldots, x_N, \beta\} = \frac{\alpha+N}{\alpha+N-1}\max\left\{\hat{\theta}_{ML}, \beta\right\}$$

---

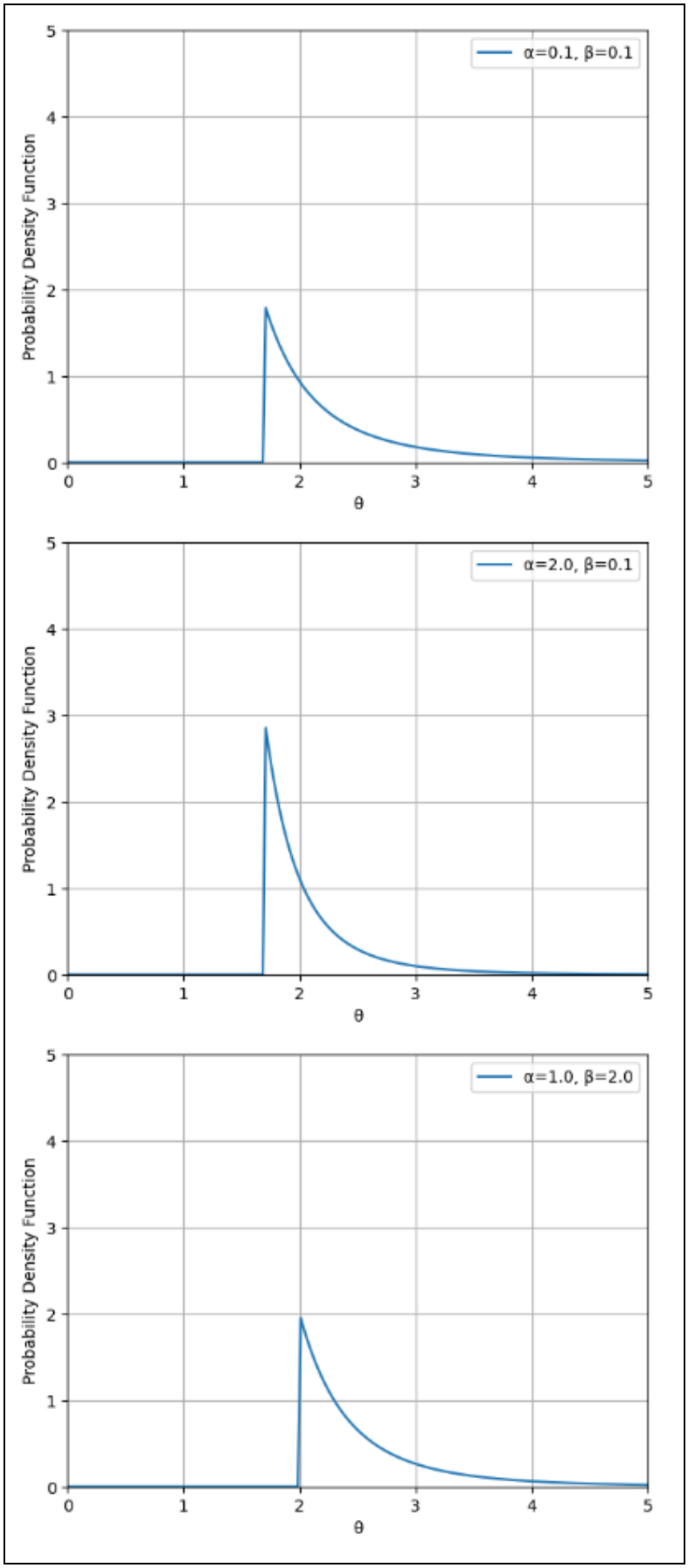
f)

- For the third prior, the posterior mean is higher than for the other two.
- $\hat{\theta}_{MAP} = 2.0$ and other two, $\hat{\theta}_{MAP} = \hat{\theta}_{ML} = 1.7$.
- The difference in the posterior mean values varies slightly depending on the chosen prior.
- For three hyperparameters, posterior mean equals 2.5095, 2.1250, and 2.6667 , respectively.

Code:

```
def pareto_pdf(theta, alpha, beta, x):
    beta = max(beta, max(x))
    alpha += len(x)
    if theta >= beta:
        return alpha * beta**alpha / theta**(alpha + 1)
    else:
        return 0

x = (0.7, 1.3, 1.7)
```

```
params = [(0.1, 0.1), (2.0, 0.1), (1.0, 2.0)]
theta_values = np.linspace(0, 5, 200)

for i, (alpha, beta) in enumerate(params, start=1):
   pdf_values = [pareto_pdf(theta, alpha, beta,x) for theta in theta_values]
   plt.plot(theta_values, pdf_values, label=f'α={alpha}, β={beta}')
   plt.xlabel('θ')
   plt.ylabel('Probability Density Function')
   plt.legend()
   plt.grid(True)
   plt.axis([0, 5, 0, 5])
   plt.show()
```


α=0.1, β=0.1
Probability Density Function
θ
α=2.0, β=0.1
Probability Density Function
θ
α=1.0, β=2.0
Probability Density Function
θ

Question 2:

Code:

```
import numpy as np
from functools import partial
import scipy.optimize
```

```
# Define the loss function
def loss_function(weights, input_data, target_labels, num_classes, alpha):
    weights = np.reshape(weights, (input_data.shape[1], num_classes))
    z = np.sum(np.exp(np.matmul(input_data, weights)), axis=1)
    regularization = (0.5 * alpha * np.sum(weights**2))
    loss = -np.sum(np.sum(input_data * np.transpose(weights[:, target_labels]), -1) -
np.log(z)) + regularization
    return loss
```

```
# Define the gradient of the loss function
def gradient_loss_function(weights, input_data, target_labels, num_classes, alpha):
    weights = np.reshape(weights, (input_data.shape[1], num_classes))
    gradient = np.zeros((input_data.shape[1], num_classes))
    unnormalized_probs = np.sum(np.exp(np.matmul(input_data, weights)), axis=1)
    for i in range(num_classes):
        target_label_mask = (target_labels == i)
        probabilities = np.exp(np.matmul(input_data, weights[:, i])) /
unnormalized_probs
        difference = target_label_mask - probabilities
        tmp = np.expand_dims(difference, 1) * input_data
        gradient[:, i] = (alpha * weights[:, i]) - np.sum(tmp, 0)
    gradient = np.reshape(gradient, (-1,))
    return gradient
```

```
# Function to generate different feature sets
def generate_features(x, feature_type):
    num_features = x.shape[-1]
    if feature_type == 0:
        # Add bias feature
        phi = np.zeros((x.shape[0], num_features + 1))
        phi[:, 0] = 1.0
        phi[:, 1:] = x
    elif feature_type == 1:
        # Add linear and quadratic features
        phi = np.zeros((x.shape[0], (num_features * 2) + 1))
```

```
        phi[:, 0] = 1.0
        phi[:, 1:(num_features + 1)] = x
        phi[:, (num_features + 1):] = x**2
    elif feature_type == 2:
        # Add interaction features
        num_phi = int(((num_features + 1) * num_features / 2) + num_features + 1)
        phi = np.zeros((x.shape[0], num_phi))
        phi[:, 0] = 1.0
        phi[:, 1:(num_features + 1)] = x
        phi[:, (num_features + 1):((2 * num_features) + 1)] = x**2
        for i in range(x.shape[0]):
            x_feats = x[i]
            curr_idx = (2 * num_features) + 1
            for j in range(x_feats.shape[0]):
                for k in range(x_feats.shape[0]):
                    if k <= j:
                        continue
                    phi[i, curr_idx] = x_feats[j] * x_feats[k]
                    curr_idx += 1
    else:
        assert False
    return phi
```

```
# Load data
data = np.load("gamma.npy", allow_pickle=True).item()
train_data = data["train"]
train_labels = data["trainLabels"]
test_data = data["test"]
test_labels = data["testLabels"]

num_classes = np.unique(train_labels).shape[0]
#regularization parameter
alpha = 1e-6
#no of feature sets
num_feature_sets = 3

# Normalize data
x_offset = (np.min(train_data, 0) + np.max(train_data, 0)) / 2.0
x_scale = (np.max(train_data, 0) - np.min(train_data, 0)) / 2.0
train_data = (train_data - x_offset) / x_scale
test_data = (test_data - x_offset) / x_scale
```

```
for t in range(num_feature_sets):
   #generate features for training and testing
   phi_train = generate_features(train_data, t)
   phi_test = generate_features(test_data, t)
   # get the size
   phi_size = phi_train.shape[1]

   w0 = np.zeros(num_classes * phi_train.shape[1])
   print("Feature set", t)

   # Partial functions for loss function and gradient
   loss_function_partial = partial(loss_function, x=phi_train, t=train_labels,
k=num_classes, alpha=alpha)
   grad_func_partial = partial(gradient_loss_function, x=phi_train, t=train_labels,
k=num_classes, alpha=alpha)

   # Optimization
   options = dict()
   options["maxiter"] = 2000
   options["ftol"] = 1e-7

   #bto optimise the result
   results = scipy.optimize.minimize(fun=loss_function_partial, x0=w0,
jac=grad_func_partial, method="L-BFGS-B", options=options)
   assert results.success

  # get the optimised weights
   w_min = results.x

   # Calculate train and test loss
   train_loss = loss_function(w_min, phi_train, train_labels, num_classes, alpha)
   test_loss = loss_function(w_min, phi_test, test_labels, num_classes, alpha)

   optimized_weights_matrix = np.reshape(w_min, (phi_size, num_classes))

   # Calculate train and test accuracy
   train_predictions = np.argmax(np.matmul(phi_train, optimized_weights_matrix), 1)
   train_accuracy = np.sum(train_predictions == train_labels) /
float(train_labels.shape[0])
```

```
    test_predictions = np.argmax(np.matmul(phi_test, optimized_weights_matrix), 1)
    test_accuracy = np.sum(test_predictions == test_labels) /
float(test_labels.shape[0])

    # Print results
    print("Training Data")
    print("Accuracy:", train_accuracy)
    print("Negative Log-Like:", train_loss)

    print("Test Data")
    print("Accuracy:", test_accuracy)
    print("Negative Log-Like:", test_loss)
```

a) Accuracy & negative log-posterior probability of the classifier when training & evaluating on train

- Feature set 0
  - Accuracy: 0.7895636172450052 = **0.790**
  - Negative Log-Like: 6992.992897614068 ≈**6990**
- Feature set 1
  - Accuracy: 0.8447029442691903 =**0.845**
  - Negative Log-Like: 5901.1892026916075 ≈**5900**
- Feature set 2
  - Accuracy: 0.8608044164037855=**0.861**
  - Negative Log-Like: 5123.7167270205955≈**5120**

---

c) Accuracy & negative log-posterior probability of the classifier when training & evaluating on test

- Feature set 0
  - Accuracy: 0.7965299684542587 = **0.797**
  - Negative Log-Like: 1706.7319877844368 ≈ **1710**
- Feature set 1
  - Accuracy: 0.852260778128286 = **0.853**
  - Negative Log-Like: 1434.2625710069 ≈ **1440**
- Feature set 2
  - Accuracy: 0.8719768664563617 = **0.871**
  - Negative Log-Like: 1274.748323030174 ≈ **1270**

---

c) *Compare the test accuracies of the logistic regression models to the Gaussian naive Bayes classifier from Homework 1. Which method is more accurate?*

1. Accuracy Comparison:

- Linear features: 79.7%
- Diagonal quadratic features: 85.3%
- General quadratic features: 87.1%
- There's no overfitting.
- This is as the most complex model (general quadratic) is also the most accurate.
- Train & test accuracies are similar.

2. Comparison with Naive Bayes:

- Using the same training data, a Gaussian naive Bayes classifier had an accuracy of 72.9%.
- Both diagonal-quadratic logistic regression & naive Bayes can represent similar decision boundaries.
- However, logistic regression models, including even the simpler linear features, perform better than naive Bayes.
- This improvement is probably because the Gamma telescope data's distribution doesn't match the assumptions of the Gaussian naive Bayes model.

Question 3:

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from functools import partial
import scipy.optimize
from plotting_utils import plotter_classifier
```

```
# Loss function for multinomial logistic regression
def loss_function(weights, input_data, target_labels, num_classes, alpha):
    weights = np.reshape(weights, (input_data.shape[1], num_classes))
    z = np.sum(np.exp(np.matmul(input_data, weights)), axis=1)
    regularization = (0.5 * alpha * np.sum(weights**2))
    loss = -np.sum(np.sum(input_data * np.transpose(weights[:, target_labels]), -1) -
np.log(z)) + regularization
    return loss

# Gradient of the loss function for multinomial logistic regression
def gradient_loss_function(weights, input_data, target_labels, num_classes, alpha):
    weights = np.reshape(weights, (input_data.shape[1], num_classes))
    gradient = np.zeros((input_data.shape[1], num_classes))
    unnormalized_probs = np.sum(np.exp(np.matmul(input_data, weights)), axis=1)
    for i in range(num_classes):
        target_label_mask = (target_labels == i)
        probabilities = np.exp(np.matmul(input_data, weights[:, i])) /
unnormalized_probs
        difference = target_label_mask - probabilities
        tmp = np.expand_dims(difference, 1) * input_data
        gradient[:, i] = (alpha * weights[:, i]) - np.sum(tmp, 0)
    gradient = np.reshape(gradient, (-1,))
    return gradient

# list of datasets
total_datasets = ["partA_two_clouds", "partB_three_triangle", "partC_three_linear"]
names1 = ["Two Clouds", "Three Triangle", "Three Linear"]

# loop over the datasets
for d, dataset in enumerate(total_datasets):
    data = np.load("{}.npy".format(dataset), allow_pickle=True).item()
```

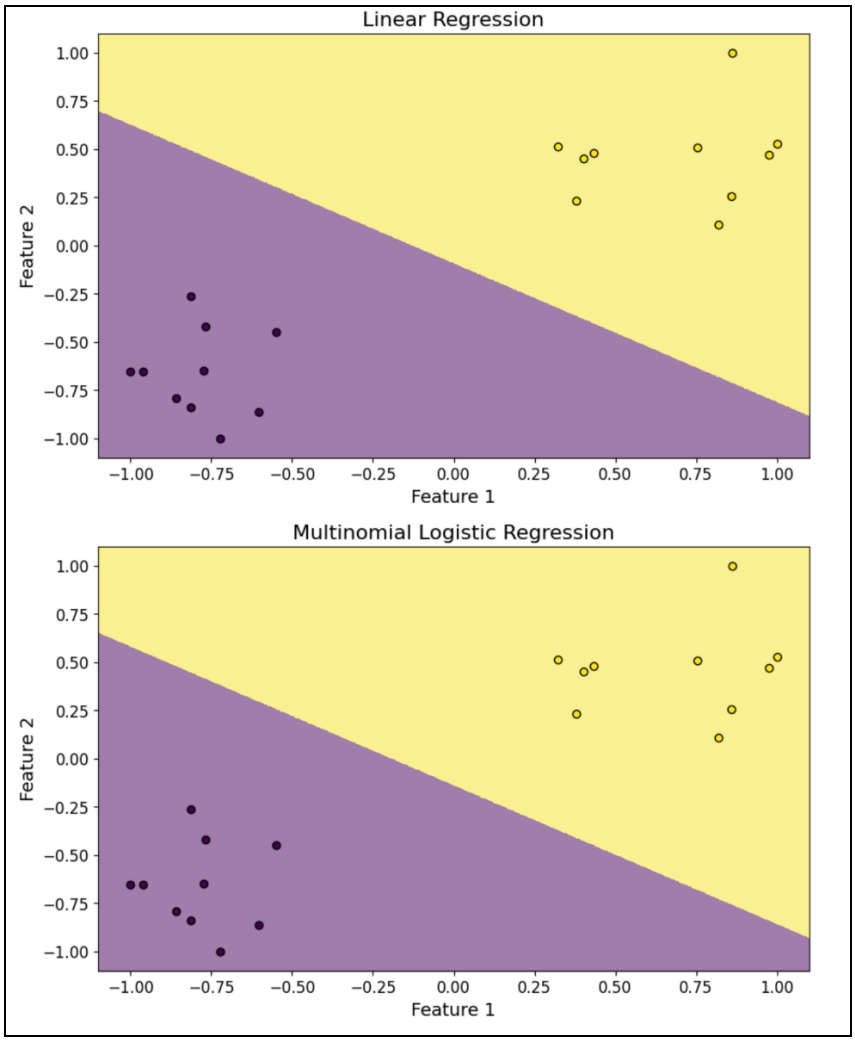
```
    # Unpack the data
    input_train = data["Xtrain"]
    input_test = data["Xtest"]
    target_train = data["Ytrain"]
    target_test = data["Ytest"]

    # Rescale inputs to [-1, 1] for numerical stability
    input_offset = (np.min(input_train) + np.max(input_train)) / 2.0
    input_scale = (np.max(input_train) - np.min(input_train)) / 2.0
    input_train_rescaled = (input_train - input_offset) / input_scale
    input_test_rescaled = (input_test - input_offset) / input_scale

    # Basis function to include bias term
    def basis_function(x):
        return np.concatenate([np.ones((x.shape[0], 1)), x], axis=-1)

    # Compute linear features
    phi_train = basis_function(input_train_rescaled)
    phi_test = basis_function(input_test_rescaled)

    # linear regression
    tmp = np.matmul(np.transpose(phi_train), phi_train)
    tmp = np.linalg.inv(tmp)
    tmp = np.matmul(tmp, np.transpose(phi_train))
    linear_weights = np.matmul(tmp, target_train)
    fhat_train = np.matmul(phi_train, linear_weights)
    fhat_test = np.matmul(phi_test, linear_weights)

    predicted_train = np.argmax(fhat_train, 1)
    predicted_test = np.argmax(fhat_test, 1)
    target_train_int = np.argmax(target_train, 1)
    target_test_int = np.argmax(target_test, 1)

    train_err = np.sum(predicted_train != target_train_int) /
target_train_int.shape[0]
    test_err = np.sum(predicted_test != target_test_int) / target_test_int.shape[0]
    print(f"Accuracy on Training Set: {(1 - train_err):.3f}")
    print(f"Accuracy on Test Set: {(1 - test_err):.3f}")

    plotter_classifier(linear_weights, basis_function, input_test_rescaled,
target_test_int, title="Linear Regression")
    plt.title("Linear Regression", fontsize=16)
```

```
    plt.xlabel("Feature 1", fontsize=14)
    plt.ylabel("Feature 2", fontsize=14)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)

    alpha = 1e-6
    num_classes = np.unique(target_train_int).shape[0]
    loss_func = partial(loss_function, \
                        input_data=phi_train, target_labels=target_train_int,
num_classes=num_classes, alpha=alpha)
    grad_func = partial(gradient_loss_function, \
                        input_data=phi_train, target_labels=target_train_int,
num_classes=num_classes, alpha=alpha)

    options = dict()
    options["maxiter"] = 2000
    options["ftol"] = 1e-7
    initial_weights = np.zeros((phi_train.shape[1] * num_classes))
    results = scipy.optimize.minimize(fun=loss_func, \
                                      x0=initial_weights, jac=grad_func,
method="L-BFGS-B", options=options)

    assert(results.success)
    optimized_weights = results.x
    softmax_weights = np.reshape(optimized_weights, (phi_train.shape[1], num_classes))

    predicted_train = np.argmax(np.matmul(phi_train, softmax_weights), 1)
    train_err = np.sum(predicted_train != target_train_int) /
float(target_train_int.shape[0])

    predicted_test = np.argmax(np.matmul(phi_test, softmax_weights), 1)
    test_err = np.sum(predicted_test != target_test_int) /
float(target_test_int.shape[0])

    print(f"Accuracy on Training Set: {(1 - train_err):.3f}")
    print(f"Accuracy on Test Set: {(1 - test_err):.3f}")

    fig2 = plt.figure(figsize=(10, 6))
    plotter_classifier(softmax_weights, basis_function, input_test_rescaled,
target_test_int, title="Multinomial Logistic Regression")
    plt.title("Multinomial Logistic Regression", fontsize=16)
    plt.xlabel("Feature 1", fontsize=14)
```

```
    plt.ylabel("Feature 2", fontsize=14)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.show()
```

a)

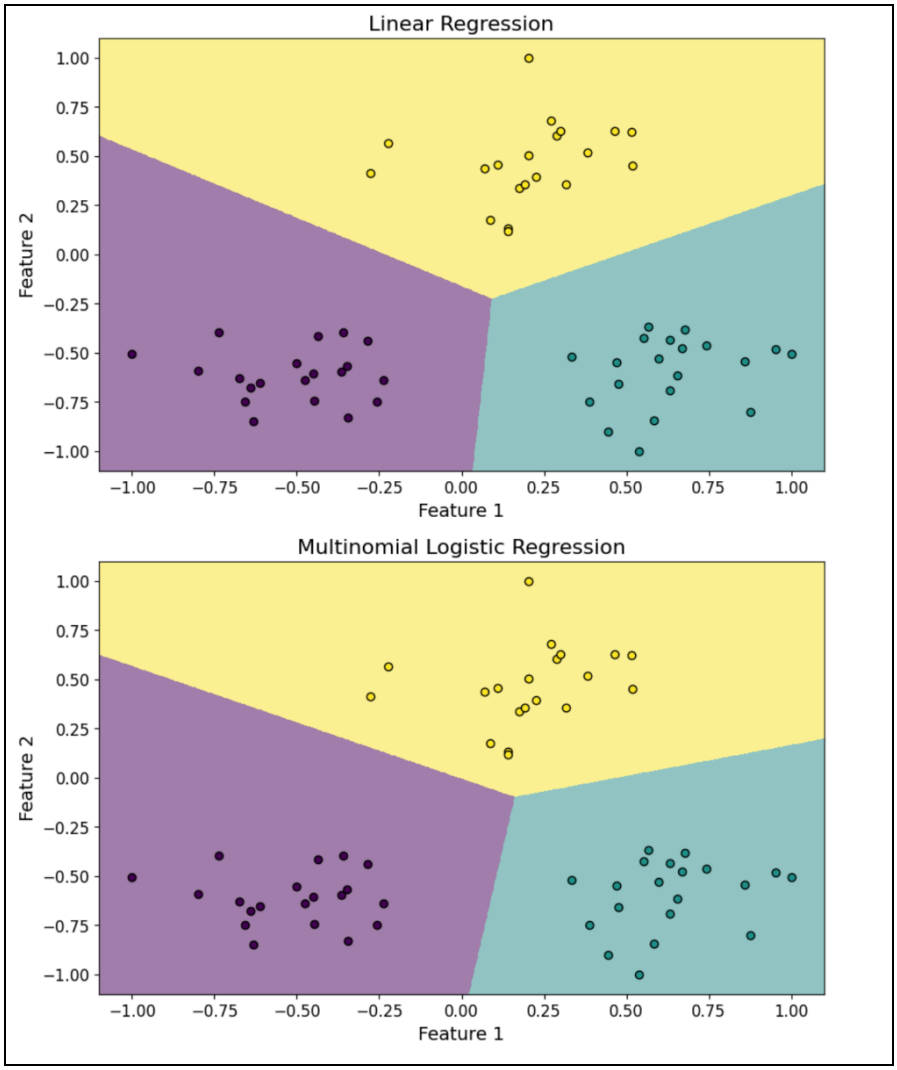
- Linear Regression Model:
  - Accuracy on Training Set: 1.000
  - Accuracy on Test Set: 1.000
- Multinomial logistic regression model:
  - Accuracy on Training Set: 1.000
  - Accuracy on Test Set: 1.000
- Both classifiers correctly label the test data without any errors.



b)

- Linear Regression Model:
  - Accuracy on Training Set: 1.000
  - Accuracy on Test Set: 1.000
- Multinomial logistic regression model:
  - Accuracy on Training Set: 1.000
  - Accuracy on Test Set: 1.000

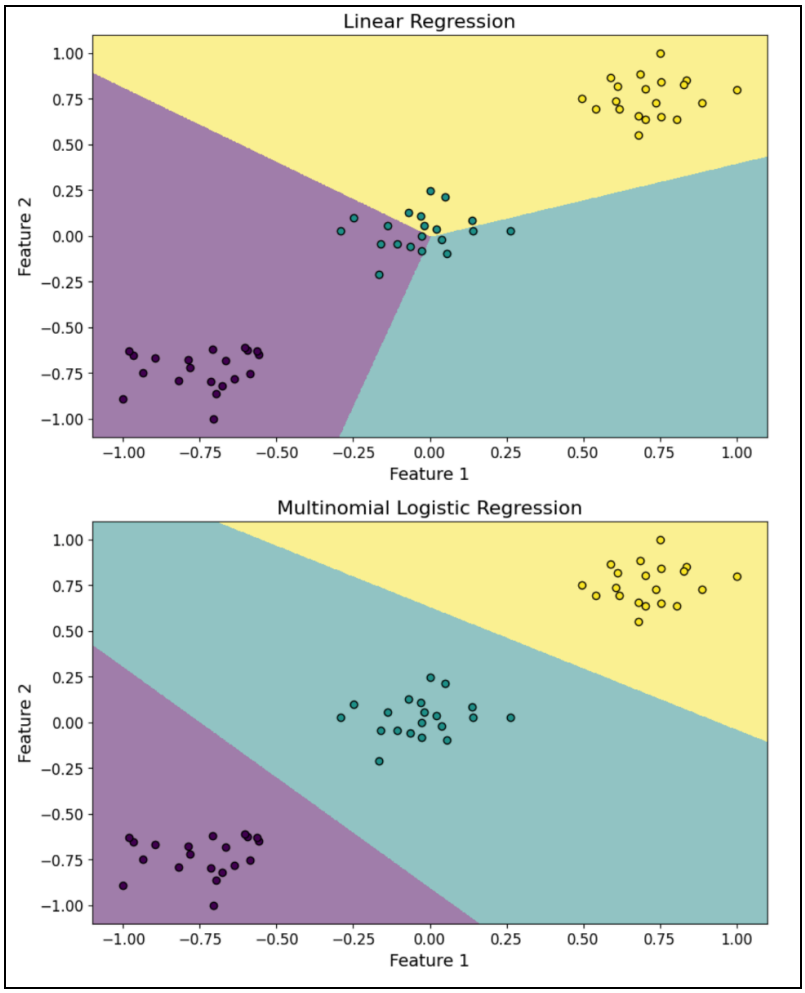
- Both classifiers correctly label the test data without any errors.



---

c)

- Linear Regression Model:
  - Accuracy on Training Set: 0.783
  - Accuracy on Test Set: 0.733
- Multinomial logistic regression model:
  - Accuracy on Training Set: 1.000
  - Accuracy on Test Set: 1.000
- The disparity in performance between the two models can be attributed to the nature of the data.
- Logistic regression's non-linear mapping between linear predictors and probabilities allows it to better capture the complex relationships between features and classes.
- As a result, it successfully classifies the test data without any errors.
- Linear regression struggles with correctly classifying the data, particularly due to the middle class where the linear fit becomes essentially flat.
- This results in many points in this class being incorrectly labeled.


Linear Regression
Feature 2
Feature 1
Multinomial Logistic Regression
Feature 2
Feature 1