assignment final with category and sub-category

1. Imports and Setup

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from transformers import BertTokenizer, BertConfig, BertMod
el, BertPreTrainedModel
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
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

- `numpy` **and** `pandas`: Used for numerical operations and data manipulation.
- `torch` **and related modules**: PyTorch library components for neural network operations and data handling.
- `transformers`: Provides tools to use pre-trained BERT models and tokenizers.
- `tensorflow.keras`: Used for defining and training LSTM and GRU models.
- `sklearn`: For preprocessing, splitting data, and evaluating models.

2. Sample Data

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

- **Data Dictionary**: Contains sample prompts along with their categories and subcategories.

3. Convert to DataFrame

```
df = pd.DataFrame(data)
texts = df['prompt'].values
categories = df['category'].values
subcategories = df['subcategory'].values
```

- **Convert Data**: Converts the dictionary into a DataFrame and extracts values for further processing.

4. Tokenization for BERT

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncase
d')
X_bert = tokenizer(texts.tolist(), padding=True, truncation
=True, max_length=50, return_tensors='pt')
X_bert_ids = X_bert['input_ids']
X_bert_masks = X_bert['attention_mask']
```

- **Tokenizer**: Loads BERT tokenizer to convert text data into token IDs and attention masks suitable for BERT.

5. Encode Labels

```
label_encoder_cat = LabelEncoder()
y_cat = label_encoder_cat.fit_transform(categories)

label_encoder_subcat = LabelEncoder()
y_subcat = label_encoder_subcat.fit_transform(subcategorie
s)
```

- **Label Encoding**: Transforms categorical labels (e.g., categories and subcategories) into numerical format.

6. Split Data

```
X_train_ids, X_test_ids, y_train_cat, y_test_cat, y_train_s
ubcat, y_test_subcat = train_test_split(
    X_bert_ids.numpy(), y_cat, y_subcat, test_size=0.2, ran
dom_state=42
)
X_train_masks, X_test_masks, _, _ = train_test_split(
    X_bert_masks.numpy(), X_bert_masks.numpy(), test_size=
0.2, random_state=42
)
```

- **Data Splitting**: Splits the data into training and testing sets, including both token IDs and attention masks.

7. Custom Dataset Class

```
class CustomDataset(Dataset):
    def __init__(self, input_ids, attention_masks, category
_labels, subcategory_labels):
        self.input_ids = input_ids
        self.attention_masks = attention_masks
        self.category_labels = category_labels
        self.subcategory_labels = subcategory_labels

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

    def __getitem__(self, idx):
        item = {
            'input_ids': torch.tensor(self.input_ids[idx],
dtype=torch.long),
            'attention_mask': torch.tensor(self.attention_m
asks[idx], dtype=torch.long),
            'category_label': torch.tensor(self.category_la
bels[idx], dtype=torch.long),
            'subcategory_label': torch.tensor(self.subcateg
ory_labels[idx], dtype=torch.long)
```

```
        }
        return item
```

- **Custom Dataset**: Defines a PyTorch dataset class to handle inputs and labels for training.

8. Create DataLoader

```
train_dataset = CustomDataset(
    input_ids=X_train_ids.tolist(),
    attention_masks=X_train_masks.tolist(),
    category_labels=y_train_cat.tolist(),
    subcategory_labels=y_train_subcat.tolist()
)

train_loader = DataLoader(train_dataset, batch_size=8, shuf
fle=True)
```

- **DataLoader**: Creates a DataLoader instance to iterate through batches of training data.

9. Define LSTM Model

```
def create_lstm_model(input_length, vocab_size, embedding_d
im, num_labels):
    model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embeddin
g_dim, input_length=input_length),
        LSTM(128),
        Dense(num_labels, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorica
l_crossentropy', metrics=['accuracy'])
    return model
```

- **LSTM Model**: Defines a Sequential LSTM model with embedding, LSTM layer, and Dense output layer.

10. Prepare LSTM Data and Train

```
max_length = 50
X_bert_padded = pad_sequences(X_bert_ids.numpy(), maxlen=ma
x_length)
vocab_size = tokenizer.vocab_size
embedding_dim = 50

lstm_model = create_lstm_model(max_length, vocab_size, embe
dding_dim, len(label_encoder_cat.classes_))
history_lstm = lstm_model.fit(
    X_bert_padded, y_cat,
    epochs=100,
    batch_size=8,
    validation_split=0.1
)
```

- **Data Preparation**: Pads the tokenized BERT IDs to a uniform length for LSTM input.
- **Train LSTM**: Trains the LSTM model on the padded input.

11. Define GRU Model

```
def create_gru_model(input_length, vocab_size, embedding_di
m, num_labels):
    model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embeddin
g_dim, input_length=input_length),
        GRU(128),
        Dense(num_labels, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorica
l_crossentropy', metrics=['accuracy'])
    return model
```

- **GRU Model**: Defines a Sequential GRU model with similar structure to the LSTM model.

12. Train GRU Model

```
gru_model = create_gru_model(max_length, vocab_size, embedd
ing_dim, len(label_encoder_cat.classes_))
history_gru = gru_model.fit(
    X_bert_padded, y_cat,
    epochs=100,
    batch_size=8,
    validation_split=0.1
)
```

- **Train GRU**: Trains the GRU model on the padded input.

13. Define BERT Model for Classification

```
class BertForTextClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(config.hidden_size, len
(label_encoder_cat.classes_))
        self.subclassifier = nn.Linear(config.hidden_size,
len(label_encoder_subcat.classes_))

    def forward(self, input_ids, attention_mask=None, label
s=None):
        outputs = self.bert(input_ids, attention_mask=atten
tion_mask)
        pooled_output = outputs[1]
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        sub_logits = self.subclassifier(pooled_output)
        return logits, sub_logits
```

- **BERT Model**: Defines a BERT-based model with two separate classification heads for categories and subcategories.

14. Initialize and Train BERT Model

```
config = BertConfig.from_pretrained('bert-base-uncased')
bert_model = BertForTextClassification(config)

optimizer = torch.optim.AdamW(bert_model.parameters(), lr=1
e-5)
loss_fn = nn.CrossEntropyLoss()

bert_model.train()
for epoch in range(50):
    for batch in train_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids']
        attention_mask = batch['attention_mask']
        category_labels = batch['category_label']
        subcategory_labels = batch['subcategory_label']

        logits, sub_logits = bert_model(input_ids, attentio
n_mask=attention_mask)
        loss_cat = loss_fn(logits, category_labels)
        loss_subcat = loss_fn(sub_logits, subcategory_label
s)
        loss = loss_cat + loss_subcat
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch + 1}: Loss = {loss.item()}")
```

- **BERT Training**: Initializes the BERT model, sets up the optimizer and loss function, and trains the model over 50 epochs.

15. Predict Using All Models

```
def predict_separate_models(texts):
    bert_inputs = tokenizer(texts, padding=True, truncation
=True, max_length=50, return_tensors='pt')
    input_ids = bert_inputs['input_ids'].clone().detach()
    attention_mask = bert_inputs['attention_mask'].clone().
detach()
```

```
    X_bert_padded_test = pad_sequences(tokenizer(texts, pad
ding=True, truncation=True, max_length=max_length, return_t
ensors='pt')['input_ids'].numpy(), maxlen=max_length)
    lstm_preds = lstm_model.predict(X_bert_padded_test)
    gru_preds = gru_model.predict(X_bert_padded_test)

    with torch.no_grad

():
        logits, sub_logits = bert_model(input_ids, attentio
n_mask=attention_mask)
        bert_preds_cat = torch.argmax(logits, dim=1).numpy
()
        bert_preds_subcat = torch.argmax(sub_logits, dim=
1).numpy()

    return {
        'LSTM': np.argmax(lstm_preds, axis=1),
        'GRU': np.argmax(gru_preds, axis=1),
        'BERT': bert_preds_cat
    }

texts_to_predict = ["Example prompt text"]
predictions = predict_separate_models(texts_to_predict)
print(predictions)
```

- **Prediction Function**: Defines a function to get predictions from the LSTM, GRU, and BERT models for a given list of texts.

16. Combine Predictions Using Majority Voting

```
def combine_predictions(predictions):
    combined_preds = []
    for i in range(len(predictions['LSTM'])):
        preds = [predictions['LSTM'][i], predictions['GRU']
[i], predictions['BERT'][i]]
        combined_pred = np.bincount(preds).argmax()
        combined_preds.append(combined_pred)
    return combined_preds

combined_preds = combine_predictions(predictions)
print(combined_preds)
```

- **Combining Predictions**: Uses majority voting to combine predictions from the three models for each text input.

This code integrates various NLP models (BERT, LSTM, and GRU) for text classification and combines their predictions for improved accuracy. It involves data preparation, model training, and evaluation steps using different machine learning techniques.