python_code

December 14, 2024

[ ]:
```
from datasets import load_dataset
from torch.utils.data import DataLoader, Dataset, random_split
import os
import yaml
import torch
import json
from torch.utils.data import DataLoader, Dataset, random_split
from torch import nn
from transformers import AutoTokenizer
import pickle
import re
from torch.nn.utils.rnn import pad_sequence
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
import subprocess
from collections import Counter
```

[30]:
```
def get_data():
    raw_data = load_dataset("codeparrot/codecomplex", split="train",␣
 ↪cache_dir='data')
    if not os.path.exists('data/dataset.json'):
        raw_data.to_json('data/dataset.json')
    return raw_data
```

[ ]:
```
complexity_classes = ['logn', 'cubic', 'linear', 'nlogn', 'quadratic', 'np',␣
 ↪'constant']

padding_idx = 2
```

[ ]:
```
def load_ast(file_path="data/ast_dataset.yaml"):
    with open(file_path, 'r', errors='ignore') as f:
        d = yaml.safe_load(f)
    return [{'src': entry['src'], 'ast': entry['ast'], 'complexity':␣
 ↪complexity_classes.index(entry['complexity'])} for entry in d.values()]
```

[ ]:
```
class CustomTokenizer:
    def __init__(self):
        self.oov_count = 0
        self.vocab = self._build_vocab()

    def _build_vocab(self):
        vocab = {'<UNK>': 0, '<OOV>': 1, '<PAD>': 2}
        return vocab

    def fit(self, dataset, label):
        for item in dataset:
            tokens = self.split(item[label])
            for token in tokens:
                if token not in self.vocab:
                    self.vocab[token] = len(self.vocab)

    def split(self, text):
        SPLIT_REGEX = r'[a-zA-Z0-9_]+|[^\w\s]'
        return re.findall(SPLIT_REGEX, text)

    def tokenize(self, text):
        tokens = self.split(text)
        token_ids = [self.vocab.get(token,'<OOV>') for token in tokens]


        return token_ids

    def transform(self, text):
        return self.tokenize(text)
```

[5]:
```
class TorchDataset(Dataset):
    def __init__(self, data, tokenizer: CustomTokenizer, max_length=None,␣
 ↪data_label='src'):
        self.data = []
        self.tokenizer = tokenizer
        self.max_length = max_length

        for item in data:
            data = item[data_label]
            label = item['complexity']
            token_ids = self.tokenizer.transform(data)

            if self.max_length:
                token_ids = token_ids[:self.max_length]

            token_tensor = torch.tensor(token_ids, dtype=torch.long)
            label_tensor = torch.tensor(label, dtype=torch.long)
```

```
            self.data.append((token_tensor, label_tensor))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

[ ]:
```
def collate_fn(batch):
    inputs, labels = zip(*batch)

    padded_inputs = pad_sequence(inputs, batch_first=True,␣
 ↪padding_value=padding_idx)

    attention_mask = (padded_inputs != padding_idx).float()

    return padded_inputs, attention_mask, torch.tensor(labels)
```

[ ]:
```
class LSTMModel(nn.Module):
    def __init__(self,
                 vocab_size,
                 embedding_dim,
                 hidden_size,
                 output_size,
                 num_layers,
                 padding_idx=padding_idx,
                 dropout_rate=0.5,  # Added dropout rate
                 batch_first=True):
        super(LSTMModel, self).__init__()

        self.class_labels = complexity_classes
        self.padding_idx = padding_idx
        self.embedding = nn.Embedding(vocab_size, embedding_dim,␣
 ↪padding_idx=padding_idx)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, num_layers,
                            batch_first=batch_first, dropout=dropout_rate if␣
 ↪num_layers > 1 else 0.0)
        self.fc = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout_rate)  # Add dropout layer

    def forward(self, x, mask):
        embedded = self.embedding(x)
        embedded = self.dropout(embedded)  # Apply dropout after embedding

        lengths = mask.sum(dim=1).cpu()
        packed_input = pack_padded_sequence(embedded, lengths,␣
 ↪batch_first=True, enforce_sorted=False)
```

```
        packed_output, (hn, cn) = self.lstm(packed_input)
        output, _ = pad_packed_sequence(packed_output, batch_first=True)

        final_output = hn[-1]
        final_output = self.dropout(final_output)  # Apply dropout before the␣
↪fully connected layer
        output = self.fc(final_output)
        return output

    def predict(self, x):
        self.eval()
        with torch.no_grad():
            mask = (x != self.padding_idx).float()
            output = self(x, mask)
        probabilities = torch.softmax(output, dim=1)
        _, predicted_class = torch.max(probabilities, dim=1)
        return self.class_labels[predicted_class]
```

[ ]:

```
def evaluate(model, test_loader, criterion, device_str='cuda'):
    device = torch.device("cuda" if torch.cuda.is_available() and device_str ==␣
↪'cuda' else "cpu")
    model.eval()  # Set model to evaluation mode

    epoch_loss = 0.0
    correct_predictions = 0

    total_samples = 0

    true_labels = []
    predictions = []

    with torch.no_grad():  # No need to track gradients during evaluation
        for inputs, attention_mask, labels in test_loader:
            if device_str == 'cuda':
                inputs = inputs.to(device)
                attention_mask = attention_mask.to(device)
                labels = labels.to(device)

            outputs = model(inputs, attention_mask)

            labels = labels.long()  # Ensure labels are in the right format
            loss = criterion(outputs, labels)

            batch_size = labels.size(0)
            epoch_loss += loss.item() * batch_size  # Weighted by batch size
            correct_predictions += (torch.argmax(outputs, dim=1) == labels).
↪sum().item()
```

```
            total_samples += batch_size

            _, predicted = torch.max(outputs, 1)  # Get predicted class labels
            true_labels.extend(labels.cpu().numpy())
            predictions.extend(predicted.cpu().numpy())


    # Calculate average loss and accuracy
    epoch_train_loss = epoch_loss / total_samples
    epoch_train_accuracy = correct_predictions / total_samples



    return epoch_train_loss, epoch_train_accuracy, true_labels, predictions
```

[ ]:
```
def train(model, train_loader, test_loader, criterion, optimizer,␣
↪scheduler=None, num_epochs=10, patience=5, device_str='cuda'):

    device = torch.device("cuda" if torch.cuda.is_available() and device_str ==␣
↪'cuda' else "cpu")

    best_loss = float('inf')
    no_improve_epochs = 0

    if device_str == 'cuda':
        model.to(device)


    train_losses = []
    train_accuracies = []
    test_losses = []
    test_accuracies = []

    for epoch in range(num_epochs):

        model.train()
        epoch_loss = 0.0
        correct_predictions = 0
        total_samples = 0

        for i, (inputs, attention_mask, labels) in enumerate(train_loader):
            print(f"\rBatch {i}", sep='', end='')

            if device_str == 'cuda':
                inputs = inputs.to(device)
                attention_mask = attention_mask.to(device)
                labels = labels.to(device)
```

```
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs, attention_mask)

            # Calculate loss
            labels = labels.long()
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Update metrics
            batch_size = labels.size(0)
            epoch_loss += loss.item() * batch_size
            correct_predictions += (torch.argmax(outputs, dim=1) == labels).
↪sum().item()
            total_samples += batch_size

        if scheduler:
            scheduler.step()

        epoch_train_loss = epoch_loss / total_samples
        epoch_train_accuracy = correct_predictions / total_samples

        train_losses.append(epoch_train_loss)
        train_accuracies.append(epoch_train_accuracy)

        test_loss, test_accuracy, _, _ = evaluate(model, test_loader,␣
↪criterion, device_str=device_str)

        test_losses.append(test_loss)
        test_accuracies.append(test_accuracy)

        print(f'\rEpoch:{epoch+1}|Train Loss:{epoch_train_loss:.4f}|Train␣
↪Accuracy:{epoch_train_accuracy:.2f}|Test Loss:{test_loss:.4f}|Test Accuracy:
↪{test_accuracy:.2f}\n', sep='', end='')

        if test_loss < best_loss:
            best_loss = test_loss
            no_improve_epochs = 0
            best_model_state = model.state_dict()  # Save the best model
        else:
            no_improve_epochs += 1
            print(f"No improvement for {no_improve_epochs} epoch(s)")

        if no_improve_epochs >= patience:
```

```
            print("Early stopping triggered!")
            model.load_state_dict(best_model_state)  # Restore the best model
            break


    return model, train_losses, train_accuracies, test_losses, test_accuracies,␣
↪epoch
```

[69]:
```
def plot_metrics(train_losses, train_accuracies, test_losses, test_accuracies):
    epochs = range(1, len(train_losses) + 1)

    # Plot training and testing loss
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)  # First subplot: Loss
    plt.plot(epochs, train_losses, label='Train Loss', color='blue', marker='o')
    plt.plot(epochs, test_losses, label='Test Loss', color='red', marker='o')
    plt.title('Train and Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    # Plot training and testing accuracy
    plt.subplot(1, 2, 2)  # Second subplot: Accuracy
    plt.plot(epochs, train_accuracies, label='Train Accuracy', color='blue',␣
↪marker='o')
    plt.plot(epochs, test_accuracies, label='Test Accuracy', color='red',␣
↪marker='o')
    plt.title('Train and Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Show the plots
    plt.tight_layout()
    plt.show()
```

[ ]:
```
def save(name, model, final_epoch, optimizer, scheduler, train_losses,␣
↪train_accuracies, test_losses, test_accuracies):

    model.to(torch.device('cpu'))
    torch.save({
        'epoch': final_epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'scheduler_state_dict': scheduler.state_dict(),
        'train_loss': train_losses,
```

```python
        'train_accuracies': train_accuracies,
        'test_loss': test_losses,
        'test_accuracies': test_accuracies,
    }, f'models/{name}.checkpoint')
```

[ ]:
```python
def read_simplified():
    with open('data/simplified_dataset.json', 'r') as file:
        return [
            {**json.loads(line), 'complexity': complexity_classes.index(json.
 ↪loads(line)['complexity'])}
            for line in file
        ]
```

[12]:
```python
simplified_dataset = read_simplified()
```

[13]:
```python
simplified_dataset[0]
```

[13]:
```
{'complexity': 4,
 'src': 'public class VAR_51 { static class VAR_6 { private final VAR_2 VAR_1;
public VAR_6() { this.VAR_1 = new VAR_2(new VAR_5(VAR_4.VAR_3)); } public void
VAR_9(VAR_10 VAR_7) throws VAR_11 { VAR_1.VAR_8(STRING_LITERAL + VAR_7); }
public void VAR_12(VAR_10 VAR_7) throws VAR_11 { VAR_9(VAR_7);
VAR_1.VAR_8(STRING_LITERAL); } public void VAR_13() throws VAR_11 {
VAR_1.VAR_13(); } } static class VAR_20 { VAR_15 VAR_14; VAR_17 VAR_16; public
VAR_20() { VAR_14 = new VAR_15(new VAR_19(VAR_4.VAR_18)); } VAR_26 VAR_27() {
while (VAR_16 == null || !VAR_16.VAR_24()) { try { VAR_16 = new
VAR_17(VAR_14.VAR_23()); } catch (VAR_11 VAR_22) { VAR_22.VAR_21(); } } return
VAR_16.VAR_25(); } int VAR_30() { return VAR_29.VAR_28(VAR_27()); } long
VAR_33() { return VAR_32.VAR_31(VAR_27()); } double VAR_36() { return
VAR_35.VAR_34(VAR_27()); } VAR_26 VAR_38() { VAR_26 VAR_37 = STRING_LITERAL; try
{ VAR_37 = VAR_14.VAR_23(); } catch (VAR_11 VAR_22) { VAR_22.VAR_21(); } return
VAR_37; } VAR_41 VAR_42() { try { return new VAR_41(VAR_38()); } catch (VAR_40
VAR_22) { throw new VAR_39(); } } } public static void VAR_49(VAR_26[] VAR_50)
throws VAR_11 { VAR_20 VAR_43 = new VAR_20(); VAR_6 VAR_44 = new VAR_6(); int
VAR_45 = VAR_43.VAR_30(); int VAR_46 = VAR_43.VAR_30(); for (int VAR_47 =
INTEGER_LITERAL; VAR_47 < VAR_45 / INTEGER_LITERAL; VAR_47++) { for (int VAR_48
= INTEGER_LITERAL; VAR_48 < VAR_46; VAR_48++) { VAR_44.VAR_12((VAR_47 +
INTEGER_LITERAL) + STRING_LITERAL + (VAR_48 + INTEGER_LITERAL));
VAR_44.VAR_12((VAR_45 - VAR_47) + STRING_LITERAL + (VAR_46 - VAR_48)); } } if
(VAR_45 % INTEGER_LITERAL != INTEGER_LITERAL) { int VAR_47 = VAR_45 /
INTEGER_LITERAL; for (int VAR_48 = INTEGER_LITERAL; VAR_48 < VAR_46 /
INTEGER_LITERAL; VAR_48++) { VAR_44.VAR_12((VAR_47 + INTEGER_LITERAL) +
STRING_LITERAL + (VAR_48 + INTEGER_LITERAL)); VAR_44.VAR_12((VAR_47 +
INTEGER_LITERAL) + STRING_LITERAL + (VAR_46 - VAR_48)); } if (VAR_46 %
INTEGER_LITERAL != INTEGER_LITERAL) VAR_44.VAR_12((VAR_47 + INTEGER_LITERAL) +
STRING_LITERAL + (VAR_46 / INTEGER_LITERAL + INTEGER_LITERAL)); }
VAR_44.VAR_13(); } }'}
```

[ ]:
```
if not os.path.exists('simplfiied_tokenizer'):
    simplified_tokenizer = CustomTokenizer()
    simplified_tokenizer.fit(simplified_dataset, label='src')
    with open('simplfiied_tokenizer', 'wb') as f:
        print("Saving tokenizer...")
        pickle.dump(simplified_dataset, f)
else:
    with open('simplfiied_tokenizer', 'rb') as f:
        print("Loading tokenizer...")
        simplfiied_tokenizer: CustomTokenizer = pickle.load(f)
```

```
Saving tokenizer…
```

[18]:
```
len(simplified_tokenizer.vocab)
```

[18]:
```
567
```

[20]:
```
if not os.path.exists('data/torch_simplified_dataset'):
    torch_simplified_datatset = TorchDataset(simplified_dataset,␣
 ↪simplified_tokenizer)
    with open('data/torch_simplified_dataset', 'wb') as f:
        print("Saving torch dataset...")
        pickle.dump(torch_simplified_datatset, f)
else:
    with open('data/torch_simplified_dataset', 'rb') as f:
        print("Loading torch dataset...")
        torch_simplified_datatset: TorchDataset = pickle.load(f)
```

```
Loading torch dataset…
```

[61]:
```
train_size = int(0.8 * len(torch_simplified_datatset))
test_size = len(torch_simplified_datatset) - train_size
vocab_size = len(simplified_tokenizer.vocab)
batch_size = 16

train_dataset, test_dataset = random_split(torch_simplified_datatset,␣
 ↪[train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,␣
 ↪collate_fn=collate_fn)

print(f"Length train_loader: {len(train_loader)}, length test_loader:␣
 ↪{len(test_loader)}")
```

```
Length train_loader: 226, length test_loader: 57
```

[62]:
```
model = LSTMModel(
    vocab_size=len(simplified_tokenizer.vocab),
    embedding_dim=100,
    hidden_size=100,
    output_size=len(complexity_classes),
    num_layers=2,
    dropout_rate=0.5
    )
```

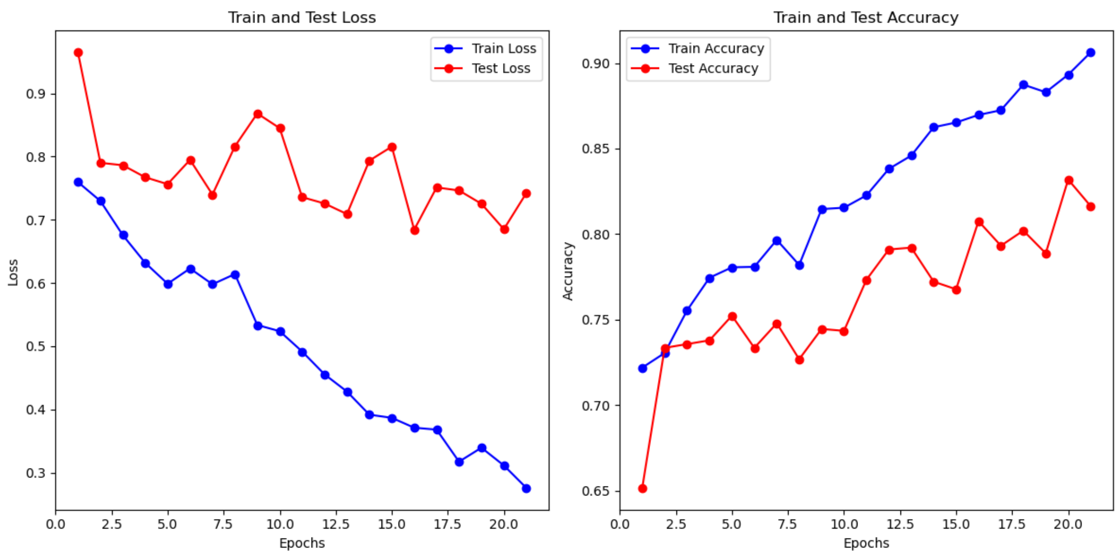
[67]:
```
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.003, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

model, train_losses, train_accuracies, test_losses, test_accuracies,␣
 ↪final_epoch = train(
    model=model,
    train_loader=train_loader,
    test_loader=test_loader,
    criterion=loss_function,
    optimizer=optimizer,
    scheduler=scheduler,
    num_epochs=1000)
```

```
Epoch:1|Train Loss:0.7597|Train Accuracy:0.72|Test Loss:0.9648|Test
Accuracy:0.65
Epoch:2|Train Loss:0.7298|Train Accuracy:0.73|Test Loss:0.7902|Test
Accuracy:0.73
Epoch:3|Train Loss:0.6758|Train Accuracy:0.76|Test Loss:0.7864|Test
Accuracy:0.74
Epoch:4|Train Loss:0.6319|Train Accuracy:0.77|Test Loss:0.7673|Test
Accuracy:0.74
Epoch:5|Train Loss:0.5988|Train Accuracy:0.78|Test Loss:0.7565|Test
Accuracy:0.75
Epoch:6|Train Loss:0.6230|Train Accuracy:0.78|Test Loss:0.7950|Test
Accuracy:0.73
No improvement for 1 epoch(s)
Epoch:7|Train Loss:0.5980|Train Accuracy:0.80|Test Loss:0.7400|Test
Accuracy:0.75
Epoch:8|Train Loss:0.6142|Train Accuracy:0.78|Test Loss:0.8162|Test
Accuracy:0.73
No improvement for 1 epoch(s)
Epoch:9|Train Loss:0.5335|Train Accuracy:0.81|Test Loss:0.8683|Test
Accuracy:0.74
No improvement for 2 epoch(s)
Epoch:10|Train Loss:0.5238|Train Accuracy:0.82|Test Loss:0.8451|Test
Accuracy:0.74
No improvement for 3 epoch(s)
Epoch:11|Train Loss:0.4916|Train Accuracy:0.82|Test Loss:0.7359|Test
```

```
Accuracy:0.77
Epoch:12|Train Loss:0.4555|Train Accuracy:0.84|Test Loss:0.7258|Test
Accuracy:0.79
Epoch:13|Train Loss:0.4283|Train Accuracy:0.85|Test Loss:0.7091|Test
Accuracy:0.79
Epoch:14|Train Loss:0.3917|Train Accuracy:0.86|Test Loss:0.7937|Test
Accuracy:0.77
No improvement for 1 epoch(s)
Epoch:15|Train Loss:0.3867|Train Accuracy:0.87|Test Loss:0.8158|Test
Accuracy:0.77
No improvement for 2 epoch(s)
Epoch:16|Train Loss:0.3711|Train Accuracy:0.87|Test Loss:0.6843|Test
Accuracy:0.81
Epoch:17|Train Loss:0.3680|Train Accuracy:0.87|Test Loss:0.7512|Test
Accuracy:0.79
No improvement for 1 epoch(s)
Epoch:18|Train Loss:0.3173|Train Accuracy:0.89|Test Loss:0.7464|Test
Accuracy:0.80
No improvement for 2 epoch(s)
Epoch:19|Train Loss:0.3394|Train Accuracy:0.88|Test Loss:0.7254|Test
Accuracy:0.79
No improvement for 3 epoch(s)
Epoch:20|Train Loss:0.3113|Train Accuracy:0.89|Test Loss:0.6852|Test
Accuracy:0.83
No improvement for 4 epoch(s)
Epoch:21|Train Loss:0.2759|Train Accuracy:0.91|Test Loss:0.7421|Test
Accuracy:0.82
No improvement for 5 epoch(s)
Early stopping triggered!
```

[70]:
```
plot_metrics(train_losses, train_accuracies, test_losses,test_accuracies)
```



[71]:

```
save("model_2_h_0.003lr",
     model=model,
     final_epoch=final_epoch,
     optimizer=optimizer,
     scheduler=scheduler,
     train_losses=train_losses,
     train_accuracies=train_accuracies,
     test_losses=test_losses,
     test_accuracies=test_accuracies
     )
```

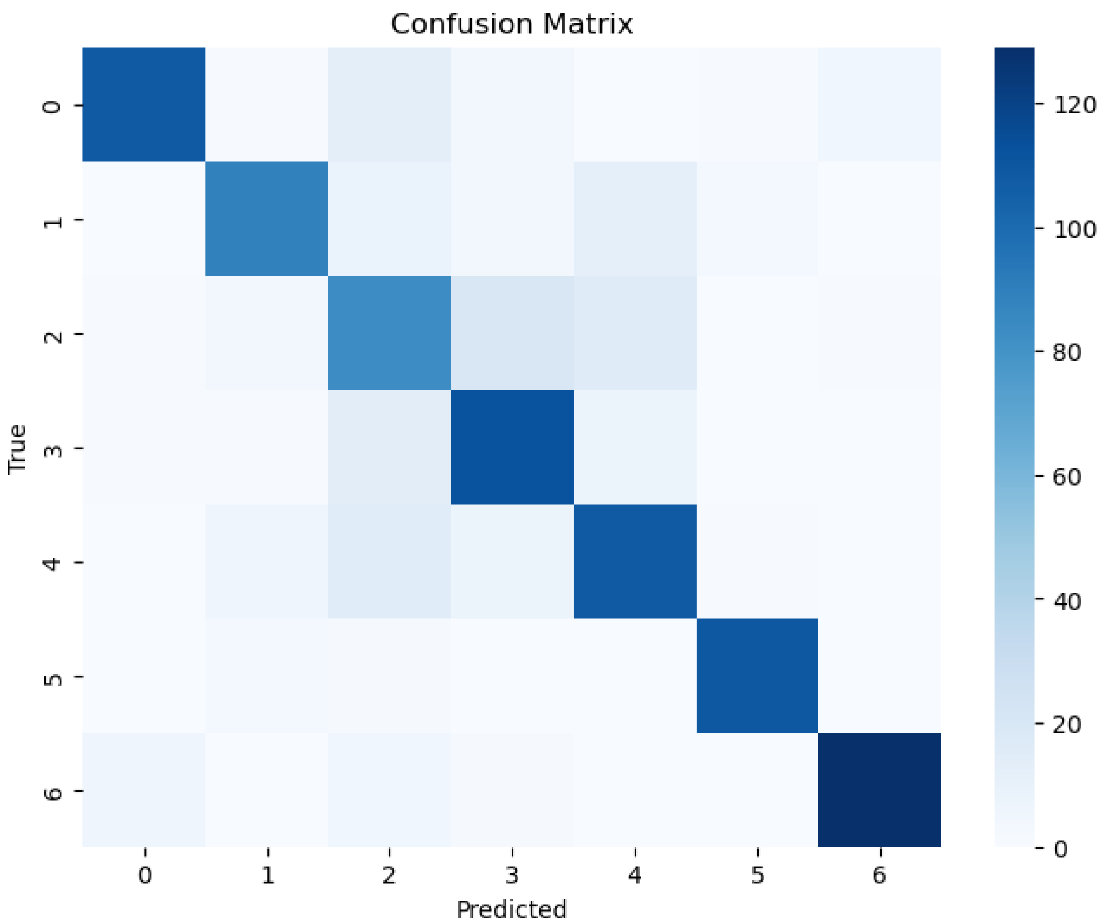
[ ]:

```
model.to(torch.device('cpu'))
_, _, true_labels, predictions = evaluate(model, test_loader=test_loader,␣
 ↪criterion=loss_function, device_str = 'cpu')

cm = confusion_matrix(true_labels, predictions, labels=np.arange(7))
```

[83]:

```
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues', xticklabels=np.arange(7),␣
 ↪yticklabels=np.arange(7))
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



[108]:
```
# Taken from chatGPT after asking it for an nlogn algorithm.

test_src = '''
public class MergeSortExample {

    public static void main(String[] args) {
        int[] array = {38, 27, 43, 3, 9, 82, 10};
        mergeSort(array, 0, array.length - 1);
        for (int num : array) {
            System.out.print(num + " ");
        }
    }

    public static void mergeSort(int[] arr, int left, int right) {
        if (left >= right) return;
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
```

```
        merge(arr, left, mid, right);
    }

    public static void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
            temp[k++] = (arr[i] <= arr[j]) ? arr[i++] : arr[j++];
        }
        while (i <= mid) temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];
        System.arraycopy(temp, 0, arr, left, temp.length);
    }
}

'''

result = subprocess.run(['java', '-jar', 'code2ast/target/code2ast-1.0-SNAPSHOT.
 ↪jar', 'print_cu', test_src], capture_output=True)
simplified_src = result.stdout.decode().strip()
test_src_tokens = simplified_tokenizer.tokenize(simplified_src)
x = torch.tensor([test_src_tokens], dtype=torch.long)
model.predict(x)
```

[108]: 'nlogn'

[107]:
```
test_src = '''
public class BinarySearch {

    public static void main(String[] args) {
        int[] array = {1, 3, 5, 7, 9, 11, 13, 15, 17};
        int target = 7;

        int index = binarySearch(array, target);
        System.out.println("Index of " + target + ": " + index);
    }

    public static int binarySearch(int[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (array[mid] == target) {
                return mid;
            } else if (array[mid] < target) {
```

```
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1;
    }
}

'''

result = subprocess.run(['java', '-jar', 'code2ast/target/code2ast-1.0-SNAPSHOT.
 ↪jar', 'print_cu', test_src], capture_output=True)
simplified_src = result.stdout.decode().strip()
test_src_tokens = simplified_tokenizer.tokenize(simplified_src)
x = torch.tensor([test_src_tokens], dtype=torch.long)
model.predict(x)
```

[107]: 'logn'

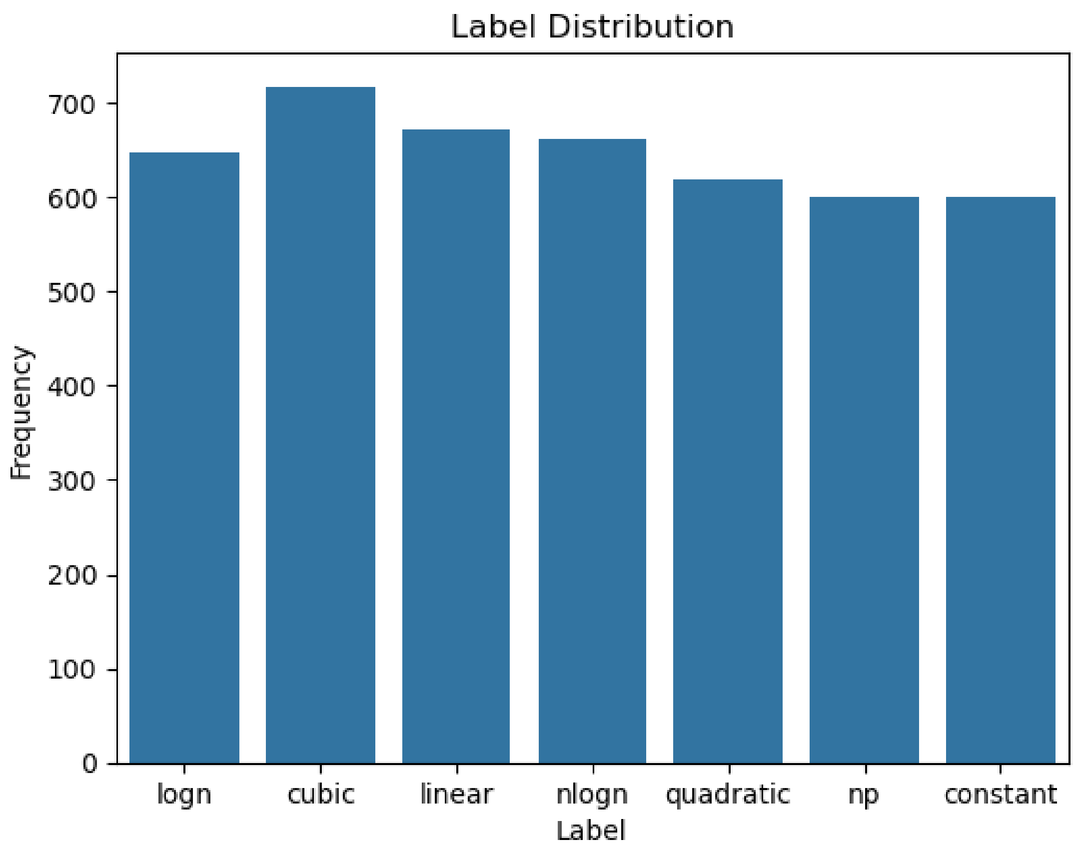
[112]:
```
labels = []
for _, label in torch_simplified_datatset:
    labels.append(label.item())


label_counts = Counter(labels)


label_names = complexity_classes
label_frequencies = list(label_counts.values())


sns.barplot(x=label_names, y=label_frequencies)

plt.title('Label Distribution')
plt.xlabel('Label')
plt.ylabel('Frequency')
plt.show()
```



[123]:
```
def get_label_counts(loader):
    labels = []
    for batch in loader:
        _, _, batch_labels = batch
        labels.extend(batch_labels.numpy())  # Convert batch labels to numpy␣
↪for easy appending
    return Counter(labels)

train_label_counts = get_label_counts(train_loader)
test_label_counts = get_label_counts(test_loader)

train_label_names = list(train_label_counts.keys())
train_label_frequencies = list(train_label_counts.values())

test_label_names = list(test_label_counts.keys())
test_label_frequencies = list(test_label_counts.values())

# Combine train and test data
all_labels = np.arange(7)
train_freqs = [train_label_counts.get(label, 0) for label in all_labels]
```

```
test_freqs = [test_label_counts.get(label, 0) for label in all_labels]

# Plotting
fig, ax = plt.subplots(figsize=(10, 6))

bar_width = 0.35
index = range(len(all_labels))

bar_train = ax.bar(index, train_freqs, bar_width, label='Train', color='blue')
bar_test = ax.bar([i + bar_width for i in index], test_freqs, bar_width,␣
 ↪label='Test', color='orange')

ax.set_xlabel('Label')
ax.set_ylabel('Frequency')
ax.set_title('Label Distribution - Train vs Test')
ax.set_xticks([i + bar_width / 2 for i in index])
ax.set_xticklabels(complexity_classes)
ax.legend()

plt.tight_layout()
plt.show()
```