Radosław_Kawa_lab1

March 12, 2025

0.1 Linear Regression and Gradient Descent - Radosław Kawa

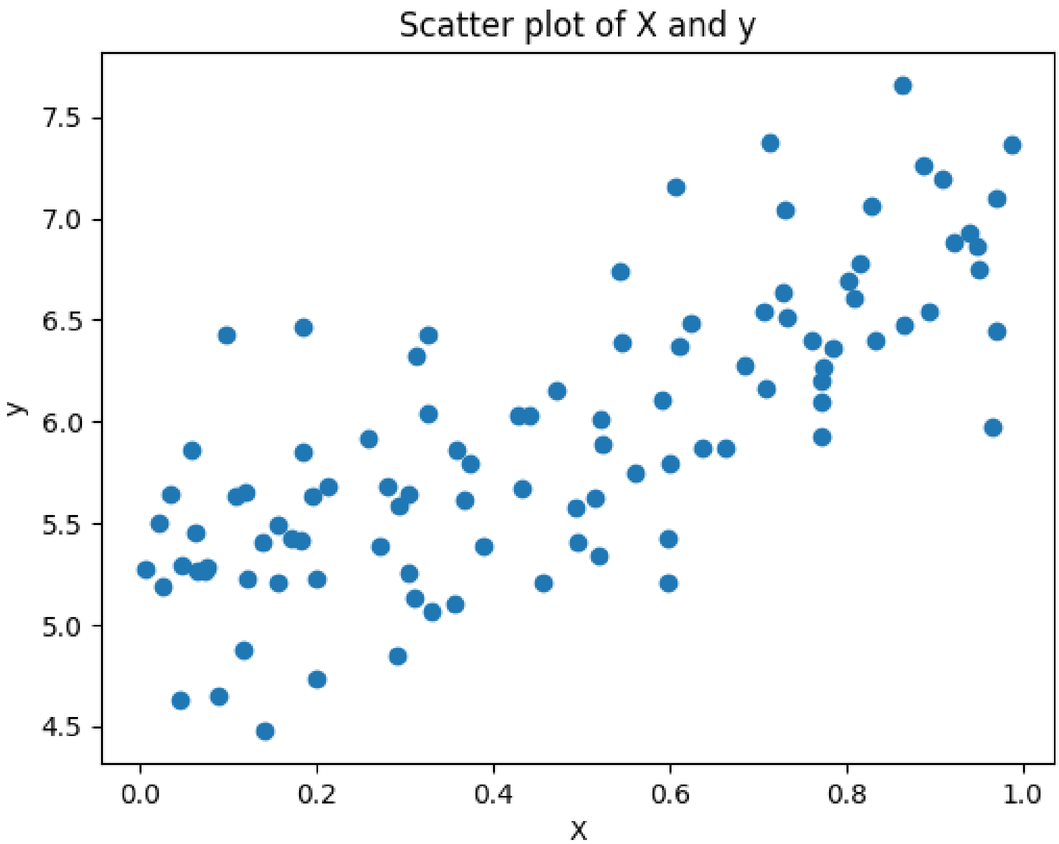
0.1.1 1. Generate Synthetic Data

[236]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

np.random.seed(42)
X = np.random.rand(100)

true_bias = 5
true_weight = 2

y = true_weight * X + true_bias + np.random.randn(100) * 0.5

pd.DataFrame({'X': X, 'y': y}).head()
```

[236]:

	X	y
0	0.374540	5.792604
1	0.950714	6.751925
2	0.731994	6.509868
3	0.598658	5.203533
4	0.156019	5.202201

[237]:
```python
plt.scatter(X, y)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Scatter plot of X and y')
plt.show()
```



0.1.2 2. Prepare the Data

[238]:
```
X_b = np.c_[np.ones(100), X]

pd.DataFrame(X_b).head()
```

[238]:

	0	1
0	1.0	0.374540
1	1.0	0.950714
2	1.0	0.731994
3	1.0	0.598658
4	1.0	0.156019

0.1.3 3. Initialize Parameters

[239]:
```
weights = np.zeros(2)

pd.DataFrame(weights).head()
```

[239]:
```
   0
0  0.0
1  0.0
```

0.1.4 4. Implement Gradient Descent

[240]:
```
def compute_gradient(X, y, weights):
    m = len(y)
    predictions = X.dot(weights)
    errors = predictions - y
    gradients = 2/m * X.T.dot(errors)
    return gradients

def gradient_descent(X, y, weights, learning_rate, num_epochs):
    loss_history = []
    for epoch in range(num_epochs):
        gradients = compute_gradient(X, y, weights)
        weights -= learning_rate * gradients
        loss = np.mean(np.square(X.dot(weights) - y))
        loss_history.append(loss)
    return weights, loss_history
```

0.1.5 5. Run the Training Loop

[241]:
```
learning_rate = 0.01
num_epochs = 1000

weights = np.zeros(2)

final_weights_base, loss_history_base = gradient_descent(X_b, y, weights,␣
 ↪learning_rate, num_epochs)
```

0.1.6 6. Check Your Results

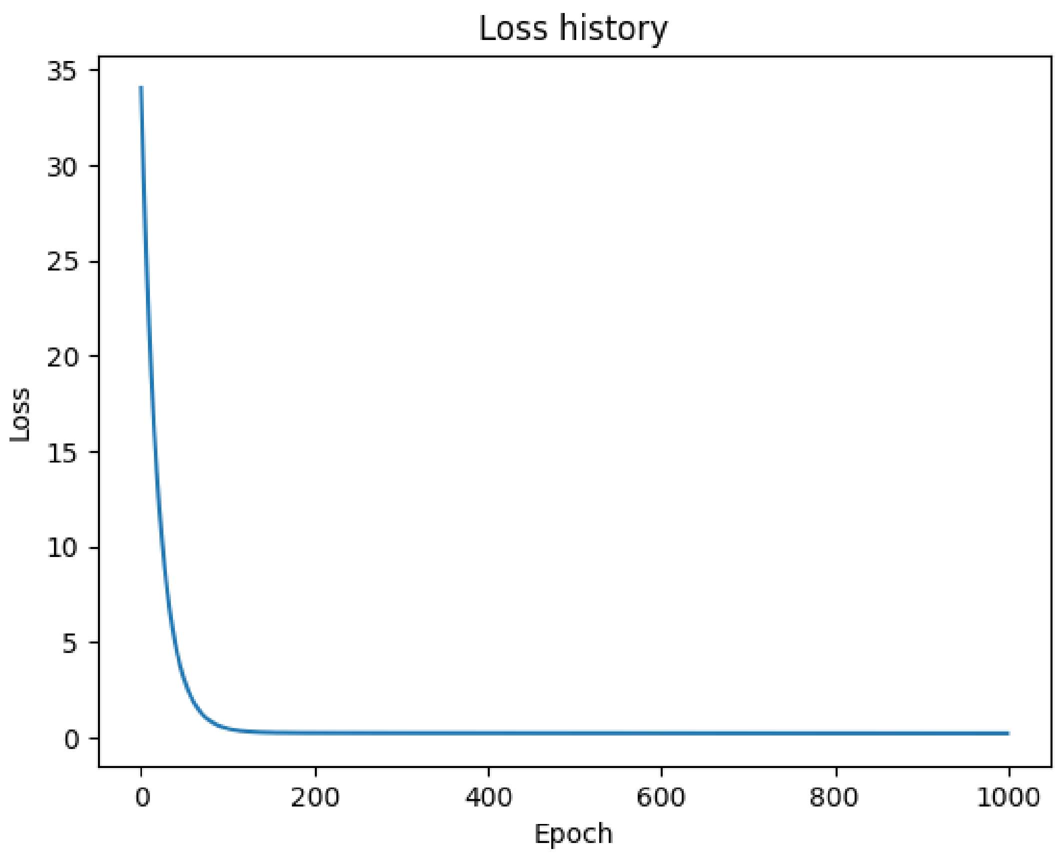
[242]:
```
pd.DataFrame({
    'Bias': [true_bias, final_weights_base[0]],
    'Weight': [true_weight, final_weights_base[1]]
}, index=['True', 'Predicted'])
```

[242]:

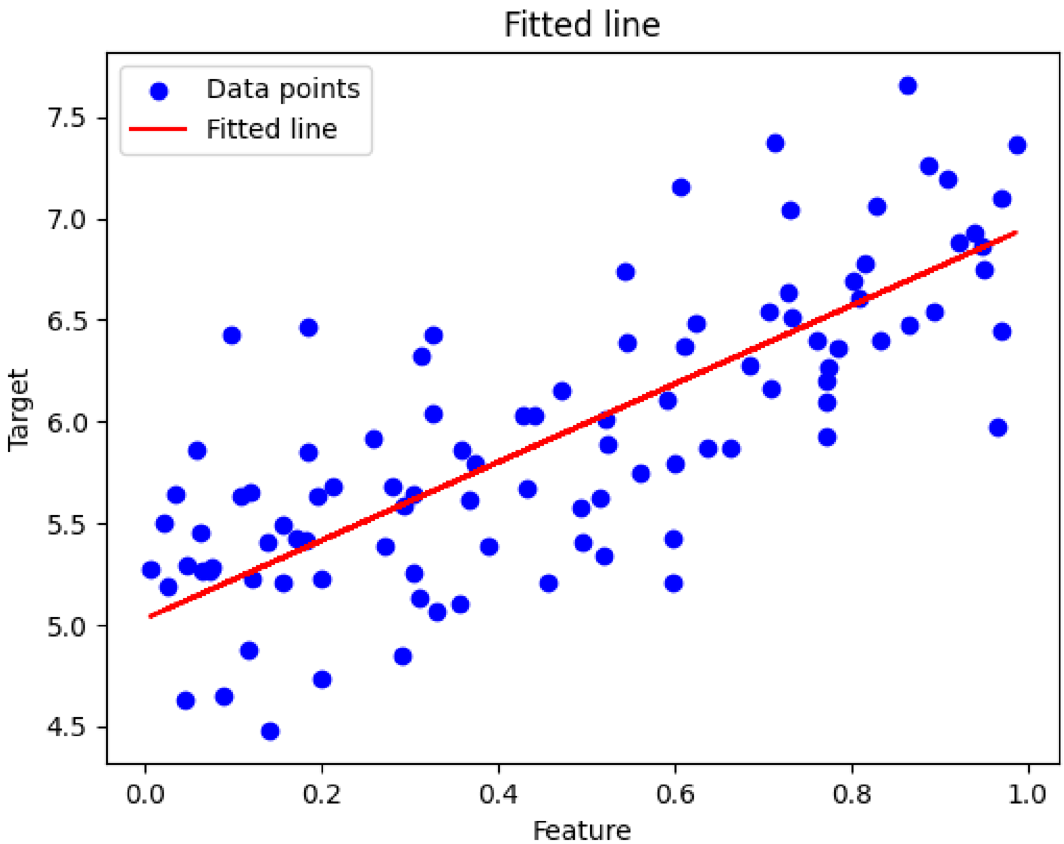
	Bias	Weight
True	5.000000	2.000000
Predicted	5.027983	1.927358

[243]:
```
plt.plot(loss_history_base)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss history')
```

```
plt.show()
```



[244]:
```
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, X_b.dot(final_weights_base), color='red', label='Fitted line')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('Fitted line')
plt.legend()
plt.show()
```



0.1.7 7. Experiment with Learning Rate

[245]:
```
num_epochs = 1000
learning_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
results = {}
losses = {}

for lr in learning_rates:
    weights = np.zeros(2)
    final_weights, loss_history = gradient_descent(X_b, y, weights, lr,␣
 ↪num_epochs)
    results[str(lr)] = final_weights
    losses[str(lr)] = loss_history

results_df = pd.DataFrame(results).T
results_df.columns = ['Bias', 'Weight']
results_df
```

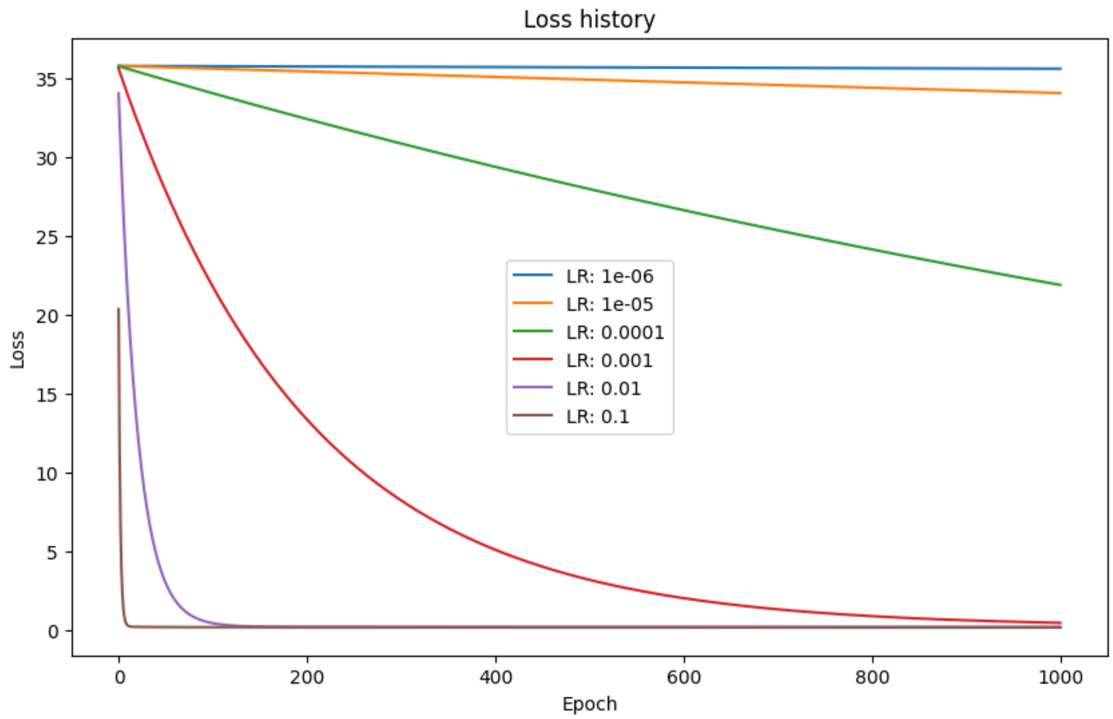
```
# When the learning rate is too small, the model will take a long time to␣
↪converge.
# When the learning rate is too large, the model may overshoot the minimum and␣
↪diverge.
# General optimal learning rate is around 0.01.
```

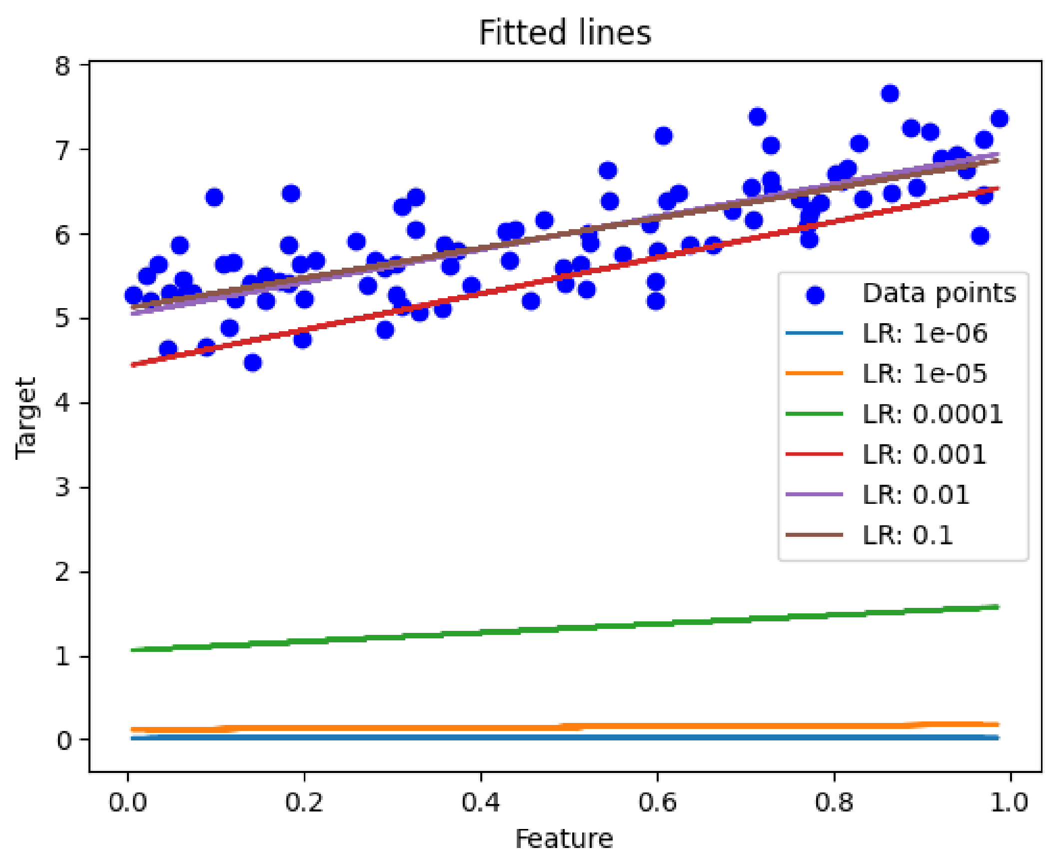
[245]:

	Bias	Weight
1e-06	0.011865	0.005888
1e-05	0.117345	0.058224
0.0001	1.052938	0.521334
0.001	4.422119	2.130079
0.01	5.027983	1.927358
0.1	5.107548	1.770114

[246]:

```
plt.figure(figsize=(10, 6))
for lr, loss_history in losses.items():
    plt.plot(loss_history, label=f'LR: {lr}')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss history')
plt.legend()
plt.show()

plt.scatter(X, y, color='blue', label='Data points')
for lr, weights in results.items():
    plt.plot(X, X_b.dot(weights), label=f'LR: {lr}')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('Fitted lines')
plt.legend()
plt.show()
```


Loss history
Loss
Epoch
LR: 1e-06
LR: 1e-05
LR: 0.0001
LR: 0.001
LR: 0.01
LR: 0.1


Fitted lines
Target
Feature
Data points
LR: 1e-06
LR: 1e-05
LR: 0.0001
LR: 0.001
LR: 0.01
LR: 0.1

0.1.8 8. Explore Training Duration

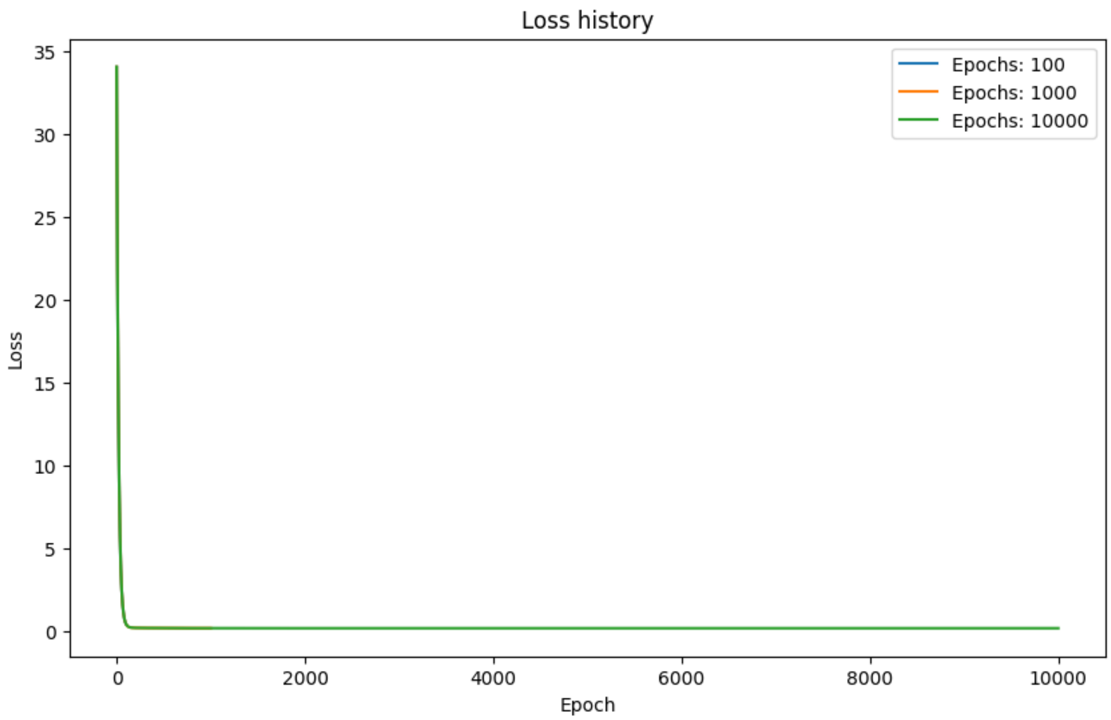
[247]:
```
num_epochs_list = [100, 1000, 10000]
results = {}
losses = {}

for epochs in num_epochs_list:
    weights = np.zeros(2)
    final_weights, loss_history = gradient_descent(X_b, y, weights,␣
 ↪learning_rate, epochs)
    results[epochs] = final_weights
    losses[epochs] = loss_history

results_df = pd.DataFrame(results).T
results_df.columns = ['Bias', 'Weight']
results_df.index.name = 'Epochs'
results_df['Bias_diff'] = results_df['Bias'] - true_bias
results_df['Weight_diff'] = results_df['Weight'] - true_weight
results_df

# After some point loss is not decreasing much and the weights are not changing␣
 ↪much
# There is also possibility of overfitting if we train the model for too long
# Around 1000 epochs is a good number of epochs to train the model
# Additionally as we can see for 10000 epochs, predicted weight are happening␣
 ↪to be a bit off
```

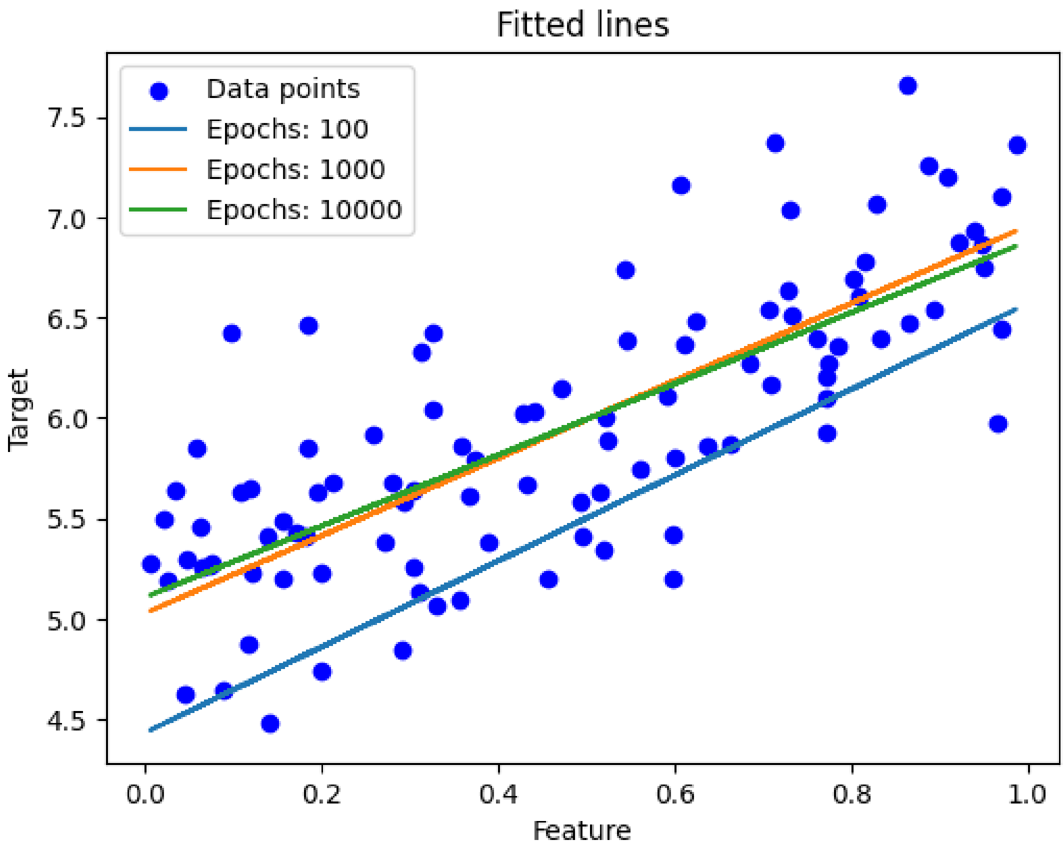
[247]:

Epochs	Bias	Weight	Bias_diff	Weight_diff
100	4.433246	2.135645	-0.566754	0.135645
1000	5.027983	1.927358	0.027983	-0.072642
10000	5.107548	1.770114	0.107548	-0.229886

[248]:
```
# Each plot will look very similar to the previous one
plt.figure(figsize=(10, 6))
for epochs, loss_history in losses.items():
    plt.plot(loss_history, label=f'Epochs: {epochs}')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss history')
plt.legend()
plt.show()

plt.scatter(X, y, color='blue', label='Data points')
```

```
for epochs, weights in results.items():
    plt.plot(X, X_b.dot(weights), label=f'Epochs: {epochs}')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('Fitted lines')
plt.legend()
plt.show()
```

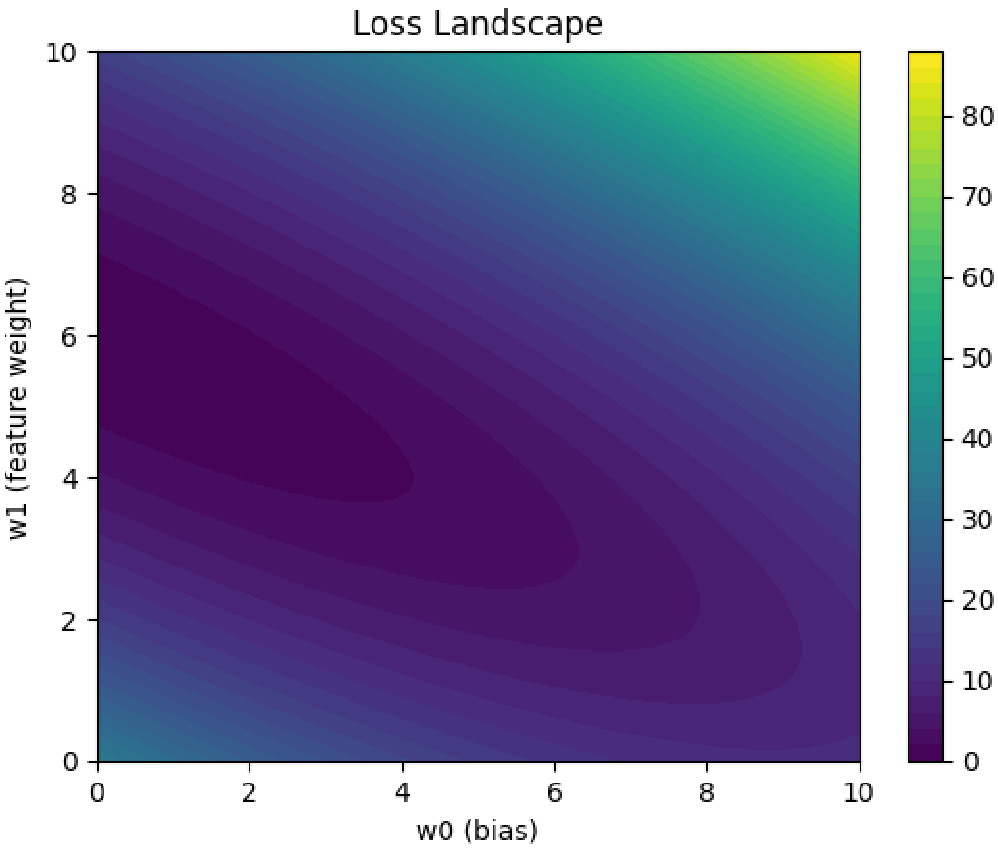




0.1.9 9. Visualize the Loss Landscape

[249]:
```
import matplotlib.pyplot as plt

w0_values = np.linspace(0, 10, 100)
w1_values = np.linspace(0, 10, 100)
loss_values = np.zeros((100, 100))

for i, w0 in enumerate(w0_values):
    for j, w1 in enumerate(w1_values):
        weights = np.array([[w0], [w1]])
        predictions = X_b.dot(weights)
        loss = np.mean((predictions - y) ** 2)
        loss_values[i, j] = loss

W0, W1 = np.meshgrid(w0_values, w1_values)
plt.contourf(W0, W1, loss_values, levels=50, cmap='viridis')
plt.colorbar()
plt.xlabel('w0 (bias)')
```

```
plt.ylabel('w1 (feature weight)')
plt.title('Loss Landscape')
plt.show()
```



0.1.10 10. Analytical Solution

It seems the more epochs we run the closer we get to the analytical solution

For this visualization i chose 1000 epochs but with 10000 epochs there is no difference between the two solutions

[250]:
```
X_b_T_X_b_inv = np.linalg.inv(X_b.T.dot(X_b))
analytical_weights = X_b_T_X_b_inv.dot(X_b.T).dot(y)

pd.DataFrame({
    'True': [true_bias, true_weight],
    'Gradient Descent': final_weights_base,
    'Analytical': analytical_weights.flatten()
}, index=['Bias', 'Weight'])
```

[250]:

	True	Gradient Descent	Analytical
Bias	5	5.027983	5.107548
Weight	2	1.927358	1.770113

Here is the comparison with more epochs - 10000

[251]:

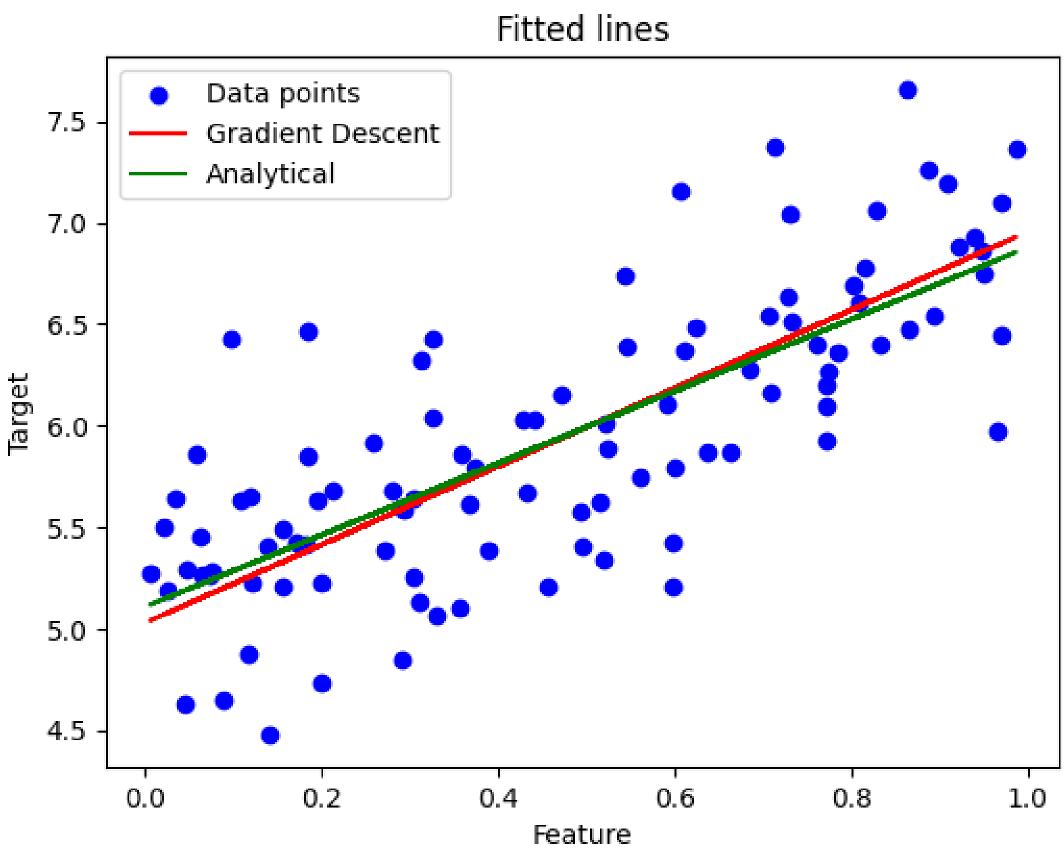
```
pd.DataFrame({
    'True': [true_bias, true_weight],
    'Gradient Descent': final_weights,
    'Analytical': analytical_weights.flatten()
}, index=['Bias', 'Weight'])
```

[251]:

	True	Gradient Descent	Analytical
Bias	5	5.107548	5.107548
Weight	2	1.770114	1.770113

[252]:

```
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, X_b.dot(final_weights_base), color='red', label='Gradient Descent')
plt.plot(X, X_b.dot(analytical_weights), color='green', label='Analytical')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('Fitted lines')
plt.legend()
plt.show()
```



0.1.11 11. High Dimensional Example

[286]:
```
def generate_synthetic_data(n_samples, n_features, true_weights, true_bias,␣
↪noise_std=0.5):
    X = np.random.rand(n_samples, n_features)
    y = X.dot(true_weights) + true_bias + np.random.randn(n_samples) * noise_std
    return X, y

def prepare_data(X):
    return np.c_[np.ones(X.shape[0]), X]

def compute_gradient(X, y, weights):
    m = len(y)
    predictions = X.dot(weights)
    errors = predictions - y
    gradients = 2/m * X.T.dot(errors)
    return gradients

def gradient_descent(X, y, weights, learning_rate, num_epochs):
```

```
    loss_history = []
    for epoch in range(num_epochs):
        gradients = compute_gradient(X, y, weights)
        weights -= learning_rate * gradients
        loss = np.mean((X.dot(weights) - y) ** 2)
        loss_history.append(loss)
    return weights, loss_history
```

[290]:
```
np.random.seed(42)
n_samples = 100
n_features = 10
true_weights = np.random.rand(n_features)
true_bias = 5

X_high_dim, y_high_dim = generate_synthetic_data(n_samples, n_features,␣
 ↪true_weights, true_bias)

X_b_high_dim = prepare_data(X_high_dim)

weights_high_dim = np.zeros(n_features + 1)

learning_rate = 0.1 # Had to increase the learning rate for high-dimensional␣
 ↪data
num_epochs = 1000 # It could be increased too instead of learning rate

final_weights_high_dim, loss_history = gradient_descent(X_b_high_dim,␣
 ↪y_high_dim, weights_high_dim, learning_rate, num_epochs)

weights_df = pd.DataFrame({
    'True Weights': np.append(true_bias, true_weights),
    'Final Weights': final_weights_high_dim
}, index=['Bias'] + [f'Feature_{i}' for i in range(n_features)])

weights_df

# And generally you need more epochs to train the model on high-dimensional␣
 ↪data or higher learning rate
# Higher learning rate in multiple dimensions seems to be better choice than␣
 ↪increasing the number of epochs
# We are achieving similar results with less epochs and higher learning rate␣
 ↪and gaining in performance in the process
```

[290]:

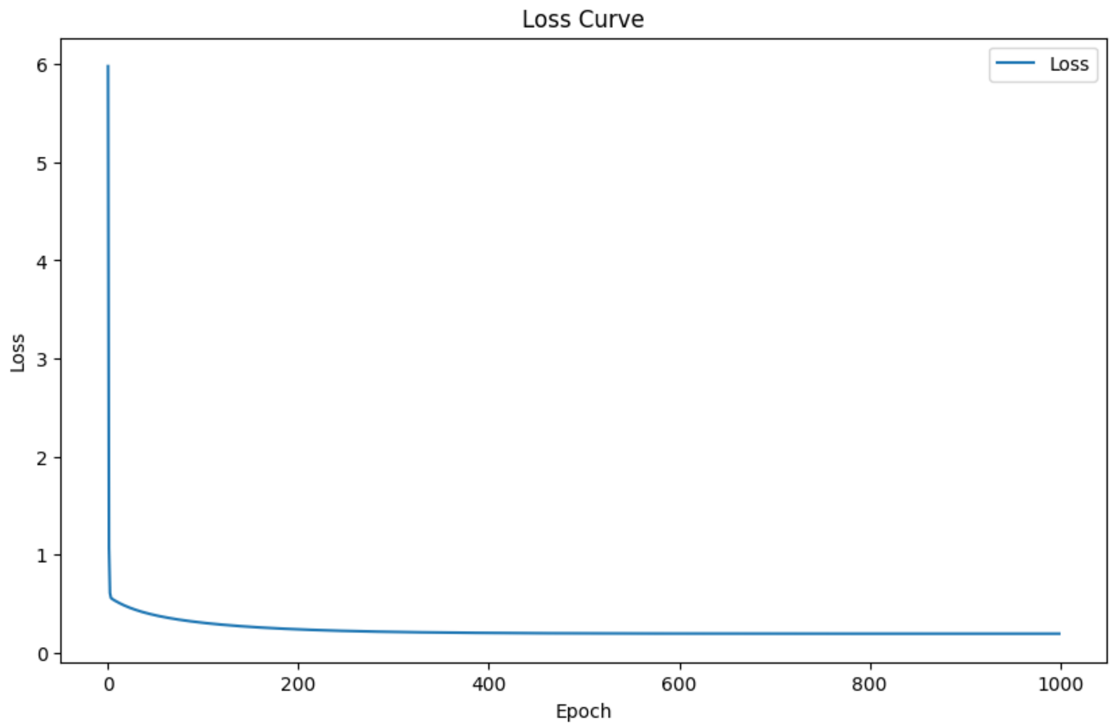
	True Weights	Final Weights
Bias	5.000000	5.096102
Feature_0	0.374540	0.213619
Feature_1	0.950714	0.693418
Feature_2	0.731994	0.942793

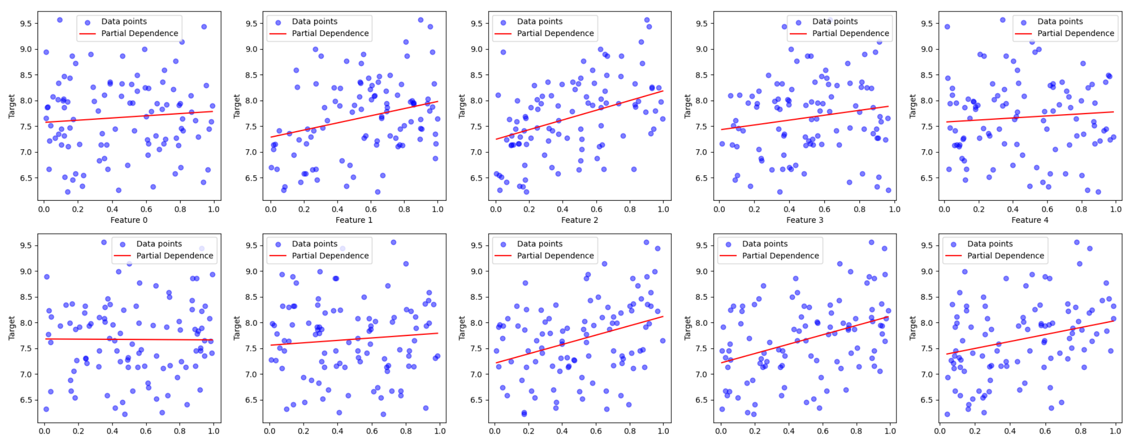
```
Feature_3      0.598658      0.474064
Feature_4      0.156019      0.199986
Feature_5      0.155995     -0.017582
Feature_6      0.058084      0.235840
Feature_7      0.866176      0.909529
Feature_8      0.601115      0.907433
Feature_9      0.708073      0.675823
```

[288]:
```
# Final loss:
final_loss = loss_history[-1]
final_loss
```

[288]: 0.1917456261375808

[289]:
```
plt.figure(figsize=(10, 6))
plt.plot(loss_history, label='Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.legend()
plt.show()

fig, axs = plt.subplots(2, 5, figsize=(20, 8))
axs = axs.flatten()
for i in range(n_features):
    x_range = np.linspace(X_high_dim[:, i].min(), X_high_dim[:, i].max(), 100)
    X_temp = np.mean(X_b_high_dim, axis=0)
    X_temp = np.tile(X_temp, (100, 1))
    X_temp[:, i + 1] = x_range

    y_pred = X_temp.dot(final_weights_high_dim)

    axs[i].scatter(X_high_dim[:, i], y_high_dim, color='blue', alpha=0.5,␣
↪label='Data points')
    axs[i].plot(x_range, y_pred, color='red', label='Partial Dependence')
    axs[i].set_xlabel(f'Feature {i}')
    axs[i].set_ylabel('Target')
    axs[i].legend()
plt.tight_layout()
plt.show()
```


Loss Curve
Loss
Epoch
Loss


Data points
Partial Dependence
Target
Feature 0
Feature 1
Feature 2
Feature 3
Feature 4
Feature 5
Feature 6
Feature 7
Feature 8
Feature 9