**Commonsense Dataset**

**BERT(base):**

import torch, os

import pandas as pd

from transformers import pipeline, RobertaForSequenceClassification, RobertaTokenizer, RobertaModel,BertTokenizerFast,BertForSequenceClassification, AutoModelForSequenceClassification

from transformers import GPT2Tokenizer, GPT2Model

from torch.utils.data import Dataset

from torch import cuda

device = 'cuda' if cuda.is\_available() else 'cpu'

device

import gc

torch.cuda.empty\_cache()

gc.collect()

df\_org= pd.read\_csv("processed\_sentiment.1600000.csv", encoding='utf-8')

df\_org = df\_org.sample(frac=1.0, random\_state=42)

df\_org.head()

df\_org=df\_org.dropna()

labels = df\_org['Class'].unique().tolist()

labels

NUM\_LABELS= len(labels)

id2label={id:label for id,label in enumerate(labels)}

label2id={label:id for id,label in enumerate(labels)}

id2label

df\_org["labels"]=df\_org.Class.map(lambda x: label2id[x.strip()])

df\_org.head()

df\_org.Class.value\_counts().plot(kind='pie', figsize=(5,5))

from transformers import AutoModel, AutoTokenizer, DistilBertTokenizer, DistilBertModel, AutoModelForSequenceClassification

import torch.nn as nn

from transformers.modeling\_outputs import TokenClassifierOutput

fchidden = 256

hiddendim\_lstm = 256

embeddim = 768

numlayers = 5

checkpoint='bert-base-cased'

class MyTaskSpecificCustomModel(nn.Module):

"""

A task-specific custom transformer model. This model loads a pre-trained transformer model and adds a new dropout

and linear layer at the end for fine-tuning and prediction on specific tasks.

"""

def \_\_init\_\_(self, checkpoint, num\_labels ):

"""

Args:

checkpoint (str): The name of the pre-trained model or path to the model weights.

num\_labels (int): The number of output labels in the final classification layer.

"""

super(MyTaskSpecificCustomModel, self).\_\_init\_\_()

self.num\_labels = num\_labels

self.model = model = AutoModel.from\_pretrained(checkpoint, config = AutoConfig.from\_pretrained(checkpoint,

output\_attention = True,

output\_hidden\_state = True ) )

# New Layer

self.dropout = nn.Dropout(0.1)

#self.lstm=nn.LSTM(768,hiddendim\_lstm,batch\_first=True)

self.classifier = nn.Linear(768, self.num\_labels )

def forward(self, input\_ids = None, attention\_mask=None, labels = None ):

"""

Forward pass for the model.

Args:

input\_ids (torch.Tensor, optional): Tensor of input IDs. Defaults to None.

attention\_mask (torch.Tensor, optional): Tensor for attention masks. Defaults to None.

labels (torch.Tensor, optional): Tensor for labels. Defaults to None.

Returns:

TokenClassifierOutput: A named tuple with the following fields:

- loss (torch.FloatTensor of shape (1,), optional, returned when label\_ids is provided) – Classification loss.

- logits (torch.FloatTensor of shape (batch\_size, num\_labels)) – Classification scores before SoftMax.

- hidden\_states (tuple(torch.FloatTensor), optional, returned when output\_hidden\_states=True is passed or when config.output\_hidden\_states=True) – Tuple of torch.FloatTensor (one for the output of the embeddings + one for the output of each layer) of shape (batch\_size, sequence\_length, hidden\_size).

- attentions (tuple(torch.FloatTensor), optional, returned when output\_attentions=True is passed or when config.output\_attentions=True) – Tuple of torch.FloatTensor (one for each layer) of shape (batch\_size, num\_heads, sequence\_length, sequence\_length).

"""

outputs = self.model(input\_ids = input\_ids, attention\_mask = attention\_mask )

last\_hidden\_state = outputs[0]

sequence\_outputs = self.dropout(last\_hidden\_state)

logits = self.classifier(sequence\_outputs[:, 0, : ].view(-1, 768 ))

loss = None

loss = None

if labels is not None:

loss\_func = nn.CrossEntropyLoss()

loss = loss\_func(logits.view(-1, self.num\_labels), labels.view(-1))

return TokenClassifierOutput(loss=loss, logits=logits, hidden\_states=outputs.hidden\_states, attentions=outputs.attentions)

tokenizer = AutoTokenizer.from\_pretrained("bert-base-uncased", max\_length=512)

model = AutoModelForSequenceClassification.from\_pretrained("bert-base-uncased", num\_labels=NUM\_LABELS, id2label=id2label, label2id=label2id)

model.to(device)

SIZE= df\_org.shape[0]

train\_texts= list(df\_org.Comments [:(90\*SIZE)//100])

val\_texts= list(df\_org.Comments [(90\*SIZE)//100:(95\*SIZE)//100 ])

test\_texts= list(df\_org.Comments [(95\*SIZE)//100:])

train\_labels= list(df\_org.labels[:(90\*SIZE)//100])

val\_labels= list(df\_org.labels[(90\*SIZE)//100:(95\*SIZE)//100 ])

test\_labels= list(df\_org.labels[(95\*SIZE)//100:])

len(train\_texts), len(val\_texts), len(test\_texts)

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

val\_encodings = tokenizer(val\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

class DataLoader(Dataset):

"""

Custom Dataset class for handling tokenized text data and corresponding labels.

Inherits from torch.utils.data.Dataset.

"""

def \_\_init\_\_(self, encodings, labels):

"""

Initializes the DataLoader class with encodings and labels.

Args:

encodings (dict): A dictionary containing tokenized input text data

(e.g., 'input\_ids', 'token\_type\_ids', 'attention\_mask').

labels (list): A list of integer labels for the input text data.

"""

self.encodings = encodings

self.labels = labels

def \_\_getitem\_\_(self, idx):

"""

Returns a dictionary containing tokenized data and the corresponding label for a given index.

Args:

idx (int): The index of the data item to retrieve.

Returns:

item (dict): A dictionary containing the tokenized data and the corresponding label.

"""

# Retrieve tokenized data for the given index

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

# Add the label for the given index to the item dictionary

item['labels'] = torch.tensor(self.labels[idx])

return item

def \_\_len\_\_(self):

"""

Returns the number of data items in the dataset.

Returns:

(int): The number of data items in the dataset.

"""

return len(self.labels)

print(train\_labels)

train\_dataloader = DataLoader(train\_encodings, train\_labels)

val\_dataloader = DataLoader(val\_encodings, val\_labels)

test\_dataset = DataLoader(test\_encodings, test\_labels)

from transformers import TrainingArguments, Trainer

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support,classification\_report

def compute\_metrics(pred):

"""

Computes accuracy, F1, precision, and recall for a given set of predictions.

Args:

pred (obj): An object containing label\_ids and predictions attributes.

- label\_ids (array-like): A 1D array of true class labels.

- predictions (array-like): A 2D array where each row represents

an observation, and each column represents the probability of

that observation belonging to a certain class.

Returns:

dict: A dictionary containing the following metrics:

- Accuracy (float): The proportion of correctly classified instances.

- F1 (float): The macro F1 score, which is the harmonic mean of precision

and recall. Macro averaging calculates the metric independently for

each class and then takes the average.

- Precision (float): The macro precision, which is the number of true

positives divided by the sum of true positives and false positives.

- Recall (float): The macro recall, which is the number of true positives

divided by the sum of true positives and false negatives.

"""

# Extract true labels from the input object

labels = pred.label\_ids

# Obtain predicted class labels by finding the column index with the maximum probability

preds = pred.predictions.argmax(-1)

# Compute macro precision, recall, and F1 score using sklearn's precision\_recall\_fscore\_support function

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, preds, average='weighted', warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None, zero\_division=0)

# Calculate the accuracy score using sklearn's accuracy\_score function

acc = accuracy\_score(labels, preds)

#mainreports=classification\_report(preds, labels, target\_names=['negative', 'positive']) #For IMDb and Sentiment140

mainreports=classification\_report(preds, labels, target\_names=['negative', 'positive'])

# Return the computed metrics as a dictionary

return {

'Accuracy': acc,

'F1': f1,

'Precision': precision,

'Recall': recall,

'reports': mainreports

}

from huggingface\_hub import notebook\_login

notebook\_login()

training\_args = TrainingArguments(

output\_dir="./imdbreviews\_classification\_roberta\_v02",

learning\_rate=1e-5,

per\_device\_train\_batch\_size=16,

per\_device\_eval\_batch\_size=16,

num\_train\_epochs=5,

#weight\_decay=0.01,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

push\_to\_hub=True,

)

trainer = Trainer(

# the pre-trained model that will be fine-tuned

model=model,

# training arguments that we defined above

tokenizer=tokenizer,

args=training\_args,

train\_dataset=train\_dataloader,

eval\_dataset=val\_dataloader,

compute\_metrics= compute\_metrics

)

trainer.train()

q=[trainer.evaluate(eval\_dataset=df\_org) for df\_org in [train\_dataloader, val\_dataloader, test\_dataset]]

pd.DataFrame(q, index=["train","val","test"]).iloc[:,:5]

**RoBERTa(base):**

import torch, os

import pandas as pd

from transformers import pipeline, RobertaForSequenceClassification, RobertaTokenizer, RobertaModel,BertTokenizerFast,BertForSequenceClassification, AutoModelForSequenceClassification

from transformers import GPT2Tokenizer, GPT2Model

from torch.utils.data import Dataset

from torch import cuda

device = 'cuda' if cuda.is\_available() else 'cpu'

device

import gc

torch.cuda.empty\_cache()

gc.collect()

df\_org= pd.read\_csv("processed\_sentiment.1600000.csv", encoding='utf-8')

df\_org = df\_org.sample(frac=1.0, random\_state=42)

df\_org.head()

df\_org=df\_org.dropna()

labels = df\_org['Class'].unique().tolist()

labels

NUM\_LABELS= len(labels)

id2label={id:label for id,label in enumerate(labels)}

label2id={label:id for id,label in enumerate(labels)}

id2label

df\_org["labels"]=df\_org.Class.map(lambda x: label2id[x.strip()])

df\_org.head()

df\_org.Class.value\_counts().plot(kind='pie', figsize=(5,5))

from transformers import AutoModel, AutoTokenizer, DistilBertTokenizer, DistilBertModel, AutoModelForSequenceClassification

import torch.nn as nn

from transformers.modeling\_outputs import TokenClassifierOutput

fchidden = 256

hiddendim\_lstm = 256

embeddim = 768

numlayers = 5

checkpoint='roberta-base'

class MyTaskSpecificCustomModel(nn.Module):

"""

A task-specific custom transformer model. This model loads a pre-trained transformer model and adds a new dropout

and linear layer at the end for fine-tuning and prediction on specific tasks.

"""

def \_\_init\_\_(self, checkpoint, num\_labels ):

"""

Args:

checkpoint (str): The name of the pre-trained model or path to the model weights.

num\_labels (int): The number of output labels in the final classification layer.

"""

super(MyTaskSpecificCustomModel, self).\_\_init\_\_()

self.num\_labels = num\_labels

self.model = model = AutoModel.from\_pretrained(checkpoint, config = AutoConfig.from\_pretrained(checkpoint,

output\_attention = True,

output\_hidden\_state = True ) )

# New Layer

self.dropout = nn.Dropout(0.1)

#self.lstm=nn.LSTM(768,hiddendim\_lstm,batch\_first=True)

self.classifier = nn.Linear(768, self.num\_labels )

def forward(self, input\_ids = None, attention\_mask=None, labels = None ):

"""

Forward pass for the model.

Args:

input\_ids (torch.Tensor, optional): Tensor of input IDs. Defaults to None.

attention\_mask (torch.Tensor, optional): Tensor for attention masks. Defaults to None.

labels (torch.Tensor, optional): Tensor for labels. Defaults to None.

Returns:

TokenClassifierOutput: A named tuple with the following fields:

- loss (torch.FloatTensor of shape (1,), optional, returned when label\_ids is provided) – Classification loss.

- logits (torch.FloatTensor of shape (batch\_size, num\_labels)) – Classification scores before SoftMax.

- hidden\_states (tuple(torch.FloatTensor), optional, returned when output\_hidden\_states=True is passed or when config.output\_hidden\_states=True) – Tuple of torch.FloatTensor (one for the output of the embeddings + one for the output of each layer) of shape (batch\_size, sequence\_length, hidden\_size).

- attentions (tuple(torch.FloatTensor), optional, returned when output\_attentions=True is passed or when config.output\_attentions=True) – Tuple of torch.FloatTensor (one for each layer) of shape (batch\_size, num\_heads, sequence\_length, sequence\_length).

"""

outputs = self.model(input\_ids = input\_ids, attention\_mask = attention\_mask )

last\_hidden\_state = outputs[0]

sequence\_outputs = self.dropout(last\_hidden\_state)

logits = self.classifier(sequence\_outputs[:, 0, : ].view(-1, 768 ))

loss = None

loss = None

if labels is not None:

loss\_func = nn.CrossEntropyLoss()

loss = loss\_func(logits.view(-1, self.num\_labels), labels.view(-1))

return TokenClassifierOutput(loss=loss, logits=logits, hidden\_states=outputs.hidden\_states, attentions=outputs.attentions)

tokenizer = AutoTokenizer.from\_pretrained("roberta-base", max\_length=512)

model = AutoModelForSequenceClassification.from\_pretrained("roberta-base", num\_labels=NUM\_LABELS, id2label=id2label, label2id=label2id)

model.to(device)

SIZE= df\_org.shape[0]

train\_texts= list(df\_org.Comments [:(90\*SIZE)//100])

val\_texts= list(df\_org.Comments [(90\*SIZE)//100:(95\*SIZE)//100 ])

test\_texts= list(df\_org.Comments [(95\*SIZE)//100:])

train\_labels= list(df\_org.labels[:(90\*SIZE)//100])

val\_labels= list(df\_org.labels[(90\*SIZE)//100:(95\*SIZE)//100 ])

test\_labels= list(df\_org.labels[(95\*SIZE)//100:])

len(train\_texts), len(val\_texts), len(test\_texts)

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

val\_encodings = tokenizer(val\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

class DataLoader(Dataset):

"""

Custom Dataset class for handling tokenized text data and corresponding labels.

Inherits from torch.utils.data.Dataset.

"""

def \_\_init\_\_(self, encodings, labels):

"""

Initializes the DataLoader class with encodings and labels.

Args:

encodings (dict): A dictionary containing tokenized input text data

(e.g., 'input\_ids', 'token\_type\_ids', 'attention\_mask').

labels (list): A list of integer labels for the input text data.

"""

self.encodings = encodings

self.labels = labels

def \_\_getitem\_\_(self, idx):

"""

Returns a dictionary containing tokenized data and the corresponding label for a given index.

Args:

idx (int): The index of the data item to retrieve.

Returns:

item (dict): A dictionary containing the tokenized data and the corresponding label.

"""

# Retrieve tokenized data for the given index

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

# Add the label for the given index to the item dictionary

item['labels'] = torch.tensor(self.labels[idx])

return item

def \_\_len\_\_(self):

"""

Returns the number of data items in the dataset.

Returns:

(int): The number of data items in the dataset.

"""

return len(self.labels)

print(train\_labels)

train\_dataloader = DataLoader(train\_encodings, train\_labels)

val\_dataloader = DataLoader(val\_encodings, val\_labels)

test\_dataset = DataLoader(test\_encodings, test\_labels)

from transformers import TrainingArguments, Trainer

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support,classification\_report

def compute\_metrics(pred):

"""

Computes accuracy, F1, precision, and recall for a given set of predictions.

Args:

pred (obj): An object containing label\_ids and predictions attributes.

- label\_ids (array-like): A 1D array of true class labels.

- predictions (array-like): A 2D array where each row represents

an observation, and each column represents the probability of

that observation belonging to a certain class.

Returns:

dict: A dictionary containing the following metrics:

- Accuracy (float): The proportion of correctly classified instances.

- F1 (float): The macro F1 score, which is the harmonic mean of precision

and recall. Macro averaging calculates the metric independently for

each class and then takes the average.

- Precision (float): The macro precision, which is the number of true

positives divided by the sum of true positives and false positives.

- Recall (float): The macro recall, which is the number of true positives

divided by the sum of true positives and false negatives.

"""

# Extract true labels from the input object

labels = pred.label\_ids

# Obtain predicted class labels by finding the column index with the maximum probability

preds = pred.predictions.argmax(-1)

# Compute macro precision, recall, and F1 score using sklearn's precision\_recall\_fscore\_support function

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, preds, average='weighted', warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None, zero\_division=0)

# Calculate the accuracy score using sklearn's accuracy\_score function

acc = accuracy\_score(labels, preds)

#mainreports=classification\_report(preds, labels, target\_names=['negative', 'positive']) #For IMDb and Sentiment140

mainreports=classification\_report(preds, labels, target\_names=['negative', 'positive'])

# Return the computed metrics as a dictionary

return {

'Accuracy': acc,

'F1': f1,

'Precision': precision,

'Recall': recall,

'reports': mainreports

}

from huggingface\_hub import notebook\_login

notebook\_login()

training\_args = TrainingArguments(

output\_dir="./imdbreviews\_classification\_roberta\_v02",

learning\_rate=1e-5,

per\_device\_train\_batch\_size=16,

per\_device\_eval\_batch\_size=16,

num\_train\_epochs=5,

#weight\_decay=0.01,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

push\_to\_hub=True,

)

trainer = Trainer(

# the pre-trained model that will be fine-tuned

model=model,

# training arguments that we defined above

tokenizer=tokenizer,

args=training\_args,

train\_dataset=train\_dataloader,

eval\_dataset=val\_dataloader,

compute\_metrics= compute\_metrics

)

trainer.train()

q=[trainer.evaluate(eval\_dataset=df\_org) for df\_org in [train\_dataloader, val\_dataloader, test\_dataset]]

pd.DataFrame(q, index=["train","val","test"]).iloc[:,:5]

**RoBERTa-RNN:**

import torch, os

import pandas as pd

from transformers import pipeline, BertModel, BertTokenizer, BertForSequenceClassification,RobertaForSequenceClassification, RobertaTokenizer,T5ForConditionalGeneration,CodeLlamaTokenizer

from torch.utils.data import Dataset

from torch import cuda

device = 'cuda' if cuda.is\_available() else 'cpu'

device

df\_org= pd.read\_csv("/content/drive/MyDrive/Dataset/processed\_Tweets.csv", encoding="utf-8")

df\_org = df\_org.sample(frac=1.0, random\_state=0)

df\_org.head()

labels = df\_org['airline\_sentiment'].unique().tolist()

labels

NUM\_LABELS= len(labels)

id2label={id:label for id,label in enumerate(labels)}

label2id={label:id for id,label in enumerate(labels)}

label2id

df\_org["labels"]=df\_org.airline\_sentiment.map(lambda x: label2id[x.strip()])

df\_org.head()

df\_org.airline\_sentiment.value\_counts().plot(kind='pie', figsize=(5,5))

from transformers import AutoModel, AutoTokenizer

import torch.nn as nn

from transformers.modeling\_outputs import TokenClassifierOutput

fchidden = 256

hiddendim\_lstm = 128

embeddim = 768

numlayers = 5

#checkpoint='google-bert/bert-base-cased'

#checkpoint='microsoft/codebert-base'

checkpoint='roberta-base'

#checkpoint='Salesforce/codet5-small'

#checkpoint='Salesforce/codet5p-220m'

class Bert\_LSTM(nn.Module):

def \_\_init\_\_(self, checkpoint, num\_labels):

super(Bert\_LSTM, self).\_\_init\_\_()

self.numclasses = num\_labels

self.embeddim = embeddim

self.numlayers = numlayers

self.hiddendim\_lstm = hiddendim\_lstm

self.model= model = BertModel.from\_pretrained(checkpoint, output\_hidden\_states=True, output\_attentions=False)

print("BERT Model Loaded")

#self.dropout = nn.Dropout(0.1)

self.lstm = nn.LSTM(self.embeddim, self.hiddendim\_lstm, batch\_first=True, bidirectional=False) # noqa

#self.dropout = nn.Dropout(0.1)

self.classifier = nn.Linear(self.embeddim, self.numclasses)

#self.classifier1 = nn.Linear(256, self.numclasses)

#def forward(self, inp\_ids, att\_mask, token\_ids):

def forward(self, input\_ids = None, attention\_mask=None, labels = None ):

outputs = self.model(input\_ids = input\_ids, attention\_mask = attention\_mask)

sequence\_outputs=outputs[0]

#sequence\_outputs = self.dropout(sequence\_outputs)

sequence\_outputs = self.lstm(sequence\_outputs)

#logits = self.classifier(sequence\_outputs[:, 0, : ].view(-1, 768 ))

logits = self.classifier(sequence\_outputs[:, -1])

loss = None

loss = None

if labels is not None:

loss\_func = nn.CrossEntropyLoss()

loss = loss\_func(logits.view(-1, self.numclasses), labels.view(-1))

return TokenClassifierOutput(loss=loss, logits=logits, hidden\_states=outputs.hidden\_states, attentions=outputs.attentions)

class BertClassifier(nn.Module):

"""Bert Model for Classification Tasks."""

#def \_\_init\_\_(self, checkpoint, num\_labels, freeze\_bert=False):

def \_\_init\_\_(self, checkpoint, num\_labels):

super(BertClassifier, self).\_\_init\_\_()

self.numclasses = num\_labels

self.embeddim = embeddim

self.numlayers = numlayers

self.hiddendim\_lstm = hiddendim\_lstm

# Specify hidden size of BERT, hidden size of our classifier, and number of labels

#D\_in, H, D\_out = 768, 50, 2

# Instantiate BERT model

self.model= model = AutoModel.from\_pretrained(checkpoint)

#self.model= model = RobertaForSequenceClassification.from\_pretrained(checkpoint)

self.dropout = nn.Dropout(0.2)

#self.activation=nn.ReLU()

self.lstm = nn.LSTM(self.embeddim, self.hiddendim\_lstm, batch\_first=True, bidirectional=False)

#self.dropout1 = nn.Dropout(0.1)

#self.linear = nn.Linear(self.hiddendim\_lstm\*2 , self.numclasses)False

self.linear = nn.Linear(self.hiddendim\_lstm, self.numclasses)

self.softmax = nn.LogSoftmax(dim=1)

# Freeze the BERT model

#if freeze\_bert:

#for param in self.model.parameters():

#param.requires\_grad = False

def forward(self, input\_ids = None, attention\_mask=None, labels = None ):

# Feed input to BERT

outputs = self.model(input\_ids=input\_ids,attention\_mask=attention\_mask)

sequence\_output = outputs[0]

#print("sequence\_output size", sequence\_output.size())

sequence\_output = self.dropout(sequence\_output)

#sequence\_output=self.activation(sequence\_output)

sequence\_output, \_ = self.lstm(sequence\_output)

#sequence\_output = self.dropout1(sequence\_output)

#print("lstm size", sequence\_output.size())

#sequence\_output = self.dropout(sequence\_output)

logits = self.linear(sequence\_output[:, -1])

logits = self.softmax(logits)

loss = None

loss = None

if labels is not None:

loss\_func = nn.CrossEntropyLoss()

loss = loss\_func(logits.view(-1, self.numclasses), labels.view(-1))

return TokenClassifierOutput(loss=loss, logits=logits, hidden\_states=outputs.hidden\_states, attentions=outputs.attentions)

tokenizer = RobertaTokenizer.from\_pretrained(checkpoint, max\_length=512)

from transformers import RobertaTokenizer, BertForSequenceClassification

model = RobertaForSequenceClassification.from\_pretrained(checkpoint, num\_labels=NUM\_LABELS, id2label=id2label, label2id=label2id)

model=BertClassifier(checkpoint,NUM\_LABELS)

model.to(device)

SIZE= df\_org.shape[0]

SIZE

SIZE= df\_org.shape[0]

train\_texts= list(df\_org.CodeFilter[:(8\*SIZE)//10])

val\_texts= list(df\_org.CodeFilter[(8\*SIZE)//10:(90\*SIZE)//100 ])

test\_texts= list(df\_org.CodeFilter[(90\*SIZE)//100:])

train\_labels= list(df\_org.labels[:(8\*SIZE)//10])

val\_labels= list(df\_org.labels[(8\*SIZE)//10:(90\*SIZE)//100 ])

test\_labels= list(df\_org.labels[(90\*SIZE)//100:])

len(train\_texts), len(val\_texts), len(test\_texts)

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

val\_encodings = tokenizer(val\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

class DataLoader(Dataset):

"""

Custom Dataset class for handling tokenized text data and corresponding labels.

Inherits from torch.utils.data.Dataset.

"""

def \_\_init\_\_(self, encodings, labels):

"""

Initializes the DataLoader class with encodings and labels.

Args:

encodings (dict): A dictionary containing tokenized input text data

(e.g., 'input\_ids', 'token\_type\_ids', 'attention\_mask').

labels (list): A list of integer labels for the input text data.

"""

self.encodings = encodings

self.labels = labels

def \_\_getitem\_\_(self, idx):

"""

Returns a dictionary containing tokenized data and the corresponding label for a given index.

Args:

idx (int): The index of the data item to retrieve.

Returns:

item (dict): A dictionary containing the tokenized data and the corresponding label.

"""

# Retrieve tokenized data for the given index

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

# Add the label for the given index to the item dictionary

item['labels'] = torch.tensor(self.labels[idx])

return item

def \_\_len\_\_(self):

"""

Returns the number of data items in the dataset.

Returns:

(int): The number of data items in the dataset.

"""

return len(self.labels)

print(train\_labels)

train\_dataloader = DataLoader(train\_encodings, train\_labels)

val\_dataloader = DataLoader(val\_encodings, val\_labels)

test\_dataset = DataLoader(test\_encodings, test\_labels)

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support, classification\_report

def compute\_metrics(pred):

"""

Computes accuracy, F1, precision, and recall for a given set of predictions.

Args:

pred (obj): An object containing label\_ids and predictions attributes.

- label\_ids (array-like): A 1D array of true class labels.

- predictions (array-like): A 2D array where each row represents

an observation, and each column represents the probability of

that observation belonging to a certain class.

Returns:

dict: A dictionary containing the following metrics:

- Accuracy (float): The proportion of correctly classified instances.

- F1 (float): The macro F1 score, which is the harmonic mean of precision

and recall. Macro averaging calculates the metric independently for

each class and then takes the average.

- Precision (float): The macro precision, which is the number of true

positives divided by the sum of true positives and false positives.

- Recall (float): The macro recall, which is the number of true positives

divided by the sum of true positives and false negatives.

"""

# Extract true labels from the input object

labels = pred.label\_ids

# Obtain predicted class labels by finding the column index with the maximum probability

preds = pred.predictions.argmax(-1)

# Compute macro precision, recall, and F1 score using sklearn's precision\_recall\_fscore\_support function

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, preds, average='weighted', zero\_division=0)

# Calculate the accuracy score using sklearn's accuracy\_score function

acc = accuracy\_score(labels, preds)

#mainreports=classification\_report(preds, labels, target\_names=['Insertion Sort', 'MST', 'Shell Sort', 'Exhaustive Search', 'BFS', 'SP', 'Linear Search', 'Selection Sort', 'Stable Sort', 'Binary Search', 'Rooted Trees', 'Bubble Sort', 'Counting Sort', 'Merge Sort', 'Projection', 'Convex Hull', 'NSS', 'Graph', 'Tree Walk', 'Puzzle', 'DFS', 'CBT', 'BST', 'Intersection', 'Binary Trees', 'Reflection', 'String Search', 'Quick Sort', 'Area'])

mainreports=classification\_report(preds, labels, target\_names=['negative', 'neutral', 'positive'])

#mainreports=classification\_report(preds, labels, target\_names=['DFS', 'Linear Search', 'Binary Search', 'Exhaustive Search', 'BFS'])

# Return the computed metrics as a dictionary

return {

'Accuracy': acc,

'F1': f1,

'Precision': precision,

'Recall': recall,

'reports': mainreports

}

from transformers import TrainingArguments, Trainer

from transformers.optimization import AdamW, Adafactor, AdafactorSchedule

import torch.optim.lr\_scheduler as lr\_scheduler

#optimizer = Adafactor(model.parameters(), relative\_step=False, lr=1e-5, weight\_decay=0.01)

#lr\_scheduler = AdafactorSchedule(optimizer)

optimizer = torch.optim.NAdam (model.parameters(), lr=2e-5)

lr\_scheduler=lr\_scheduler.ReduceLROnPlateau(optimizer)

training\_args = TrainingArguments(

output\_dir="./imdbreviews\_classification\_roberta\_v02",

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

push\_to\_hub=True,

)

trainer = Trainer(

# the pre-trained model that will be fine-tuned

model=model,

# Optimizer

optimizers=(optimizer, lr\_scheduler),

# training arguments that we defined above

tokenizer=tokenizer,

args=training\_args,

train\_dataset=train\_dataloader,

eval\_dataset=val\_dataloader,

compute\_metrics= compute\_metrics

)

trainer.train()

q=[trainer.evaluate(eval\_dataset=df\_org) for df\_org in [train\_dataloader, val\_dataloader, test\_dataset]]

pd.DataFrame(q, index=["train","val","test"]).iloc[:,:5]

**Coding Dataset:**

**Roberta(Base)(Benchmark):**

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

# Load the dataset

dataset = load\_dataset("code\_x\_glue\_cc\_defect\_detection")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the tokenizer and model

model\_name = "roberta-base"

tokenizer = RobertaTokenizer.from\_pretrained(model\_name)

model = RobertaForSequenceClassification.from\_pretrained(model\_name, num\_labels=2)

# Preprocess the data

def preprocess\_function(examples):

inputs = examples['func']

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length')

model\_inputs["labels"] = torch.tensor(examples["target"], dtype=torch.int64)

return model\_inputs

# Apply preprocessing to datasets

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

# Format datasets for PyTorch

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Define training arguments

training\_args = TrainingArguments(

output\_dir='./results',

evaluation\_strategy="epoch",

save\_strategy="epoch",

learning\_rate=1e-5,

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

weight\_decay=0.01,

logging\_dir='./logs',

logging\_steps=100,

load\_best\_model\_at\_end=True,

metric\_for\_best\_model="eval\_loss",

report\_to="none"

)

# Define data collator for dynamic padding

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define custom evaluation metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, predictions, average='macro')

acc = accuracy\_score(labels, predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

# Define a custom Trainer to ensure correct loss computation

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return (loss, outputs) if return\_outputs else loss

# Initialize the Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate on validation and test datasets

print("Validation Results:")

val\_results = trainer.evaluate()

print(val\_results)

print("Test Results:")

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print(test\_results)

**RoBERTa-RNN(Benchmark):**

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, RobertaModel, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

# Load the dataset

dataset = load\_dataset("code\_x\_glue\_cc\_defect\_detection")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "roberta-base"

tokenizer = RobertaTokenizer.from\_pretrained(model\_name)

encoder\_model = RobertaModel.from\_pretrained(model\_name, num\_labels=2)

# Define the custom model

class RoBERTaWithGRU(nn.Module):

def \_\_init\_\_(self, encoder\_model, hidden\_dim, num\_labels):

super(RoBERTaWithGRU, self).\_\_init\_\_()

self.encoder = encoder\_model

self.gru = nn.LSTM(input\_size=encoder\_model.config.hidden\_size, hidden\_size=hidden\_dim, bidirectional=False, batch\_first=True)

self.classifier = nn.Linear(hidden\_dim, num\_labels)

def forward(self, input\_ids, attention\_mask, labels=None):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask)

last\_hidden\_states = encoder\_outputs.last\_hidden\_state

\_, hn = self.gru(last\_hidden\_states)

logits = self.classifier(hn[-1])

loss = None

if labels is not None:

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return {"loss": loss, "logits": logits}

num\_labels = 2 # Assuming binary classification (defect or no defect)

hidden\_dim = 125 # Example hidden dimension for GRU

model = RoBERTaWithGRU(encoder\_model, hidden\_dim, num\_labels)

# Preprocess the data

def preprocess\_function(examples):

inputs = examples['func']

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length')

# Ensure labels are of type Long

model\_inputs["labels"] = examples["target"]

model\_inputs["labels"] = torch.tensor(model\_inputs["labels"], dtype=torch.long)

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=20,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=1e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, predictions, average='macro',warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None,zero\_division=0)

acc = accuracy\_score(labels, predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

# Custom Trainer to ensure loss function is handled correctly

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return (loss, outputs) if return\_outputs else loss

#optimizer = Adafactor(model.parameters(), relative\_step=False, lr=1e-5, weight\_decay=0.01)

#lr\_scheduler = AdafactorSchedule(optimizer)

optimizer = torch.optim.NAdam(model.parameters(), lr=1e-5)

lr\_scheduler=lr\_scheduler.ReduceLROnPlateau(optimizer)

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Validation Results:",eval\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Testing Results:",test\_results)

**CodeBERT(Base)(Benchmark):**

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

# Load the dataset

dataset = load\_dataset("code\_x\_glue\_cc\_defect\_detection")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "microsoft/codebert-base"

tokenizer = RobertaTokenizer.from\_pretrained(model\_name)

model = RobertaForSequenceClassification.from\_pretrained(model\_name, num\_labels=2)

# Preprocess the data

def preprocess\_function(examples):

inputs = examples['func']

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length')

# Ensure labels are of type Long

model\_inputs["labels"] = examples["target"]

model\_inputs["labels"] = torch.tensor(model\_inputs["labels"], dtype=torch.long)

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

evaluation\_strategy="epoch",

learning\_rate=1e-5,

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=3,

#weight\_decay=0.01,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, predictions, average='macro')

acc = accuracy\_score(labels, predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

# Custom Trainer to ensure loss function is handled correctly

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return (loss, outputs) if return\_outputs else loss

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print(eval\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print(test\_results)

**CodeBERT-RNN(Benchmark):**

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, RobertaModel, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

# Load the dataset

dataset = load\_dataset("code\_x\_glue\_cc\_defect\_detection")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "microsoft/codebert-base"

tokenizer = RobertaTokenizer.from\_pretrained(model\_name)

encoder\_model = RobertaModel.from\_pretrained(model\_name, num\_labels=2)

# Define the custom model

class RoBERTaWithGRU(nn.Module):

def \_\_init\_\_(self, encoder\_model, hidden\_dim, num\_labels):

super(RoBERTaWithGRU, self).\_\_init\_\_()

self.encoder = encoder\_model

self.gru = nn.LSTM(input\_size=encoder\_model.config.hidden\_size, hidden\_size=hidden\_dim, bidirectional=False, batch\_first=True)

self.classifier = nn.Linear(hidden\_dim, num\_labels)

def forward(self, input\_ids, attention\_mask, labels=None):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask)

last\_hidden\_states = encoder\_outputs.last\_hidden\_state

\_, hn = self.gru(last\_hidden\_states)

logits = self.classifier(hn[-1])

loss = None

if labels is not None:

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return {"loss": loss, "logits": logits}

num\_labels = 2 # Assuming binary classification (defect or no defect)

hidden\_dim = 125 # Example hidden dimension for GRU

model = RoBERTaWithGRU(encoder\_model, hidden\_dim, num\_labels)

# Preprocess the data

def preprocess\_function(examples):

inputs = examples['func']

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length')

# Ensure labels are of type Long

model\_inputs["labels"] = examples["target"]

model\_inputs["labels"] = torch.tensor(model\_inputs["labels"], dtype=torch.long)

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=20,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=1e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, predictions, average='macro',warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None,zero\_division=0)

acc = accuracy\_score(labels, predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

# Custom Trainer to ensure loss function is handled correctly

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits, labels)

return (loss, outputs) if return\_outputs else loss

#optimizer = Adafactor(model.parameters(), relative\_step=False, lr=1e-5, weight\_decay=0.01)

#lr\_scheduler = AdafactorSchedule(optimizer)

optimizer = torch.optim.NAdam(model.parameters(), lr=1e-5)

lr\_scheduler=lr\_scheduler.ReduceLROnPlateau(optimizer)

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Validation Results:",eval\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Testing Results:",test\_results)

**Codet5(base)(Benchmark)**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from datasets import load\_dataset

import time

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.fc = nn.Linear(768, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

pooled\_output = encoder\_outputs[:, 0] # Use [CLS] token representation

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Load and preprocess the dataset

def load\_code\_x\_glue\_defect\_detection():

dataset = load\_dataset('code\_x\_glue\_cc\_defect\_detection')

train\_codes = dataset['train']['func']

train\_labels = dataset['train']['target']

valid\_codes = dataset['validation']['func']

valid\_labels = dataset['validation']['target']

test\_codes = dataset['test']['func']

test\_labels = dataset['test']['target']

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, tokenizer, max\_length, batch\_size):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs=5, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

# Print total trainable parameters

total\_params = sum(p.numel() for p in model.parameters() if p.requires\_grad)

print(f"Total trainable parameters: {total\_params:,}")

# Start timing

start\_time = time.time()

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

val\_metrics = evaluate\_model(model, val\_loader, device, silent=True)

print(f'Validation Accuracy: {val\_metrics["accuracy"]:.4f}')

# End timing

end\_time = time.time()

training\_time = end\_time - start\_time

print(f"Total training time: {training\_time:.2f} seconds")

# Evaluate the model

def evaluate\_model(model, data\_loader, device, silent=False):

model.eval()

preds = []

labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, label in data\_loader:

input\_ids, attention\_mask, label = input\_ids.to(device), attention\_mask.to(device), label.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, label)

total\_loss += loss.item()

pred = torch.argmax(outputs, dim=1)

preds.extend(pred.cpu().numpy())

labels.extend(label.cpu().numpy())

accuracy = accuracy\_score(labels, preds)

precision = precision\_score(labels, preds, average='weighted')

recall = recall\_score(labels, preds, average='weighted')

f1 = f1\_score(labels, preds, average='weighted')

avg\_loss = total\_loss / len(data\_loader)

metrics = {

"loss": avg\_loss,

"accuracy": accuracy,

"precision": precision,

"recall": recall,

"f1": f1

}

if not silent:

print("Evaluation Metrics:")

print(f'Loss: {avg\_loss:.4f}')

print(f'Accuracy: {accuracy:.4f}')

print(f'Precision: {precision:.4f}')

print(f'Recall: {recall:.4f}')

print(f'F1 Score: {f1:.4f}')

# Generate classification report

report = classification\_report(labels, preds, target\_names=['True', 'False'])

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(labels, preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['True', 'False'], yticklabels=['True', 'False'])

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes):

model = CodeClassifier(encoder, num\_classes, 512)

model.load\_state\_dict(torch.load(path))

return model

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load the dataset

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_code\_x\_glue\_defect\_detection()

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5-base" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), tokenizer, max\_length, batch\_size)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, 512)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'code\_classifier.pth')

# Load the model for evaluation

loaded\_model = load\_model('code\_classifier.pth', encoder\_model, num\_classes)

loaded\_model.to(device)

# Evaluate the model

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, train\_loader, device)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, val\_loader, device)

# Prepare test data loader

test\_dataset = CodeDataset(test\_codes, test\_labels, tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, test\_loader, device)

# Print metrics in desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

# Save the model

#model.save\_pretrained('./codebert-defect-detection')

#tokenizer.save\_pretrained('./codebert-defect-detection')

**Codet5-RNN(Benchmark):**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from datasets import load\_dataset

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.GRU(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=True)

self.fc = nn.Linear(hidden\_dim\*2, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Load and preprocess the dataset

def load\_code\_x\_glue\_defect\_detection():

dataset = load\_dataset('code\_x\_glue\_cc\_defect\_detection')

train\_codes = dataset['train']['func']

train\_labels = dataset['train']['target']

valid\_codes = dataset['validation']['func']

valid\_labels = dataset['validation']['target']

test\_codes = dataset['test']['func']

test\_labels = dataset['test']['target']

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs=5, learning\_rate=1e-6):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

val\_metrics = evaluate\_model(model, val\_loader, device, silent=True)

print(f'Validation Accuracy: {val\_metrics["accuracy"]:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, data\_loader, device, silent=False):

model.eval()

preds = []

labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, label in data\_loader:

input\_ids, attention\_mask, label = input\_ids.to(device), attention\_mask.to(device), label.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, label)

total\_loss += loss.item()

pred = torch.argmax(outputs, dim=1)

preds.extend(pred.cpu().numpy())

labels.extend(label.cpu().numpy())

accuracy = accuracy\_score(labels, preds)

precision = precision\_score(labels, preds, average='weighted')

recall = recall\_score(labels, preds, average='weighted')

f1 = f1\_score(labels, preds, average='weighted')

avg\_loss = total\_loss / len(data\_loader)

metrics = {

"loss": avg\_loss,

"accuracy": accuracy,

"precision": precision,

"recall": recall,

"f1": f1

}

if not silent:

print("Evaluation Metrics:")

print(f'Loss: {avg\_loss:.4f}')

print(f'Accuracy: {accuracy:.4f}')

print(f'Precision: {precision:.4f}')

print(f'Recall: {recall:.4f}')

print(f'F1 Score: {f1:.4f}')

# Generate classification report

report = classification\_report(labels, preds, target\_names=['True', 'False'])

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(labels, preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['True', 'False'], yticklabels=['True', 'False'])

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load the dataset

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_code\_x\_glue\_defect\_detection()

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5-base" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, train\_loader, device)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, val\_loader, device)

# Prepare test data loader

test\_dataset = CodeDataset(test\_codes, test\_labels, tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, test\_loader, device)

# Print metrics in desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

**codet5-RNN(search dataset) Real world:**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.GRU(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=False)

self.fc = nn.Linear(hidden\_dim, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'search\_5\_classes\_row\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5-base" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

Codet5-rnn(sort-search) Real world:

#LSTM

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.LSTM(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=False)

self.fc = nn.Linear(hidden\_dim, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'sort\_search\_row\_converted\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5-base" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 512

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

**Codet5-rnn(sort-search-graph)Realworld:**

#Bi-LSTM

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.LSTM(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=True)

self.fc = nn.Linear(hidden\_dim\*2, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=1e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.RMSprop(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'sort\_search\_graph\_tree\_row\_converted\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5-base" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

**codet5(+)-Base(Benchmark Dataset):**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from datasets import load\_dataset

import time

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.fc = nn.Linear(768, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

pooled\_output = encoder\_outputs[:, 0] # Use [CLS] token representation

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Load and preprocess the dataset

def load\_code\_x\_glue\_defect\_detection():

dataset = load\_dataset('code\_x\_glue\_cc\_defect\_detection')

train\_codes = dataset['train']['func']

train\_labels = dataset['train']['target']

valid\_codes = dataset['validation']['func']

valid\_labels = dataset['validation']['target']

test\_codes = dataset['test']['func']

test\_labels = dataset['test']['target']

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, tokenizer, max\_length, batch\_size):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs=5, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

# Print total trainable parameters

total\_params = sum(p.numel() for p in model.parameters() if p.requires\_grad)

print(f"Total trainable parameters: {total\_params:,}")

# Start timing

start\_time = time.time()

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

val\_metrics = evaluate\_model(model, val\_loader, device, silent=True)

print(f'Validation Accuracy: {val\_metrics["accuracy"]:.4f}')

# End timing

end\_time = time.time()

training\_time = end\_time - start\_time

print(f"Total training time: {training\_time:.2f} seconds")

# Evaluate the model

def evaluate\_model(model, data\_loader, device, silent=False):

model.eval()

preds = []

labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, label in data\_loader:

input\_ids, attention\_mask, label = input\_ids.to(device), attention\_mask.to(device), label.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, label)

total\_loss += loss.item()

pred = torch.argmax(outputs, dim=1)

preds.extend(pred.cpu().numpy())

labels.extend(label.cpu().numpy())

accuracy = accuracy\_score(labels, preds)

precision = precision\_score(labels, preds, average='weighted')

recall = recall\_score(labels, preds, average='weighted')

f1 = f1\_score(labels, preds, average='weighted')

avg\_loss = total\_loss / len(data\_loader)

metrics = {

"loss": avg\_loss,

"accuracy": accuracy,

"precision": precision,

"recall": recall,

"f1": f1

}

if not silent:

print("Evaluation Metrics:")

print(f'Loss: {avg\_loss:.4f}')

print(f'Accuracy: {accuracy:.4f}')

print(f'Precision: {precision:.4f}')

print(f'Recall: {recall:.4f}')

print(f'F1 Score: {f1:.4f}')

# Generate classification report

report = classification\_report(labels, preds, target\_names=['True', 'False'])

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(labels, preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['True', 'False'], yticklabels=['True', 'False'])

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes):

model = CodeClassifier(encoder, num\_classes, 512)

model.load\_state\_dict(torch.load(path))

return model

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load the dataset

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_code\_x\_glue\_defect\_detection()

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5p-220m" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), tokenizer, max\_length, batch\_size)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, 512)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'code\_classifier.pth')

# Load the model for evaluation

loaded\_model = load\_model('code\_classifier.pth', encoder\_model, num\_classes)

loaded\_model.to(device)

# Evaluate the model

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, train\_loader, device)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, val\_loader, device)

# Prepare test data loader

test\_dataset = CodeDataset(test\_codes, test\_labels, tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, test\_loader, device)

# Print metrics in desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

**codet5(+)-RNN(Benchmark Dataset):**

#Bi-GRU-256

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from datasets import load\_dataset

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.GRU(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=True)

self.fc = nn.Linear(hidden\_dim\*2, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Load and preprocess the dataset

def load\_code\_x\_glue\_defect\_detection():

dataset = load\_dataset('code\_x\_glue\_cc\_defect\_detection')

train\_codes = dataset['train']['func']

train\_labels = dataset['train']['target']

valid\_codes = dataset['validation']['func']

valid\_labels = dataset['validation']['target']

test\_codes = dataset['test']['func']

test\_labels = dataset['test']['target']

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs=5, learning\_rate=1e-6):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

val\_metrics = evaluate\_model(model, val\_loader, device, silent=True)

print(f'Validation Accuracy: {val\_metrics["accuracy"]:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, data\_loader, device, silent=False):

model.eval()

preds = []

labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, label in data\_loader:

input\_ids, attention\_mask, label = input\_ids.to(device), attention\_mask.to(device), label.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, label)

total\_loss += loss.item()

pred = torch.argmax(outputs, dim=1)

preds.extend(pred.cpu().numpy())

labels.extend(label.cpu().numpy())

accuracy = accuracy\_score(labels, preds)

precision = precision\_score(labels, preds, average='weighted')

recall = recall\_score(labels, preds, average='weighted')

f1 = f1\_score(labels, preds, average='weighted')

avg\_loss = total\_loss / len(data\_loader)

metrics = {

"loss": avg\_loss,

"accuracy": accuracy,

"precision": precision,

"recall": recall,

"f1": f1

}

if not silent:

print("Evaluation Metrics:")

print(f'Loss: {avg\_loss:.4f}')

print(f'Accuracy: {accuracy:.4f}')

print(f'Precision: {precision:.4f}')

print(f'Recall: {recall:.4f}')

print(f'F1 Score: {f1:.4f}')

# Generate classification report

report = classification\_report(labels, preds, target\_names=['True', 'False'])

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(labels, preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['True', 'False'], yticklabels=['True', 'False'])

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load the dataset

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_code\_x\_glue\_defect\_detection()

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5p-220m" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, train\_loader, device)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, val\_loader, device)

# Prepare test data loader

test\_dataset = CodeDataset(test\_codes, test\_labels, tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, test\_loader, device)

# Print metrics in desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

Code classification(real world dataset):

codet5(+) - rnn(search dataset):

#GRU

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.GRU(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=False)

self.fc = nn.Linear(hidden\_dim, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'search\_5\_classes\_row\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5p-220m" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

codet5(+)-rnn(sort-search):

#LSTM

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.LSTM(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=False)

self.fc = nn.Linear(hidden\_dim, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=2e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.NAdam(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'sort\_search\_row\_converted\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5p-220m" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 512

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

Codet5(+)-rnn(sort-search-graph):

#Bi-LSTM

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from transformers import T5Tokenizer, T5EncoderModel, AutoTokenizer

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

import os

# Dataset class

class CodeDataset(Dataset):

def \_\_init\_\_(self, codes, labels, tokenizer, max\_length):

self.codes = codes

self.labels = labels

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_len\_\_(self):

return len(self.codes)

def \_\_getitem\_\_(self, idx):

code = self.codes[idx]

label = self.labels[idx]

if isinstance(code, list):

code = [str(i) for i in code]

else:

code = [str(code)]

encodings = self.tokenizer(code, truncation=True, padding='max\_length', max\_length=self.max\_length, return\_tensors="pt")

input\_ids = encodings['input\_ids'].squeeze()

attention\_mask = encodings['attention\_mask'].squeeze()

return input\_ids, attention\_mask, label

# Model class

class CodeClassifier(nn.Module):

def \_\_init\_\_(self, encoder, num\_classes, hidden\_dim):

super(CodeClassifier, self).\_\_init\_\_()

self.encoder = encoder

self.gru = nn.LSTM(input\_size=768, hidden\_size=hidden\_dim, batch\_first=True, bidirectional=True)

self.fc = nn.Linear(hidden\_dim\*2, num\_classes)

self.dropout = nn.Dropout(0.2)

def forward(self, input\_ids, attention\_mask):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask).last\_hidden\_state

gru\_outputs, \_ = self.gru(encoder\_outputs)

pooled\_output = torch.max(gru\_outputs, 1)[0]

pooled\_output = self.dropout(pooled\_output)

logits = self.fc(pooled\_output)

return logits

# Function to load and split the custom dataset

def load\_and\_split\_dataset(file\_path, test\_size=0.2, val\_size=0.1):

df = pd.read\_csv(file\_path)

codes = df['Code'].tolist()

labels = df['Class'].tolist()

train\_val\_codes, test\_codes, train\_val\_labels, test\_labels = train\_test\_split(codes, labels, test\_size=test\_size, random\_state=42)

train\_codes, valid\_codes, train\_labels, valid\_labels = train\_test\_split(train\_val\_codes, train\_val\_labels, test\_size=val\_size, random\_state=42)

return (train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels)

# Encode labels

def encode\_labels(labels):

encoder = LabelEncoder()

encoded\_labels = encoder.fit\_transform(labels)

return encoded\_labels, encoder

# Prepare DataLoader

def create\_dataloaders(train\_data, valid\_data, batch\_size, tokenizer, max\_length):

train\_dataset = CodeDataset(train\_data[0], train\_data[1], tokenizer, max\_length)

valid\_dataset = CodeDataset(valid\_data[0], valid\_data[1], tokenizer, max\_length)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size)

return train\_loader, val\_loader

# Train the model

def train\_model(model, train\_loader, val\_loader, device, num\_epochs, learning\_rate=1e-5):

criterion = nn.CrossEntropyLoss()

optimizer = optim.RMSprop(model.parameters(), lr=learning\_rate)

model.to(device)

for epoch in range(num\_epochs):

model.train()

total\_loss = 0

for input\_ids, attention\_mask, labels in train\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_train\_loss = total\_loss / len(train\_loader)

print(f'Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}')

# Validation

model.eval()

val\_preds = []

val\_labels = []

with torch.no\_grad():

for input\_ids, attention\_mask, labels in val\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

preds = torch.argmax(outputs, dim=1)

val\_preds.extend(preds.cpu().numpy())

val\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(val\_labels, val\_preds)

print(f'Validation Accuracy: {accuracy:.4f}')

# Evaluate the model and print metrics

def evaluate\_model(model, test\_data, device, encoder, batch\_size, tokenizer, max\_length):

test\_dataset = CodeDataset(test\_data[0], test\_data[1], tokenizer, max\_length)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

model.eval()

test\_preds = []

test\_labels = []

total\_loss = 0

criterion = nn.CrossEntropyLoss()

with torch.no\_grad():

for input\_ids, attention\_mask, labels in test\_loader:

input\_ids, attention\_mask, labels = input\_ids.to(device), attention\_mask.to(device), labels.to(device)

outputs = model(input\_ids, attention\_mask)

loss = criterion(outputs, labels)

total\_loss += loss.item()

preds = torch.argmax(outputs, dim=1)

test\_preds.extend(preds.cpu().numpy())

test\_labels.extend(labels.cpu().numpy())

accuracy = accuracy\_score(test\_labels, test\_preds)

precision = precision\_score(test\_labels, test\_preds, average='weighted')

recall = recall\_score(test\_labels, test\_preds, average='weighted')

f1 = f1\_score(test\_labels, test\_preds, average='weighted')

metrics = {

'loss': total\_loss / len(test\_loader),

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'f1': f1

}

# Generate classification report

report = classification\_report(test\_labels, test\_preds, target\_names=encoder.classes\_)

print("Classification Report: \n", report)

# Compute confusion matrix

conf\_matrix = confusion\_matrix(test\_labels, test\_preds)

# Plot confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes\_, yticklabels=encoder.classes\_)

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

return metrics

# Save the model

def save\_model(model, path='code\_classifier.pth'):

create\_directory\_if\_not\_exists(path)

torch.save(model.state\_dict(), path)

# Load the model

def load\_model(path, encoder, num\_classes, hidden\_dim):

model = CodeClassifier(encoder, num\_classes, hidden\_dim)

model.load\_state\_dict(torch.load(path))

return model

# Ensure directory creation for saving models

def create\_directory\_if\_not\_exists(path):

directory = os.path.dirname(path)

if directory and not os.path.exists(directory):

os.makedirs(directory)

# Main script

if \_\_name\_\_ == "\_\_main\_\_":

# Load and split the dataset

dataset\_file = 'sort\_search\_graph\_tree\_row\_converted\_filtered.csv'

(train\_codes, train\_labels), (valid\_codes, valid\_labels), (test\_codes, test\_labels) = load\_and\_split\_dataset(dataset\_file)

# Encode labels

train\_labels, label\_encoder = encode\_labels(train\_labels)

valid\_labels = label\_encoder.transform(valid\_labels)

test\_labels = label\_encoder.transform(test\_labels)

# Load tokenizer and preprocess data

pretrained\_model\_name = "Salesforce/codet5p-220m" # Replace with your pre-trained model name

tokenizer = AutoTokenizer.from\_pretrained(pretrained\_model\_name)

max\_length = 512

# Prepare dataloaders

batch\_size = 8

train\_loader, val\_loader = create\_dataloaders((train\_codes, train\_labels), (valid\_codes, valid\_labels), batch\_size, tokenizer, max\_length)

# Load the encoder model

encoder\_model = T5EncoderModel.from\_pretrained(pretrained\_model\_name)

# Build and train the model

num\_classes = len(label\_encoder.classes\_)

hidden\_dim = 256

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = CodeClassifier(encoder\_model, num\_classes, hidden\_dim)

train\_model(model, train\_loader, val\_loader, device, num\_epochs=5)

# Save the trained model

save\_model(model, 'model\_saves/code\_classifier.pth')

# Load the model for prediction

loaded\_model = load\_model('model\_saves/code\_classifier.pth', encoder\_model, num\_classes, hidden\_dim)

loaded\_model.to(device)

# Evaluate the model on training, validation, and test datasets

print("Train Dataset:")

train\_metrics = evaluate\_model(loaded\_model, (train\_codes, train\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Validation Dataset:")

val\_metrics = evaluate\_model(loaded\_model, (valid\_codes, valid\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

print("Test Dataset:")

test\_metrics = evaluate\_model(loaded\_model, (test\_codes, test\_labels), device, label\_encoder, batch\_size, tokenizer, max\_length)

# Print metrics in the desired format

print("\neval\_loss\teval\_accuracy\teval\_f1\teval\_precision\teval\_recall")

print(f"train\t{train\_metrics['loss']:.6f}\t{train\_metrics['accuracy']:.6f}\t{train\_metrics['f1']:.6f}\t{train\_metrics['precision']:.6f}\t{train\_metrics['recall']:.6f}")

print(f"val\t{val\_metrics['loss']:.6f}\t{val\_metrics['accuracy']:.6f}\t{val\_metrics['f1']:.6f}\t{val\_metrics['precision']:.6f}\t{val\_metrics['recall']:.6f}")

print(f"test\t{test\_metrics['loss']:.6f}\t{test\_metrics['accuracy']:.6f}\t{test\_metrics['f1']:.6f}\t{test\_metrics['precision']:.6f}\t{test\_metrics['recall']:.6f}")

**Biomedical Dataset(ncbi)**

**RoBERTa-Base:**

#roberta-bigru

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, RobertaModel, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

from transformers import RobertaTokenizerFast

# Load the dataset

dataset = load\_dataset("ncbi/ncbi\_disease")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "roberta-base"

tokenizer = RobertaTokenizerFast.from\_pretrained(model\_name, add\_prefix\_space=True)

model = RobertaForSequenceClassification.from\_pretrained(model\_name, num\_labels=3)

# Preprocess the data

def preprocess\_function(examples):

# Ensure inputs are always lists

inputs = [x if isinstance(x, list) else [x] for x in examples['tokens']]

labels = examples['ner\_tags']

# Tokenize the inputs with padding and truncation

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length', is\_split\_into\_words=True)

all\_new\_labels = []

for i in range(len(examples['tokens'])):

word\_ids = model\_inputs.word\_ids(batch\_index=i) # Get word IDs for each example

new\_labels = []

previous\_word\_idx = None

for word\_idx in word\_ids:

if word\_idx is None:

new\_labels.append(-100)

elif word\_idx != previous\_word\_idx:

new\_labels.append(labels[i][word\_idx]) # Access labels for the current example using index 'i'

else:

new\_labels.append(-100)

previous\_word\_idx = word\_idx

all\_new\_labels.append(new\_labels)

# Assign the list of new\_labels to the 'labels' key

model\_inputs["labels"] = all\_new\_labels

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=1e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

# Flatten the labels and predictions and remove -100 values

true\_labels = [l for label\_list in labels for l in label\_list if l != -100]

true\_predictions = []

#Adjust predictions length to match true\_labels

for i, label\_list in enumerate(labels):

true\_predictions.extend([predictions[i]] \* len([l for l in label\_list if l != -100]))

precision, recall, f1, \_ = precision\_recall\_fscore\_support(true\_labels, true\_predictions, average='weighted', warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None, zero\_division=0)

acc = accuracy\_score(true\_labels, true\_predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False, num\_items\_in\_batch=None): # Add num\_items\_in\_batch argument

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

main\_labels = []

for label\_sequence in labels:

try:

main\_label\_index = next((i for i, label in enumerate(label\_sequence) if label != -100), None)

if main\_label\_index is not None:

main\_labels.append(label\_sequence[main\_label\_index])

else:

# Handle cases where all labels are -100

main\_labels.append(0) # You might need to adjust this default value

except StopIteration:

# Handle empty label sequences

main\_labels.append(0) # Adjust default value as needed

main\_labels = torch.tensor(main\_labels, device=logits.device,dtype=torch.long) # Convert to tensor and ensure it's on the same device as logits

# Now use main\_labels for the loss calculation

loss = loss\_fct(logits, main\_labels) # Reshape logits to (batch\_size, num\_labels)

return (loss, outputs) if return\_outputs else loss

#optimizer = Adafactor(model.parameters(), relative\_step=False, lr=1e-5, weight\_decay=0.01)

#lr\_scheduler = AdafactorSchedule(optimizer)

#optimizer = torch.optim.NAdam(model.parameters(), lr=1e-5)

#lr\_scheduler=lr\_scheduler.ReduceLROnPlateau(optimizer)

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Training Results:",eval\_results)

val\_results = trainer.evaluate(eval\_dataset=val\_dataset)

print("Validation Results:",val\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Test Results:",test\_results)

**RoBERTa-RNN**

#robera-bilstm

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, RobertaModel, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

from transformers import RobertaTokenizerFast

# Load the dataset

dataset = load\_dataset("ncbi/ncbi\_disease")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "roberta-base"

tokenizer = RobertaTokenizerFast.from\_pretrained(model\_name, add\_prefix\_space=True)

encoder\_model = RobertaModel.from\_pretrained(model\_name, num\_labels=3)

# Define the custom model

class RoBERTaWithBiLSTM(nn.Module):

def \_\_init\_\_(self, encoder\_model, hidden\_dim, num\_labels):

super(RoBERTaWithBiLSTM, self).\_\_init\_\_()

self.encoder = encoder\_model

self.bilstm = nn.LSTM(

input\_size=encoder\_model.config.hidden\_size,

hidden\_size=hidden\_dim,

bidirectional=True,

batch\_first=True

)

self.classifier = nn.Linear(hidden\_dim \* 2, num\_labels) # Multiply hidden\_dim by 2 for bidirectional LSTM

def forward(self, input\_ids, attention\_mask, labels=None):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask)

last\_hidden\_states = encoder\_outputs.last\_hidden\_state

# Get the last hidden state of BiLSTM for each sequence in the batch

output, (hn, cn) = self.bilstm(last\_hidden\_states)

# Concatenate the forward and backward hidden states

logits = self.classifier(torch.cat((hn[-2], hn[-1]), dim=1))

loss = None

if labels is not None:

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits.view(-1, num\_labels), labels.view(-1))

return {"loss": loss, "logits": logits}

num\_labels = 3 # Assuming binary classification (defect or no defect)

hidden\_dim = 512 # Example hidden dimension for BiLSTM

model = RoBERTaWithBiLSTM(encoder\_model, hidden\_dim, num\_labels)

# Preprocess the data

def preprocess\_function(examples):

# Ensure inputs are always lists

inputs = [x if isinstance(x, list) else [x] for x in examples['tokens']]

labels = examples['ner\_tags']

# Tokenize the inputs with padding and truncation

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length', is\_split\_into\_words=True)

all\_new\_labels = []

for i in range(len(examples['tokens'])):

word\_ids = model\_inputs.word\_ids(batch\_index=i) # Get word IDs for each example

new\_labels = []

previous\_word\_idx = None

for word\_idx in word\_ids:

if word\_idx is None:

new\_labels.append(-100)

elif word\_idx != previous\_word\_idx:

new\_labels.append(labels[i][word\_idx]) # Access labels for the current example using index 'i'

else:

new\_labels.append(-100)

previous\_word\_idx = word\_idx

all\_new\_labels.append(new\_labels)

model\_inputs["labels"] = all\_new\_labels

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=2e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

true\_labels = [l for label\_list in labels for l in label\_list if l != -100]

true\_predictions = []

for i, label\_list in enumerate(labels):

true\_predictions.extend([predictions[i]] \* len([l for l in label\_list if l != -100]))

precision, recall, f1, \_ = precision\_recall\_fscore\_support(

true\_labels, true\_predictions, average='weighted', warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None, zero\_division=0

)

acc = accuracy\_score(true\_labels, true\_predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False, num\_items\_in\_batch=None):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

main\_labels = []

for label\_sequence in labels:

try:

main\_label\_index = next((i for i, label in enumerate(label\_sequence) if label != -100), None)

if main\_label\_index is not None:

main\_labels.append(label\_sequence[main\_label\_index])

else:

main\_labels.append(0)

except StopIteration:

main\_labels.append(0)

main\_labels = torch.tensor(main\_labels, device=logits.device, dtype=torch.long)

loss = loss\_fct(logits, main\_labels)

return (loss, outputs) if return\_outputs else loss

optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

lr\_scheduler = lr\_scheduler.ReduceLROnPlateau(optimizer)

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Training Results:", eval\_results)

val\_results = trainer.evaluate(eval\_dataset=val\_dataset)

print("Validation Results:", val\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Test Results:", test\_results)

**BioLinkBERT-Base:**

import numpy as np

from datasets import load\_dataset

from transformers import RobertaTokenizer, RobertaForSequenceClassification, RobertaModel, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

from transformers import RobertaTokenizerFast

from transformers import AutoTokenizer, AutoModelForSequenceClassification # Import AutoModelForSequenceClassification

# Load the dataset

dataset = load\_dataset("ncbi/ncbi\_disease")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "michiyasunaga/BioLinkBERT-base"

tokenizer = AutoTokenizer.from\_pretrained(model\_name, add\_prefix\_space=True)

model = AutoModelForSequenceClassification.from\_pretrained(model\_name, num\_labels=3)

# Preprocess the data

def preprocess\_function(examples):

# Ensure inputs are always lists

inputs = [x if isinstance(x, list) else [x] for x in examples['tokens']]

labels = examples['ner\_tags']

# Tokenize the inputs with padding and truncation

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length', is\_split\_into\_words=True)

all\_new\_labels = []

for i in range(len(examples['tokens'])):

word\_ids = model\_inputs.word\_ids(batch\_index=i) # Get word IDs for each example

new\_labels = []

previous\_word\_idx = None

for word\_idx in word\_ids:

if word\_idx is None:

new\_labels.append(-100)

elif word\_idx != previous\_word\_idx:

new\_labels.append(labels[i][word\_idx]) # Access labels for the current example using index 'i'

else:

new\_labels.append(-100)

previous\_word\_idx = word\_idx

all\_new\_labels.append(new\_labels)

# Assign the list of new\_labels to the 'labels' key

model\_inputs["labels"] = all\_new\_labels

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=1e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

# Flatten the labels and predictions and remove -100 values

true\_labels = [l for label\_list in labels for l in label\_list if l != -100]

true\_predictions = []

#Adjust predictions length to match true\_labels

for i, label\_list in enumerate(labels):

true\_predictions.extend([predictions[i]] \* len([l for l in label\_list if l != -100]))

precision, recall, f1, \_ = precision\_recall\_fscore\_support(true\_labels, true\_predictions, average='weighted', warn\_for=('precision', 'recall', 'f-score'), sample\_weight=None, zero\_division=0)

acc = accuracy\_score(true\_labels, true\_predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False, num\_items\_in\_batch=None): # Add num\_items\_in\_batch argument

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

main\_labels = []

for label\_sequence in labels:

try:

main\_label\_index = next((i for i, label in enumerate(label\_sequence) if label != -100), None)

if main\_label\_index is not None:

main\_labels.append(label\_sequence[main\_label\_index])

else:

# Handle cases where all labels are -100

main\_labels.append(0) # You might need to adjust this default value

except StopIteration:

# Handle empty label sequences

main\_labels.append(0) # Adjust default value as needed

main\_labels = torch.tensor(main\_labels, device=logits.device,dtype=torch.long) # Convert to tensor and ensure it's on the same device as logits

# Now use main\_labels for the loss calculation

loss = loss\_fct(logits, main\_labels) # Reshape logits to (batch\_size, num\_labels)

return (loss, outputs) if return\_outputs else loss

#optimizer = Adafactor(model.parameters(), relative\_step=False, lr=1e-5, weight\_decay=0.01)

#lr\_scheduler = AdafactorSchedule(optimizer)

#optimizer = torch.optim.NAdam(model.parameters(), lr=1e-5)

#lr\_scheduler=lr\_scheduler.ReduceLROnPlateau(optimizer)

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Training Results:",eval\_results)

val\_results = trainer.evaluate(eval\_dataset=val\_dataset)

print("Validation Results:",val\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Test Results:",test\_results)

**BioLinkBERT-RNN:**

import numpy as np

from datasets import load\_dataset

from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Trainer, DataCollatorWithPadding

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

import torch

import torch.nn as nn

import torch.optim.lr\_scheduler as lr\_scheduler

# Load the dataset

dataset = load\_dataset("ncbi/ncbi\_disease")

train\_dataset = dataset['train']

val\_dataset = dataset['validation']

test\_dataset = dataset['test']

# Load the model and tokenizer

model\_name = "michiyasunaga/BioLinkBERT-base"

tokenizer = AutoTokenizer.from\_pretrained(model\_name, add\_prefix\_space=True)

encoder\_model = AutoModelForSequenceClassification.from\_pretrained(model\_name, num\_labels=3)

# Define the BioLinkBERT-BiGRU model

class BioLinkBERTBiGRU(nn.Module):

def \_\_init\_\_(self, encoder\_model, hidden\_dim, num\_labels):

super(BioLinkBERTBiGRU, self).\_\_init\_\_()

self.encoder = encoder\_model

self.gru = nn.GRU(input\_size=encoder\_model.config.hidden\_size, hidden\_size=hidden\_dim, bidirectional=True, batch\_first=True)

self.classifier = nn.Linear(hidden\_dim \* 2, num\_labels) # Hidden\_dim \* 2 for bidirectional GRU

def forward(self, input\_ids, attention\_mask, labels=None):

encoder\_outputs = self.encoder(input\_ids=input\_ids, attention\_mask=attention\_mask, output\_hidden\_states=True)

# Get the last hidden state

encoder\_outputs = encoder\_outputs.hidden\_states[-1]

# Get GRU outputs

output, hn = self.gru(encoder\_outputs) # hn will have 2 hidden states (forward and backward)

# Concatenate forward and backward hidden states

logits = self.classifier(torch.cat((hn[-2], hn[-1]), dim=-1)) # Concatenate forward and backward GRU outputs

loss = None

if labels is not None:

loss\_fct = nn.CrossEntropyLoss()

loss = loss\_fct(logits.view(-1, num\_labels), labels.view(-1))

return {"loss": loss, "logits": logits}

# Set up the model

num\_labels = 3

hidden\_dim = 512

model = BioLinkBERTBiGRU(encoder\_model, hidden\_dim, num\_labels)

# Preprocess the data

def preprocess\_function(examples):

inputs = [x if isinstance(x, list) else [x] for x in examples['tokens']]

labels = examples['ner\_tags']

model\_inputs = tokenizer(inputs, max\_length=512, truncation=True, padding='max\_length', is\_split\_into\_words=True)

all\_new\_labels = []

for i in range(len(examples['tokens'])):

word\_ids = model\_inputs.word\_ids(batch\_index=i)

new\_labels = []

previous\_word\_idx = None

for word\_idx in word\_ids:

if word\_idx is None:

new\_labels.append(-100)

elif word\_idx != previous\_word\_idx:

new\_labels.append(labels[i][word\_idx])

else:

new\_labels.append(-100)

previous\_word\_idx = word\_idx

all\_new\_labels.append(new\_labels)

model\_inputs["labels"] = all\_new\_labels

return model\_inputs

train\_dataset = train\_dataset.map(preprocess\_function, batched=True)

val\_dataset = val\_dataset.map(preprocess\_function, batched=True)

test\_dataset = test\_dataset.map(preprocess\_function, batched=True)

train\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

val\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

test\_dataset.set\_format(type='torch', columns=['input\_ids', 'attention\_mask', 'labels'])

# Training arguments

training\_args = TrainingArguments(

output\_dir='./results',

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=5,

evaluation\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

learning\_rate=1e-5,

)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Define metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

true\_labels = [l for label\_list in labels for l in label\_list if l != -100]

true\_predictions = []

for i, label\_list in enumerate(labels):

true\_predictions.extend([predictions[i]] \* len([l for l in label\_list if l != -100]))

precision, recall, f1, \_ = precision\_recall\_fscore\_support(true\_labels, true\_predictions, average='weighted', zero\_division=0)

acc = accuracy\_score(true\_labels, true\_predictions)

return {

'accuracy': acc,

'f1': f1,

'precision': precision,

'recall': recall

}

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False, num\_items\_in\_batch=None):

labels = inputs.pop("labels")

outputs = model(\*\*inputs)

logits = outputs.get("logits")

loss\_fct = nn.CrossEntropyLoss()

main\_labels = []

for label\_sequence in labels:

try:

main\_label\_index = next((i for i, label in enumerate(label\_sequence) if label != -100), None)

if main\_label\_index is not None:

main\_labels.append(label\_sequence[main\_label\_index])

else:

main\_labels.append(0)

except StopIteration:

main\_labels.append(0)

main\_labels = torch.tensor(main\_labels, device=logits.device, dtype=torch.long)

loss = loss\_fct(logits, main\_labels)

return (loss, outputs) if return\_outputs else loss

optimizer = torch.optim.NAdam(model.parameters(), lr=1e-5)

lr\_scheduler = lr\_scheduler.ReduceLROnPlateau(optimizer)

# Initialize Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=val\_dataset,

tokenizer=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

# Train the model

trainer.train()

# Evaluate the model

eval\_results = trainer.evaluate()

print("Training Results:", eval\_results)

val\_results = trainer.evaluate(eval\_dataset=val\_dataset)

print("Validation Results:", val\_results)

test\_results = trainer.evaluate(eval\_dataset=test\_dataset)

print("Test Results:", test\_results)