The lead TA for this assignment is Shuyu Gan (gan00067@umn.edu). Please communicate with the lead TA via Slack or office hours. All questions MUST be discussed in the homework channel (i.e., #HW3). Questions through emails, Direct Messages, and other channels will not be answered.

This assignment explores the authorship attribution problem, i.e. determine who wrote a given text. You will write both generative (Transformer or RNN) and discriminative (sequence classifier) solutions and compare the results.

Academic Honesty Policy. Make sure to (a) cite any tools or papers you reference/use, and (b) credit anyone youve discussed the assignment with. It is considered academic dishonesty if you reference any tool/paper/person without proper attribution.

Setup

I have created four source files containing excerpts from multiple works by different authors: Jane Austen, Charles Dickens, Leo Tolstoy, and Oscar Wilde. You can find these files in this link. Download the files from the folder. You will need to decide which encoding you want to use. I have posted the UTF-8 encoded text and the ASCII "transliteration" of the UTF-8 encodings. Some peculiar things are going on; in particular, look at the first few paragraphs of Tolstoy's text to see some examples of the differences in encodings.

```
ASCII   possible. Kut'uzov himself with all his transport took the ..
UTF8    possible. Kutzov himself with all his transport took the ..
```

Note that UTF-8 encoding is **recommended in most cases** as most text editors, websites, text data on the Internet, and many programming languages use UTF-8 by default.

Your task is to implement a text classification system that can attribute a new text to its original author.

Your Task

You will implement a text classification system in a **Jupyter notebook** (Google Colab recommended). Your notebook should contain two separate approaches:

1. **Generative model classifier** (language-model based)
2. **Discriminative model classifier** (sequence classification based)

Both approaches must be implemented and evaluated. The training data consists of several text files, each containing works from a single author. For example:

```
austen.txt
dickens.txt
tolstoy.txt
wilde.txt
```

Each file corresponds to one author and provides the data you will use to train(or test) your models.

Required Behavior

1. For each author, split the data into 90% training and 10% development.

2. Train the model on the training portion.
3. Evaluate on the development portion.
4. Save predictions to predictions files and record the test accuracy.

Note: For the generative model classifier, you should train a separate classifier for each author. For the discriminative model classifier, you only need to train a single classifier that distinguishes between all authors.

Suggested Notebook Structure

1. **Setup and Imports:** Import necessary libraries (e.g., PyTorch or other libraries you need).
2. **Data Loading:** Load the author files and the test file if provided.
3. **Preprocessing:** Tokenize and clean the text; split into train/dev if no test file is provided.
4. **Generative Model Classifier:** Implement and train generative (language-model based) classifiers; evaluate and record predictions.
5. **Discriminative Model Classifier:** Implement and train a discriminative (sequence classification) model; evaluate and record predictions.
6. **Output:** Save predictions to files (e.g., `predictions_generative.txt` and `predictions_discriminative.txt`) and report evaluation metrics.

Note: Since you will write a separate report for this homework, you do not need to include extensive explanations or detailed comments inside the notebook itself. However, your report should describe your implementation process as clearly and thoroughly as possible. Please also make sure that all code cells in your notebook have been executed and retain their execution traces, including both training and evaluation outputs.

In addition, you must include **test accuracy results** in both the notebook and the report. For example, results on the development set may be summarized as follows:

```
Results on dev set:
austen      61.4% correct
dickens     73.3% correct
tolstoy     57.7% correct
wilde       67.3% correct
```

Your *predictions files* should look list this, without comments in the parentheses:

```
austen (the first line of text is predicted as austen)
austen
wilde
austen
tolstoy
...
```

Step 1: Generative Authorship Classifier

Build your generative classifier as follows:

- Build a neural language model (LM) for each author (Austen, Dickens, Tolstoy, Wilde). Use either a small **RNN-based LM** or a **lightweight Transformer decoder**. Do not use HuggingFace models or tokenizers.

- Tokenization. Train a shared tokenizer with on the concatenation of all authors texts (so all LMs share the same vocabulary). In the example code we use **SentencePiece (BPE)**, You may choose a different tokenizer if you prefer.
- Training objective. Train each author-specific LM with teacher forcing to minimize token-level loss, and use a 10% dev split for early stopping (when no external test file is provided).
- (Task 1) Classification by perplexity. For a test sentence $x_{1:T}$, compute its average token-level negative log-likelihood (NLL) under each author LM and convert to **perplexity (PPL)**:

$$\mathrm{PPL}(x_{1:T}) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log p(x_t \mid x_{<t})\right).$$

  Predict the author with the lowest PPL. (Using the average over tokens ensures fair comparison across different lengths.)
- (Task 2) Generation & qualitative analysis. Design five different prompts (sentence beginnings). For each prompt, generate one sentence continuation from each author LM. Briefly compare stylistic differences among authors under the same prompt.

Below is some basic code to preprocess the training data with a shared SentencePiece BPE tokenizer, train neural language models (RNN or Transformer) for each author, and calculate perplexity. Please refer to the accompanying tutorial and documentation for the detailed process.

1. Preparing Data (SentencePiece BPE, sample)

```
# Train a shared BPE tokenizer with SentencePiece
import sentencepiece as spm

spm.SentencePieceTrainer.Train(
    "--input=all_authors.txt --model_prefix=authors_bpe "
    "--vocab_size=4000 --model_type=bpe --unk_id=0 "
    "--pad_id=1 --bos_id=2 --eos_id=3"
)

sp = spm.SentencePieceProcessor(model_file="authors_bpe.model")
```

2. Dataset Construction

```
from torch.utils.data import Dataset, DataLoader
import torch

def encode_line(text):
    # TODO: implement
    pass

class LineDataset(Dataset):
    def __init__(self, path, max_len=256):
        # TODO: implement
        self.samples = []

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

    def __getitem__(self, i):
        # TODO: implement
        pass

def collate(batch):
    # TODO: implement
    pass

author_dataset = DataLoader(
    LineDataset("austen.txt"),
    batch_size=32, shuffle=True, collate_fn=collate
)
```

3. Training RNN LM (sample)

```python
import torch, torch.nn as nn

class RNNLM(nn.Module):
    def __init__(self, vocab_size, emb_dim=256, hidden_dim=512, pad_id=1):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, emb_dim, padding_idx=pad_id)
        # Vanilla RNN instead of LSTM
        self.rnn = nn.RNN(
            input_size=emb_dim,
            hidden_size=hidden_dim,
            num_layers=1,
            nonlinearity='tanh',
            batch_first=True
        )
        self.fc = nn.Linear(hidden_dim, vocab_size)
    def forward(self, x):
        # x: [B, T]
        emb = self.emb(x)          # [B, T, D]
        out, _ = self.rnn(emb)     # [B, T, H]
        return self.fc(out)        # [B, T, V]

# TODO: Training and Evaluation
# More details in Transformer Code Sample
```

4. Training Transformer LM (sample)

```python
class TinyTransformerLM(nn.Module):
    class TinyTransformerLM(nn.Module):
    def __init__(self, vocab_size, d_model=256, n_head=4, n_layer=4, d_ff=1024, max_len=512, dropout=0.1):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model, padding_idx=pad_id)
        self.pos_emb = nn.Embedding(max_len, d_model)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=n_head, dim_feedforward=d_ff,
        ↪  dropout=dropout, batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=n_layer)
        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        B, T = x.size()
        pos = torch.arange(T, device=x.device).unsqueeze(0).expand(B, T)
        h = self.tok_emb(x) + self.pos_emb(pos)

        # Causal mask: prevent attending to future positions
        # shape [T, T], True means "mask out"
        causal_mask = torch.triu(torch.ones(T, T, device=x.device, dtype=torch.bool), diagonal=1)
        # Padding mask: True at pad positions
        pad_mask = (x == pad_id)  # shape [B, T]

        h = self.encoder(h, mask=causal_mask, src_key_padding_mask=pad_mask)
        return self.fc(h)

# TODO: initialize model
# model = ...
# TODO: define loss function
# loss_fn = ...
# TODO: define optimizer
# opt = ...
for epoch in range(EPOCHS):
    model.train()
    for x, y in DataLoader(author_dataset):
        # TODO: forward pass
        # logits = ...
        # TODO: compute loss
        # loss = ...
        # TODO: backward + update
        pass
    model.eval()
    #TODO: Evaluation
        pass
```

4. Inference & Generation

```
# Compute perplexity for a given sentence
def compute_ppl(model, sentence):
    # TODO: implement
    # Hint: encode with <bos> ... <eos>, shift for x/y,
    # run model, compute average token NLL, return exp(loss)
    pass


# Generate a continuation given a prompt
def generate(model, prompt, max_len=50):
    # TODO: implement
    # Hint: iterative decoding:
    #  - encode prompt
    #  - loop: feed current ids, take last logits
    #  - apply softmax + sampling, append new token
    #  - stop at <eos> or max_len
    pass

print(generate(model, "It was a cold winter evening"))
```

Note: The model design and the hyperparameter choices in the example code (e.g., embedding size, hidden size, number of layers) are for illustration only. You are encouraged to experiment with your own settings.

Step 2: Discriminative Authorship Classifier

Build your discriminative classifier as follows:

- Use Huggingface to create a sequence classification model. Your model should have k labels, where k is the number of authors.
- Process your data such that each text is labeled with the appropriate author; create your train and test dataloaders. For examples of creating dataloaders, you can refer to your HW1 solution and/or the Pytorch Tutorial from Week 2.
- Train your classifier. You may use the Huggingface Trainer class as shown in the Classification Tutorial.

Recall how to instantiate a classifier using Huggingface Transformers:

```
from transformers import AutoModelForSequenceClassification
# This automodel class gives us the model with pretrained weights
# and a sequence classification head. When you instantiate it,
# you'll use the following arguments:
#
# a string with the model name as found on the Huggingface hub,
# e.g. 'distilbert-base-uncased'
#
# num_labels: an int that corresponds to the number of classes
# in your classification problem
#
# id2label: a dictionary that maps from label id  (an int
# in range(0, num_labels)) to the human-readable label name (a string)
#
# label2id: the inverse mapping from id2label
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=num_labels,
    id2label=id2label,
    label2id=label2id
)
```

Deliverables

Please upload your code and report to Canvas by Oct 5, 11:59pm.

Code(Jupyter Notebook) and test result: You should provide a **zipped file** containing your training/inference scripts and prediction files.

Report: Maximum **five pages PDF**. The page limit of homework doesn't include references and an appendix with additional information. For report, you must use this LaTex template (link). Please present your results using tables or plots whenever appropriate. Please try to avoid copying and pasting directly from sources, and ensure your results are formatted nicely.

Your report needs to include the following content:

- Generative Classifier:
  - What encoding type your program uses (e.g., SentencePiece BPE, unigram LM, etc.)
  - What model architecture you implemented (RNN-based LM or Transformer-based LM), and what hyperparameters you chose.
  - How you trained your model (loss function, teacher forcing, handling of `<pad>` tokens, early stopping).
  - Any other tweaks you made to improve results (e.g., dropout, weight decay, gradient clipping).
  - Design five prompts. For each prompt, generate one continuation from each author LM (so 5 prompts $\times$ 4 authors = 20 generations total). Report the perplexity score of each generated sample under the corresponding LM, and briefly compare stylistic differences.
  - Can you extract any representative stylistic features for each author from your trained LMs (e.g., frequent phrases, characteristic word choices)?
- Comparison Between Generative and Discriminative Classifiers:
  - The results (i.e. accuracy for each author) you get with the given data with an automatically-extracted development set (i.e. the output from running it without the -test flag). Show this for **both generative and discriminative models**.
  - What are some failure cases (i.e. errors made during testing) for each model?
  - Based on your results, discuss some advantages and disadvantages of generative and discriminative approaches to classification.

Formatting convention: All your files submitted should follow this naming convention: **CSCI5541-F25-HW3-{First Name}-{Last Name}.{zip,pdf}**.

Rubric (40 points + 10 bonus points)

- Code (25 points)
  - Code is Error-Free (2 points)
    - Code for **generative** model looks good, i.e., program runs as directed without error and outputs requested results (+1)
    - Code for **discriminative** model looks good, i.e., program runs as directed without error and outputs requested results (+1)
  - Data Processing (11 points)
    - Data is properly processed for generative model (i.e., tokenization with tokenizer, dataset split into input/target, and correct batching/padding) (+3)
    - Data is properly processed for discriminative model (i.e., dataloaders are created with correct labels) (+3)
    - Training/Dev set are correctly created (+5)
  - Modeling (12 points)

    - ∗ Generative classifier (RNN or Transformer LM) is trained correctly and achieves accuracy above random chance (i.e., 25%) using perplexity-based classification (+5)
    - ∗ Model training loop includes standard practices (teacher forcing, loss masking for pad tokens) (+4)
    - ∗ Correct implementation of Huggingface sequence classifier (+3)
- Report (15 points)
  - Includes appropriate references (+1)
  - Description of the encoding type, model choice, and training setup is clearly explained in report (+2)
  - Accuracy numbers for **both** models are reported (+2)
  - Report includes generated samples and perplexity scores from generative models as described in assignment (+5)
  - Report includes failure cases from each model (+1)
  - Report includes discussion comparing two models (+2)
  - The comparison of the two models goes beyond just comparing accuracy numbers and failure cases (e.g., it also mentions success cases, stylistic differences, theoretical trade-offs, etc) (+2)
- **Bonus Points (Max: +10 points)**
  - Implement an author-conditioned LM (a single shared model with an additional author embedding), and compare its perplexity/classification accuracy to four separate LMs (+3)
  - Implement top-k or nucleus (top-p) sampling, and compare them against greedy decoding and multinomial sampling (+2)
  - Show top-5 features for each author in the discriminative classifier (+2)
  - Show the best result (lowest perplexity or highest accuracy in the class) for each model (+1)
  - Show results and analyses on various hyperparameters/model variants (e.g., different hidden sizes, number of layers, vocab sizes, smoothing/regularization strategies) (+1∼2)