CONTLO ASSIGNMENT

Name : A Sneha Roopa Sri
Roll No : BT20CSE161

Task 0: Q/A assignment

Q 1)
Answer:

When we duplicate a feature n into a new feature (n+1) and retrain a logistic regression model, the likely relationship between the new weights (w_new_0, w_new_1, w_new_n, and w_new_{n+1}) depends on how the duplicated feature impacts the model.

1. If the duplicated feature provides similar information to the original, the weights are likely to be similar.
2. If the duplicated feature adds valuable information, w_new_{n+1} may be greater than w_new_n.
3. If the duplicated feature is redundant, w_new_{n+1} may be smaller than w_new_n.

```
for i in {0, 1, 2, ….., n-2, n-1}:
    w_new_i = w_i

for i in {n, n+1}:
    w_new_i = (w_n / 2)

w_new_0 = w_0
w_new_1 = w_1
w_new_n = w_n / 2
w_new_n+1 = w_n / 2
```

In summary, the relationship between the weights depends on whether the duplicated feature is helpful, similar or redundant compared to the original.

Q 2)
Answer:

Given the information provided, the statement ‘b’ is correct.

b. E is better than A with over 95% confidence, B is worse than A with over 95% confidence. You need to run the test for longer to tell where C and D compare to A with 95% confidence.

Justification:
1. Template E has a higher click-through rate 14% than template A 10% with a significant difference, and this conclusion is made with over 95% confidence.
2. Template B has a lower click-through rate 7% than template A with over 95% confidence.
3. To confidently compare templates C and D to A with 95% confidence, more data or a longer test duration may be necessary. The current information isn't sufficient to draw conclusions about the relative performance of C and D compared to A.

Q 3)
Answer:

In modern well written packages, such as those based on efficient linear algebra libraries the computational cost is often dominated by the matrix operations involved in gradient descent. In the case of sparse feature vectors, the cost is reduced due to the sparsity.

The main components of the computational cost:

Forward Pass (Prediction):
1. Computing the linear combination of weights and features.
2. Complexity: O(m * n), where m is the number of training examples, and n is the number of features.

Compute Loss:
1. Calculating the logistic loss.
2. Complexity: O(m).

Backward Pass (Gradient Calculation):
1. Computing the gradient of the loss with respect to the weights.
2. Complexity: O(m * n * k), where k is the average number of non-zero entries in each training example. This is due to the sparsity of feature vectors.

Parameter Update:
1. Updating the weights using the computed gradient.

2. Complexity: O(n * k), where k is the average number of non-zero entries.

Overall, the dominant factor in the computational cost is often the product of the number of training examples (m) and the number of features (n), but the sparsity of feature vectors helps in reducing the cost, making it more efficient for sparse datasets. As a result, the approximate computational cost of each gradient descent iteration for logistic regression with sparse feature vectors is often in the order of O(m * n) with optimizations for sparsity.

Q 4)

Answer:

Considering the different methods to generate additional training data for building Classifier V2, we can make some informed predictions about their potential impact on accuracy:

1. Run V1 Classifier on 1 Million Random Stories:

- These stories are likely to be close to the decision boundary according to the V1 classifier.
- Since the V1 classifier might have misclassified these examples, the new model (V2) might inherit some of the misclassifications.
- The accuracy of V2 may not significantly improve, and it might even introduce some noise.

2. Get 10k Random Labeled Stories:

- Randomly selecting labeled stories doesn't consider the decision boundary of the V1 classifier.
- The accuracy of V2 is likely to depend on the quality and diversity of the randomly selected stories.
- It might perform better than Method 1 if the random selection includes informative and representative examples.

3. Pick a Random Sample of 1 Million Stories:

- Labeling the entire set and then selecting the subset where V1 is both wrong and far from the decision boundary implies picking ambiguous or difficult-to-classify examples.
- This method might lead to a more challenging and diverse set of training data for V2.
- V2 may have improved accuracy compared to Method 1 if it can learn from the challenging cases.

Ranking based on Accuracy Prediction:

- Method 3 might have the potential to yield the highest accuracy for V2 if the challenging cases help the model generalize better.

- Method 2 could be second, depending on the quality and diversity of the randomly selected labeled stories.
- Method 1 might result in the lowest accuracy for V2, as it focuses on examples close to the decision boundary of V1, which might be inherently difficult for the model.

In summary, while the accuracy ranking is somewhat speculative, Method 3 appears to have the potential to provide the best training data for improving the accuracy of Classifier V2, followed by Method 2, and then Method 1.

Q 5)
Answer:

The estimates for **p** using the three methods:

1. **Maximum Likelihood Estimate (MLE):** The MLE for (p) is the ratio of the number of heads (k) to the total number of coin tosses (n).

   MLE: p_{MLE} = k / n

2. **Bayesian Estimate:** Given a uniform prior over the range [0,1], the posterior distribution is a Beta distribution. The expected value of the posterior distribution is the Bayesian estimate for (p).

   Bayesian Estimate: $p_{Bayesian}$ = {k+1} / {n+2}

3. **Maximum a Posteriori (MAP) Estimate:** The MAP estimate is the mode of the posterior distribution, which, in this case, is also the peak of the Beta distribution.

   MAP Estimate: MLE: p_{MAP} = k / n

MLE is directly based on observed frequencies and doesn't incorporate any prior information. Bayesian Estimate incorporates a uniform prior and is a compromise between the prior and the likelihood. MAP Estimate considers the mode of the posterior distribution and, in this case, aligns with the MLE due to the specific choice of a uniform prior.

Task 3: Training Loop Implementation

Creating a full training loop with support for single GPU, Distributed Data Parallel (DDP), and Fully Sharded Data Parallel (FSDP) is quite extensive and involves multiple components. A simplified example to get started.

Training Loop with Single GPU, DDP, and FSDP Support:

Code:

```
import torch

import torch.nn as nn

from torch.nn.parallel import DistributedDataParallel

from torch.cuda.amp import GradScaler, autocast

from torch.utils.data import DataLoader


# Assume you have a GPT-2 model and dataset defined

model = GPT2(embed_size=256, heads=8, num_layers=6, vocab_size=10000)

dataset = rDataset(...)


# Initialize Distributed Data Parallel (DDP) if available

if torch.cuda.device_count() > 1:

    model = nn.DataParallel(model)


# Initialize Fully Sharded Data Parallel (FSDP) if available

if fsdp_available:

    from fsdp import FullyShardedDataParallel

    model = FullyShardedDataParallel(model)


# Set device and move model to GPU

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = model.to(device)
```

```
# Define optimizer and other necessary components
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)

# Training loop
num_epochs = 10
train_loader = DataLoader(dataset, batch_size=32, shuffle=True)
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        inputs, targets = batch
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()

        # Use autocast for mixed-precision training (optional)
        with autocast():
            outputs = model(inputs)

            # Define your loss function and compute the loss
            loss = your_loss_function(outputs, targets)

        scaler = GradScaler()
        scaler.scale(loss).backward()

        # Gradient clipping if needed
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        scaler.step(optimizer)
        scaler.update()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}")
    torch.save(model.state_dict(), "model.pth")
```

Adaptations for Visualizations:

1. Single GPU: Visualize GPU usage, memory consumption, and training metrics. Tools like `nvidia-smi` or PyTorch's `torch.utils.tensorboard` can be helpful.

2. DDP:Visualize GPU usage, memory consumption, and training metrics for each GPU separately. Compare training speed and resource utilization with the single GPU setup.

3. FSDP: Visualize the sharding of parameters and gradients. Compare training speed and resource utilization with both single GPU and DDP setups.