Assignment @ SuperAGI

Q/A -

1. Wnew +W new (n+1) ≈ wn matlab weights wnew and wnew(n+1) will adjust in such a way that their combined effect is similar to the effect of regional weight wn.
2. 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.
3. O(mk) -> computational cost for each gradient descent iteration in logistic regression , when dealing with sparse data in an optimized package.
4. Approach 3 has the potential to be the most beneficial for improving the accuracy of the V2 classifier, as it focuses on challenging cases where the V1 model was confidently wrong and far from the decision boundary.
   Approach 1 could also provide useful data, but the proximity to the decision boundary doesn't guarantee that the V1 classifier was wrong.
   Approach 2, while providing labeled data, may not specifically target challenging cases where the classifier is likely to make errors.
   Ranking (from likely most helpful to least):
    1. Approach 3
    2. Approach 1
    3. Approach 2
5. MLE -> k/n , Bayesian : K+1/n+2 , map estimate : k+1/n+2

Coding

All these 3 tasks are done with help of chatgpt.

Task 1

The following python code is as follows-

```
import torch
import torch.nn as nn

class GPT2Model(nn.Module):
def __init__(self, vocab_size, d_model=768, nhead=12, num_layers=12):
super(GPT2Model, self).__init__()

self.token_embedding = nn.Embedding(vocab_size, d_model)
self.positional_encoding = self.positional_encoding(d_model)
```

```
            self.transformer_layers = nn.ModuleList([
            nn.TransformerEncoderLayer(d_model, nhead) for _ in range(num_layers)
        ])

        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        x = self.token_embedding(x)

        x += self.positional_encoding[:x.size(1), :].unsqueeze(0)

        for layer in self.transformer_layers:
            x = layer(x)

        x = self.fc(x)

        return x

    def positional_encoding(self, d_model, max_len=512):
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.log(torch.tensor(10000.0)) /
d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        return pe

vocab_size = 30000  # Adjust based on your dataset
model = GPT2Model(vocab_size)

checkpoint = torch.load("path/to/gpt2_model_checkpoint.pth")
model.load_state_dict(checkpoint)
model.eval()

input_sequence = torch.randint(0, vocab_size, (1, 50))  # Adjust sequence length as needed
output = model(input_sequence)
print(output.shape)
```

Task 2

Rotatory positional embedding

```
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F

class GPT2ModelRotary(nn.Module):
    def __init__(self, vocab_size, d_model=768, nhead=12, num_layers=12):
        super(GPT2ModelRotary, self).__init__()

        self.token_embedding = nn.Embedding(vocab_size, d_model)

        self.rotary_positional_encoding = self.rotary_positional_encoding(d_model)

        self.transformer_layers = nn.ModuleList([
            nn.TransformerEncoderLayer(d_model, nhead) for _ in range(num_layers)
        ])

        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        x = self.token_embedding(x)

        x += self.rotary_positional_encoding[:x.size(1), :].unsqueeze(0)

        for layer in self.transformer_layers:
            x = layer(x)

        x = self.fc(x)

        return x

    def rotary_positional_encoding(self, d_model, max_len=512):
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.log(torch.tensor(10000.0)) /
d_model))
        sin_vals = torch.sin(position * div_term)
        cos_vals = torch.cos(position * div_term)
        return torch.cat([sin_vals, cos_vals], dim=1)

model_rotary = GPT2ModelRotary(vocab_size)
output_rotary = model_rotary(input_sequence)
print(output_rotary.shape)
```

Group Query Attention

```
class GPT2ModelGroupQueryAttention(GPT2ModelRotary):
    def __init__(self, vocab_size, d_model=768, nhead=12, num_layers=12, num_groups=4):
        super(GPT2ModelGroupQueryAttention, self).__init__(vocab_size, d_model, nhead, num_layers)

        self.num_groups = num_groups

        for layer in self.transformer_layers:
            layer.self_attn = nn.MultiheadAttention(d_model, nhead, kdim=d_model // num_groups,
vdim=d_model // num_groups)

model_group_query_attention = GPT2ModelGroupQueryAttention(vocab_size)
output_group_query_attention = model_group_query_attention(input_sequence)
print(output_group_query_attention.shape)
```

Sliding Window Attention

```
class GPT2ModelSlidingWindow(GPT2ModelRotary):
    def __init__(self, vocab_size, d_model=768, nhead=12, num_layers=12, window_size=512):
        super(GPT2ModelSlidingWindow, self).__init__(vocab_size, d_model, nhead, num_layers)

        self.window_size = window_size

        for layer in self.transformer_layers:
            layer.self_attn = SlidingWindowAttention(d_model, nhead, window_size)

class SlidingWindowAttention(nn.Module):
    def __init__(self, embed_dim, num_heads, window_size):
        super(SlidingWindowAttention, self).__init__()
        self.window_size = window_size
        self.attention = nn.MultiheadAttention(embed_dim, num_heads)

    def forward(self, x):
        return x
model_sliding_window = GPT2ModelSlidingWindow(vocab_size)
output_sliding_window = model_sliding_window(input_sequence)
print(output_sliding_window.shape)
```

Task 3

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split


class MyDataset(torch.utils.data.Dataset):

vocab_size = 30000  # Adjust based on your dataset
model = GPT2Model(vocab_size)

batch_size = 64
learning_rate = 1e-3
epochs = 10

dataset = MyDataset()
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

def train_step(model, data, criterion, optimizer):
    model.train()
    inputs, targets = data
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss.item()

def validate(model, val_loader, criterion):
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for inputs, targets in val_loader:
            outputs = model(inputs)
```

```
            loss = criterion(outputs, targets)
            total_loss += loss.item()
    return total_loss / len(val_loader)

# Single GPU Training Loop
for epoch in range(epochs):
    for i, data in enumerate(train_loader):
        loss = train_step(model, data, criterion, optimizer)
        print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train_loader)}, Loss: {loss}")

# Distributed Data Parallel (DDP) Training Loop
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

model = model.to("cuda")
model = nn.parallel.DistributedDataParallel(model)

for epoch in range(epochs):
    for i, data in enumerate(train_loader):
        loss = train_step(model, data, criterion, optimizer)
        print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train_loader)}, Loss: {loss}")

# Fully Sharded Data Parallel (FSDP) Training Loop
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

from fairscale.nn.data_parallel import FullyShardedDataParallel

model = model.to("cuda")
model = FullyShardedDataParallel(model)

for epoch in range(epochs):
    for i, data in enumerate(train_loader):
        loss = train_step(model, data, criterion, optimizer)
        print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train_loader)}, Loss: {loss}")
```