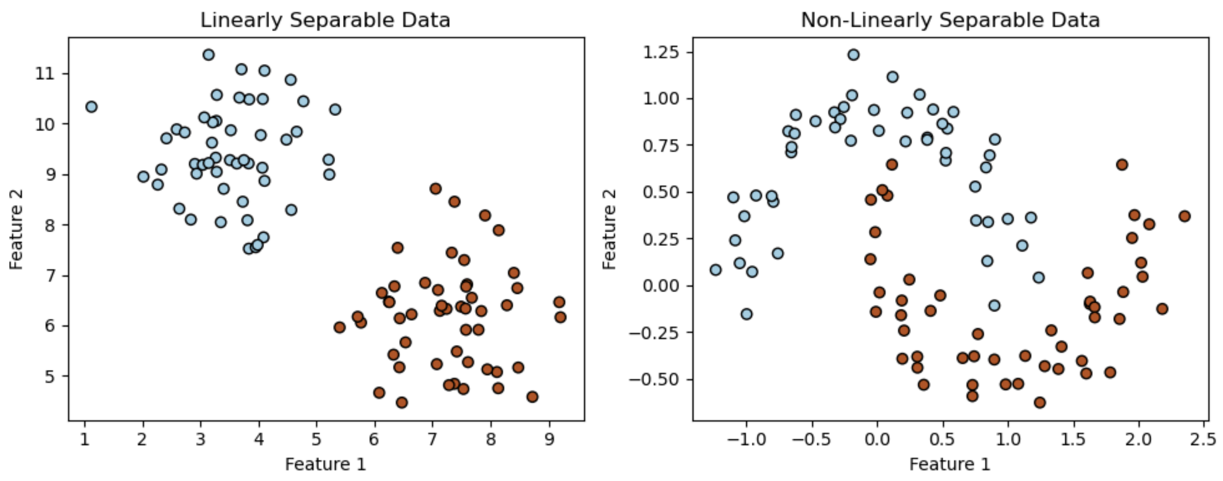
Homework 6

Problem 1

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
```

In [2]:
```python
# Create the data
from sklearn.datasets import make_blobs, make_moons

# Creating a linearly separable dataset
X1, y1 = make_blobs(n_samples=100, centers=2, n_features=2, center_box=(0, 10), random_state=42)
# Creating a non-linearly separable dataset
X2, y2 = make_moons(n_samples=100, noise=0.13, random_state=42)

# Plotting the datasets
plt.figure(figsize=(10, 4))

# Linearly separable dataset
plt.subplot(1, 2, 1)
plt.scatter(X1[:, 0], X1[:, 1], c=y1, cmap='Paired', edgecolor='k')
plt.title('Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Non-linearly separable dataset
plt.subplot(1, 2, 2)
plt.scatter(X2[:, 0], X2[:, 1], c=y2, cmap='Paired', edgecolor='k')
plt.title('Non-Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

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



(a) The starter code `logistic starter.ipynb` defines the two datasets shown below – one is linearly separable and the other is not. Do you expect to correctly classify all samples from the first dataset when using a logistic regression model? Answer the same question for the second dataset

For linearly separable data sets, logistic regression can find a suitable hyperplane to correctly classify all samples in the data set. However, it's impossible to correctly classify all the non-linear separable datasets, since LR cannot find a suitable hyperplane.

(b) Train a logistic regression model on each dataset and report the number of training samples that were misclassified. Assume the standard probability threshold p_{thr} = 0.5. Refer to `sklearn.linear model.LogisticRegression`.

In [3]:
```python
# Part f)
# Train two LogisticRegression models. One on X1, y1 and another on X2, y2.
# Report number of misclassifications.

model1 = LogisticRegression().fit(X1, y1)
print('Model for linear data: intercept =', model1.intercept_, 'coefficients =', model1.coef_)

model2 = LogisticRegression().fit(X2, y2)
print('Model for linear data: intercept =', model2.intercept_, 'coefficients =', model2.coef_)
```

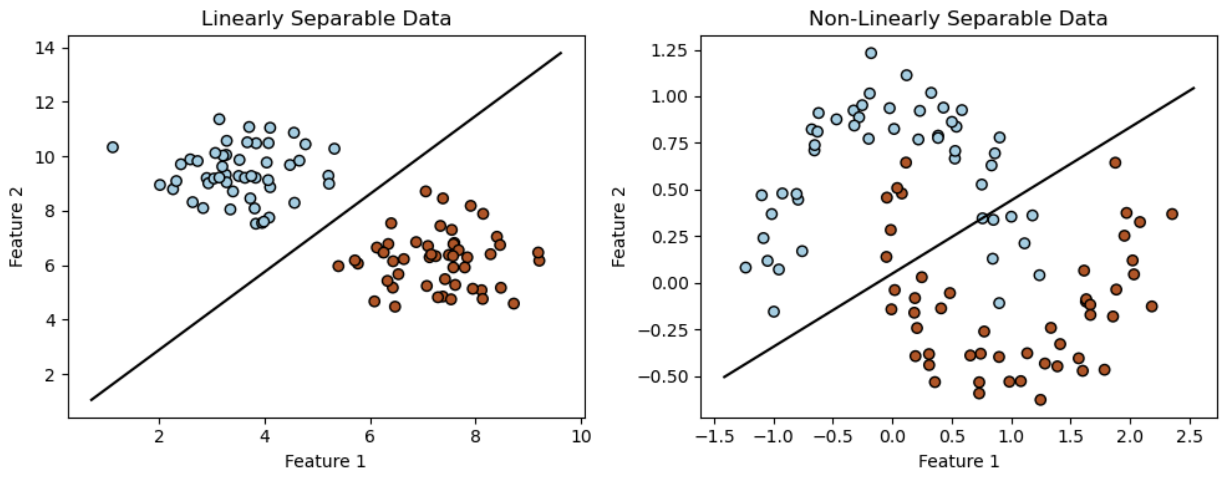
```
Model for linear data: intercept = [0.0192667] coefficients = [[ 1.86293484 -1.29989556]]
Model for linear data: intercept = [0.13961158] coefficients = [[ 1.12303974 -2.86034584]]
```

In [4]:
```
from sklearn.metrics import accuracy_score

n_errors1 = np.floor(100 * (1.0 - accuracy_score(model1.predict(X1), y1)))
print(f'Num errors in linear data:', n_errors1)

n_errors2 = np.floor(100 * (1.0 - accuracy_score(model2.predict(X2), y2)))
print(f'Num errors in non-linear data:', n_errors2)
```

```
Num errors in linear data: 0.0
Num errors in non-linear data: 14.0
```

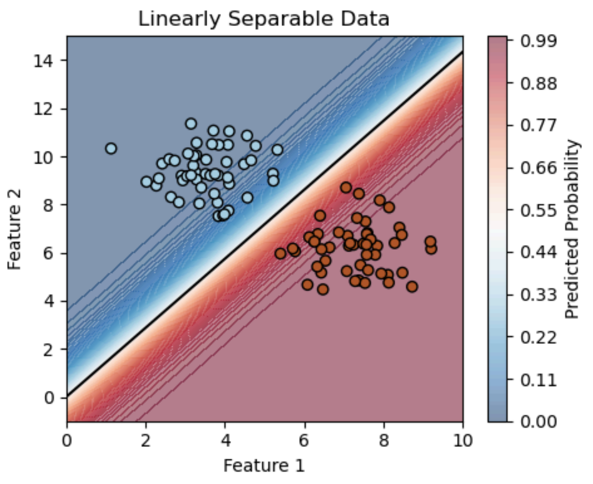
(c) Using your results from part (d) of Activity 14 plot the decision boundaries on top of the scatter plots for the corresponding datasets.

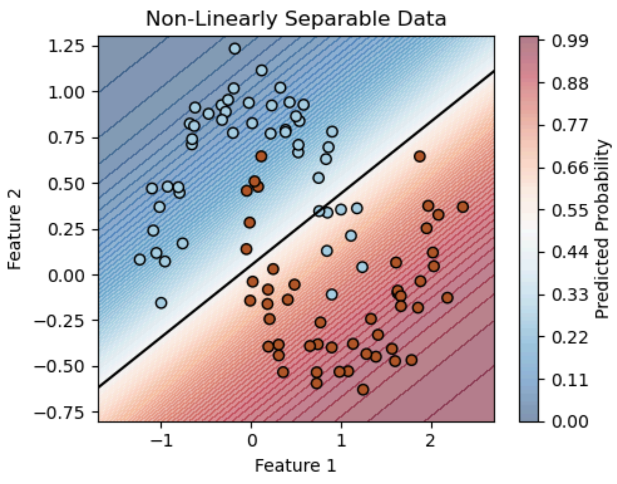
In [5]:
```
# Part g) Plotting decision boundaries
def decision_boundary(w, b, xlim):
    # This assumes 2d feature vectors. w must be a 1x2 matrix.
    assert w.shape == (1, 2)

    # When w_2 is 0, the decision boundary is a vertical line. Ignore this case for simplicity.
    assert w[0, 1] != 0

    x1_boundary = np.linspace(xlim[0], xlim[1], 100)   # Define sequence of x1 values between xlim[0], xlim[1]
    x2_boundary = -(w[0, 0]/w[0, 1])*x1_boundary - b/w[0, 1] + np.log(1)/w[0, 1]  # Decision boundary equation
    return x1_boundary, x2_boundary

plt.figure(figsize=(10, 4))

# Linearly separable dataset
plt.subplot(1, 2, 1)
plt.title('Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.scatter(X1[:, 0], X1[:, 1], c=y1, cmap='Paired', edgecolor='k')

x1_boundary, x2_boundary = decision_boundary(model1.coef_, model1.intercept_, xlim=plt.gca().get_xlim())
plt.plot(x1_boundary, x2_boundary, '-k')

# Non-linearly separable dataset
plt.subplot(1, 2, 2)
plt.title('Non-Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.scatter(X2[:, 0], X2[:, 1], c=y2, cmap='Paired', edgecolor='k')

x_boundary, y_boundary = decision_boundary(model2.coef_, model2.intercept_, xlim=plt.gca().get_xlim())
plt.plot(x_boundary, y_boundary, '-k')

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



(d) The provided starter code overlays the predicted probabilities over the entire feature space as a colormap, with darker blues indicating certain of negative class and darker reds indicating certainty of positive class. How would you expect the predicted probabilities to change on the first dataset, if the positive and negative clusters of samples overlapped with each other.

In [6]:
```python
# Part h) How would you expect the predicted probabilities to change on the first dataset,
#         if the positive and negative clusters of samples overlapped with each other?
#        [Nothing to code.]
def overlay_probabilities(model, xlim, ylim):
    # Create a grid to evaluate the model at every point in the 2D space
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy_grid = np.stack([XX.reshape(-1), YY.reshape(-1)], axis=1)

    probs = model.predict_proba(xy_grid)[:, 1].reshape(YY.shape)
    plt.contourf(XX, YY, probs, levels=100, cmap='RdBu_r', alpha=0.5)
    plt.colorbar(label='Predicted Probability')


plt.figure(figsize=(10, 4))

# Linearly separable dataset
plt.subplot(1, 2, 1)
plt.title('Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
overlay_probabilities(model1, xlim=(0, 10), ylim=(-1, 15))
plt.scatter(X1[:, 0], X1[:, 1], c=y1, cmap='Paired', edgecolor='k')
x_boundary, y_boundary = decision_boundary(model1.coef_, model1.intercept_, xlim=plt.gca().get_xlim())
plt.plot(x_boundary, y_boundary, '-k')

# Non-linearly separable dataset
plt.subplot(1, 2, 2)
plt.title('Non-Linearly Separable Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
overlay_probabilities(model2, xlim=(-1.7, 2.7), ylim=(-0.8, 1.3))
plt.scatter(X2[:, 0], X2[:, 1], c=y2, cmap='Paired', edgecolor='k')
x_boundary, y_boundary = decision_boundary(model2.coef_, model2.intercept_, xlim=plt.gca().get_xlim())
plt.plot(x_boundary, y_boundary, '-k')

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





Predicted probabilities in overlapping regions to be around 0.5, indicating high uncertainty. A greater amount of lighter colors in the colormap, signifying areas where the model is unsure about its predictions. Overall, the visualization will reflect the challenges of the model in making confident classifications due to the inherent ambiguity in the feature space caused by overlapping clusters.

Problem 2

In [2]:
```
import random
import torch
import numpy as np
import matplotlib.pyplot as plt

# for easier reading np
np.set_printoptions(precision=3,suppress=True)
```

In [3]:
```
from sklearn import datasets
iris = datasets.load_iris()
X = torch.tensor(iris.data[:, :3], dtype=torch.float32)  # we only take the first three features.
y = torch.tensor(iris.data[:, 3], dtype=torch.float32)   # we use the fourth feature as the target.

# Uncomment this line for problem 2 (logistic regression)
# X = torch.tensor(iris.data, dtype=torch.float32)
# y = torch.tensor(iris.target == 2, dtype=torch.float32)
```

In [4]:
```
# Partition the data into Training and Testing (80:20 split)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, shuffle=True, random_state=0)

print('X_train', X_train.shape)
print('X_test', X_test.shape)
```

```
X_train torch.Size([120, 3])
X_test torch.Size([30, 3])
```

In [5]:
```
# Reading the dataset
def data_iter(batch_size, features, labels):
    num_examples = len(features)

    # The examples are read at random, in no particular order
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = indices[i:i + batch_size]
        yield features[j], labels[j]

# Check data reader
for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
    print('X_batch', X_batch.shape, X_batch[0])
    print('y_batch', y_batch.shape, y_batch[0])
    break
```

```
X_batch torch.Size([10, 3]) tensor([6.9000, 3.2000, 5.7000])
y_batch torch.Size([10]) tensor(2.3000)
```

In [6]:
```
# Initializing Model Parameters
w = torch.nn.Parameter(data=torch.zeros((3, 1)), requires_grad=True)
torch.nn.init.normal_(w, mean=0, std=0.01)
b = torch.nn.Parameter(data=torch.zeros((1, 1)), requires_grad=True)
print('w', w)
print('b', b)
```

```
w Parameter containing:
tensor([[ 0.0013],
        [-0.0035],
        [-0.0019]], requires_grad=True)
b Parameter containing:
tensor([[0.]], requires_grad=True)
```

In [7]:
```
# Defining the Model
def linreg(X, w, b):
    """The linear regression model."""
    return X@w + b

# Check model
for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
    out_batch = linreg(X_batch, w, b)
    print('X_batch', X_batch.shape, X_batch[0])
    print('out_batch', out_batch.shape, out_batch[0])
    break
```

```
X_batch torch.Size([10, 3]) tensor([6.7000, 3.0000, 5.0000])
out_batch torch.Size([10, 1]) tensor([-0.0117], grad_fn=<SelectBackward0>)
```

In [8]:
```python
# Defining the Loss Function
def squared_loss(y_hat, y):
    """Squared loss."""
    return torch.mean((y_hat - y.view(y_hat.shape))**2 / 2)

# Check loss
err = squared_loss(torch.tensor([1, 2, 3]), torch.tensor([3, 2, 1]))
print('err =', err)
```

```
err = tensor(1.3333)
```

In [9]:
```python
# Defining the Optimization Algorithm
def sgd(params, grads, lr):
    """Minibatch stochastic gradient descent."""
    for p, g in zip(params, grads):
        p.data -= lr * g
```

(a) After creating a copy of the code, modify it to compute both the training and testing R^2 at the end of each epoch. Plot the training R^2 vs epoch and test R^2 vs epoch. Discuss your observations.

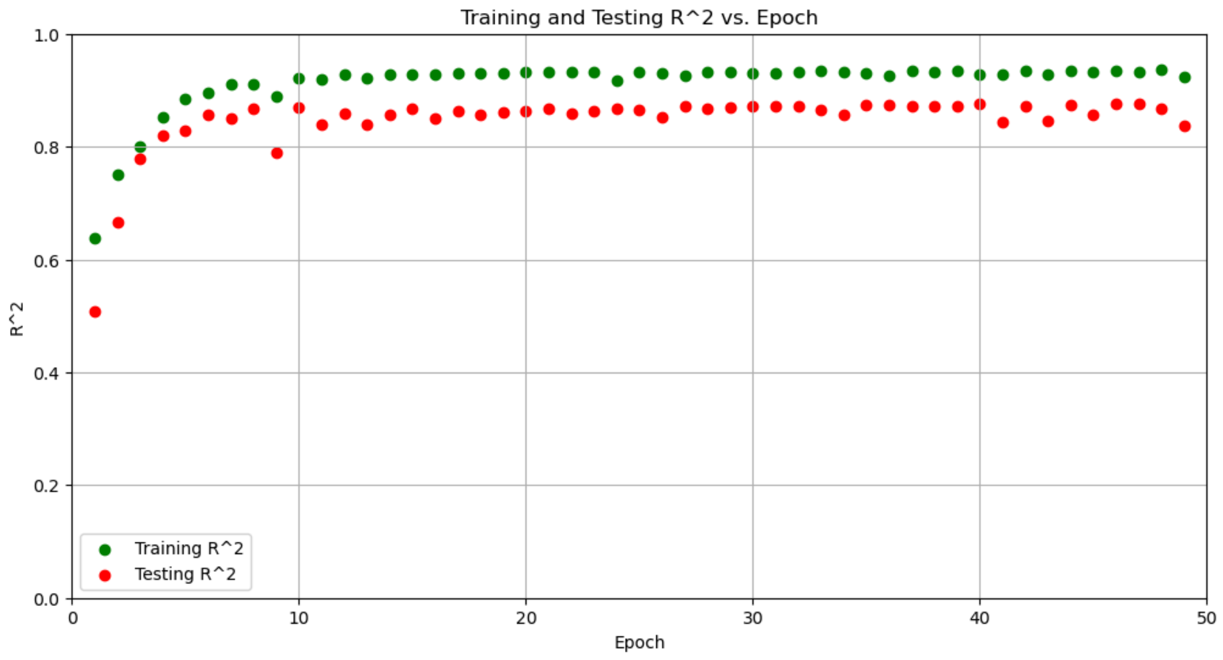
Training

In [10]:
```python
lr = 0.01
batch_size = 10
num_epochs = 50
net = linreg
loss = squared_loss
```

In [11]:
```python
# Initialize the parameters of the model
torch.nn.init.normal_(w, mean=0, std=0.01)
torch.nn.init.zeros_(b)

test_R2 = []
train_R2 = []

for epoch in range(num_epochs):
    # Evaluate model
    with torch.no_grad():
        yhat = net(X_test, w, b)[:, 0]
        mse = torch.sum((y_test - yhat)**2)
        var = torch.sum((y_test - torch.mean(y_test))**2)
        R_sq = 1 - mse / var
        test_R2.append(R_sq)
        test_l = loss(yhat, y_test)
        print(f'epoch {epoch:03d}, test loss {float(test_l):.5f}, Rsquare {R_sq:.3f}')

        yhat_train = net(X_train, w, b)[:, 0]
        mse_train = torch.sum((y_train - yhat_train)**2)
        var_train = torch.sum((y_train - torch.mean(y_train))**2)
        R_sq_train = 1 - mse_train / var_train
        train_R2.append(R_sq_train)
        train_l = loss(yhat_train, y_train)
        print(f'epoch {epoch:03d}, train loss {float(train_l):.5f}, Rsquare {R_sq_train:.3f}')

    # Train for one epoch
    for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
        # Use model to compute predictions
        yhat = net(X_batch, w, b)
        l = loss(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        sgd([w, b], [w.grad, b.grad], lr)

        # Reset gradients
        w.grad = b.grad = None
```

```
epoch 000, test loss 0.83192, Rsquare -2.466
epoch 000, train loss 1.07697, Rsquare -2.613
epoch 001, test loss 0.11819, Rsquare 0.508
epoch 001, train loss 0.10785, Rsquare 0.638
epoch 002, test loss 0.08037, Rsquare 0.665
epoch 002, train loss 0.07408, Rsquare 0.751
epoch 003, test loss 0.05317, Rsquare 0.778
epoch 003, train loss 0.05944, Rsquare 0.801
epoch 004, test loss 0.04339, Rsquare 0.819
epoch 004, train loss 0.04412, Rsquare 0.852
epoch 005, test loss 0.04114, Rsquare 0.829
epoch 005, train loss 0.03441, Rsquare 0.885
epoch 006, test loss 0.03457, Rsquare 0.856
epoch 006, train loss 0.03126, Rsquare 0.895
epoch 007, test loss 0.03618, Rsquare 0.849
epoch 007, train loss 0.02672, Rsquare 0.910
epoch 008, test loss 0.03176, Rsquare 0.868
epoch 008, train loss 0.02640, Rsquare 0.911
epoch 009, test loss 0.05061, Rsquare 0.789
epoch 009, train loss 0.03294, Rsquare 0.890
epoch 010, test loss 0.03130, Rsquare 0.870
epoch 010, train loss 0.02326, Rsquare 0.922
epoch 011, test loss 0.03847, Rsquare 0.840
epoch 011, train loss 0.02393, Rsquare 0.920
epoch 012, test loss 0.03367, Rsquare 0.860
epoch 012, train loss 0.02154, Rsquare 0.928
epoch 013, test loss 0.03866, Rsquare 0.839
epoch 013, train loss 0.02348, Rsquare 0.921
epoch 014, test loss 0.03419, Rsquare 0.858
epoch 014, train loss 0.02121, Rsquare 0.929
epoch 015, test loss 0.03174, Rsquare 0.868
epoch 015, train loss 0.02123, Rsquare 0.929
epoch 016, test loss 0.03594, Rsquare 0.850
epoch 016, train loss 0.02159, Rsquare 0.928
epoch 017, test loss 0.03309, Rsquare 0.862
epoch 017, train loss 0.02063, Rsquare 0.931
epoch 018, test loss 0.03459, Rsquare 0.856
epoch 018, train loss 0.02090, Rsquare 0.930
epoch 019, test loss 0.03348, Rsquare 0.861
epoch 019, train loss 0.02054, Rsquare 0.931
epoch 020, test loss 0.03281, Rsquare 0.863
epoch 020, train loss 0.02040, Rsquare 0.932
epoch 021, test loss 0.03203, Rsquare 0.867
epoch 021, train loss 0.02048, Rsquare 0.931
epoch 022, test loss 0.03368, Rsquare 0.860
epoch 022, train loss 0.02038, Rsquare 0.932
epoch 023, test loss 0.03276, Rsquare 0.864
epoch 023, train loss 0.02021, Rsquare 0.932
epoch 024, test loss 0.03182, Rsquare 0.867
epoch 024, train loss 0.02450, Rsquare 0.918
epoch 025, test loss 0.03213, Rsquare 0.866
epoch 025, train loss 0.02014, Rsquare 0.932
epoch 026, test loss 0.03522, Rsquare 0.853
epoch 026, train loss 0.02079, Rsquare 0.930
epoch 027, test loss 0.03102, Rsquare 0.871
epoch 027, train loss 0.02230, Rsquare 0.925
epoch 028, test loss 0.03190, Rsquare 0.867
epoch 028, train loss 0.01997, Rsquare 0.933
epoch 029, test loss 0.03109, Rsquare 0.870
epoch 029, train loss 0.02050, Rsquare 0.931
epoch 030, test loss 0.03088, Rsquare 0.871
epoch 030, train loss 0.02109, Rsquare 0.929
epoch 031, test loss 0.03089, Rsquare 0.871
epoch 031, train loss 0.02054, Rsquare 0.931
epoch 032, test loss 0.03093, Rsquare 0.871
epoch 032, train loss 0.02025, Rsquare 0.932
epoch 033, test loss 0.03224, Rsquare 0.866
epoch 033, train loss 0.01966, Rsquare 0.934
epoch 034, test loss 0.03465, Rsquare 0.856
epoch 034, train loss 0.02032, Rsquare 0.932
epoch 035, test loss 0.03053, Rsquare 0.873
epoch 035, train loss 0.02053, Rsquare 0.931
epoch 036, test loss 0.03047, Rsquare 0.873
epoch 036, train loss 0.02196, Rsquare 0.926
epoch 037, test loss 0.03099, Rsquare 0.871
epoch 037, train loss 0.01962, Rsquare 0.934
epoch 038, test loss 0.03055, Rsquare 0.873
epoch 038, train loss 0.01991, Rsquare 0.933
epoch 039, test loss 0.03095, Rsquare 0.871
epoch 039, train loss 0.01946, Rsquare 0.935
epoch 040, test loss 0.03001, Rsquare 0.875
epoch 040, train loss 0.02120, Rsquare 0.929
epoch 041, test loss 0.03739, Rsquare 0.844
epoch 041, train loss 0.02173, Rsquare 0.927
epoch 042, test loss 0.03095, Rsquare 0.871
epoch 042, train loss 0.01927, Rsquare 0.935
epoch 043, test loss 0.03701, Rsquare 0.846
epoch 043, train loss 0.02146, Rsquare 0.928
epoch 044, test loss 0.03033, Rsquare 0.874
epoch 044, train loss 0.01938, Rsquare 0.935
epoch 045, test loss 0.03463, Rsquare 0.856
epoch 045, train loss 0.02015, Rsquare 0.932
epoch 046, test loss 0.03001, Rsquare 0.875
epoch 046, train loss 0.01949, Rsquare 0.935
epoch 047, test loss 0.02958, Rsquare 0.877
epoch 047, train loss 0.02013, Rsquare 0.932
epoch 048, test loss 0.03202, Rsquare 0.867
epoch 048, train loss 0.01911, Rsquare 0.936
epoch 049, test loss 0.03906, Rsquare 0.837
epoch 049, train loss 0.02271, Rsquare 0.924
```

In [12]:
```python
plt.figure(figsize=(12, 6))
plt.scatter(range(num_epochs), train_R2, label='Training R^2', color='green')
plt.scatter(range(num_epochs), test_R2, label='Testing R^2', color='red')
plt.xlabel('Epoch')
plt.ylabel('R^2')
plt.title('Training and Testing R^2 vs. Epoch')
plt.xlim(0, 50)
plt.ylim(0, 1.0)
plt.legend()
plt.grid(True)
plt.show()
```

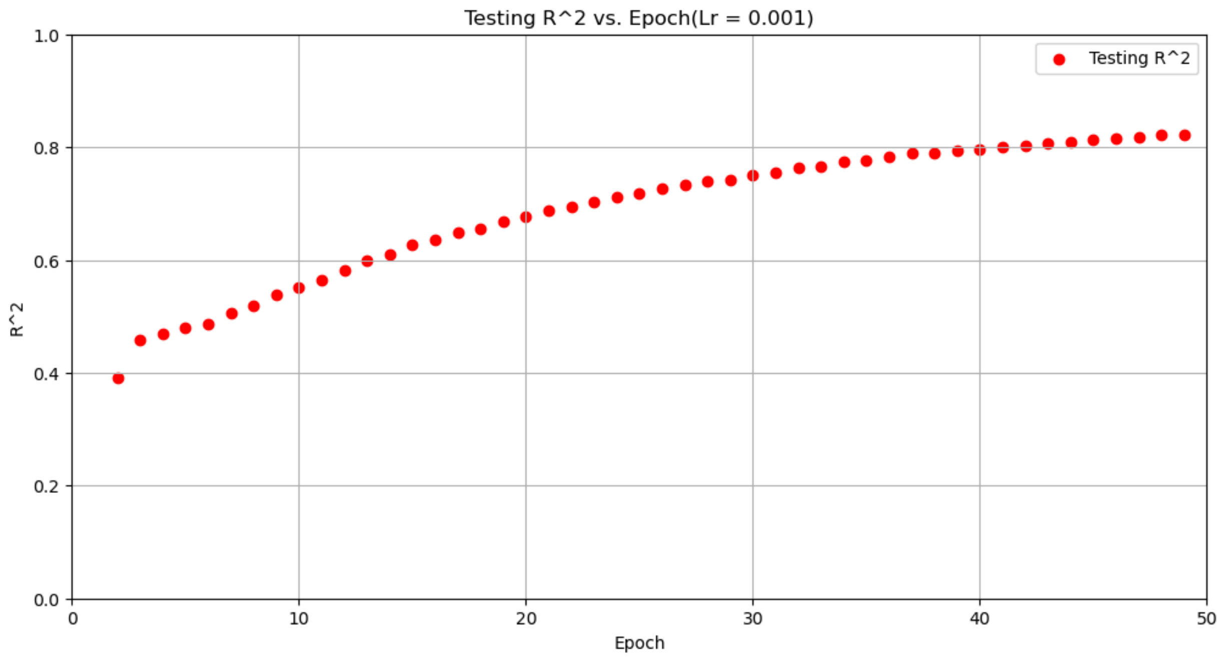


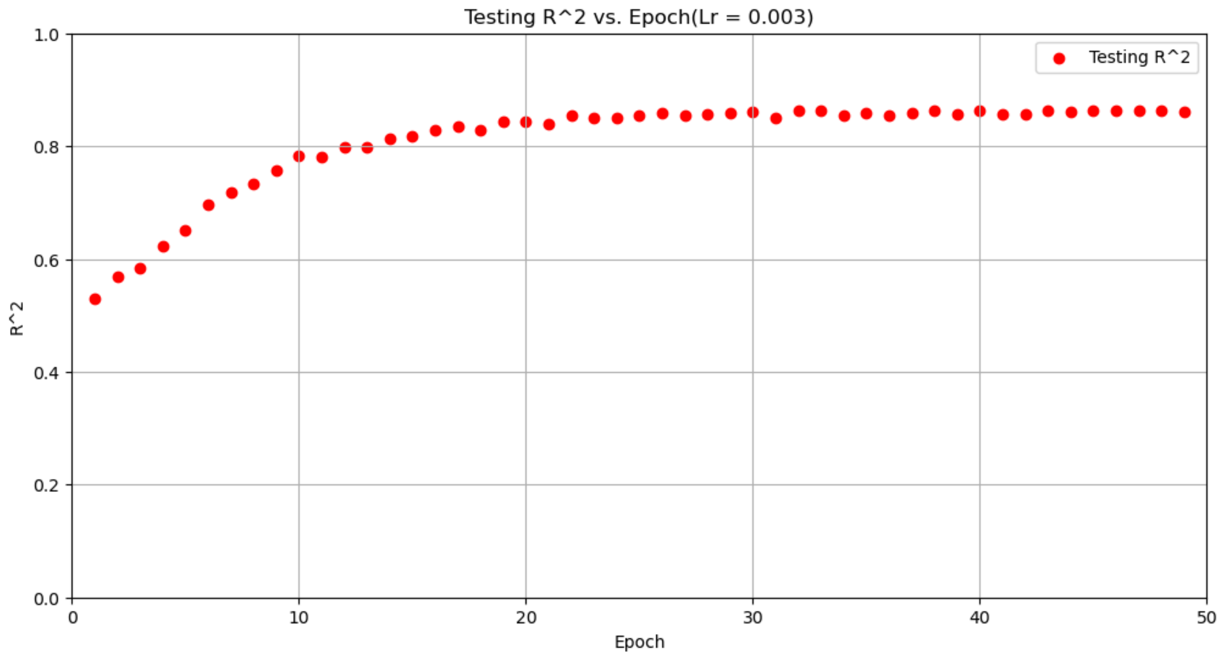
In [13]:
```python
# R2 = 1 - MSE/var(y)
print('Intercept = ', b.detach().numpy())
print('Coefficients = \n', w.detach().numpy())

with torch.no_grad():
    yhat = net(X_train, w, b)[:, 0]
    mse = torch.sum((y_train - yhat)**2)
    var = torch.sum((y_train - torch.mean(y_train))**2)
    R_sq_train = 1 - mse / var

    yhat = net(X_test, w, b)[:, 0]
    mse = torch.sum((y_test - yhat)**2)
    var = torch.sum((y_test - torch.mean(y_test))**2)
    R_sq_test = 1 - mse / var

print('Train R square = ', format(R_sq_train.numpy(),".3f"))
print('Test R square = ', format(R_sq_test.numpy(),".3f"))
```

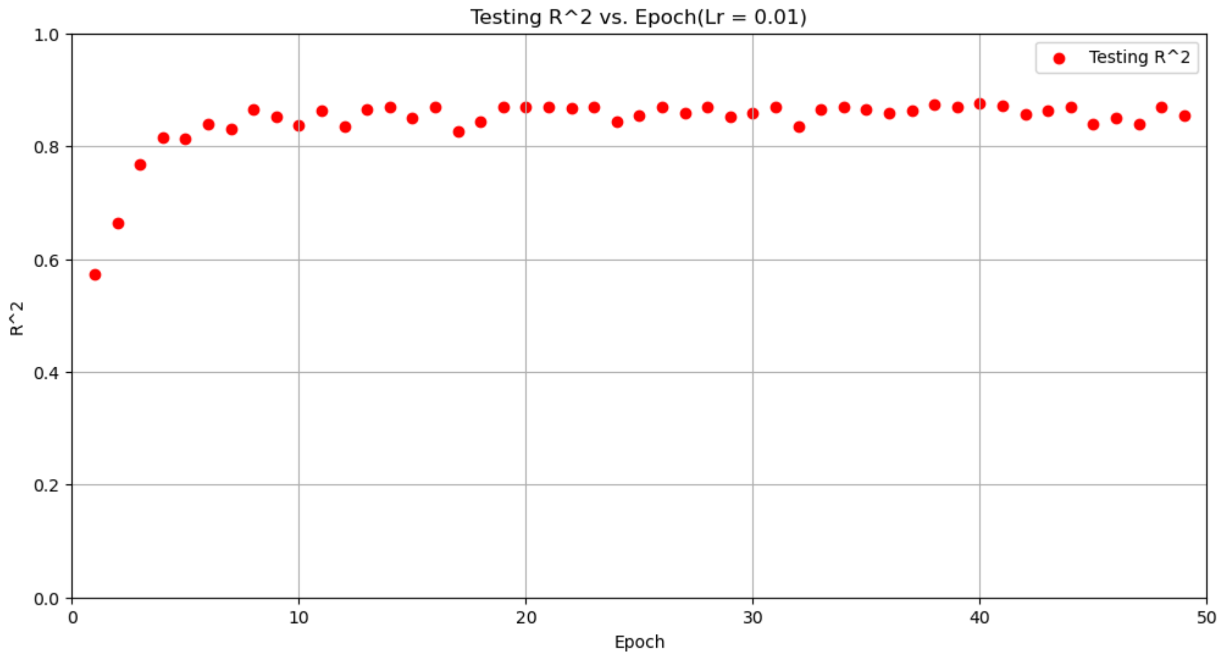
```
Intercept =  [[-0.041]]
Coefficients =
 [[-0.054]
 [-0.032]
 [ 0.429]]
Train R square =  0.931
Test R square =  0.877
```

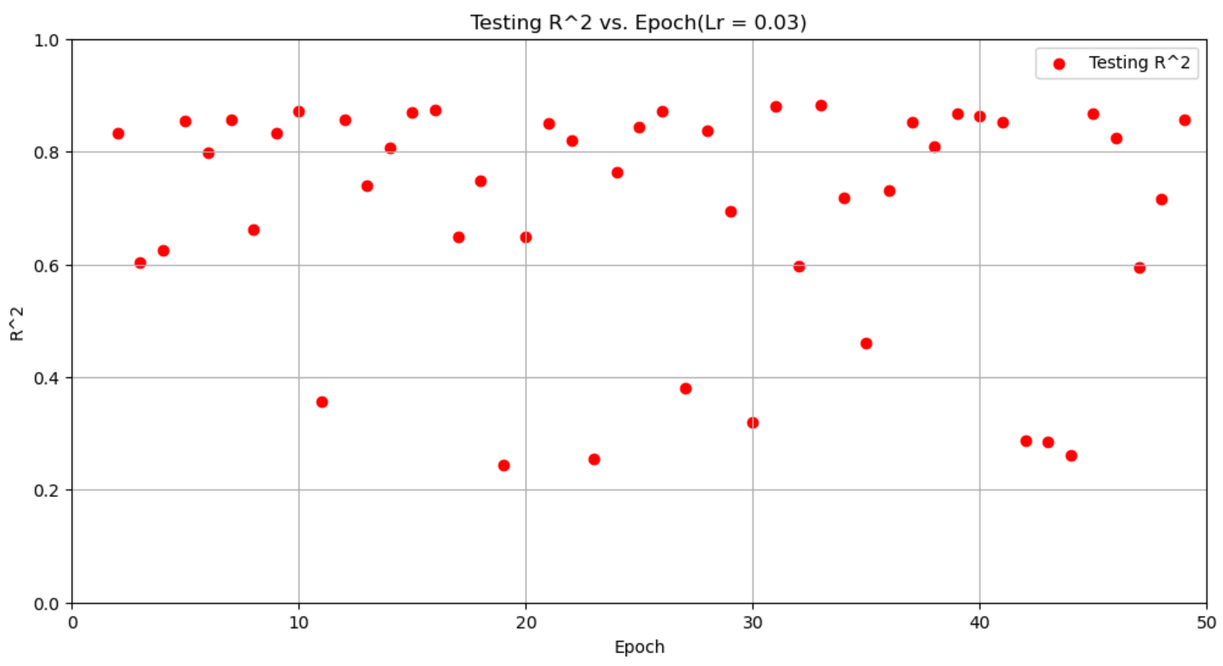
(b) Try different learning rates (use lr=0.03, 0.01, 0.003, 0.001). Plot the test R2 vs epoch for each run. Discuss your observations.

In [14]:
```python
learning_rate = [0.001, 0.003, 0.01, 0.03]

for lr in learning_rate:
    # Initialize the parameters of the model
    torch.nn.init.normal_(w, mean=0, std=0.01)
    torch.nn.init.zeros_(b)

    test_R2 = []
    train_R2 = []

    for epoch in range(num_epochs):
        # Evaluate model
        with torch.no_grad():
            yhat = net(X_test, w, b)[:, 0]
            mse = torch.sum((y_test - yhat)**2)
            var = torch.sum((y_test - torch.mean(y_test))**2)
            R_sq = 1 - mse / var
            test_R2.append(R_sq)
            test_l = loss(yhat, y_test)
            # print(f'epoch {epoch:03d}, test loss {float(test_l):.5f}, Rsquare {R_sq:.3f}')

        # Train for one epoch
        for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
            # Use model to compute predictions
            yhat = net(X_batch, w, b)
            l = loss(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

            # Compute gradients by back propagation
            l.backward()

            # Update parameters using their gradient
            sgd([w, b], [w.grad, b.grad], lr)

            # Reset gradients
            w.grad = b.grad = None

    plt.figure(figsize=(12, 6))
    plt.scatter(range(num_epochs), test_R2, label='Testing R^2', color='red')
    plt.xlabel('Epoch')
    plt.ylabel('R^2')
    plt.title(f'Testing R^2 vs. Epoch(Lr = {lr})')
    plt.xlim(0, 50)
    plt.ylim(0, 1.0)
    plt.legend()
    plt.grid(True)
    plt.show()
```


Testing R^2 vs. Epoch(Lr = 0.001)
Testing R^2
R^2
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50


Testing R^2 vs. Epoch(Lr = 0.003)
Testing R^2
R^2
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

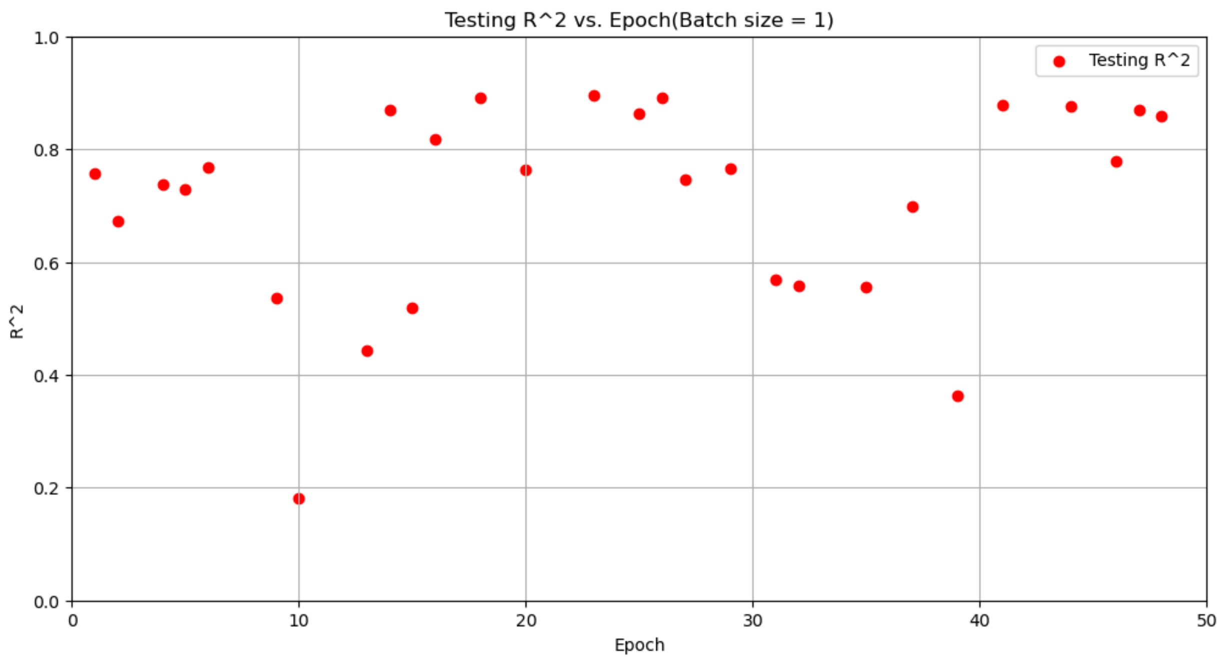

Testing R^2 vs. Epoch(Lr = 0.01)
Testing R^2
R^2
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

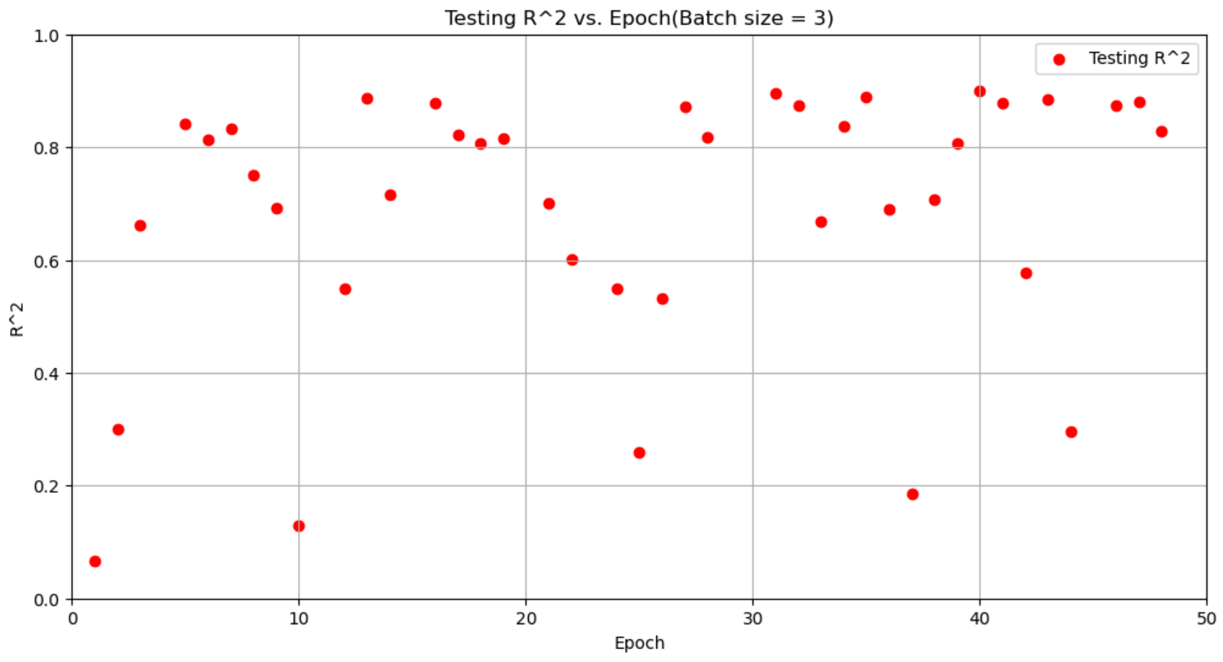


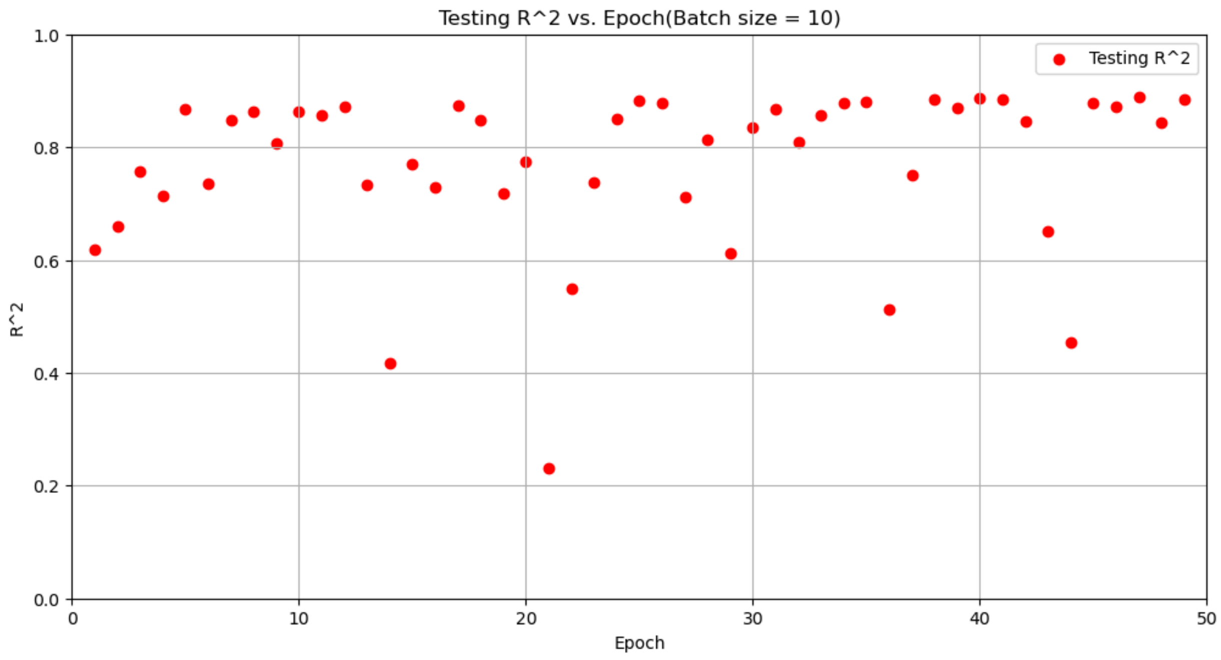
As the learning rate increases, the model can quickly find a better solution, thereby increasing the R2 value. When the learning rate is too high, the model may begin to oscillate around the optimal solution, resulting in performance degradation and a decrease in the R2 value.

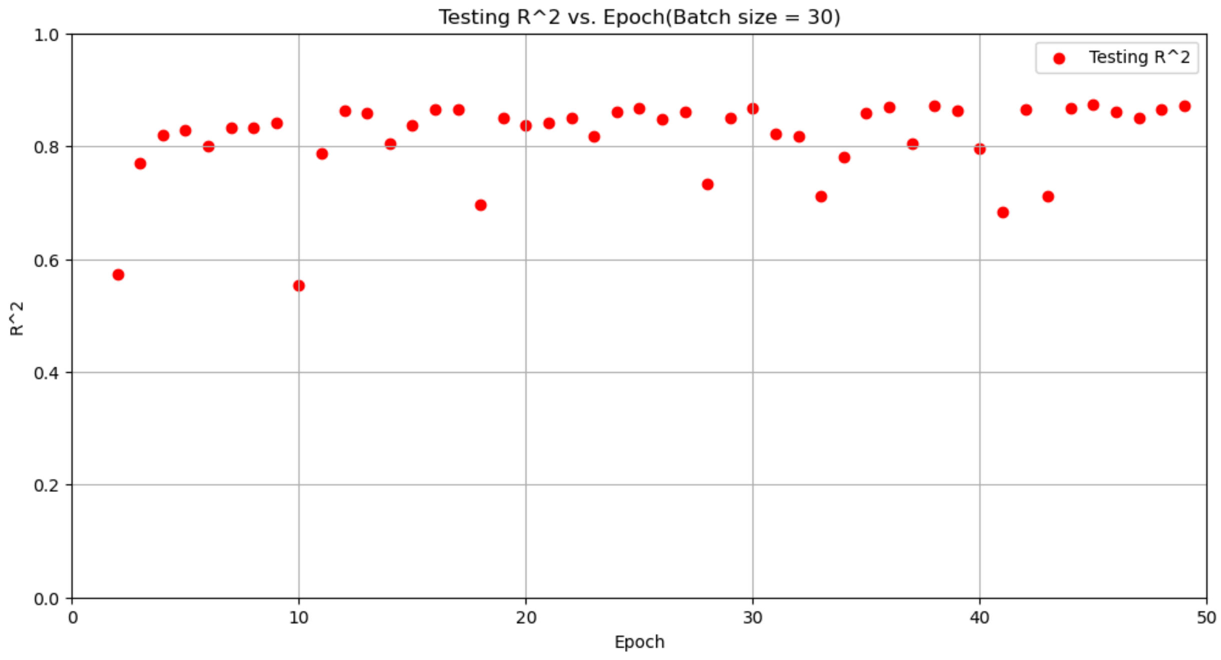
(c) Try different batch sizes (use batch size=1, 3, 10, 30, 100). Plot R2 vs epoch for each run. Discuss your observations.

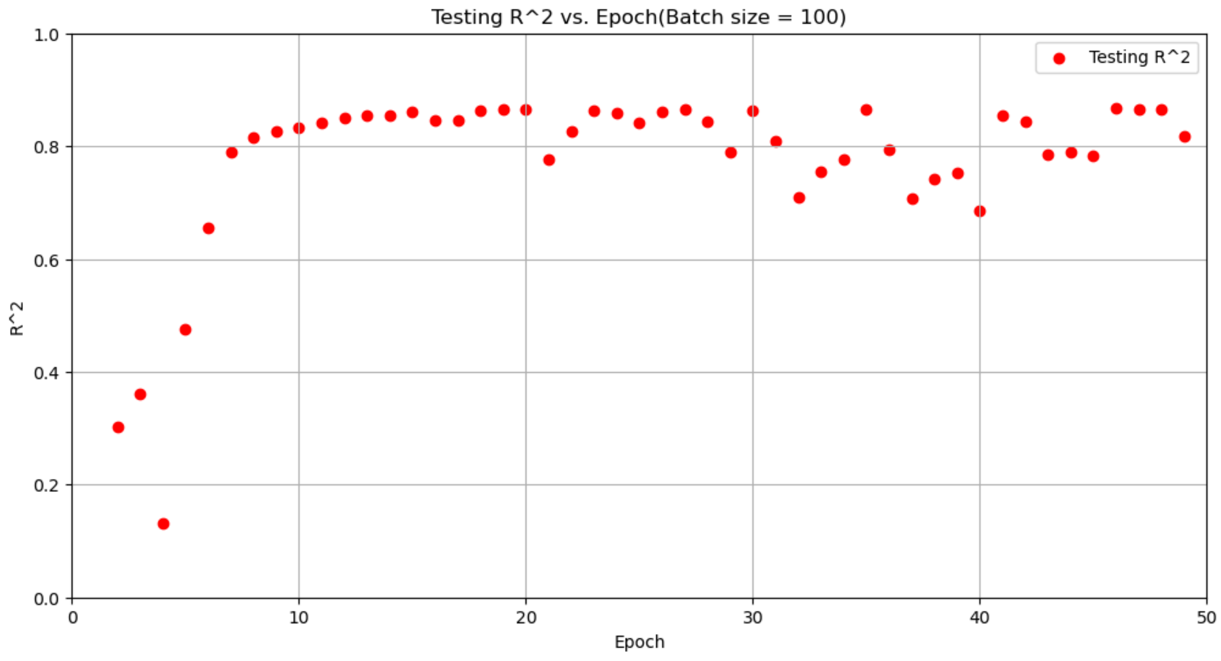
In [15]:
```
batch_sizes = [1, 3, 10, 30, 100]

for batch_size in batch_sizes:
    # Initialize the parameters of the model
    torch.nn.init.normal_(w, mean=0, std=0.01)
    torch.nn.init.zeros_(b)

    test_R2 = []
    train_R2 = []

    for epoch in range(num_epochs):
        # Evaluate model
        with torch.no_grad():
            yhat = net(X_test, w, b)[:, 0]
            mse = torch.sum((y_test - yhat)**2)
            var = torch.sum((y_test - torch.mean(y_test))**2)
            R_sq = 1 - mse / var
            test_R2.append(R_sq)
            test_l = loss(yhat, y_test)
            # print(f'epoch {epoch:03d}, test loss {float(test_l):.5f}, Rsquare {R_sq:.3f}')

        # Train for one epoch
        for X_batch, y_batch in data_iter(batch_size, features=X_train, labels=y_train):
            # Use model to compute predictions
            yhat = net(X_batch, w, b)
            l = loss(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

            # Compute gradients by back propagation
            l.backward()

            # Update parameters using their gradient
            sgd([w, b], [w.grad, b.grad], lr)

            # Reset gradients
            w.grad = b.grad = None

    plt.figure(figsize=(12, 6))
    plt.scatter(range(num_epochs), test_R2, label='Testing R^2', color='red')
    plt.xlabel('Epoch')
    plt.ylabel('R^2')
    plt.title(f'Testing R^2 vs. Epoch(Batch size = {batch_size})')
    plt.xlim(0, 50)
    plt.ylim(0, 1.0)
    plt.legend()
    plt.grid(True)
    plt.show()
```






Testing R^2 vs. Epoch(Batch size = 10)
R^2
Epoch
● Testing R^2
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50


Testing R^2 vs. Epoch(Batch size = 30)
R^2
Epoch
● Testing R^2
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50


Testing R^2 vs. Epoch(Batch size = 100)
R^2
Epoch
● Testing R^2
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

Smaller batch sizes may lead to a noisier model learning process, but may be able to find better solutions when exploring feature space. Larger batch sizes may make the model more likely to fall into local optimal solutions because its update process is smoother and may lack exploration.

Problem 3

In [16]:
```
X = torch.tensor(iris.data, dtype=torch.float32)
y = torch.tensor(iris.target, dtype=torch.float32).reshape(-1, 1)
y = (y == 2).float()

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, shuffle=True, random_state=0)

print('X_train', X_train.shape)
print('X_test', X_test.shape)
```

```
X_train torch.Size([120, 4])
X_test torch.Size([30, 4])
```

In [17]:
```
# Reading the dataset
def data_iter(batch_size, features, labels):
    num_examples = len(features)

    # The examples are read at random, in no particular order
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = indices[i:i + batch_size]
        yield features[j], labels[j]

# Check data reader
for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
    print('X_batch', X_batch.shape, X_batch[0])
    print('y_batch', y_batch.shape, y_batch[0])
    break
```

```
X_batch torch.Size([10, 4]) tensor([5.6000, 3.0000, 4.1000, 1.3000])
y_batch torch.Size([10, 1]) tensor([0.])
```

In [18]:
```
# Initializing Model Parameters
w = torch.nn.Parameter(data=torch.zeros((4, 1)), requires_grad=True)
torch.nn.init.normal_(w, mean=0, std=0.01)
b = torch.nn.Parameter(data=torch.zeros((1, 1)), requires_grad=True)
print('w', w)
print('b', b)
```

```
w Parameter containing:
tensor([[5.7846e-03],
        [2.2764e-05],
        [6.9745e-03],
        [4.9972e-03]], requires_grad=True)
b Parameter containing:
tensor([[0.]], requires_grad=True)
```

In [19]:
```
# Defining the Model
def logireg(X, w, b):
    """The logistic regression model."""
    return torch.sigmoid(X@w + b)

# Check model
for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
    out_batch = logireg(X_batch, w, b)
    print('X_batch', X_batch.shape, X_batch[0])
    print('out_batch', out_batch.shape, out_batch[0])
    break
```

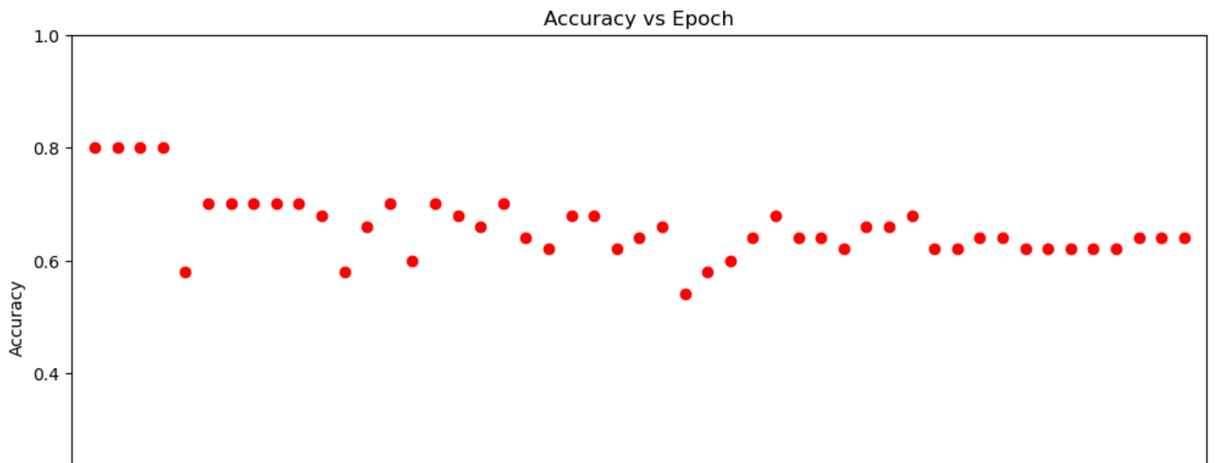
```
X_batch torch.Size([10, 4]) tensor([5.5000, 2.4000, 3.7000, 1.0000])
out_batch torch.Size([10, 1]) tensor([0.5157], grad_fn=<SelectBackward0>)
```

In [20]:
```
# Defining the Loss Function
def crossenty_loss(y_hat, y):
    """Cross-Entropy loss."""
    epsilon = 1e-15
    y_hat = torch.clamp(y_hat, epsilon, 1 - epsilon)
    return -torch.mean(y * torch.log(y_hat) + (1 - y) * torch.log(1 - y_hat))

# Check loss
err = crossenty_loss(torch.tensor([0.2, 0.7]), torch.tensor([0,1]))
print('err =', err)
```

```
err = tensor(0.2899)
```

In [21]:
```
# Defining the Optimization Algorithm
def sgd(params, grads, lr):
    """Minibatch stochastic gradient descent."""
    for p, g in zip(params, grads):
        p.data -= lr * g

lr = 0.01
batch_size = 10
num_epochs = 50
net = logireg
loss = crossenty_loss


# Initialize the parameters of the model
torch.nn.init.normal_(w, mean=0, std=0.01)
torch.nn.init.zeros_(b)
accuracies = []

for epoch in range(num_epochs):
    # Evaluate model
    with torch.no_grad():
        yhat = net(X_test, w, b)[:, 0]
        ypred = (yhat > 0.5).float()
        accuracy = (ypred == y_test).float().mean()
        accuracies.append(accuracy)
        test_l = loss(yhat, y_test)
        print(f'epoch {epoch:03d}, test loss {float(test_l):.5f}, accuracy {float(accuracy):.5f}')

    # Train for one epoch
    for X_batch, y_batch in data_iter(batch_size=10, features=X_train, labels=y_train):
        # Use model to compute predictions
        yhat = net(X_batch, w, b)
        ypred = (yhat > 0.5).float()
        l = loss(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        sgd([w, b], [w.grad, b.grad], lr)

        # Reset gradients
        w.grad = b.grad = None


epochs = range(num_epochs)
plt.figure(figsize=(12, 6))
plt.scatter(epochs, accuracies, label='Accuracy', color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Epoch')
plt.xlim(0, 50)
plt.ylim(0, 1.0)

plt.show()
```

```
epoch 000, test loss 0.71557, accuracy 0.20000
epoch 001, test loss 0.65009, accuracy 0.80000
epoch 002, test loss 0.62615, accuracy 0.80000
epoch 003, test loss 0.60138, accuracy 0.80000
epoch 004, test loss 0.60526, accuracy 0.80000
epoch 005, test loss 0.65187, accuracy 0.58000
epoch 006, test loss 0.61135, accuracy 0.70000
epoch 007, test loss 0.60678, accuracy 0.70000
epoch 008, test loss 0.60919, accuracy 0.70000
epoch 009, test loss 0.61204, accuracy 0.70000
epoch 010, test loss 0.60313, accuracy 0.70000
epoch 011, test loss 0.61860, accuracy 0.68000
epoch 012, test loss 0.65060, accuracy 0.58000
epoch 013, test loss 0.63313, accuracy 0.66000
epoch 014, test loss 0.60638, accuracy 0.70000
epoch 015, test loss 0.64998, accuracy 0.60000
epoch 016, test loss 0.61337, accuracy 0.70000
epoch 017, test loss 0.62932, accuracy 0.68000
epoch 018, test loss 0.64591, accuracy 0.66000
epoch 019, test loss 0.62046, accuracy 0.70000
epoch 020, test loss 0.65600, accuracy 0.64000
epoch 021, test loss 0.66185, accuracy 0.62000
epoch 022, test loss 0.65263, accuracy 0.68000
epoch 023, test loss 0.65514, accuracy 0.68000
epoch 024, test loss 0.67309, accuracy 0.62000
epoch 025, test loss 0.67301, accuracy 0.64000
epoch 026, test loss 0.66939, accuracy 0.66000
epoch 027, test loss 0.71345, accuracy 0.54000
epoch 028, test loss 0.69814, accuracy 0.58000
epoch 029, test loss 0.69706, accuracy 0.60000
epoch 030, test loss 0.68485, accuracy 0.64000
epoch 031, test loss 0.67827, accuracy 0.68000
epoch 032, test loss 0.69161, accuracy 0.64000
epoch 033, test loss 0.70035, accuracy 0.64000
epoch 034, test loss 0.70955, accuracy 0.62000
epoch 035, test loss 0.69603, accuracy 0.66000
epoch 036, test loss 0.69794, accuracy 0.66000
epoch 037, test loss 0.68954, accuracy 0.68000
epoch 038, test loss 0.72270, accuracy 0.62000
epoch 039, test loss 0.72125, accuracy 0.62000
epoch 040, test loss 0.71715, accuracy 0.64000
epoch 041, test loss 0.72433, accuracy 0.64000
epoch 042, test loss 0.72963, accuracy 0.62000
epoch 043, test loss 0.73505, accuracy 0.62000
epoch 044, test loss 0.73682, accuracy 0.62000
epoch 045, test loss 0.74726, accuracy 0.62000
epoch 046, test loss 0.74768, accuracy 0.62000
epoch 047, test loss 0.73926, accuracy 0.64000
epoch 048, test loss 0.74017, accuracy 0.64000
epoch 049, test loss 0.73875, accuracy 0.64000
```


Accuracy vs Epoch
Accuracy
Epochs