Fraud_Email_MNB.ipynb — Detailed Overview

1. Notebook Overview

This notebook is aimed at detecting fraudulent emails using BERT, a transformer model originally designed for natural language processing. In a fraud email detection pipeline, the notebook will typically:

- **Import and initialize libraries**: Load PyTorch, Hugging Face's Transformers, Pandas, and other utilities.
- **Load and preprocess the dataset**: Read email data (often from a CSV or JSON file), clean the text, and possibly generate labels indicating fraud (fraudulent or not).
- **Tokenize the text with BERT's tokenizer**: Convert raw text into token IDs and attention masks.
- **Define a classifier model**: Create a PyTorch model that uses a pre-trained BERT as the encoder and adds one or more additional layers (such as a fully connected layer) to output a fraud probability.
- **Train the model**: Run training over a number of epochs while evaluating validation metrics.
- **Evaluate and visualize the results**: Test on a hold-out set, report accuracy, F1-score, etc., and optionally show examples of predictions.
- **Save the trained model**: Persist the weights for future inference or deployment.

2. Detailed Walkthrough (Cell by Cell)

Below, I explain in detail each segment that such a notebook may contain.

Cell 1: Import Libraries and Set Up Environment

python
Copy code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from transformers import BertTokenizer, BertModel,
BertForSequenceClassification
import re
```

- **Line 1–3:**
  Import the PyTorch modules needed for tensor computation, neural network building blocks, and optimization routines.
- **Line 4:**
  Imports the PyTorch dataset and dataloader classes. These help you load and iterate over your training data.
- **Line 5:**
  Pandas is used for data manipulation, typically to read CSV files with emails and labels.
- **Line 6:**
  Numpy is imported for numerical operations, if needed (e.g., handling arrays).
- **Line 7:**
  Matplotlib is often used to visualize loss curves or other statistics.
- **Line 8:**
  The Hugging Face Transformers package provides the BERT model and tokenizer. A common pattern is to use either the plain BERT model or, even more likely, `BertForSequenceClassification` which automatically attaches a classification head to the base model.
- **Line 9:**
  The `re` module is used for regular expressions—often employed to clean text data (e.g., removing punctuation or extra spaces).

Cell 2: Data Loading and Preprocessing

python
Copy code

```
# Load the dataset
df = pd.read_csv('fraud_emails.csv')  # Assumes a CSV file with email
texts and a label column

# Preview the data
print(df.head())

# Function to clean email text
```

```
def clean_text(text):
    text = text.lower()  # Convert to lower-case
    text = re.sub(r'\s+', ' ', text)  # Remove extra spaces
    text = re.sub(r'\W', ' ', text)  # Remove non-word characters
    return text.strip()

# Apply cleaning on the email text column
df['cleaned_text'] = df['email_text'].apply(clean_text)
```

- **Line 1:**
  Reads a CSV file (named 'fraud_emails.csv') into a DataFrame. This file is assumed to contain a column of email texts and one or more columns for the label (e.g., a fraud indicator).
- **Line 4:**
  Prints the first few rows of the DataFrame to confirm data loading.
- **Lines 7–11:**
  Define a `clean_text` function which applies standard text-cleaning steps:

Convert the text to lower case (this helps reduce vocabulary size).

Use regular expressions to remove extra whitespace and non-word characters.

Strip leading and trailing spaces.

- **Line 14:**
  Applies the cleaning function to all email texts in the DataFrame, storing the cleaned version for tokenization later.

Cell 3: Tokenization and Dataset Preparation

python
Copy code

```
# Initialize a BERT tokenizer from the pre-trained model (e.g., 'bert-
base-uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Define a custom dataset
class FraudEmailDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
```

```
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        # Use the tokenizer to convert text to inputs required for
BERT
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            truncation=True,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),  # flatten
removes extra batch dimension
            'attention_mask': encoding['attention_mask'].flatten(),
            'label': torch.tensor(label, dtype=torch.long)
        }

# Prepare training and validation splits
from sklearn.model_selection import train_test_split
train_texts, val_texts, train_labels, val_labels = train_test_split(
    df['cleaned_text'].tolist(), df['label'].tolist(), test_size=0.2,
random_state=42
)

train_dataset = FraudEmailDataset(train_texts, train_labels,
tokenizer)
```

```
val_dataset = FraudEmailDataset(val_texts, val_labels, tokenizer)

# Create DataLoader objects
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
```

- **Line 1:**
  Loads a pre-trained BERT tokenizer (here using the 'bert-base-uncased' model) that splits text into tokens and maps them to their corresponding IDs.
- **Lines 3–21:**
  Defines a custom subclass of `Dataset` called `FraudEmailDataset`.

The constructor takes a list of texts, labels, a tokenizer, and a maximum sequence length.

The `__len__` method returns the number of examples.

The `__getitem__` method retrieves an item at a given index, tokenizes the text (using `encode_plus` which adds special tokens, pads, truncates, and creates an attention mask), and returns a dictionary with keys for input IDs, attention mask, and label. This dictionary is the format expected for feeding data into a BERT model.

- **Line 23:**
  Uses Scikit-learn's `train_test_split` function to divide the dataset into training and validation sets.
- **Line 28–29:**
  Instantiates training and validation datasets with the cleaned texts and corresponding labels.
- **Line 32–33:**
  Creates DataLoader objects for training and validation—batching the data and shuffling training examples for better learning.

Cell 4: Define and Initialize the BERT Classifier

python
Copy code

```
# Option 1: Using Hugging Face's BertForSequenceClassification
(simpler and pre-built)
model = BertForSequenceClassification.from_pretrained('bert-base-
uncased', num_labels=2)
```

```
model = model.to(torch.device("cuda" if torch.cuda.is_available() else
"cpu"))
```

- **Line 1–2:**
  Here we use the built-in model `BertForSequenceClassification`, which automatically adds a classification head on top of BERT.
  - `num_labels=2` indicates a binary classification (fraudulent or not).
- **Line 3:**
  Moves the model to the appropriate device (GPU if available, else CPU).

Alternatively, you might see a custom model definition that uses `BertModel` as the encoder, followed by additional linear (fully connected) layers to map the [CLS] token's embedding to a prediction. In that case, the notebook would include a class definition extending `nn.Module` with a custom forward method.

Cell 5: Define Optimizer, Scheduler, and Loss Function

python
Copy code

```
optimizer = optim.AdamW(model.parameters(), lr=2e-5,
correct_bias=False)
from transformers import get_linear_schedule_with_warmup
epochs = 3
total_steps = len(train_loader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=0, num_training_steps=total_steps)
loss_fn = nn.CrossEntropyLoss()
```

- **Line 1:**
  Defines an optimizer (AdamW, which is well-suited to transformer training) with a low learning rate (2e-5 is common for fine-tuning BERT).
- **Line 2:**
  Imports a learning rate scheduler from the Transformers library.
- **Line 3–5:**
  Calculates the total number of training steps and initializes a linear scheduler that will linearly reduce the learning rate during training.
- **Line 6:**
  Creates a loss function (Cross Entropy Loss is standard for classification tasks).

Cell 6: Training Loop

python
Copy code

```
for epoch in range(epochs):
    model.train()  # set the model to training mode
    total_train_loss = 0

    for batch in train_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(model.device)
        attention_mask = batch['attention_mask'].to(model.device)
        labels = batch['label'].to(model.device)

        outputs = model(input_ids=input_ids,
attention_mask=attention_mask, labels=labels)
        loss = outputs.loss  # When passing labels, Hugging Face
models return the loss
        total_train_loss += loss.item()

        loss.backward()
        optimizer.step()
        scheduler.step()

    avg_train_loss = total_train_loss / len(train_loader)
    print(f"Epoch {epoch+1}: Training Loss = {avg_train_loss}")

    # Validation Loop (set model to eval mode to disable dropout,
etc.)
    model.eval()
    total_val_loss = 0
    correct_predictions = 0

    with torch.no_grad():
        for batch in val_loader:
            input_ids = batch['input_ids'].to(model.device)
            attention_mask = batch['attention_mask'].to(model.device)
            labels = batch['label'].to(model.device)
```

```
            outputs = model(input_ids=input_ids,
attention_mask=attention_mask, labels=labels)
            loss = outputs.loss
            total_val_loss += loss.item()
            logits = outputs.logits
            preds = torch.argmax(logits, dim=1)
            correct_predictions += torch.sum(preds == labels)

    avg_val_loss = total_val_loss / len(val_loader)
    val_accuracy = correct_predictions.double() / len(val_dataset)
    print(f"Epoch {epoch+1}: Validation Loss = {avg_val_loss},
Accuracy = {val_accuracy}")
```

- **Line 1:**
  Loops through each training epoch.
- **Line 2:**
  Puts the model in training mode (activates dropout, etc.).
- **Line 4:**
  Resets the cumulative loss for the epoch.
- **Inside the Training Loop:**

Line 6: Zeroes out gradients from the previous step.

Line 7–9: Extracts input IDs, attention masks, and labels from the batch and moves them to the device.

Line 11: Feeds these inputs to the model. When you pass labels, the model automatically computes the loss.

Line 12: Retrieves the loss from the model's output.

Line 14–17: Performs backpropagation, updates the optimizer's parameters, and steps the scheduler.

- **After Training Loop:**
  Computes and prints the average training loss.
- **Validation Block:**
  With gradients turned off (via `torch.no_grad()`), the notebook loops through the validation data, computes the loss for each batch, and collects predictions to calculate accuracy.

- **Final Lines:**
  Print the average loss and accuracy on the validation set.

Cell 7: Testing/Inference and Saving the Model

python
Copy code

```
# Save the model weights
model.save_pretrained('fraud_email_bert_model')

# Example inference
def predict_email(email_text):
    model.eval()
    text_clean = clean_text(email_text)
    encoding = tokenizer.encode_plus(
        text_clean,
        add_special_tokens=True,
        max_length=128,
        truncation=True,
        padding='max_length',
        return_attention_mask=True,
        return_tensors='pt'
    )
    input_ids = encoding['input_ids'].to(model.device)
    attention_mask = encoding['attention_mask'].to(model.device)
    outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
    logits = outputs.logits
    probabilities = torch.softmax(logits, dim=1)
    return probabilities.detach().cpu().numpy()

# Run a sample prediction
sample_email = "Congratulations! You have won a lottery. Please send
your bank details."
print(predict_email(sample_email))
```

- **Lines 1–2:**
  The model's weights are saved using the `save_pretrained` method (this creates files that can later be reloaded with `from_pretrained`).
- **Lines 4–20:**
  Defines a helper function `predict_email` to run inference on a single email. It:

Sets the model to evaluation mode.

Cleans the provided email text.

Uses the same tokenizer to prepare input tensors.

Runs the model to get output logits.

Applies softmax to convert logits to probabilities.

Returns these probabilities as a NumPy array.

- **Line 23–25:**
  Provides an example email text and prints the resulting prediction probabilities.

Final Notes

- **Modularity:**
  The notebook is organized so each cell corresponds to a distinct step in the pipeline. This improves readability and debugging.
- **Transformer-specific Methods:**
  The use of Hugging Face's `BertForSequenceClassification` simplifies many tasks such as loss computation and output formatting.
- **Line-by-Line Details:**
  Each line in this explanation plays a role in ensuring that the raw email text is properly processed, tokenized, fed into BERT, and that the classifier is fine-tuned to detect fraudulent emails.
- **Customization and Further Improvements:**
  In practice, you might add more preprocessing (e.g., handling stop words or domain-specific terms), experiment with hyperparameters (learning rate, batch size, epochs), or incorporate more detailed logging and visualization for training progress.