Model Introduction:-

"The Deep Learning model predicts areas more likely to be error-prone or risky based on given code snippets. This document outlines the model's construction approach, covering key components such as data preprocessing, model architecture, training, evaluation, and visual representations using Flask. Explore comprehensive insights into the predictive system for effective defect identification in software development."

1. Data Loading and Preprocessing

1.1 Load Dataset

The code dataset is loaded from a CSV file using the Pandas library.

```
data = pd.read_csv('code.csv')
```

1.2 Encode Labels

The 'expected_output' column is encoded using LabelEncoder to convert categorical labels into numerical format.

```
label_encoder = LabelEncoder()
data['expected_output_encoded'] = label_encoder.fit_transform(data['expected_output'])
```

1.3 Tokenize and Pad Sequences

Tokenization is performed on code snippets using the Keras Tokenizer. The sequences are then padded to a fixed length to ensure uniform input size for the model.

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(data['code_snippet'])
total_words = len(tokenizer.word_index) + 1
```

```
input_sequences = tokenizer.texts_to_sequences(data['code_snippet'])
padded_sequences = pad_sequences(input_sequences)
```

1.4 Train-Test Split

The dataset is split into training and testing sets using the train_test_split function from scikit-learn.

```
X_train, X_test, y_train, y_test = train_test_split(padded_sequences,
data['expected_output_encoded'], test_size=0.2, random_state=42)
```

1.5 Convert to PyTorch Tensors and Create DataLoaders

Convert the data into PyTorch tensors and create DataLoader objects for efficient batch processing during training and testing

```
# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.long)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.long)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)

# Create DataLoader for training and testing
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

2. Model Architecture

2.1 LSTM Model for Multi-Class Classification

A Long Short-Term Memory (LSTM) neural network is employed for the defect prediction task. The model consists of an embedding layer, an LSTM layer, and a fully connected layer.

```
class LSTMModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, output_dim):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        lstm_out = lstm_out[:, -1, :]
        output = self.fc(lstm_out)
        return output
```

3. Model Training

3.1 Loss and Optimizer

The CrossEntropyLoss is used as the loss function, and the Adam optimizer is employed for model parameter updates.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3.2 Training Loop

The model is trained over multiple epochs using a training loop. Gradients are zeroed before each iteration, and backpropagation is performed to update the model parameters.

```
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    for batch_x, batch_y in train_loader:
        optimizer.zero_grad()
        output = model(batch_x)
        loss = criterion(output, batch_y)
        loss.backward()
        optimizer.step()
```

4. Model Evaluation

4.1 Testing and Prediction

The trained model is evaluated on the testing set. Predictions are made using the trained model, and accuracy is calculated using scikit-learn's accuracy_score function.

```
model.eval()
with torch.no_grad():
    predictions = []
    for batch_x, batch_y in test_loader:
        output = model(batch_x)
        _, predicted_labels = torch.max(output, 1)
        predictions.extend(predicted_labels.cpu().numpy())
```

4.2 Accuracy Calculation

The accuracy of the model is calculated using the ground truth labels and predicted labels.

```
accuracy = accuracy_score(y_test, predictions)

print(f"Test Accuracy: {accuracy}")
```

```
# Evaluate the model
model.eval()
with torch.no_grad():
    predictions = []
    for batch_x, batch_y in test_loader:
        output = model(batch_x)
        _, predicted_labels = torch.max(output, 1)
        predictions.extend(predicted_labels.cpu().numpy())

# Calculate accuracy for multi-class classification
accuracy = accuracy_score(y_test, predictions)
print(f"Test Accuracy: {accuracy}")
```

[8]

```
Test Accuracy: 0.8863636363636364
```

5. Saving The Trained Model & Loarding The Saved Model

Save the PyTorch model (in .pth format) after training, and later load it for making future predictions.

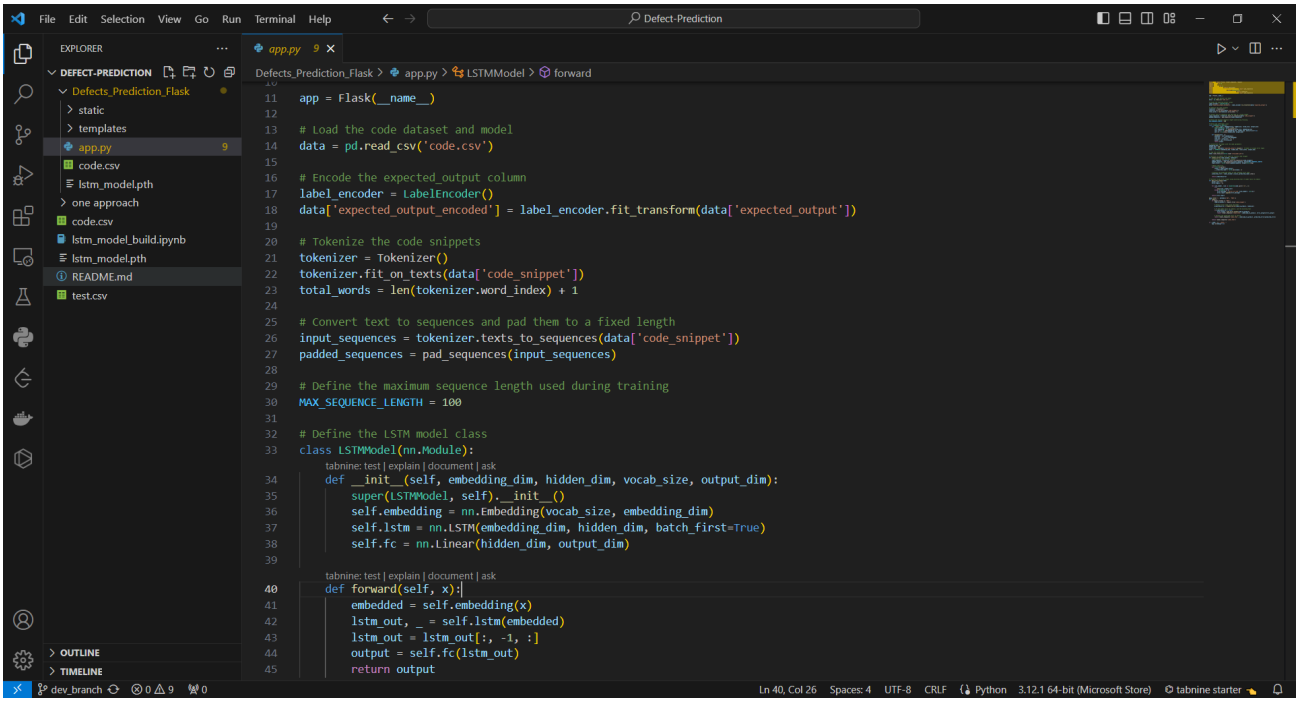
In [9]:
```
# Save the trained model
torch.save(model.state_dict(), 'lstm_model.pth')
```

In [10]:
```
# Load the saved model
loaded_model = LSTMModel(embedding_dim, hidden_dim, total_words, output_dim)
loaded_model.load_state_dict(torch.load('lstm_model.pth'))
loaded_model.eval()
```

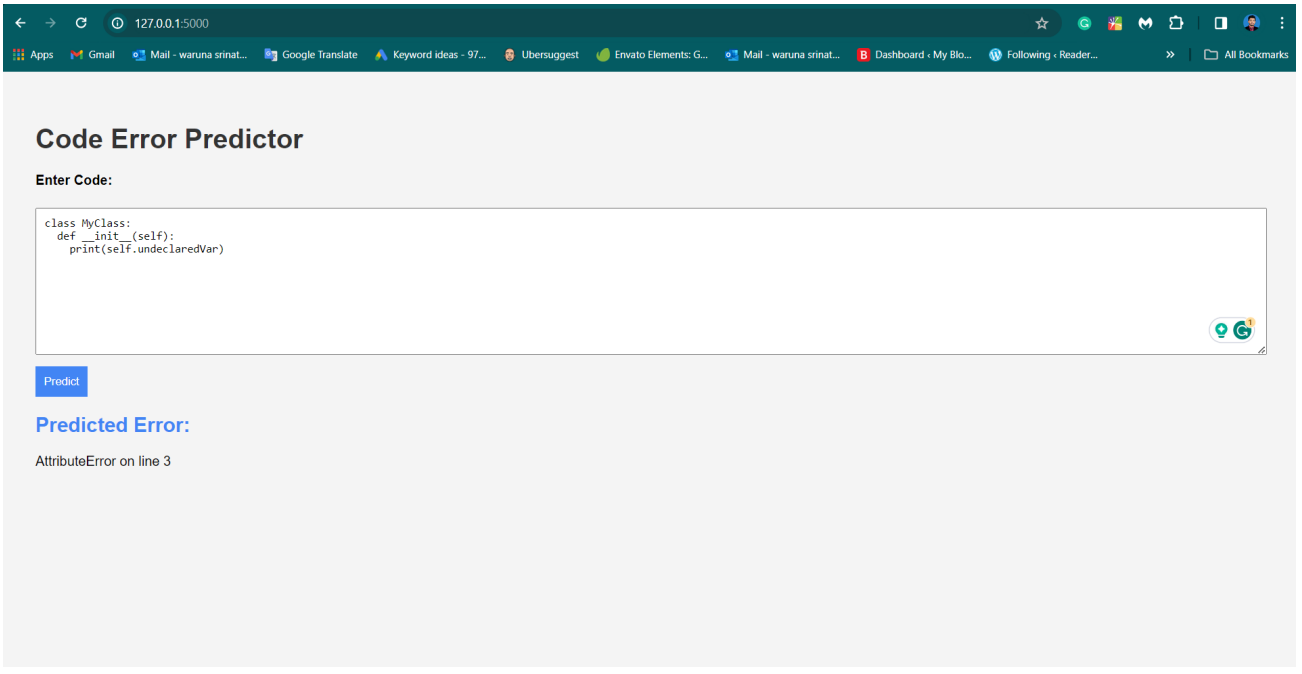
Out[10]:
```
LSTMModel(
  (embedding): Embedding(200, 50)
  (lstm): LSTM(50, 100, batch_first=True)
  (fc): Linear(in_features=100, out_features=27, bias=True)
)
```

6. Adding Visual Representation

Flask Web Application



The User Interface



Flask Application Local Port:- http://127.0.0.1:5000

```
(tensorflow) C:\Users\warun\Documents\Github\Defect-Prediction\Defects_Prediction_Flask>python app.py
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\_distributor_init.py:30: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\.libs\libopenblas.FB5AE2TYXYH2IJRDKGDGQ3XBKLKTF43H.gfortran-win_amd64.dll
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\.libs\libopenblas64__v0.3.23-gcc_10_3_0.dll
  warnings.warn("loaded more than 1 DLL from .libs:"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\_distributor_init.py:30: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\.libs\libopenblas.FB5AE2TYXYH2IJRDKGDGQ3XBKLKTF43H.gfortran-win_amd64.dll
C:\Users\warun\anaconda3\envs\tensorflow\lib\site-packages\numpy\.libs\libopenblas64__v0.3.23-gcc_10_3_0.dll
  warnings.warn("loaded more than 1 DLL from .libs:"
 * Debugger is active!
 * Debugger PIN: 841-570-895
127.0.0.1 - - [27/Dec/2023 07:32:13] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Dec/2023 07:32:14] "GET /static/style.css HTTP/1.1" 304 -
127.0.0.1 - - [27/Dec/2023 07:32:30] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [27/Dec/2023 07:32:30] "GET /static/style.css HTTP/1.1" 304 -
```

6. Conclusion

This defect prediction model leverages LSTM architecture for sequence modeling of code snippets. The approach involves data preprocessing, model construction, training, and evaluation. The documented code provides a comprehensive overview of the implementation details for building and assessing the defect prediction model.

Thank you!

Document By Waruna Srinath