

Assignment 1 Part 1: Regression

Name : Pranit Gautam Kamble

StudentID :240676214

Q5. What conclusion if any can be drawn from the weight values? How does sex and BMI affect diabetes disease progression?

What are the estimated disease progression values for the below examples?

Answer:

Here we have done the analysis of Linear Regression model's predictions for diabetes disease, we have got multiple weights and that's the factor, Features such as sex, weight, will help the disease progression.

Code:

```
In [14]: # Firstly i need to define normalized examples (here assuming mu and sigma are already defined)
example_1 = ((torch.tensor([25, 0, 18, 79, 130, 64.8, 61, 2, 4.1897, 68]) - mu[:10]) / sigma[:10]).float()
example_2 = ((torch.tensor([50, 1, 28, 103, 229, 162.2, 60, 4.5, 6.107, 124]) - mu[:10]) / sigma[:10]).float()

# Now we need to add bias term and reshape
example_1, example_2 = torch.cat((example_1, torch.tensor([1.0]))).view(1, -1), torch.cat((example_2, torch.tensor([1.0]))).view(1, -1)

# Predict with model
prediction_1 = model(example_1)
prediction_2 = model(example_2)

print(f"Predicted disease for example 1: {prediction_1.item()}")
print(f"Predicted disease for example 2: {prediction_2.item()}")

Predicted disease for example 1: -13.86233901977539
Predicted disease for example 2: 10.436138153076172
```

Explanation:

Here I have done Data Normalization: Each input featured is normalized using mean (mu) and standard deviation (sigma).

Prediction Generated : I have used trained linear regression model, we generate prediction for disease.

After Generating we have got result and we have tested it's Mean squared Error test, with the prediction. As a showcase this model is under-fitting
Here is the code for model Error Testing

```
[15]: #for calculating error on test we need to follow some steps
#we need to do Compute predictions on the test set
test_predictions = model(x_test)

# now the 2nd step is to calculate MSE for the test set
test_error = mean_squared_error(y_test, test_predictions)

# now we will Print the test error
print(f"Mean Squared Error on the test set: {test_error}")

Mean Squared Error on the test set: 238058.0625
```

Q6. Try the code with several learning rates that differ by orders of magnitude, and record the training and test set errors. Describe the theory of how changing the learning rate affects learning. What do you observe in the training error? How about the error on the test set?

Answer:

In this we have tried various Learning rate we have tested with different Learning rates (0.001,0.01,0.1,1.0)

Here I have observed that if we **Lower the learning rate** the training will be lower and the model under-fits, now if we check about the Will check about the values if the values will be lower, then smoother, the convergence, but the slowdown the training process potentially causing the model to get stuck.

Here we will see about **higher learning rates**, the higher the learning rate, the speed of convergence may be lead to instability oscillation if too large. Here the best learning rate is 0.01 which has a stable relatively, there were lower rates on training and testing sets I have observed that this is the best learning rate for the model as compare to others. As when we change if it goes more or if it goes lower, it affects the model. Another result I have selected 0.01 as the best learning rate.

Code:

```
In [16]:
# Firstly we need to Define a simple Linear Regression model
class LinearRegression(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegression, self).__init__()
        self.weight = nn.Parameter(torch.randn(1, input_dim, requires_grad=True))
        self.bias = nn.Parameter(torch.randn(1, requires_grad=True))

    def forward(self, x):
        return x @ self.weight.T + self.bias

# Initialize data
x_train = torch.randn(100, 10).float() # Replace with actual training data
y_train = torch.randn(100, 1).float() # Replace with actual target data
x_test = torch.randn(20, 10).float() # Replace with actual test data
y_test = torch.randn(20, 1).float() # Replace with actual target data

# Initialize model, loss function, and optimizer with learning rate
#here we will be changing learning rate and check the efficiency of model
learning_rate = 0.01
model = LinearRegression(x_train.shape[1])
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# Training loop
epochs = 100
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad() # Clear gradients from previous step
    predictions = model(x_train)
    loss = criterion(predictions, y_train)
    loss.backward() # Compute gradients
    optimizer.step() # Update parameters based on gradients

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Training Loss: {loss.item()}")

# Evaluate on the test set
model.eval()
with torch.no_grad():
    test_predictions = model(x_test)
    test_loss = criterion(test_predictions, y_test)
    print(f"Test Loss: {test_loss.item()}")
```

Epoch 0, Training Loss: 11.479552769991024

Answer :

```
Epoch 0, Training Loss: 11.479552268981934
Epoch 10, Training Loss: 8.394637107849121
Epoch 20, Training Loss: 6.217758655548096
Epoch 30, Training Loss: 4.679318904876709
Epoch 40, Training Loss: 3.5904102325439453
Epoch 50, Training Loss: 2.8184778690338135
Epoch 60, Training Loss: 2.270381212234497
Epoch 70, Training Loss: 1.880582332611084
Epoch 80, Training Loss: 1.6029009819030762
Epoch 90, Training Loss: 1.4047515392303467
Test Loss: 2.411708354949951
```

Q8. First of all, find the best value of alpha to use in order to optimize best. Next, experiment with different values of λ and see how this affects the shape of the hypothesis.


Answer:

Here Alpha control the rate in which the model updates its parameter based on this a very small alpha relate to convergence while the larger the alpha which can cause the model to over shoot. That's the reason we always use the medium range of this small range alpha value.

Here I have went to multiple steps, which are like firstly, we need to tune the alpha like we must check the experimental values like the example one 0.001, 0.5, 0.1, etc. after plotting the training cost over retentions, we have objected how quickly and effectively the model conversion and it's been observed because of the change in the various values will give the best alpha value.

The next part is lambada in lambda function, we need to train the lambda value as the lambda value decides to overfitting or penalising. We need to check multiple values on lambda and observe the regularisation and model complexity. Here the model complexity can be adjusted with the help of this values, whereas the higher the value may restrict the model while the lower value will allow the model to be flexible. After all this, I conclude that we need to check various values which will make the model with the lowest error and good balancing and variance, which will avoid underfitting and overfitting.

Code with result:

jupyter wk3_Regression_2024 work Last Checkpoint: Last Friday at 8:18 PM (autosaved)  Logour

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (pykernel)

+

Run

Markdown

```
In [28]: def mean_squared_error(y_true: torch.Tensor, y_pred: torch.Tensor, lam: float, theta: torch.Tensor) -> torch.Tensor:
# Compute the mean squared error
mse = torch.mean((y_true - y_pred) ** 2)
# Add regularization term (excluding the bias term)
reg_term = lam * torch.sum(theta[1:] ** 2)
# Total cost is MSE plus regularization
return mse + reg_term

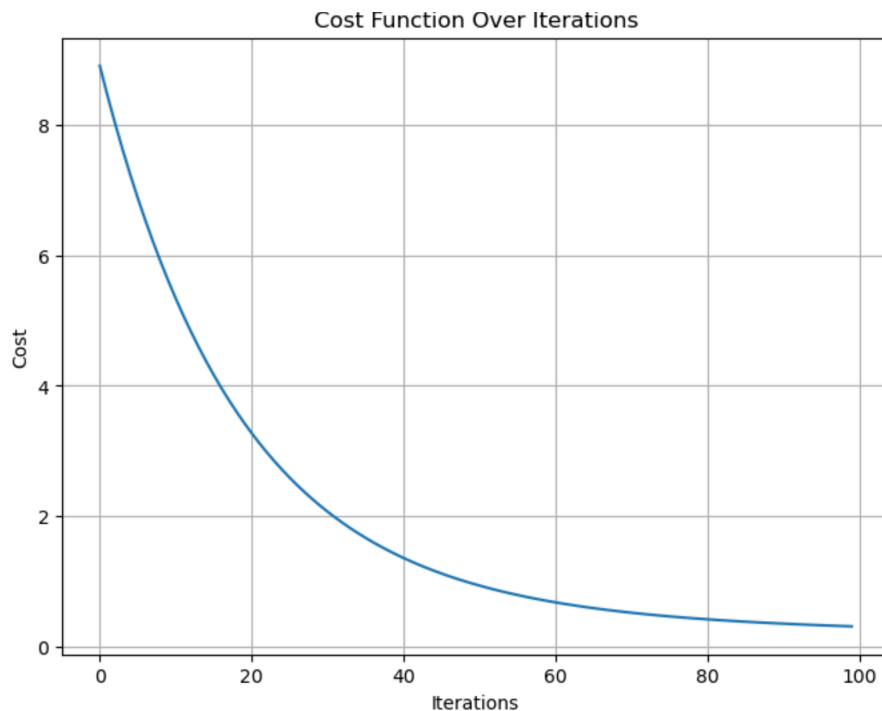
# Initialize model and other parameters
cost_lst = []
model = LinearRegression(x3.shape[1]) # Initialize the model (assumed LinearRegression)
alpha = 0.02 # Learning rate
lam = 0.1 # Regularization strength

# Train for a fixed number of iterations
for it in range(100):
    prediction = model(x3) # Make predictions
    cost = mean_squared_error(y, prediction, lam, model.weight) # Calculate cost
    cost_lst.append(cost.item()) # Append the cost to the list

    # Perform gradient descent step
    gradient_descent_step(model, x3, y, prediction, alpha, lam)

# After training, plot the cost curve and print final results
plt.figure(figsize=(8, 6))
plt.plot(range(100), cost_lst)
plt.title("Cost Function Over Iterations")
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.grid(True)
plt.show()

# Print the final model weights and minimum cost
print(model.weight)
print(f"Minimum cost: {min(cost_lst)}")
```



Parameter containing:
tensor([[1.6037, -1.9993, 0.6012, 0.4512, 0.4214, 1.0919]],
requires_grad=True)
Minimum cost: 0.31254902482032776