FUNDRAISING/best_fundraising_cross_val.py

```python
# ==============================================================================
# SET-UP AND IMPORTS
# ==============================================================================
import os
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset, ConcatDataset
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler

# Set directory and device setup
os.chdir('/home/dan/FUNDRAISING')
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# ==============================================================================
# PREPARE DATA
# ==============================================================================
df = pd.read_csv('./data/df.csv', index_col='Unnamed: 0')
df['zipconvert'] = df[['zipconvert2', 'zipconvert3', 'zipconvert4', 'zipconvert5']].apply(
    lambda row: 'zc' + str(row.idxmax()[-1]) if pd.notna(row.idxmax()) else 'unknown', axis=1
)
test = df[df['type']=='test']

df = df[df['type']!='test']

df.drop(['zipconvert2', 'zipconvert3', 'zipconvert4', 'zipconvert5', 'type'], axis=1, inplace=True)
test.drop(['zipconvert2', 'zipconvert3', 'zipconvert4', 'zipconvert5', 'type'], axis=1, inplace=True)

cat_cols = ['homeowner', 'female', 'zipconvert', 'wealth', 'income', 'num_child']
cont_cols = [col for col in df.columns if col not in cat_cols + ['target']]
emb_sizes = [(df[col].astype('category').cat.codes.max() + 1, min(50, (df[col].nunique() + 1) // 2)) for col in cat_cols]

# Function to process datasets
def process_data(data):
    cats = np.stack([data[col].astype('category').cat.codes.values for col in cat_cols], axis=1)
    conts = np.stack([data[col].values for col in cont_cols], axis=1)
    scaler = StandardScaler()

    conts = scaler.fit_transform(conts)
    y = data['target'].map({'Donor': 1, 'No Donor': 0}).values
    return torch.tensor(cats, dtype=torch.int64), torch.tensor(conts, dtype=torch.float), torch.tensor(y, dtype=torch.long)


cats, conts, targets = process_data(df)
test_cats, test_conts, test_targets = process_data(test)

dataset = TensorDataset(cats, conts, targets)

test_dataset = TensorDataset(test_cats, test_conts, test_targets)
test_loader = DataLoader(test_dataset, batch_size=2048, shuffle=False)

# ==============================================================================
# DEFINE A TABULAR MODEL
# ==============================================================================
class TabularModel(nn.Module):
    def __init__(self, emb_sizes, n_cont, out_sz, layers, p=0.5):
        super().__init__()
        self.embeds = nn.ModuleList([nn.Embedding(ni, nf) for ni, nf in emb_sizes])
        self.emb_drop = nn.Dropout(p)
        self.bn_cont = nn.BatchNorm1d(n_cont)

        # Calculate the total embedding output size
        total_emb_size = sum(nf for _, nf in emb_sizes)
        n_in = total_emb_size + n_cont  # Total input size for the first linear layer

        layerlist = nn.ModuleList()
        for output_size in layers:
            layerlist.append(nn.Linear(n_in, output_size))
            layerlist.append(nn.ReLU())
            layerlist.append(nn.BatchNorm1d(output_size))
            layerlist.append(nn.Dropout(p))
            n_in = output_size  # Update n_in to the output size of the current layer

        layerlist.append(nn.Linear(layers[-1], out_sz))  # Final output layer
        self.layers = nn.Sequential(*layerlist)

    def forward(self, x_cat, x_cont):
        embeddings = [e(x_cat[:, i]) for i, e in enumerate(self.embeds)]
        x = torch.cat(embeddings, 1)
        x = self.emb_drop(x)
```

```
        x_cont = self.bn_cont(x_cont)
        x = torch.cat([x, x_cont], 1)
        return self.layers(x)

# Initialize model
model = TabularModel(emb_sizes, len(cont_cols), 2, [200, 100], p=0.4)
model = model.to(device)


#endregion
#region # EARLY STOPPING
# ================================================================================
# EARLY STOPPING
# ================================================================================
class EarlyStopping:
    def __init__(self, patience=5, verbose=False, delta=0):
        self.patience = patience
        self.verbose = verbose
        self.delta = delta
        self.best_score = None
        self.early_stop = False
        self.counter = 0

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.verbose:
                print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.counter = 0


# ================================================================================
# CROSS-VALIDATION SETUP
# ================================================================================
def calculate_accuracy(y_pred, y_true):
    y_pred_classes = torch.argmax(y_pred, dim=1)
    correct = (y_pred_classes == y_true).float()  # convert into float for division
    acc = correct.sum() / len(correct)
    return acc

kf = KFold(n_splits=20, shuffle=True, random_state=42)
early_stopping = EarlyStopping(patience=5, verbose=True)
results = []

for fold, (train_idx, val_idx) in enumerate(kf.split(dataset)):
    train_subsampler = torch.utils.data.SubsetRandomSampler(train_idx)
    val_subsampler = torch.utils.data.SubsetRandomSampler(val_idx)
    train_loader = DataLoader(dataset, batch_size=32, sampler=train_subsampler)
    val_loader = DataLoader(dataset, batch_size=32, sampler=val_subsampler)

    model = TabularModel(emb_sizes, len(cont_cols), 2, [100,200], p=0.4).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)

    for epoch in range(50):  # Adjust as needed
        model.train()
        total_loss, total_acc = 0, 0
        for cats, conts, y in train_loader:
            cats, conts, y = cats.to(device), conts.to(device), y.to(device)
            optimizer.zero_grad()
            outputs = model(cats, conts)
            loss = criterion(outputs, y)
            acc = calculate_accuracy(outputs, y)
            total_loss += loss.item()
            total_acc += acc.item()
            loss.backward()
            optimizer.step()
        avg_train_loss = total_loss / len(train_loader)
        avg_train_acc = total_acc / len(train_loader)

        model.eval()
        val_loss, val_acc = 0, 0
        with torch.no_grad():
            for cats, conts, y in val_loader:
                cats, conts, y = cats.to(device), conts.to(device), y.to(device)
                outputs = model(cats, conts)
                loss = criterion(outputs, y)
                acc = calculate_accuracy(outputs, y)
                val_loss += loss.item()
```

```
                val_acc += acc.item()
        avg_val_loss = val_loss / len(val_loader)
        avg_val_acc = val_acc / len(val_loader)

        print(f'Fold {fold+1}, Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Train Acc: {avg_train_acc:.4f}, Val Loss: {avg_val_loss:.4f}, Val Acc: {avg_val_acc:.4f}')

        # Call early stopping
        early_stopping(avg_val_loss, model)
        if early_stopping.early_stop:
            print("Early stopping")
            break

    results.append((avg_val_loss, avg_val_acc))

average_val_loss = sum(x[0] for x in results) / len(results)
average_val_acc = sum(x[1] for x in results) / len(results)
print(f'Average Validation Loss: {average_val_loss:.4f}, Average Validation Accuracy: {average_val_acc:.4f}')

preds = []
model.eval()

with torch.no_grad():
    for cats, conts, y in test_loader:
        cats, conts, y = cats.to(device), conts.to(device), y.to(device)
        output = model(cats, conts)
        predicted = output.argmax(dim=1)  # Ensure you use dim=1
        # print(predicted)
        preds.append(predicted.cpu())  # Move predictions to CPU
# print(type(preds[0]))
# Concatenate all batch predictions into a single tensor
# preds = torch.cat(preds)
preds = preds[0].tolist()
# print(preds)
final_preds = []
donor=1
no_donor=1
for i in preds:
    if i == 1:
        final_preds.append('Donor')
        donor +=1
    else:
        final_preds.append('No Donor')
        no_donor+=1
print(f'% Donor = {donor / (donor+no_donor)}')


save_df = pd.DataFrame(final_preds, columns=['values'])
save_df.to_csv('./preds/preds.csv', index=False)
```