assignment with only prompt and category

Documentation: Multi-Model Text Classification System

Overview

This script implements a multi-model text classification system using three different models: LSTM, GRU, and BERT. The models are trained to classify prompts into various categories. The system uses different vectorization techniques for feature extraction and includes functions for training, evaluating, and making predictions with the models.

Installation

Before running the code, ensure you have the required packages installed. You can install them using pip:

```
pip install numpy pandas scikit-learn nltk tensorflow trans
formers torch textblob gensim
```

Code Description

1. Imports and Setup

```
import numpy as np
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import CountVectorize
r, TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_seq
uences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, D
ense, Dropout
from transformers import BertTokenizer, BertForSequenceClas
sification
import torch
from torch.utils.data import DataLoader, Dataset
```

- **nltk**: Used for text preprocessing (stop words removal and stemming).
- **sklearn**: For text vectorization (BoW, TF-IDF) and model evaluation.
- **gensim**: For Word2Vec word embeddings.
- **tensorflow**: For LSTM and GRU model creation.
- **transformers**: For BERT model setup.
- **torch**: For BERT model training.

2. Data Preparation

Sample Data

The sample data includes text prompts and their corresponding categories. It is used to train and test the models.

```
data = {
    'prompt': [...],
    'category': [...]
}
```

- **prompt**: Textual prompts.
- **category**: Categories for each prompt.

DataFrame Creation

```
df = pd.DataFrame(data)
```

Creates a DataFrame from the sample data.

Text Preprocessing

```
stop_words = set(stopwords.words('english'))
stemmer = PorterStemmer()

def preprocess_text(text):
    tokens = simple_preprocess(text)
    tokens = [word for word in tokens if word not in stop_w
ords]
    tokens = [stemmer.stem(word) for word in tokens]
    return ' '.join(tokens)

df['cleaned_prompt'] = df['prompt'].apply(preprocess_text)
```

- **stopwords**: English stop words used to filter out common words.
- **PorterStemmer**: Used for stemming (reducing words to their root form).

Vectorization

BoW and TF-IDF

```
bow_vectorizer = CountVectorizer()
tfidf_vectorizer = TfidfVectorizer()
X_bow = bow_vectorizer.fit_transform(df['cleaned_prompt'])
X_tfidf = tfidf_vectorizer.fit_transform(df['cleaned_promp
t'])
```

- **CountVectorizer**: Converts text to a matrix of token counts.
- **TfidfVectorizer**: Converts text to a matrix of TF-IDF features.

Word2Vec Embeddings

```
sentences = [simple_preprocess(text) for text in df['promp
t']]
word2vec_model = Word2Vec(sentences, vector_size=100, windo
w=5, min_count=1, workers=4)

def get_word2vec_features(text):
    tokens = simple_preprocess(text)
    feature_vec = np.zeros(100)
    count = 0
    for word in tokens:
        if word in word2vec_model.wv:
            feature_vec += word2vec_model.wv[word]
            count += 1
    if count > 0:
        feature_vec /= count
    return feature_vec

X_word2vec = np.array([get_word2vec_features(text) for text
in df['prompt']])
```

- **Word2Vec**: Generates word embeddings for texts.
- **get_word2vec_features**: Converts text to a vector by averaging the word embeddings.

Label Encoding

```
label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['category'])
num_classes = len(label_encoder.classes_)
```

Encodes the category labels into numerical values.

3. Train-Test Split

```
X_train_bow, X_test_bow, y_train, y_test = train_test_split
(X_bow, df['label'], test_size=0.2, random_state=42)
X_train_tfidf, X_test_tfidf, _, _ = train_test_split(X_tfid
f, df['label'], test_size=0.2, random_state=42)
```

```
X_train_word2vec, X_test_word2vec, _, _ = train_test_split
(X_word2vec, df['label'], test_size=0.2, random_state=42)
```

Splits the data into training and testing sets.

4. Model Creation

RNN Models with LSTM and GRU

```
def create_rnn_model(vocab_size, embedding_dim, input_lengt
h):
    model = Sequential()
    model.add(Embedding(vocab_size, embedding_dim, input_le
ngth=input_length))
    model.add(LSTM(128, return_sequences=True))
    model.add(Dropout(0.2))
    model.add(LSTM(128))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorica
l_crossentropy', metrics=['accuracy'])
    return model

def create_gru_model(vocab_size, embedding_dim, input_lengt
h):
    model = Sequential()
    model.add(Embedding(vocab_size, embedding_dim, input_le
ngth=input_length))
    model.add(GRU(128, return_sequences=True))
    model.add(Dropout(0.2))
    model.add(GRU(128))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorica
l_crossentropy', metrics=['accuracy'])
    return model
```

- **create_rnn_model**: Creates an LSTM-based RNN model.
- **create_gru_model**: Creates a GRU-based RNN model.

5. Training and Evaluation

```
def train_and_evaluate(model, X_train, X_test, y_train, y_t
est):
    model.fit(X_train, y_train, epochs=150, batch_size=8, v
alidation_split=0.2, verbose=1)
    y_pred = model.predict(X_test)
    y_pred_classes = np.argmax(y_pred, axis=1)
    unique_labels = np.unique(y_test)
    target_names = [label_encoder.classes_[i] for i in uniq
ue_labels]
    print(classification_report(y_test, y_pred_classes, lab
els=unique_labels, target_names=target_names))
    return y_pred_classes

print("Training LSTM model...")
y_pred_lstm = train_and_evaluate(model_lstm, X_train_bow, X
_test_bow, y_train, y_test)

print("Training GRU model...")
y_pred_gru = train_and_evaluate(model_gru, X_train_bow, X_t
est_bow, y_train, y_test)
```

- **train_and_evaluate**: Trains the model and evaluates it using the test set.

6. BERT Model Setup and Training

```
bert_tokenizer = BertTokenizer.from_pretrained('bert-base-u
ncased')
bert_model = BertForSequenceClassification.from_pretrained
('bert-base-uncased', num_labels=num_classes)

class CustomDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
```

```
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt'
        )
        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].fl
atten(),
            'label': torch.tensor(label, dtype=torch.long)
        }

max_len = 50
train_dataset = CustomDataset(df['prompt'].tolist(), df['la
bel'].tolist(), bert_tokenizer, max_len)
train_loader = DataLoader(train_dataset, batch_size=8, shuf
fle=True)

def train_bert_model(model, train_loader, epochs):
    model.train()
    optimizer = torch.optim.AdamW(model.parameters(), lr=2e
-5)
    for epoch in range(epochs):
        for batch in train_loader:
            optimizer.zero_grad()
            input_ids = batch['input_ids']
            attention_mask = batch['attention_mask']
            labels = batch['label']
            outputs = model(input_ids, attention_mask=atten
```

```
tion_mask, labels=labels)
            loss = outputs.loss
            loss.backward()
            optimizer.step()
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss.ite
m()}")

print("Training BERT model...")
train_bert_model(bert_model, train_loader, epochs=60)

def evaluate_bert_model(model, texts, labels):
    model

.eval()
    tokenizer = BertTokenizer.from_pretrained('bert-base-un
cased')
    encodings = tokenizer(texts, truncation=True, padding=T
rue, max_length=50, return_tensors='pt')
    input_ids = encodings['input_ids']
    attention_mask = encodings['attention_mask']
    with torch.no_grad():
        outputs = model(input_ids, attention_mask=attention
_mask)
    predictions = torch.argmax(outputs.logits, dim=1)
    print(classification_report(labels, predictions))
    return predictions

print("Evaluating BERT model...")
y_pred_bert = evaluate_bert_model(bert_model, df['prompt'].
tolist(), df['label'].tolist())
```

- **CustomDataset**: Creates a custom dataset for BERT.
- **train_bert_model**: Trains the BERT model.
- **evaluate_bert_model**: Evaluates the BERT model.

Usage

1. **Prepare Data**: Update the `data` dictionary with your dataset.

2. **Run the Script**: Execute the script to train and evaluate the models.
3. **Review Results**: Check the classification reports for model performance.

Notes

- Ensure that your environment supports GPU acceleration for faster training, especially for the BERT model.
- Adjust hyperparameters (e.g., epochs, batch size) based on the size of your dataset and available computational resources.

Conclusion

This multi-model text classification system provides a comprehensive approach to text classification using various techniques, including traditional machine learning models (LSTM, GRU) and state-of-the-art models like BERT.