DATA37711_AS1_Adrian Cao

November 6, 2024

1 Question 1: Utility and Decision

1.1 (a) Prove that if Victor's utility function is linear in money, then he will bet all his wealth on Biden winning.

Let $U(y) = y$ be Victor's utility function, where y is the amount of money he has, and let x be the amount he bets on Biden winning. His wealth depends on the outcome of the bet:

- If Biden wins, Victor receives a return of $\frac{1}{0.6}$ per dollar bet, since the betting markets imply $P(\text{Biden wins}) = 0.6$. Thus, his total wealth will be $d + x \times \frac{2}{3}$.
- If Biden loses, he loses his bet and his wealth will be $d - x$.

Thus, Victor's expected utility is:

$$\mathbb{E}[U(x)] = 0.85 \times \left(d + x \times \frac{2}{3}\right) + 0.15 \times (d - x).$$

Simplifying:

$$\mathbb{E}[U(x)] = 0.85\left(d + \frac{2x}{3}\right) + 0.15(d - x) = 0.85d + \frac{17x}{30} + 0.15d - 0.15x.$$

$$\mathbb{E}[U(x)] = d + x\left(\frac{17}{30} - 0.15\right).$$

To maximize expected utility, we differentiate $\mathbb{E}[U(x)]$ with respect to x:

$$\frac{d}{dx}\mathbb{E}[U(x)] = \frac{17}{30} - 0.15.$$

Since $\frac{0.85}{0.6} - 0.15 = \frac{5}{12} > 0$, the derivative is positive, meaning the expected utility increases as x increases. Therefore, Victor will bet his entire wealth d on Biden winning.

Conclusion: If Victor's utility function is linear in money and the betting markets give $P(\text{Biden wins}) = 0.6$, he will bet all his wealth d on Biden winning.

1.2 (b) Suppose Victor's utility from y dollars is $U(y) = \log(y)$. How much of his d dollars should he bet on Biden winning?

Now, Victor's utility function is $U(y) = \log(y)$, which reflects risk aversion. His wealth after the bet will be:

- If Biden wins, his wealth is $d + x \times \frac{2}{3}$.

- If Biden loses, his wealth is $d - x$.

Thus, his expected utility is:

$$\mathbb{E}[U(x)] = 0.85 \log\left(d + x \times \frac{2}{3}\right) + 0.15 \log(d - x).$$

To find the optimal bet size x, we differentiate the expected utility with respect to x:

$$\frac{d}{dx}\mathbb{E}[U(x)] = 0.85 \frac{1}{d + \frac{2x}{3}} \times \frac{2}{3} - 0.15 \frac{1}{d - x}.$$

Setting this equal to zero to find the critical point:

$$0.85 \frac{2}{3d + 2x} = 0.15 \frac{1}{d - x}.$$

Multiply both sides by $(3d + 2x)(d - x)$:

$$0.85 \times 2 \times (d - x) = 0.15 \times (3d + 2x).$$

Simplifying further:

$$(d - x) = \frac{0.15}{1.7} \times (3d + 2x).$$

$$d - x = \frac{9}{34}d + \frac{6}{34}x.$$

This simplifies to:

$$\frac{25}{34}d = \frac{40}{34}x,$$

$$x = \frac{25}{40}d = 0.675d.$$

Thus, Victor should bet approximately 67.5% of his wealth on Biden winning.

1.3 (c) Suppose you had a net worth of $500,000. Would you personally make the bet from part (b)?

From part (b), Victor would bet $0.675d$. If your net worth is $500,000, this means you would bet:

$$0.675 \times 500,000 = 337,500.$$

Whether you would personally make this bet depends on your risk tolerance. Betting more half of your net worth on one event is quite risky, even with a belief that the probability of winning is high. Most individuals, depending on their risk aversion, would likely bet a smaller portion of their wealth.

2 Question 2: Cross Entropy as proper scoring rule

Let Y be a binary random variable (i.e., $Y \in \{0, 1\}$), and let X be the covariates. A loss function L on categorical data is called a **proper scoring rule** if the predictive function f^* that minimizes the implied risk is the conditional probability $P(Y|X = x)$.

The **Cross Entropy loss** for binary classification is given by:

$$L(Y, f(X)) = -Y \log(f(X)) - (1 - Y) \log(1 - f(X)),$$

where $f(X)$ represents the predicted probability that $Y = 1$ given X, i.e., $f(X) = P(Y = 1|X)$.

The **expected risk** associated with the loss function $L(Y, f(X))$ is:

$$R(f) = \mathbb{E}_{X,Y}[L(Y, f(X))].$$

We need to minimize this expected risk to find the optimal predictive function $f^*(X)$.

The expected risk can be written as:

$$R(f) = \mathbb{E}_{X,Y}\left[-Y \log(f(X)) - (1 - Y) \log(1 - f(X))\right].$$

We can expand this using the law of total expectation:

$$R(f) = \mathbb{E}_X\left[\mathbb{E}_Y\left[-Y \log(f(X)) - (1 - Y) \log(1 - f(X)) \mid X\right]\right].$$

Let $p(X) = P(Y = 1 \mid X)$ be the true conditional probability that $Y = 1$ given X. Then, we have:

$$R(f) = \mathbb{E}_X\left[-p(X) \log(f(X)) - (1 - p(X)) \log(1 - f(X))\right].$$

To minimize the expected risk, we differentiate the risk $R(f)$ with respect to $f(X)$. The derivative is:

$$\frac{d}{df(X)} R(f) = -\frac{p(X)}{f(X)} + \frac{1 - p(X)}{1 - f(X)}.$$

Setting this derivative equal to zero to find the critical points:

$$-\frac{p(X)}{f(X)} + \frac{1 - p(X)}{1 - f(X)} = 0.$$

Multiplying both sides by $f(X)(1 - f(X))$, we get:

$$-p(X)(1 - f(X)) + (1 - p(X))f(X) = 0.$$

Expanding and simplifying:

$$-p(X) + p(X)f(X) + f(X) - p(X)f(X) = 0,$$

$$f(X) = p(X).$$

The function $f^*(X)$ that minimizes the expected risk is given by $f^*(X) = P(Y = 1 \mid X)$, which is the true conditional probability. Thus, we have shown that the Cross Entropy loss is a **proper scoring rule**.

3 Question 3: Constent step size in finite sample

In practice, machine learning models are typically trained on **finite datasets**, and the objective is to achieve good generalization performance rather than to precisely minimize the objective function. While convergence theory requires the step size ρ_t to decay (i.e., $\sum \rho_t^2 < \infty$) for stochastic gradient descent (SGD) to converge to the global minimum, this is primarily relevant for **infinite sample** settings. In the finite sample setting, it is often reasonable to use a constant step size because it allows for faster learning and avoids the vanishing updates that occur with decaying step sizes. Constant step sizes help the model explore the parameter space more thoroughly, allowing it to escape poor local minima or saddle points and find solutions that generalize well to unseen data, even if the exact convergence conditions are violated.

In conclusion, using a constant step size can be reasonable because:

- The number of iterations is finite, so exact convergence is not necessary.
- Constant step size helps balance bias and variance, which improves generalization.
- It allows for faster convergence to a near-optimal solution, which is often sufficient in practical machine learning.
- Early stopping can regularize the learning process, making decaying step sizes less important.

4 Question 4: Testing Residuals for Normality

4.1 Load and Preprocess the Dataset

We load the ALE dataset and split it into predictors (the first four variables) and the target variable (ALE). We normalize the predictors if necessary and split the data into training and testing sets.

4.2 Fit Machine Learning Models

We fit several models from the `sklearn` package, such as:

- Linear regression
- Support Vector Machine (SVM) regression
- Decision Tree regression

The residuals are calculated as: [residuals = Y_{true} - Y_{pred}] where (Y_{true}) are the actual ALE values, and (Y_{pred}) are the predicted values from the models.

4.3 Normality Testing

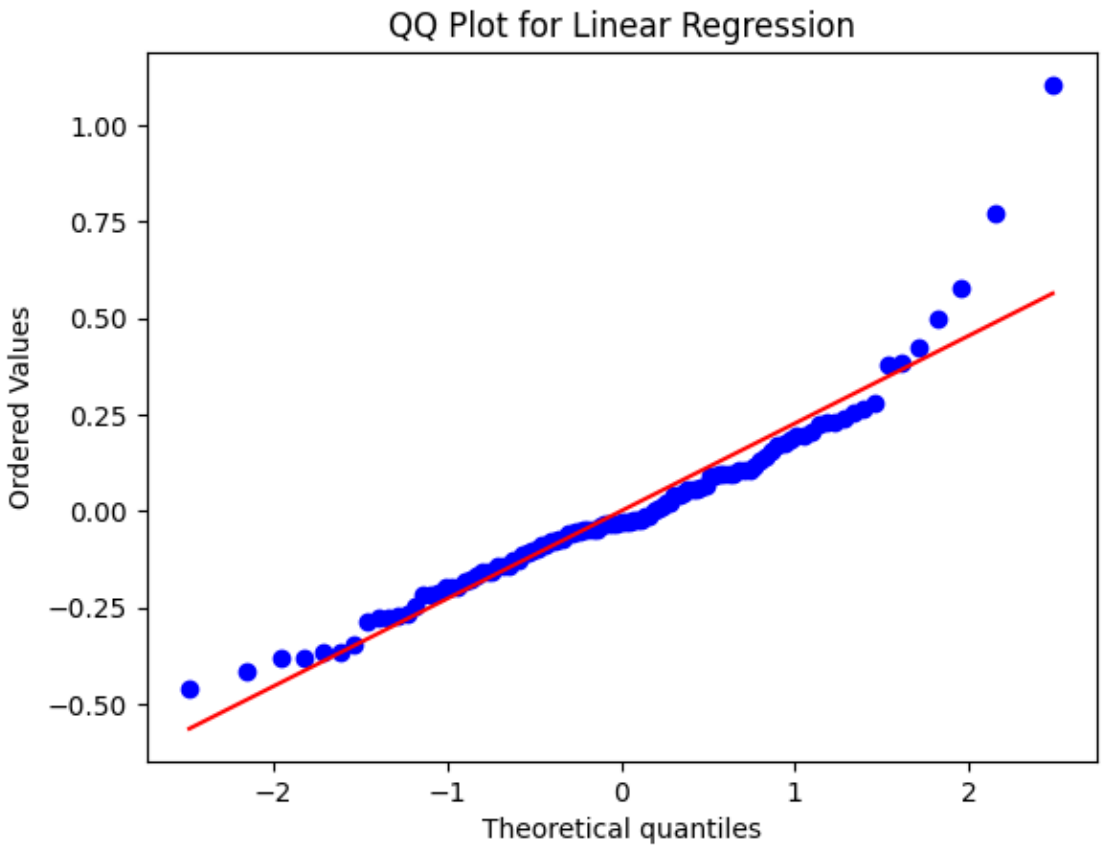
To check if the residuals are approximately normally distributed, we perform the following tests:

- **QQ Plot**: A QQ plot compares the distribution of the residuals to a normal distribution. If the residuals follow a straight line, they are approximately normal.
- **Normality Test**: We apply `scipy.stats.normaltest`, which returns a p-value. If the p-value is small (e.g., less than 0.05), we reject the null hypothesis that the residuals are normally distributed.

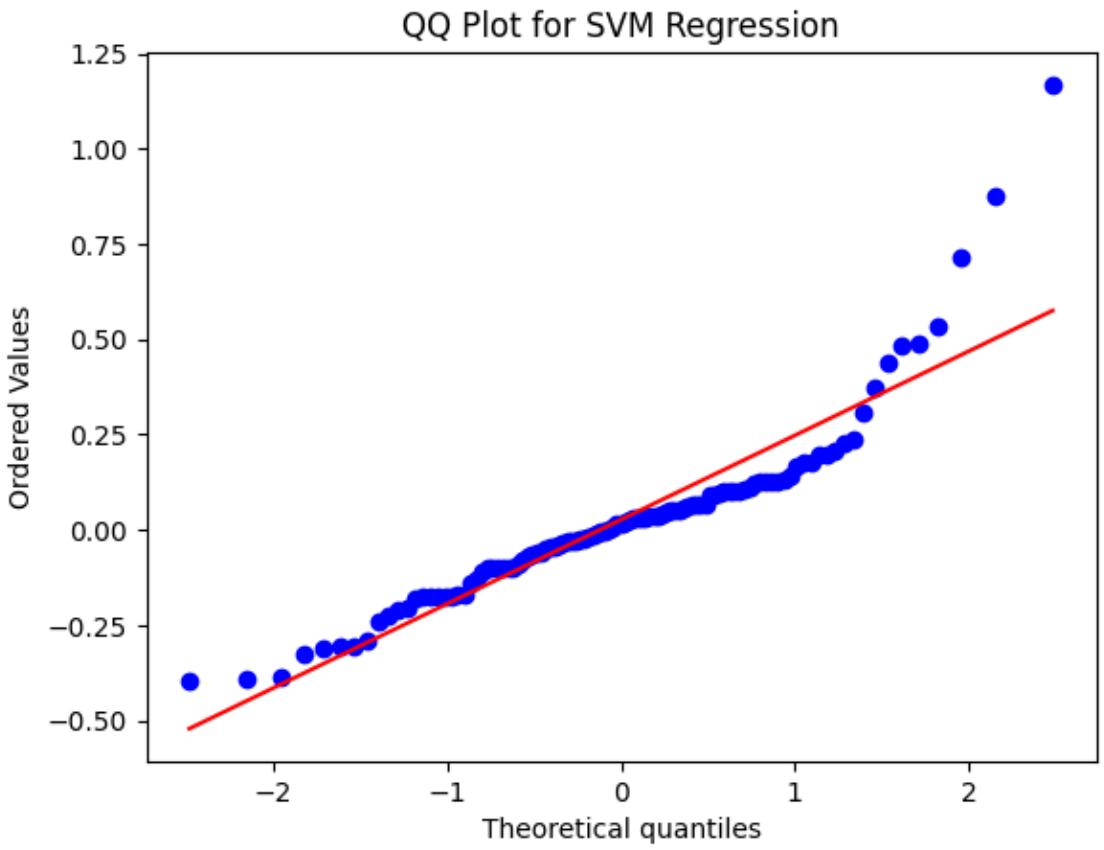
4.4 Conclusion

By fitting different models and examining the residuals, we can conclude the normality assumption is not fitted for the linear regression and SVM model. The residuals from the regularized decision tree model are close to normal, but there are some deviations at the extremes, as shown in the QQ plot.

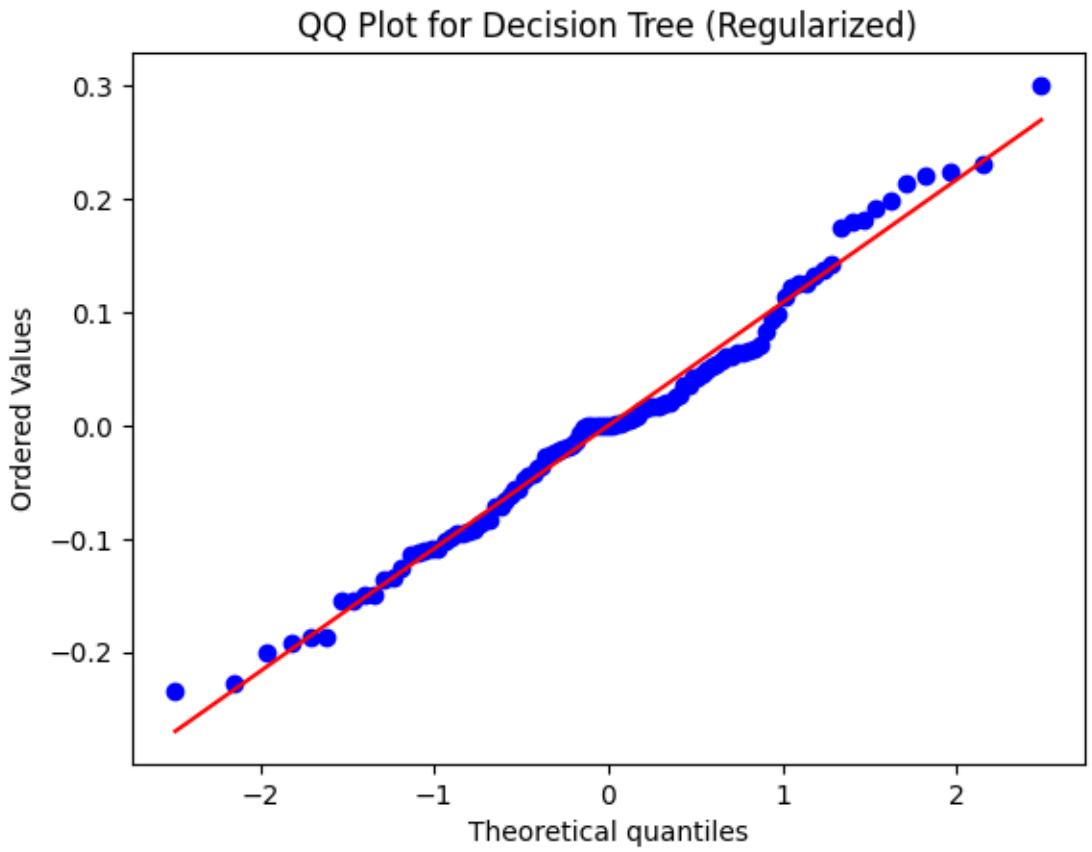
[53]:
```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt
import scipy.stats as stats

# Load the ALE dataset
data = pd.read_csv('mcs_ds_edited_iter_shuffled.csv')

# Extract features (first four variables) and target (ALE)
X = data.iloc[:, :4]  # First four variables as predictors
y = data['ale']  # ALE target

# Initialize models
models = {
    'Linear Regression': LinearRegression(),
    'SVM Regression': SVR(),
    'Decision Tree (Regularized)': DecisionTreeRegressor(max_depth=5)  #␣
 ↪Regularized Decision Tree
}

# Fit models and compute residuals
residuals = {}

for name, model in models.items():
    model.fit(X, y)
    y_pred = model.predict(X)
    residuals[name] = y - y_pred

    # Plot QQ plot
    plt.figure()
    stats.probplot(residuals[name], dist="norm", plot=plt)
    plt.title(f'QQ Plot for {name}')
    plt.show()

    # Perform normality test
    stat, p_value = stats.normaltest(residuals[name])
    print(f'{name}: Normaltest p-value = {p_value:.4f}')
```




```
Linear Regression: Normaltest p-value = 0.0000
```




```
SVM Regression: Normaltest p-value = 0.0000
```




```
Decision Tree (Regularized): Normaltest p-value = 0.5008
```

5 Question 5

5.1 Classification

[54]:
```python
import numpy as np
import torch
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load dataset
data = load_breast_cancer()
X = data['data']
y = data['target']

# Split dataset (80/20 split)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Baseline using sklearn's logistic regression
baseline_model = LogisticRegression(max_iter=1000)
baseline_model.fit(X_train, y_train)
y_pred_baseline = baseline_model.predict(X_test)

# Calculate accuracy
baseline_accuracy = accuracy_score(y_test, y_pred_baseline)
print(f"Baseline Logistic Regression Accuracy: {baseline_accuracy:.4f}")
```

```
Baseline Logistic Regression Accuracy: 0.9561
```

```
c:\Users\a_i_b\miniconda3\Lib\site-
packages\sklearn\linear_model\_logistic.py:469: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[55]:
```
# (a)

import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Convert original (unscaled) data to PyTorch tensors
X_train_torch = torch.tensor(X_train, dtype=torch.float32)
X_test_torch = torch.tensor(X_test, dtype=torch.float32)
y_train_torch = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  #␣
 ↪reshape to column vector
y_test_torch = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

# Logistic regression model definition
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)  # Linear layer: w^T x + b

    def forward(self, x):
```

```
        return torch.sigmoid(self.linear(x))  # Sigmoid activation function

# Function to train model
def train_model(model, X_train_torch, y_train_torch, num_epochs=1000, lr=0.01):
    criterion = nn.BCELoss()  # Binary Cross Entropy Loss
    optimizer = optim.SGD(model.parameters(), lr=lr)  # SGD optimizer
    for epoch in range(num_epochs):
        # Forward pass: Compute predictions and loss
        outputs = model(X_train_torch)
        loss = criterion(outputs, y_train_torch)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print loss every 100 epochs
        if (epoch+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
    return model

# Function to evaluate model
def evaluate_model(model, X_test_torch, y_test_torch):
    with torch.no_grad():
        y_pred_torch = model(X_test_torch)
        y_pred_torch = (y_pred_torch > 0.5).float()  # Threshold at 0.5
    accuracy_torch = accuracy_score(y_test_torch, y_pred_torch)
    return accuracy_torch

# Train and evaluate on unscaled data
print("Training and evaluating on unscaled data:")
input_dim = X_train.shape[1]
model_unscaled = LogisticRegressionModel(input_dim)
model_unscaled = train_model(model_unscaled, X_train_torch, y_train_torch, lr=0.
 ↪01)
accuracy_unscaled = evaluate_model(model_unscaled, X_test_torch, y_test_torch)
print(f"PyTorch Logistic Regression Accuracy (Unscaled Data):␣
 ↪{accuracy_unscaled:.4f}")

# Standardize the data (scaled data)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert scaled data to PyTorch tensors
X_train_scaled_torch = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_scaled_torch = torch.tensor(X_test_scaled, dtype=torch.float32)
```

```
# Train and evaluate on scaled data
print("\nTraining and evaluating on scaled data:")
model_scaled = LogisticRegressionModel(input_dim)
model_scaled = train_model(model_scaled, X_train_scaled_torch, y_train_torch,␣
 ↪lr=0.01)
accuracy_scaled = evaluate_model(model_scaled, X_test_scaled_torch,␣
 ↪y_test_torch)
print(f"PyTorch Logistic Regression Accuracy (Scaled Data): {accuracy_scaled:.
 ↪4f}")
```

```
Training and evaluating on unscaled data:
Epoch [100/1000], Loss: 37.1429
Epoch [200/1000], Loss: 37.1429
Epoch [300/1000], Loss: 37.1429
Epoch [400/1000], Loss: 37.1429
Epoch [500/1000], Loss: 37.1429
Epoch [600/1000], Loss: 37.1429
Epoch [700/1000], Loss: 37.1429
Epoch [800/1000], Loss: 37.1429
Epoch [900/1000], Loss: 37.1429
Epoch [1000/1000], Loss: 37.1429
PyTorch Logistic Regression Accuracy (Unscaled Data): 0.6228

Training and evaluating on scaled data:
Epoch [100/1000], Loss: 0.2500
Epoch [200/1000], Loss: 0.1909
Epoch [300/1000], Loss: 0.1636
Epoch [400/1000], Loss: 0.1472
Epoch [500/1000], Loss: 0.1361
Epoch [600/1000], Loss: 0.1279
Epoch [700/1000], Loss: 0.1215
Epoch [800/1000], Loss: 0.1164
Epoch [900/1000], Loss: 0.1122
Epoch [1000/1000], Loss: 0.1086
PyTorch Logistic Regression Accuracy (Scaled Data): 0.9737
```

If we don't perform scalization, the PyTorch implementation of logistic regression has significantly lower accuracy compared to the baseline model from sklearn. But with normalization, it performs even better.

[56]:
```
# (b)
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Normal
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

```python
# Probit regression model definition
class ProbitRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(ProbitRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)  # Linear layer: w^T x + b

    def forward(self, x):
        z = self.linear(x)
        return Normal(0, 1).cdf(z)  # CDF of standard normal distribution

# Function to train probit model
def train_probit_model(model, X_train_torch, y_train_torch, num_epochs=1000,␣
 ↪lr=0.01):
    criterion = nn.BCELoss()  # Binary Cross Entropy Loss
    optimizer = optim.SGD(model.parameters(), lr=lr)  # SGD optimizer
    for epoch in range(num_epochs):
        # Forward pass: Compute predictions and loss
        outputs = model(X_train_torch)
        loss = criterion(outputs, y_train_torch)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print loss every 100 epochs
        if (epoch+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Probit Loss: {loss.item():.
 ↪4f}')
    return model

# Function to evaluate probit model
def evaluate_probit_model(model, X_test_torch, y_test_torch):
    with torch.no_grad():
        y_pred_torch = model(X_test_torch)
        y_pred_torch = (y_pred_torch > 0.5).float()  # Threshold at 0.5
    accuracy_torch = accuracy_score(y_test_torch, y_pred_torch)
    return accuracy_torch

# Train and evaluate probit regression on unscaled data
print("Training and evaluating Probit model on unscaled data:")
input_dim = X_train.shape[1]
model_probit_unscaled = ProbitRegressionModel(input_dim)
model_probit_unscaled = train_probit_model(model_probit_unscaled,␣
 ↪X_train_torch, y_train_torch, lr=0.01)
```

```
accuracy_probit_unscaled = evaluate_probit_model(model_probit_unscaled,␣
 ↪X_test_torch, y_test_torch)
print(f"Probit Regression Accuracy (Unscaled Data): {accuracy_probit_unscaled:.
 ↪4f}")

# Standardize the data (scaled data)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert scaled data to PyTorch tensors
X_train_scaled_torch = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_scaled_torch = torch.tensor(X_test_scaled, dtype=torch.float32)

# Train and evaluate probit regression on scaled data
print("\nTraining and evaluating Probit model on scaled data:")
model_probit_scaled = ProbitRegressionModel(input_dim)
model_probit_scaled = train_probit_model(model_probit_scaled,␣
 ↪X_train_scaled_torch, y_train_torch, lr=0.01)
accuracy_probit_scaled = evaluate_probit_model(model_probit_scaled,␣
 ↪X_test_scaled_torch, y_test_torch)
print(f"Probit Regression Accuracy (Scaled Data): {accuracy_probit_scaled:.4f}")
```

```
Training and evaluating Probit model on unscaled data:
Epoch [100/1000], Probit Loss: 37.1429
Epoch [200/1000], Probit Loss: 37.1429
Epoch [300/1000], Probit Loss: 37.1429
Epoch [400/1000], Probit Loss: 37.1429
Epoch [500/1000], Probit Loss: 37.1429
Epoch [600/1000], Probit Loss: 37.1429
Epoch [700/1000], Probit Loss: 37.1429
Epoch [800/1000], Probit Loss: 37.1429
Epoch [900/1000], Probit Loss: 37.1429
Epoch [1000/1000], Probit Loss: 37.1429
Probit Regression Accuracy (Unscaled Data): 0.6228

Training and evaluating Probit model on scaled data:
Epoch [100/1000], Probit Loss: 0.1593
Epoch [200/1000], Probit Loss: 0.1232
Epoch [300/1000], Probit Loss: 0.1082
Epoch [400/1000], Probit Loss: 0.0996
Epoch [500/1000], Probit Loss: 0.0938
Epoch [600/1000], Probit Loss: 0.0896
Epoch [700/1000], Probit Loss: 0.0863
Epoch [800/1000], Probit Loss: 0.0837
Epoch [900/1000], Probit Loss: 0.0815
Epoch [1000/1000], Probit Loss: 0.0796
```

```
Probit Regression Accuracy (Scaled Data): 0.9737
```

With the change of the Probit model, it does not help with the better predicting. Similiarly, for the unscaled data, it perform much worse compared to the baseline. However, with scaling, they have significantly better performance compared to the baseline model.

5.2 Regression

[57]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

# Load the ALE dataset
data = pd.read_csv('mcs_ds_edited_iter_shuffled.csv')

# Assuming ALE dataset is loaded into a DataFrame 'df'
X = data.drop('ale', axis=1).values  # Features
y = data['ale'].values  # Target variable (ALE)

# Split into training and testing sets (80/20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Baseline model using sklearn's linear regression
baseline_model = LinearRegression()
baseline_model.fit(X_train, y_train)
y_pred_baseline = baseline_model.predict(X_test)

# Evaluate baseline performance using Mean Absolute Error (MAE)
baseline_mae = mean_absolute_error(y_test, y_pred_baseline)
print(f"Baseline Linear Regression MAE: {baseline_mae:.4f}")
```

```
Baseline Linear Regression MAE: 0.1861
```

[58]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error

# Convert unscaled data to tensors
X_train_torch_unscaled = torch.tensor(X_train, dtype=torch.float32)
X_test_torch_unscaled = torch.tensor(X_test, dtype=torch.float32)
y_train_torch = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  #␣
 ↪Reshape to column vector
y_test_torch = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

# Standardize the data (scaled data)
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert scaled data to tensors
X_train_torch_scaled = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_torch_scaled = torch.tensor(X_test_scaled, dtype=torch.float32)

# Define the linear regression model
class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)  # Linear model: w^T x + b

    def forward(self, x):
        return self.linear(x)  # f(x) = w^T x + b

# Initialize model, loss function, and optimizer
input_dim = X_train.shape[1]

def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight)
        nn.init.zeros_(m.bias)

# Function to train the model
def train_model(model, X_train_torch, y_train_torch, num_epochs=1000, lr=0.001):
    criterion = nn.MSELoss()  # Mean squared error loss
    optimizer = optim.SGD(model.parameters(), lr=lr)  # SGD optimizer with␣
 ↪learning rate
    model.apply(initialize_weights)  # Initialize weights
    for epoch in range(num_epochs):
        # Forward pass: compute predictions and loss
        outputs = model(X_train_torch)
        loss = criterion(outputs, y_train_torch)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()

        # Gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()

        # Print loss every 100 epochs
        if (epoch + 1) % 100 == 0:
```

```
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
    return model

# Function to evaluate the model
def evaluate_model(model, X_test_torch, y_test_torch):
    with torch.no_grad():
        y_pred_torch = model(X_test_torch)

        # Check for NaN or infinite values
        if torch.isnan(y_pred_torch).any() or torch.isinf(y_pred_torch).any():
            print("NaN or infinity detected in predictions!")

        mae_torch = mean_absolute_error(y_test_torch, y_pred_torch)
    return mae_torch

# Training and evaluating on unscaled data
print("Training and evaluating on unscaled data:")
model_unscaled = LinearRegressionModel(input_dim)
model_unscaled = train_model(model_unscaled, X_train_torch_unscaled,␣
 ↪y_train_torch)
mae_unscaled = evaluate_model(model_unscaled, X_test_torch_unscaled,␣
 ↪y_test_torch)
print(f"PyTorch Linear Regression MAE (Unscaled Data): {mae_unscaled:.4f}")

# Training and evaluating on scaled data
print("\nTraining and evaluating on scaled data:")
model_scaled = LinearRegressionModel(input_dim)
model_scaled = train_model(model_scaled, X_train_torch_scaled, y_train_torch)
mae_scaled = evaluate_model(model_scaled, X_test_torch_scaled, y_test_torch)
print(f"PyTorch Linear Regression MAE (Scaled Data): {mae_scaled:.4f}")
```

```
Training and evaluating on unscaled data:
Epoch [100/1000], Loss: 702.0674
Epoch [200/1000], Loss: 543.7897
Epoch [300/1000], Loss: 406.3213
Epoch [400/1000], Loss: 289.6502
Epoch [500/1000], Loss: 193.7674
Epoch [600/1000], Loss: 118.6549
Epoch [700/1000], Loss: 64.2747
Epoch [800/1000], Loss: 30.5154
Epoch [900/1000], Loss: 16.6147
Epoch [1000/1000], Loss: 12.7474
PyTorch Linear Regression MAE (Unscaled Data): 3.5756

Training and evaluating on scaled data:
Epoch [100/1000], Loss: 3.7506
Epoch [200/1000], Loss: 3.3337
```

```
Epoch [300/1000], Loss: 2.9432
Epoch [400/1000], Loss: 2.5788
Epoch [500/1000], Loss: 2.2403
Epoch [600/1000], Loss: 1.9275
Epoch [700/1000], Loss: 1.6400
Epoch [800/1000], Loss: 1.3777
Epoch [900/1000], Loss: 1.1400
Epoch [1000/1000], Loss: 0.9269
PyTorch Linear Regression MAE (Scaled Data): 0.6378
```

The PyTorch model using the normal distribution loss is currently underperforming compared to the baseline model, no matter scaling or not. However, scaling does help with better MAE performance.

[59]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error
from torch.distributions import StudentT

# Define the linear regression model
class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)  # Linear model: w^T x + b

    def forward(self, x):
        return self.linear(x)  # f(x) = w^T x + b

# Initialize model, optimizer, and t-distribution parameters
input_dim = X_train.shape[1]

def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight)
        nn.init.zeros_(m.bias)

# Define a custom t-distribution loss function using negative log-likelihood
def t_distribution_loss(y_true, y_pred, nu=5):
    t_dist = StudentT(nu)  # Create the StudentT distribution with nu degrees␣
 ↪of freedom
    residuals = y_true - y_pred  # Calculate residuals
    log_likelihood = t_dist.log_prob(residuals)  # Calculate log probability of␣
 ↪residuals
    return -log_likelihood.mean()  # Return the negative log-likelihood

# Function to train the model with t-distribution loss
```

```
def train_model_t_dist(model, X_train_torch, y_train_torch, num_epochs=1000,␣
 ↪lr=0.001, nu=5):
    optimizer = optim.SGD(model.parameters(), lr=lr)  # Optimizer with learning␣
 ↪rate
    model.apply(initialize_weights)  # Initialize weights
    for epoch in range(num_epochs):
        # Forward pass: compute predictions and loss
        outputs = model(X_train_torch)
        loss = t_distribution_loss(y_train_torch, outputs, nu=nu)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()

        # Gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()

        # Print loss every 100 epochs
        if (epoch + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], T-Loss: {loss.item():.4f}')
    return model

# Function to evaluate the model
def evaluate_model_t_dist(model, X_test_torch, y_test_torch):
    with torch.no_grad():
        y_pred_t = model(X_test_torch)

        # Check for NaN or infinite values
        if torch.isnan(y_pred_t).any() or torch.isinf(y_pred_t).any():
            print("NaN or infinity detected in predictions!")

        mae_t_dist = mean_absolute_error(y_test_torch, y_pred_t)
    return mae_t_dist

# Training and evaluating on unscaled data
print("Training and evaluating on unscaled data:")
model_unscaled = LinearRegressionModel(input_dim)
model_unscaled = train_model_t_dist(model_unscaled, X_train_torch_unscaled,␣
 ↪y_train_torch)
mae_unscaled = evaluate_model_t_dist(model_unscaled, X_test_torch_unscaled,␣
 ↪y_test_torch)
print(f"PyTorch Linear Regression (t-Distribution) MAE (Unscaled Data):␣
 ↪{mae_unscaled:.4f}")

# Training and evaluating on scaled data
```

```
print("\nTraining and evaluating on scaled data:")
model_scaled = LinearRegressionModel(input_dim)
model_scaled = train_model_t_dist(model_scaled, X_train_torch_scaled,␣
 ↪y_train_torch)
mae_scaled = evaluate_model_t_dist(model_scaled, X_test_torch_scaled,␣
 ↪y_test_torch)
print(f"PyTorch Linear Regression (t-Distribution) MAE (Scaled Data):␣
 ↪{mae_scaled:.4f}")
```

```
Training and evaluating on unscaled data:
Epoch [100/1000], T-Loss: 12.1823
Epoch [200/1000], T-Loss: 11.4535
Epoch [300/1000], T-Loss: 10.7043
Epoch [400/1000], T-Loss: 9.8376
Epoch [500/1000], T-Loss: 8.8243
Epoch [600/1000], T-Loss: 7.7079
Epoch [700/1000], T-Loss: 6.4692
Epoch [800/1000], T-Loss: 5.0717
Epoch [900/1000], T-Loss: 4.6699
Epoch [1000/1000], T-Loss: 4.4979
PyTorch Linear Regression (t-Distribution) MAE (Unscaled Data): 5.5522

Training and evaluating on scaled data:
Epoch [100/1000], T-Loss: 1.7748
Epoch [200/1000], T-Loss: 1.6942
Epoch [300/1000], T-Loss: 1.6180
Epoch [400/1000], T-Loss: 1.5465
Epoch [500/1000], T-Loss: 1.4800
Epoch [600/1000], T-Loss: 1.4186
Epoch [700/1000], T-Loss: 1.3624
Epoch [800/1000], T-Loss: 1.3116
Epoch [900/1000], T-Loss: 1.2659
Epoch [1000/1000], T-Loss: 1.2253
PyTorch Linear Regression (t-Distribution) MAE (Scaled Data): 0.6551
```

The baseline linear regression model using MSE provides better predictive performance on this dataset compared to the PyTorch model using the t-distribution loss. While the t-distribution loss helps manage outliers, it may not be necessary for this dataset. The MSE loss seems to work better, likely because the dataset does not have extreme outliers, and MSE is more suited to normally distributed residuals.