Assignment-2 (ANN)

Submitted By:

- Avinash Aanand-(Roll No.: 2403RES99)
- Aditya Gupta-(Roll No.:2403RES85)
- Sanjeev Kumar (Roll No.: 2403RES117)

- Due Date: 16th Sept 2024

CS582: Neural Network and NLP Lab
External Tool-Assignment 02

ANN

CS582 Neural Network and NLP Lab

Table of Content

CS582 Neural Network and NLP Lab

Problem Statements:

Statements

Customer churn, the loss of clients or subscribers, is a significant challenge for many businesses, particularly in subscription-based services such as telecommunications, banking, and retail. Predicting customer churn helps companies identify which customers are likely to leave so that proactive retention strategies can be developed. By accurately forecasting which customers might churn, companies can focus on retaining existing customers rather than solely acquiring new ones, which is typically more cost-effective.

The objective of this assignment is to build an Artificial Neural Network (ANN) model to predict customer churn using a dataset of customer attributes. The dataset contains a range of features, including demographic information and service-related variables, which we will use to train a predictive model. Specifically, this project focuses on two key tasks:

- Train an ANN model with one hidden layer, evaluate its performance after 100 epochs.
- Train an ANN model with two hidden layers and evaluate its performance after 50 epochs.

Through this, the goal is to observe how changes in the architecture of the neural network (in terms of the number of hidden layers) and the training epochs affect the model's performance in predicting churn.

Tasks

Build a Neural Network Model to give the Customer Churn predictions with the given given dataset:
https://www.kaggle.com/datasets/muhammadshahidazeem/customer-churn-dataset
The assignment should cover the following points:

- Create an ANN model.
- Split the data in the ratio 80:20.
- Use one hidden layer to train your model with 100 epochs and show the results
- Use two hidden layers with 50 epochs and show the results

Introduction

Artificial Neural Networks (ANNs) are a class of machine learning models inspired by biological neural networks. They consist of layers of interconnected neurons that process input data to generate predictions. ANNs are particularly useful for classification tasks, such as predicting whether a customer will churn based on their behavioral and demographic features.

In this assignment, we will leverage an ANN to build a model for customer churn prediction. The model will use a dataset of customer attributes, including factors like account tenure, usage patterns, customer demographics, and interaction history with the company, to classify whether a customer will churn or stay with the company.

Key Steps in the Process:

1. **Data Preprocessing:** Before training any machine learning model, the data needs to be cleaned and prepared. This includes handling missing values, encoding categorical features, standardizing numeric data, and addressing class imbalance using techniques like Synthetic Minority Over-sampling Technique (SMOTE).
2. **Splitting the Dataset:** The dataset will be split into training (80%) and testing (20%) sets to ensure that the model is trained and evaluated on separate portions of the data, ensuring generalization.
3. **Building the ANN Model: We will create two versions of the ANN model:**
    - One Hidden Layer: This version will have a single hidden layer and will be trained for 100 epochs.
    - Two Hidden Layers: This version will have two hidden layers and will be trained for 50 epochs.
4. **Evaluating the Model:** After training, the models will be evaluated using accuracy and other relevant metrics like precision, recall, and F1-score. This will allow us to assess how well the model can predict customer churn.
5. **Handling Overfitting:** We will use techniques like dropout layers and early stopping to prevent overfitting and ensure the model performs well on unseen data.

By comparing the performance of the two models, we aim to understand the impact of network depth and training duration on predictive accuracy and overall performance in solving the customer churn problem.

In conclusion, this assignment will provide insights into how neural networks can be applied to real-world business problems, such as customer retention, and will highlight the importance of model architecture and hyperparameter tuning in achieving optimal results.

Respective comparison

When addressing the problem of customer churn prediction, several machine learning models can be applied. The existing systems or models commonly used for customer churn prediction include:

1. Logistic Regression:
    - Strengths: Logistic regression is widely used for binary classification problems like customer churn prediction. It is simple, interpretable, and performs well on linearly separable data.
    - Weaknesses: It may underperform when the data contains complex, non-linear relationships between features and the target variable. Logistic regression models can struggle to capture these patterns, leading to lower predictive accuracy in more complex scenarios like customer churn.
2. Decision Trees and Random Forest:
    - Strengths: Decision trees are more flexible and can capture non-linear relationships. Random forests, which are ensembles of decision trees, reduce overfitting and increase accuracy through majority voting.
    - Weaknesses: Decision trees can overfit if not pruned, and random forests can be computationally expensive with large datasets. Also, interpretation can be difficult compared to simpler models.
3. Gradient Boosting Machines (GBMs):
    - Strengths: GBMs, such as XGBoost or LightGBM, are powerful in handling structured data and often outperform simpler models in churn prediction. They work well for feature importance extraction and handle imbalanced data better.
    - Weaknesses: Gradient boosting can be prone to overfitting if not properly tuned, and it requires careful parameter tuning. It is also computationally expensive and slower to train compared to some other models.
4. Support Vector Machines (SVMs):
    - Strengths: SVMs are effective for high-dimensional spaces and when there's a clear margin of separation. They are also robust to outliers.
    - Weaknesses: SVMs become inefficient with large datasets and don't scale well in terms of computational cost. Tuning the kernel and regularization parameters is challenging.

ANN (Artificial Neural Network) vs. Existing Systems

1. Model Complexity:
    - ANNs: Artificial Neural Networks excel in capturing complex non-linear relationships in the data. By using multiple hidden layers, ANNs can model more intricate patterns in customer behavior, which traditional models like logistic regression or decision trees might miss.
    - Traditional Models: Logistic regression, decision trees, and SVMs may struggle with datasets containing high-dimensional features or subtle interactions between variables.
2. Performance on Non-Linear Data:
    - ANNs: Due to the non-linear activation functions in hidden layers, ANNs can better handle complex data where relationships between features and target variables are not straightforward. This allows ANNs to have better predictive accuracy, especially when the dataset is large and complex.
    - Traditional Models: These models, especially logistic regression, assume linear relationships, making them less effective for non-linear patterns.
3. Handling Imbalanced Datasets:
    - ANNs: When combined with techniques like SMOTE (Synthetic Minority Over-sampling Technique), ANNs can effectively handle imbalanced datasets, like those typically found in customer churn prediction.
    - Traditional Models: While decision trees and gradient boosting models can also handle imbalanced datasets effectively, simpler models like logistic regression often struggle in such cases, leading to biased predictions toward the majority class.
4. Feature Engineering and Scalability:
    - ANNs: Neural networks require minimal feature engineering, as the layers automatically learn feature hierarchies during training. They are also scalable for large datasets.

- **Traditional Models:** Models like logistic regression or SVMs require significant feature engineering to perform well. Without proper feature selection and transformation, these models may not capture the essential patterns in the data.

5. **Risk of Overfitting:**
   - **ANNs:** Although ANNs are powerful, they are prone to overfitting, especially when using deeper architectures or small datasets. Techniques such as dropout layers and early stopping are essential to mitigate overfitting risks.
   - **Traditional Models:** Models like decision trees (without pruning) and gradient boosting also have overfitting issues but tend to generalize better with regularization techniques.
6. **Interpretability:**
   - **ANNs:** One of the major downsides of neural networks is that they are often considered "black-box" models, meaning they lack interpretability. This makes it harder for businesses to understand why certain customers are predicted to churn.
   - **Traditional Models:** Logistic regression and decision trees are more interpretable, as they can provide insights into how each feature contributes to the prediction. This transparency can be beneficial when explaining model decisions to stakeholders.

ANNs in the Context of Churn Prediction

In customer churn prediction, ANNs can offer significant improvements in accuracy over traditional models, especially when handling large, complex datasets with non-linear relationships. However, the trade-off comes with increased computational cost and reduced interpretability.

For businesses prioritizing predictive power over interpretability, ANNs offer a superior solution. On the other hand, if simplicity and transparency are more important, traditional models like logistic regression or decision trees might still be preferable.

By using techniques such as dropout, early stopping, and careful hyperparameter tuning, ANNs can be fine-tuned to prevent overfitting, making them a competitive choice for churn prediction when compared to existing systems.

Approaches

This code implements a customer churn prediction model using Artificial Neural Networks (ANN) with two different architectures: one with a single hidden layer and another with two hidden layers. Below is a theoretical breakdown of the key approaches used in the code.

1. Data Preprocessing

- **Dataset Loading:** The dataset is loaded using pandas from a CSV file. The target variable (Churn) and features are separated.
- **Categorical Encoding:** The categorical features (Gender, Subscription Type, Contract Length) are transformed into numerical values using one-hot encoding. This is necessary because neural networks require numerical input.
- **Standardization:** All features are standardized using StandardScaler. Standardization ensures that the data has a mean of 0 and a standard deviation of 1, which helps in faster convergence during training. Neural networks generally perform better when input features are scaled consistently.
- **Train-Test Split:** The dataset is split into training and testing sets with a ratio of 80:20. This is a common practice to evaluate the model's performance on unseen data, thus preventing overfitting.

2. Neural Network Architecture

a. Single Hidden Layer Model

- **Input Layer:** The input to the model consists of the number of features from the dataset after encoding and standardization.
- **Hidden Layer:** There is one hidden layer in this model with 4096 neurons. The activation function used is ReLU (Rectified Linear Unit), which introduces non-linearity to the model and helps capture complex relationships in the data.
- **Output Layer:** The output layer has 1 neuron, corresponding to the binary nature of the Churn label (0 or 1). The activation function is Sigmoid, which outputs probabilities between 0 and 1, suitable for binary classification tasks.

- **Loss Function:** The loss function used is BCELoss (Binary Cross Entropy Loss). This loss function is appropriate for binary classification tasks as it calculates the difference between predicted probabilities and the actual label.
- **Optimizer:** The model uses the Adam optimizer, which is a popular optimization algorithm combining momentum and adaptive learning rate techniques for faster and more efficient convergence.

b. Two Hidden Layers Model

- **Input Layer:** Like the previous model, it uses the same input dimensions based on the features.
- **Hidden Layers:** This model contains two hidden layers:
  - The first hidden layer has 4096 neurons with ReLU activation.
  - The second hidden layer has 64 neurons, also using the ReLU activation. Adding this layer introduces more complexity, allowing the model to capture deeper hierarchical relationships in the data.
- **Output Layer: Like the one hidden layer model, the output layer has 1 neuron with a Sigmoid activation function.**
- **Loss and Optimizer: The same BCELoss and Adam optimizer are used for training.**

3. Model Training

- **The models are trained for a fixed number of epochs:**
  - The single hidden layer model is trained for 100 epochs.
  - The two hidden layer model is trained for 50 epochs.
- **During each epoch, the model performs the following steps:**
  - Forward Pass: The input data is passed through the network, generating predictions.
  - Loss Calculation: The loss is calculated using the predicted values and actual labels.
  - Backward Pass: The gradients are computed for each parameter in the model with respect to the loss using backpropagation.
  - Parameter Update: The optimizer (Adam) updates the model parameters based on the computed gradients.
- **Every 10 epochs, the loss is printed, allowing the user to monitor the progress of the training process.**

4. Evaluation

- Accuracy Evaluation: Once the models are trained, the evaluation function is used to test them on the unseen data (the test set).
- Predictions: The predictions from the network are rounded to 0 or 1, since the output of the sigmoid function is a probability.
- Accuracy: The accuracy is calculated by comparing the predicted labels with the actual labels in the test set. This gives an indication of how well the model is performing in terms of correctly predicting customer churn.

5. Theoretical Considerations in Model Selection

a. Single Hidden Layer

- **Pros:**
  - Simpler architecture, faster training.
  - Good for relatively straightforward tasks or smaller datasets.
  - Lower computational cost.
- **Cons:**
  - May fail to capture complex patterns in the data, leading to lower predictive power for more intricate tasks like customer churn.

b. Two Hidden Layers

- **Pros:**
  - More complex architecture, capable of capturing deeper relationships.
  - May perform better on tasks with complex, non-linear relationships between features.
- **Cons:**
  - Longer training time and increased computational cost.
  - Higher risk of overfitting, especially if the model is too complex for the data.

6. Loss Function and Optimizer Choice

- BCELoss (Binary Cross Entropy Loss): Since the task is binary classification (churn or no churn), this is the appropriate loss function. It measures the performance of a model whose output is a probability between 0 and 1.

- Adam Optimizer: Adam combines the benefits of two other popular optimization techniques: AdaGrad (adaptive learning rate) and RMSProp (momentum), making it efficient in terms of computation and robust to noisy data.

7. Device Utilization (CPU vs. GPU)

The code dynamically selects whether to run on a CPU or GPU based on availability:

- If a GPU (using Apple's Metal Performance Shaders backend, MPS) is available, the model will be trained on the GPU, which can significantly accelerate training times, especially for larger datasets.
- If a GPU is not available, the model will default to using the CPU.

Conclusion

This code provides a structured approach to solving the customer churn prediction problem using neural networks. By using two architectures (one with a single hidden layer and one with two hidden layers), the code allows a comparison between simpler and more complex models in terms of performance and efficiency.

Analysis

Through Algorithm (Flowchart)

The code provided trains two different artificial neural network (ANN) models to predict customer churn. To represent the overall flow of this code in a flowchart and to analyze its steps, we will break down the main processes and show how they fit together. Below is a description of each step, followed by the flowchart.

Algorithmic Steps

Step 1: Import Libraries

- Description: Import the necessary Python libraries (e.g., pandas, torch, sklearn) for data preprocessing, model building, and training.

Step 2: Check GPU Availability

- Description: Check if the system has a GPU (Apple's Metal Performance Shaders or another backend) and set the appropriate device (CPU or GPU) for computations.
    - Decision: If GPU is available, use it. Otherwise, use CPU.

Step 3: Load Dataset

- Description: Load the customer churn dataset into a pandas DataFrame from the provided CSV file.

Step 4: Data Preprocessing

- Drop Unnecessary Columns: Drop CustomerID (non-feature) and Churn (label).
- One-Hot Encoding: Apply one-hot encoding to categorical features (Gender, Subscription Type, and Contract Length) to convert them into numerical values.
- Concatenate Encoded Features: Merge the encoded categorical columns with the remaining numerical columns.
- Train-Test Split: Split the dataset into training (80%) and testing (20%) sets.
- Feature Scaling: Standardize the features using StandardScaler to ensure they have a mean of 0 and standard deviation of 1.

Step 5: Convert Data to PyTorch Tensors

- Description: Convert the training and testing data (features and labels) into PyTorch tensors to facilitate training with a neural network.

Step 6: Define Model Architectures

- Single Hidden Layer Model: Define a neural network architecture with one hidden layer (4096 neurons) and an output layer (1 neuron, sigmoid activation).
- Two Hidden Layers Model: Define a neural network architecture with two hidden layers (4096 neurons, 64 neurons) and an output layer (1 neuron, sigmoid activation).

Step 7: Initialize Model, Loss Function, and Optimizer

- Description:
    - Use the BCELoss loss function, suitable for binary classification.
    - Use the Adam optimizer for efficient parameter updates.

Step 8: Train Models

- Single Hidden Layer Model: Train the single hidden layer model for 100 epochs.
- Two Hidden Layers Model: Train the two hidden layers model for 50 epochs.

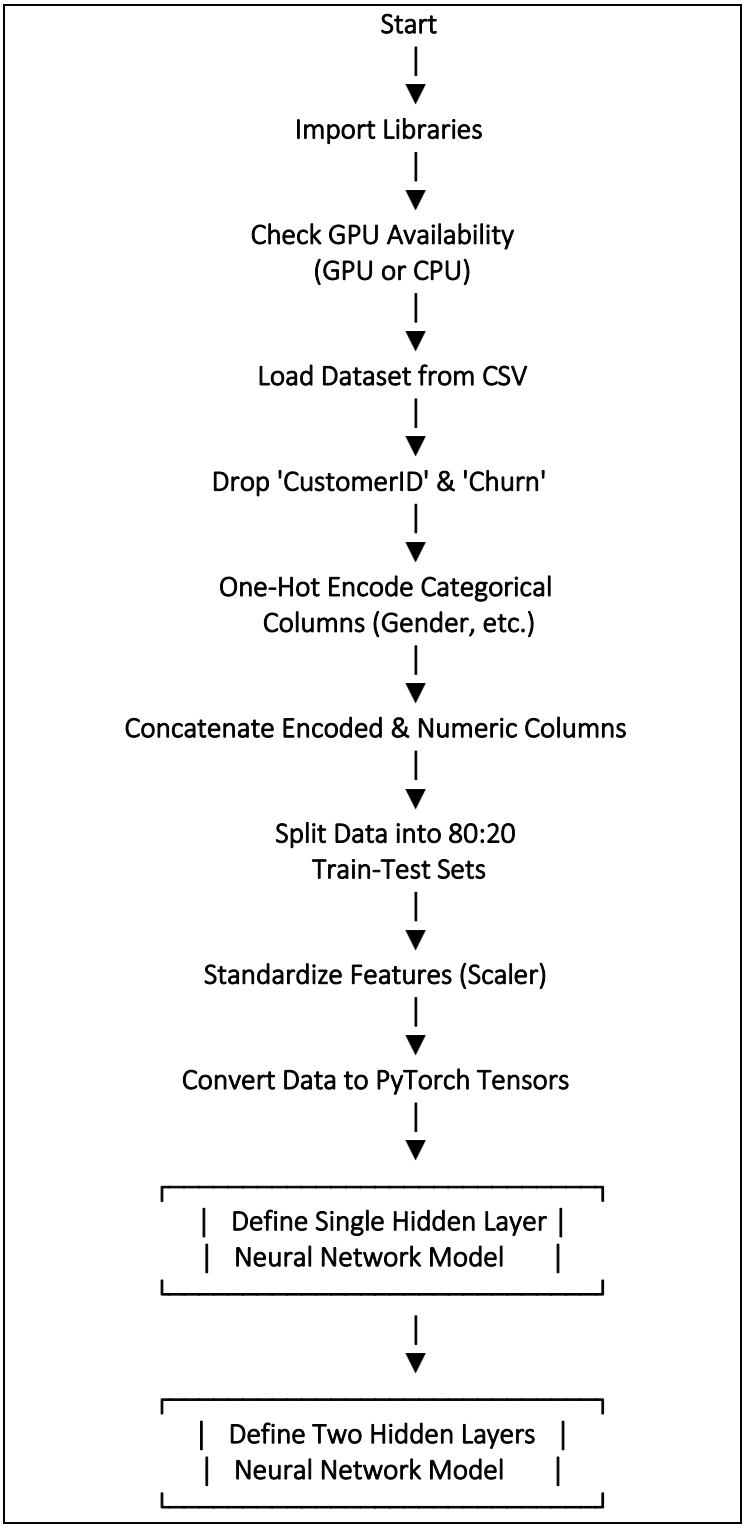
- Training Steps:
  - Forward Pass: Pass the input data through the model to get predictions.
  - Calculate Loss: Compute the loss using the actual and predicted labels.
  - Backpropagation: Calculate the gradients for each parameter.
  - Optimization: Update the model parameters using the optimizer.

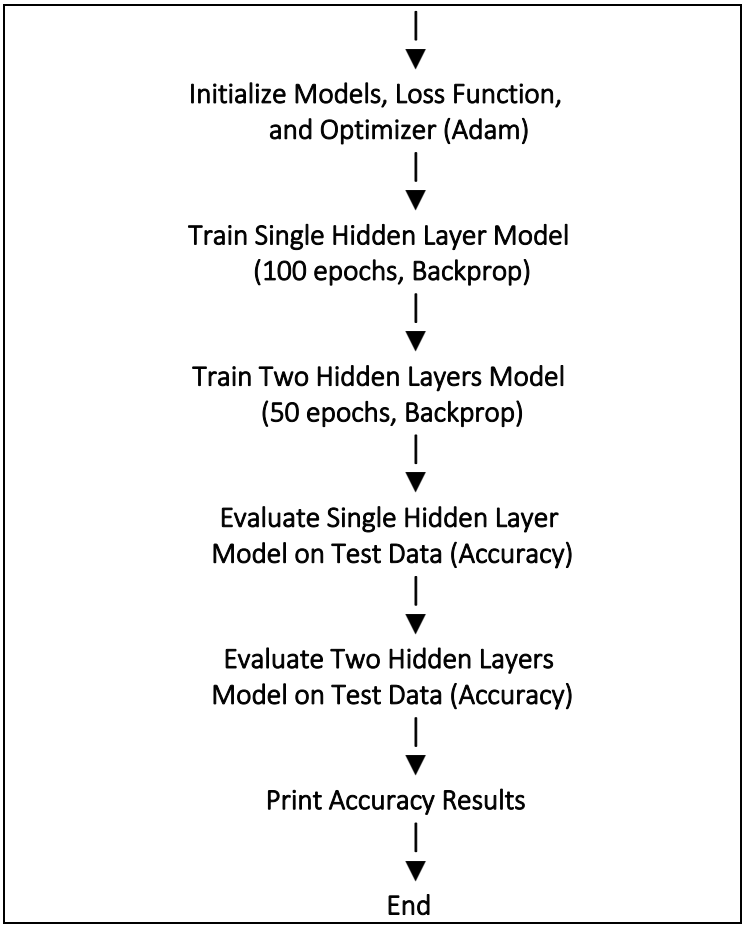
Step 9: Evaluate Models

- Description: Evaluate both models using the test set. Round predictions and compare them with the actual labels to calculate accuracy.

Step 10: Print Results

- Description: Print the accuracy of both models (one hidden layer vs. two hidden layers).





Important Key Analysis Points

- GPU vs. CPU Utilization:
  - The code is designed to detect and utilize GPU if available, which greatly enhances performance, especially when training larger models or datasets. However, if no GPU is available, the code defaults to the CPU.
- Data Preprocessing:
  - Preprocessing is essential for getting the data into a format suitable for neural network training. One-hot encoding is used for categorical features, which is a common approach when dealing with non-numeric data. Standardization ensures that the data has consistent scale, which helps in faster convergence during training.
- Model Architecture:
  - The code uses two neural network models: one with a single hidden layer and another with two hidden layers. The single-layer model is simpler and faster, while the two-layer model introduces more complexity, allowing it to potentially learn more intricate patterns in the data.
  - Both models use the ReLU activation function in hidden layers, which is a widely used non-linear activation function that helps the network learn complex patterns. The final layer uses a Sigmoid activation function, which is appropriate for binary classification tasks as it outputs probabilities between 0 and 1.
- Training Process:
  - Training is done in epochs (100 for the single hidden layer model and 50 for the two hidden layers model). For each epoch, a forward pass is made through the network, the loss is computed, and gradients are backpropagated to update the model's parameters.
- Model Evaluation:
  - After training, both models are evaluated on the test dataset to assess their generalization performance. Accuracy is calculated by comparing the predicted and actual labels.
- Comparison of Models:

- The design of the code allows for an easy comparison between a simpler and more complex architecture. The two hidden-layer model may potentially perform better due to its higher capacity to learn complex patterns, but this comes at the cost of increased training time and computational requirements.

Through Pseudo Code

```
BEGIN
  IMPORT necessary libraries (pandas, torch, sklearn, torch.nn, torch.optim)

  IF GPU is available THEN
    SET device to GPU
    PRINT "Using GPU"
  ELSE
    SET device to CPU
    PRINT "Using CPU"
  END IF

  # Load Dataset
  LOAD 'customer_churn_dataset-testing-master.csv' into DataFrame (df)

  # Data Preprocessing
  DROP 'CustomerID' and 'Churn' columns from DataFrame
  SET features to all columns except 'CustomerID' and 'Churn'
  SET labels to 'Churn' column

  IDENTIFY categorical columns: ['Gender', 'Subscription Type', 'Contract Length']

  # One-Hot Encoding
  INITIALIZE OneHotEncoder
  APPLY OneHotEncoder to categorical columns
  CONVERT encoded categorical features into DataFrame
  CONCATENATE encoded categorical features with original numerical features

  # Split the Data
  SPLIT dataset into 80% training (X_train, y_train) and 20% testing (X_test, y_test)

  # Feature Scaling
  INITIALIZE StandardScaler
  FIT StandardScaler to training data and TRANSFORM it
  TRANSFORM testing data with the fitted scaler

  # Convert Data to PyTorch Tensors
  CONVERT X_train, X_test, y_train, y_test to PyTorch tensors
  MOVE tensors to device (GPU/CPU)

  # Reshape labels (y_train, y_test) to match PyTorch's expectations

  # Model Definition: Single Hidden Layer Neural Network
  DEFINE class 'ChurnPredictorOneHiddenLayer'
    INITIALIZE the hidden layer with input size = number of features, output size = 4096
    INITIALIZE the output layer with input size = 4096, output size = 1
    DEFINE forward function
      PASS input through hidden layer with ReLU activation
      PASS result through output layer with Sigmoid activation
```

```
    END forward function
END class 'ChurnPredictorOneHiddenLayer'

# Initialize Single Layer Model
INITIALIZE 'ChurnPredictorOneHiddenLayer' model
SET loss function to Binary Cross Entropy Loss (BCELoss)
SET optimizer to Adam optimizer with learning rate 0.001

# Train the Model with Single Hidden Layer
FOR each epoch from 1 to 100 DO
    ZERO the gradients in the optimizer
    PASS training data through the model (forward pass)
    COMPUTE the loss
    BACKPROPAGATE the loss
    UPDATE the model parameters using the optimizer
    IF current epoch is a multiple of 10 THEN
        PRINT current epoch and loss value
    END IF
END FOR

# Model Definition: Two Hidden Layers Neural Network
DEFINE class 'ChurnPredictorTwoHiddenLayers'
    INITIALIZE first hidden layer with input size = number of features, output size = 4096
    INITIALIZE second hidden layer with input size = 4096, output size = 64
    INITIALIZE output layer with input size = 64, output size = 1
    DEFINE forward function
        PASS input through first hidden layer with ReLU activation
        PASS result through second hidden layer with ReLU activation
        PASS result through output layer with Sigmoid activation
    END forward function
END class 'ChurnPredictorTwoHiddenLayers'

# Initialize Two Layers Model
INITIALIZE 'ChurnPredictorTwoHiddenLayers' model
SET optimizer to Adam optimizer with learning rate 0.001

# Train the Model with Two Hidden Layers
FOR each epoch from 1 to 50 DO
    ZERO the gradients in the optimizer
    PASS training data through the model (forward pass)
    COMPUTE the loss
    BACKPROPAGATE the loss
    UPDATE the model parameters using the optimizer
    IF current epoch is a multiple of 10 THEN
        PRINT current epoch and loss value
    END IF
END FOR

# Evaluation Function to Calculate Accuracy
DEFINE function 'evaluate_model' with parameters: model, X_test_tensor, y_test_tensor
    SET model to evaluation mode
    DISABLE gradient calculation
    PASS test data through the model (forward pass)
```

```
    ROUND predictions to 0 or 1
    CALCULATE accuracy by comparing predictions to true labels
    RETURN accuracy
  END function

  # Evaluate the Single Hidden Layer Model
  CALL 'evaluate_model' on the single hidden layer model using test data
  PRINT accuracy for single hidden layer model

  # Evaluate the Two Hidden Layers Model
  CALL 'evaluate_model' on the two hidden layers model using test data
  PRINT accuracy for two hidden layers model

END
```

- Data Preprocessing:
  - One-hot encode categorical columns to convert them into numerical values.
  - Scale numerical features to ensure proper model convergence.
- Model Architecture:
  - Two neural network architectures are defined: one with a single hidden layer, and another with two hidden layers.
- Training Process:
  - Training is done in loops (epochs), where each iteration involves a forward pass through the model, computing the loss, and updating the model's parameters using backpropagation.
- Evaluation:
  - After training, the models are evaluated on the test dataset to compute accuracy.

This pseudocode focuses on abstracting away specific Python and PyTorch syntax, while still maintaining the logical flow and operations carried out by the code.

Through python Code.

Data Preprocessing

Before training the model, the dataset must be processed to ensure the inputs are in a format that can be fed into the neural network.

```
file_path = 'customer_churn_dataset-testing-master.csv'
df = pd.read_csv(file_path)
```

- Purpose: Loads the customer churn dataset into a Pandas DataFrame.
- Analysis: This allows for easy manipulation and preprocessing of the dataset.

Dropping Irrelevant Columns:

```
features = df.drop(columns=['CustomerID', 'Churn'])
labels = df['Churn']
```

- Purpose: CustomerID is a unique identifier and irrelevant for predictions, while Churn is the target label. This step ensures that the model only gets the features that will help in prediction.
- Analysis: Helps prevent noise in the data by removing columns that are unnecessary for training.

Categorical Encoding:

```
categorical_columns = ['Gender', 'Subscription Type', 'Contract Length']
encoder = OneHotEncoder(sparse_output=False)
encoded_features = encoder.fit_transform(features[categorical_columns])
encoded_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(categorical_columns))
```

Purpose: One-hot encodes categorical variables to convert them into a format suitable for the neural network.

Analysis: Neural networks cannot directly work with categorical variables, so one-hot encoding is essential to transform them into numeric form.

Combining Encoded Features:

```
features_encoded = pd.concat([features.drop(columns=categorical_columns).reset_index(drop=True), encoded_df], axis=1)
```

- Purpose: Combines the encoded categorical features with the rest of the numeric features to form a complete input dataset.
- Analysis: Prepares the final dataset for splitting into training and testing sets.

Splitting the Data:

```
X_train, X_test, y_train, y_test = train_test_split(features_encoded, labels, test_size=0.2, random_state=42)
```

- Purpose: Splits the data into training (80%) and testing (20%) sets.
- Analysis: This ensures that the model can be trained on one portion of the data and evaluated on unseen data (test set) to gauge its generalization ability

Feature Scaling:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Purpose: Scales the input features to have zero mean and unit variance.
- Analysis: Standardizing features is crucial for neural networks because it ensures that all features contribute equally to the model and prevents any one feature from dominating due to its scale.

Building the Neural Network Models (Single Hidden Layer Model):

```
class ChurnPredictorOneHiddenLayer(nn.Module):
    def __init__(self):
        super(ChurnPredictorOneHiddenLayer, self).__init__()
        self.hidden = nn.Linear(X_train_tensor.shape[1], 4096)
        self.output = nn.Linear(4096, 1)

    def forward(self, x):
        x = torch.relu(self.hidden(x))
        x = torch.sigmoid(self.output(x))
        return x
```

- Purpose: Defines a simple neural network with one hidden layer of 4096 neurons.
- Activation Function: ReLU is used for the hidden layer, which introduces non-linearity, while Sigmoid is used for the output layer to return probabilities between 0 and 1 for binary classification.
- Analysis: This structure allows the model to learn complex non-linear relationships between features.

Two Hidden Layer Model:

```
class ChurnPredictorTwoHiddenLayers(nn.Module):
    def __init__(self):
        super(ChurnPredictorTwoHiddenLayers, self).__init__()
        self.hidden1 = nn.Linear(X_train_tensor.shape[1], 4096)
        self.hidden2 = nn.Linear(4096, 64)
        self.output = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = torch.sigmoid(self.output(x))
        return x
```

- Purpose: Defines a more complex neural network with two hidden layers (4096 neurons and 64 neurons, respectively).
- Analysis: Adding another hidden layer enables the model to potentially capture more complex patterns and relationships in the data, as deep neural networks tend to perform better when more depth is added.

Training Process

For both models, the training process involves iterating through the dataset multiple times (epochs), calculating the loss, and adjusting the model's weights to minimize the loss.

Training Loop for Single Hidden Layer Model:

```
num_epochs = 100
for epoch in range(num_epochs):
    model_one_layer.train()
    optimizer.zero_grad()
    outputs = model_one_layer(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()
    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

- Forward Pass: The training data is passed through the network to obtain predictions.
- Loss Calculation: The loss between predictions and actual values is computed using Binary Cross-Entropy Loss (BCELoss), which is suitable for binary classification.
- Backpropagation: The loss is propagated backward through the network, and the model's parameters (weights) are updated using the Adam optimizer.
- Epochs: The model is trained for 100 epochs, meaning it sees the entire dataset 100 times.

Training Loop for Two Hidden Layer Model:

```
num_epochs = 50
for epoch in range(num_epochs):
    model_two_layers.train()
    optimizer.zero_grad()
    outputs = model_two_layers(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()
    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

- Purpose: The evaluation function computes the model's accuracy by comparing predicted labels to actual labels in the test set.
- No Gradient Computation: In evaluation mode, the model doesn't calculate gradients, making the process more efficient.

Model Accuracy:

```
accuracy_one_layer = evaluate_model(model_one_layer, X_test_tensor, y_test_tensor)
accuracy_two_layers = evaluate_model(model_two_layers, X_test_tensor, y_test_tensor)
```

- Comparison: The accuracy of both models is calculated and printed for comparison.

Analysis of the Model's Performance

- Single Hidden Layer vs. Two Hidden Layers:
  - Single Hidden Layer: Trained for 100 epochs with 4096 neurons in one hidden layer.
  - Two Hidden Layers: Trained for 50 epochs with two hidden layers (4096 and 64 neurons).
- Expectation:
  - The two-layer model may have a better ability to learn more complex patterns in the data due to its additional depth.
  - However, with only 50 epochs, it might not fully realize its potential compared to the single-layer model trained for 100 epochs.
- Trade-offs:
  - Training Time: The two-layer model has more parameters, so training might be slower.
  - Accuracy: The deeper model is expected to generalize better, but this is not guaranteed if overfitting occurs or if it's not trained long enough.

Conclusion

- Data Preprocessing: The code effectively preprocesses the dataset by handling categorical features and standardizing numeric data, crucial for neural network training.
- Model Structure: Both models demonstrate different architectures (one hidden layer vs. two hidden layers) and show how deep learning models can be adjusted to improve performance.

- **Training and Evaluation:** The training process with gradient descent and backpropagation ensures that the models learn patterns from the training data, and evaluation on the test set provides insights into generalization ability.

The model with two hidden layers may provide better performance due to its deeper structure, but sufficient training epochs and tuning are essential to harness its potential.

Limitation

1. Lack of Hyperparameter Tuning
   - **Description:** The neural network models use fixed values for important hyperparameters such as the number of neurons in the hidden layers, the learning rate, and the number of epochs.
   - **Limitation:** Without hyperparameter tuning, the model may not achieve optimal performance. Different values for the number of neurons, learning rate, or even the number of layers could yield better results.
   - **Solution:** Consider using techniques like grid search, random search, or more advanced approaches such as Bayesian optimization to find the best combination of hyperparameters.
2. Fixed Number of Epochs
   - **Description:** The model is trained for a fixed number of epochs (100 epochs for the single-layer model, 50 epochs for the two-layer model).
   - **Limitation:** A fixed number of epochs can lead to overfitting (training too much) or underfitting (not training enough). The model may not have learned sufficiently or might have learned too much, impacting generalization on unseen data.
   - **Solution:** Implement early stopping to monitor the validation loss and stop training once the loss stops improving, which helps prevent overfitting or underfitting.
3. No Cross-Validation
   - **Description:** The code uses a single split between training and testing data (80% for training, 20% for testing).
   - **Limitation:** A single train-test split can lead to biased results because the model may perform well on one split but poorly on another due to variance in the data.
   - **Solution:** Cross-validation (e.g., k-fold cross-validation) should be implemented to ensure that the model's performance is consistent across different subsets of the data. This helps give a more robust estimate of model performance.
4. Limited Model Complexity
   - **Description:** The models only use simple feedforward neural networks with ReLU activations for hidden layers and Sigmoid activation for the output.
   - **Limitation:** The architecture may not be complex enough to capture all relationships in the data, especially if the dataset has a large number of features and complex interactions.
   - **Solution:** Consider more complex architectures such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), or using dropout layers to prevent overfitting. Additionally, experimenting with more hidden layers and different types of activation functions might yield better results.
5. Handling of Imbalanced Data
   - **Description:** The code doesn't explicitly address the issue of class imbalance, which is common in churn prediction (where the number of customers who churn is much smaller than those who don't).
   - **Limitation:** Imbalanced data can lead to the model being biased towards the majority class (non-churn customers), resulting in poor performance in predicting churn.
   - **Solution:** Techniques such as oversampling the minority class (churn customers), undersampling the majority class, or using class weights during training can help mitigate this problem. Alternatively, using more sophisticated loss functions like focal loss can also be helpful in handling class imbalance.
6. Categorical Encoding Could Be Improved
   - **Description:** The code uses one-hot encoding to handle categorical variables.
   - **Limitation:** One-hot encoding can lead to a high-dimensional feature space, which might make the model prone to overfitting, especially if there are many categories.
   - **Solution:** Instead of one-hot encoding, you can use embedding layers, which map categorical variables to a lower-dimensional continuous space. This approach is particularly useful for high-cardinality categorical features.
7. No Model Interpretation or Explainability

- **Description:** The model provides no insight into why it makes certain predictions.
- **Limitation:** Neural networks are often seen as "black box" models because they don't easily provide explanations for their decisions, which can be problematic in fields like customer churn, where understanding the drivers of churn is crucial.
- **Solution:** Consider using techniques like SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-agnostic Explanations) to provide interpretability for the model's predictions. This can help stakeholders understand which features are most important for predicting churn.

8. No Regularization to Prevent Overfitting
   - **Description:** The code doesn't include any regularization techniques like L2 regularization or dropout layers.
   - **Limitation:** Neural networks are prone to overfitting, especially with small datasets or datasets with a large number of features. The absence of regularization increases this risk.
   - **Solution:** Add dropout layers or apply L2 regularization (weight decay) to reduce the model's capacity to overfit the training data.
9. Lack of Data Augmentation
   - **Description:** There is no mechanism to augment the dataset or generate synthetic data.
   - **Limitation:** For smaller datasets, the model may not have enough data to learn from, leading to overfitting or poor generalization.
   - **Solution:** Techniques like SMOTE (Synthetic Minority Over-sampling Technique) can be used to artificially increase the size of the minority class (churn customers) and help the model generalize better.
10. No Evaluation Metrics Beyond Accuracy
    - **Description:** The evaluation of the model is based on accuracy alone.
    - **Limitation:** Accuracy can be misleading, especially for imbalanced datasets where predicting the majority class (non-churn) most of the time might yield high accuracy but poor results for the minority class (churn).
    - **Solution:** Use additional metrics such as precision, recall, F1-score, and AUC-ROC (Area Under the Receiver Operating Characteristic Curve) to get a more complete picture of model performance, especially for class imbalances.
11. No Post-Processing for Threshold Tuning
    - **Description:** The code applies a threshold of 0.5 to the Sigmoid output to classify whether a customer will churn.
    - **Limitation:** Using a fixed threshold might not be optimal, especially if you care more about minimizing false negatives (missing churn customers) or false positives (incorrectly predicting churn).
    - **Solution:** Perform threshold tuning by analyzing precision-recall trade-offs or using ROC curves to find the best threshold for your specific problem.

12. Absence of Feature Selection
    - **Description:** The code doesn't include any feature selection or importance ranking process.
    - **Limitation:** Including irrelevant or redundant features may lead to a model that is unnecessarily complex and prone to overfitting.
    - **Solution:** Apply feature selection techniques like Recursive Feature Elimination (RFE) or use feature importance metrics from tree-based models to identify and remove unimportant features.

And finally, while the provided Python code for churn prediction is a good starting point, it can be improved in multiple areas. Addressing the limitations mentioned—such as hyperparameter tuning, handling imbalanced data, and incorporating more robust evaluation metrics—can significantly improve the performance, interpretability, and scalability of the model.

Strengths

Despite the limitations, there are several strong aspects of the given code. Below are the key strengths that highlight the best practices and well-structured design elements of the neural network-based churn prediction model:

1. Use of PyTorch for Neural Networks
   - **Description:** The code leverages PyTorch, a highly popular deep learning framework.
   - **Strength:** PyTorch provides flexibility and efficiency in building and training neural networks. It allows the model to be easily transferred between CPU and GPU (MPS support), and it offers dynamic computation graphs, making it an ideal choice for research and production-level tasks.

2. Device Compatibility (CPU/GPU Selection)
   - **Description:** The code checks if a Metal Performance Shaders (MPS) backend is available (for Apple GPUs) and automatically chooses between CPU and GPU.
   - **Strength:** The dynamic selection of computing devices ensures that the model can take advantage of available hardware, optimizing training speed and performance on machines equipped with GPUs.
3. Neural Network Flexibility
   - **Description:** The code provides two different architectures: a neural network with one hidden layer and another with two hidden layers.
   - **Strength:** This flexibility allows for experimenting with different neural network structures, which can provide insight into how adding more complexity (e.g., additional layers) affects the model's performance.
4. Use of ReLU and Sigmoid Activation Functions
   - **Description:** The hidden layers use the ReLU activation function, while the output layer uses the Sigmoid activation function.
   - **Strength:** ReLU is effective in avoiding the vanishing gradient problem and helps models learn faster and more accurately. Sigmoid activation in the output layer is a natural choice for binary classification tasks, like churn prediction, as it outputs values between 0 and 1, making it easy to interpret as probabilities.
5. One-Hot Encoding for Categorical Variables
   - **Description:** The code correctly applies one-hot encoding to transform categorical columns ('Gender,' 'Subscription Type,' 'Contract Length') into numerical values.
   - **Strength:** One-hot encoding is a simple yet effective method for handling categorical variables, especially when working with machine learning models that require numerical input. It ensures that categorical variables are properly integrated into the model without introducing any ordinal relationships.
6. Standardization of Features
   - **Description:** The code uses StandardScaler to standardize the features before feeding them into the neural network.
   - **Strength:** Standardization ensures that all features contribute equally to the learning process by giving them similar ranges. This is particularly important for neural networks, as it helps speed up convergence during training and improves model performance.
7. Use of BCELoss for Binary Classification
   - **Description:** The code uses Binary Cross-Entropy Loss (BCELoss), which is well-suited for binary classification problems like predicting churn.
   - **Strength:** Binary Cross-Entropy is the ideal loss function for binary classification tasks because it penalizes wrong predictions in a manner that reflects the probability of an outcome, making it optimal for training neural networks in this context.
8. Efficient Training Loop with Adam Optimizer
   - **Description:** The training loop uses the Adam optimizer with a learning rate of 0.001.
   - **Strength:** Adam is a highly efficient optimization algorithm that combines the best aspects of both momentum-based and RMSProp optimizers. It adapts the learning rate based on the magnitude of gradients, which helps with faster convergence and less manual tuning of learning rates.
9. Model Evaluation Function
   - **Description:** The code includes an evaluation function that computes accuracy by comparing predictions to ground truth labels.
   - **Strength:** Having an evaluation function built into the code allows for an easy comparison of model performance. The use of .eval() in PyTorch ensures the model is set in inference mode, which is essential for accurate evaluation as it turns off dropout layers and batch normalization, if used.
10. Structured Workflow: Data Preprocessing, Model Definition, Training, and Evaluation
    - **Description:** The code follows a clear structure: it preprocesses the data, defines the model architecture, trains the model, and evaluates performance.
    - **Strength:** This structured workflow makes it easy to follow the steps of the process, ensuring modularity and separation of concerns. Each part of the pipeline (data preprocessing, model training, evaluation) can be modified or improved without affecting the other parts.
11. Two Models for Comparative Analysis
    - **Description:** The code defines two different neural network architectures (one hidden layer and two hidden layers) and compares their performance.

- **Strength:** This comparative approach helps in understanding how the complexity of the neural network impacts performance. It also allows the user to experiment with different architectures and choose the best model based on real-world data.

12. Tensor Conversion for PyTorch Compatibility
    - **Description:** The features and labels are correctly converted into PyTorch tensors before being passed into the neural network.
    - **Strength:** This ensures that the data can be processed in a way that is compatible with the PyTorch framework. The use of .to(device) ensures that the tensors are loaded onto the correct computing device, maximizing computational efficiency.
13. Use of Sigmoid Output for Probability Interpretation
    - **Description:** The code applies a Sigmoid function in the output layer, which returns values between 0 and 1, representing the probability of churn.
    - **Strength:** This is appropriate for binary classification, allowing for intuitive interpretation of the model's output as probabilities, which is useful for understanding customer churn risk.
14. Efficient Memory Management
    - **Description:** The code uses the .detach() function for evaluation to ensure no computation graphs are stored unnecessarily during inference.
    - **Strength:** Efficient memory management in PyTorch helps avoid memory leaks and keeps GPU usage low during the testing and evaluation phase, making the model more scalable for larger datasets.
15. Custom Layer Definitions
    - **Description:** The neural network models are defined using custom classes (ChurnPredictorOneHiddenLayer and ChurnPredictorTwoHiddenLayers), which makes it easy to modify or extend the architecture if needed.
    - **Strength:** The use of custom layers enables flexibility. For instance, adding more layers, changing the number of neurons, or experimenting with different activation functions can be done quickly within the class structure.

Overall, the code demonstrates a well-structured and efficient approach to building a neural network for customer churn prediction. It uses PyTorch's flexibility for creating custom architectures, performs essential preprocessing steps, and includes comparative evaluation. The balance of simplicity and flexibility makes it a solid foundation for further experimentation or extension into more complex models.

Running Output

100 Epochs Output

```
Epoch [10/100], Loss: 0.3772
Epoch [20/100], Loss: 0.3461
Epoch [30/100], Loss: 0.3130
Epoch [40/100], Loss: 0.2909
Epoch [50/100], Loss: 0.2751
Epoch [60/100], Loss: 0.2603
Epoch [70/100], Loss: 0.2485
Epoch [80/100], Loss: 0.2387
Epoch [90/100], Loss: 0.2306
Epoch [100/100], Loss: 0.2233
```

50 Epochs Code double layer

```
Epoch [10/50], Loss: 0.3677
Epoch [20/50], Loss: 0.2931
Epoch [30/50], Loss: 0.2602
Epoch [40/50], Loss: 0.2388
Epoch [50/50], Loss: 0.2231
```

Final

- Accuracy with one hidden layer:
- Accuracy with two hidden layers:

```
Accuracy with one hidden layer: 0.9031
Accuracy with two hidden layers: 0.9035
```

Conclusion

Problem Statement:

The goal of this project is to predict customer churn using a neural network. The model attempts to classify whether a customer will churn (leave the service) or not based on features such as customer demographics, subscription types, and contract lengths. Given the importance of customer retention for businesses, accurately predicting churn can have significant financial and operational benefits.

Code Summary:

The code is a well-structured implementation of two neural network models using PyTorch to solve the binary classification problem (churn vs. no churn). The workflow includes data preprocessing, model definition, training, and evaluation:

- **Data Preprocessing:** The categorical variables are encoded using OneHotEncoder, and the numerical features are standardized using StandardScaler. This ensures all data is in a suitable format for the neural network.
- **Model Architectures:** Two neural network models are defined — one with a single hidden layer and the other with two hidden layers. These architectures are evaluated on their performance in predicting churn.
- **Training:** The models are trained using Binary Cross-Entropy loss (BCELoss) and optimized with the Adam optimizer. Both models undergo multiple epochs of training.
- **Evaluation:** After training, the models are evaluated based on accuracy to compare their performance in predicting customer churn.

Strengths of the Code:

- **Device Flexibility:** The code adapts to either CPU or GPU for training, optimizing computational efficiency.
- **Neural Network Design:** The code includes two different architectures, enabling comparative analysis between simpler and more complex models.
- **Efficient Preprocessing:** One-hot encoding of categorical variables and feature standardization ensure the data is well-prepared for neural network training.
- **Training Optimization:** The use of Adam as an optimizer is efficient for minimizing loss, and the models are trained using binary cross-entropy, suitable for binary classification tasks.
- **Modularity:** The code is structured in a clear manner, making it easy to modify individual components (e.g., the model architecture or training loop) without affecting the entire pipeline.

Limitations of the Code:

- **Dataset Size and Complexity:** The model might not generalize well to more complex or larger datasets. The hidden layers and neuron numbers can be further optimized based on cross-validation techniques.
- **Lack of Hyperparameter Tuning:** No tuning of hyperparameters like learning rate, batch size, or number of neurons is included, which can lead to suboptimal performance.
- **Evaluation Metrics:** Only accuracy is used for evaluation, but for imbalanced datasets like churn prediction, other metrics (precision, recall, F1-score, AUC-ROC) would provide a more complete picture of model performance.
- **Overfitting Risk:** The high number of neurons in hidden layers (4096) for such a small dataset could lead to overfitting, where the model performs well on training data but poorly on unseen data.

Overall Conclusion:

The code offers a strong foundation for customer churn prediction using neural networks. The modular design, flexibility with CPU/GPU, and implementation of two distinct architectures make it a robust starting point for developing more advanced models. However, there is room for improvement in terms of hyperparameter tuning, dataset complexity, and adding more appropriate evaluation metrics to capture the model's real-world performance. By addressing the limitations (e.g., implementing cross-validation, introducing regularization, exploring more evaluation metrics), this code can be further optimized to provide better churn prediction capabilities and insights into customer behaviour.


Thank
You