The lead TA for this assignment is Risako Owan (owan0002@umn.edu). Please communicate with the lead TA via Slack or office hours. All questions MUST be discussed in the homework channel (i.e., #HW2). 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 (n-gram language model) and discriminative (sequence classifier) solutions and compare the results. Please carefully read Section 3.5 of Jurafsky and Martin and NLTK's LM package before you start.

**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 write a program that will build a language model for each author and map a new text to the author who originally wrote it.

## Your Task

Write a program `classifier.py` that can be run with the following command-line setups:

```
python3 classifier.py authorlist -approach [generative|discriminative]

python3 classifier.py authorlist -approach [generative|discriminative] -test testfile
```

Where *authorlist* is a file containing a list of file names like the following:

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

That gives the file names of the training sets that you will use.

The *approach* flag's value determines whether the classification model will be a generative (n-gram) language model, or a discriminative (sequence classification) model. More details in Steps 1 and 2.

When the program is run without the *-test* flag, your program should automatically extract a development set (10%) from the author data, train the model on the remaining training data, and run the task on the development data and output the results to a new file *predictions.txt*.

When the program is run with the *-test* flag, your program should use the entirety of data in each author file to train a language model, then output classification results for each line in the given file *testfile* to the file *predictions.txt*. You may assume that each line of *testfile* is an entire sentence.

### Sample Runs

```
$ python3 classifier.py authorlist -approach generative
splitting into training and development...
training LMs... (this may take a while)
Results on dev set:
austen      61.4% correct
dickens     73.3% correct
tolstoy     57.7% correct
wilde       67.3% correct
```

```
$ python3 classifier.py authorlist -approach generative -test test_sents.txt
Training LMs... (this may take a while)
Training complete. Generating predictions to predictions.txt
```

Your *predictions.txt* 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:

- For each data set, create an n-gram language model using NLTK's LM package as a baseline model.
- Improve your n-gram language models (i.e., reduce perplexity) by using different types of smoothing, backoff, and interpolation. Carefully read Section 3.5 of Jurafsky and Martin and use default functions implemented in NLTK: smoothing, backoff, and interpolation. Feel free to try a few different combinations of models (e.g., uni+bigram model with smoothing) and see which works the best for this task.
- For each of your language models, compute the perplexity of the test item. Whichever language model gives the lowest perplexity should be how you classify the test item.
- For each of your language models, generate five samples of each author given the same prompt you specify and compare them.
- (optional, bonus point +5) Implement ngram language models without using NLTK. You can implement it using Numpy or PyTorch from scratch. For instance, this tutorial and code by Andrej

Karpathy describes how to build GPT language model using PyTorch, but you will only see the video frames from 7:52 to 42:12 for bigram language model implementation.

Below is some basic code to preprocess the training data, train N-gram language models, and calculate perplexity, taken from the NLTK's LM package tutorial. Carefully read the detailed process in the tutorial.

```
% Prepare Data
% Note: You may use other libraries for tokenization and sentence segmentation
>>> from nltk.lm.preprocessing import pad_both_ends, flatten,padded_everygram_pipeline
>>> list(pad_both_ends(text[0], n=2))
['<s>', 'a', 'b', 'c', '</s>']
>>> list(bigrams(pad_both_ends(text[0], n=2)))
[('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]
>>> list(flatten(pad_both_ends(sent, n=2) for sent in text))
['<s>', 'a', 'b', 'c', '</s>', '<s>', 'a', 'c', 'd', 'c', 'e', 'f', '</s>']
>>> train, vocab = padded_everygram_pipeline(2, text)

% Training
>>> from nltk.lm import MLE
>>> lm = MLE(2)
>>> lm.fit(train, vocab)
>>> print(lm.vocab)
<Vocabulary with cutoff=1 unk_label='<UNK>' and 9 items>
>>> len(lm.vocab)
9
>>> lm.vocab.lookup(text[0])
('a', 'b', 'c')
>>> lm.vocab.lookup(["aliens", "from", "Mars"])
('<UNK>', '<UNK>', '<UNK>')

% Inference
>>> test = [('a', 'b'), ('c', 'd')]
>>> lm.perplexity(test)
2.449489742783178
```

### 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:

```python
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 Mar 4, 11:59pm.

**Code:** You should provide a **zipped file** containing your training/inference scripts.

**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 runs on
  - What information is in your model (bigrams, trigrams, etc)
  - What method of smoothing your model uses
  - How do you deal with out-of-vocabulary words during run time when you build a language model?
  - Any other tweaks you made to improve results (backoff, etc.)
  - Write five prompts. For each of your language models for the four authors, generate one sample of each author. For all twenty samples (five prompts x four authors), report the perplexity score of each language model.
  - Can you extract the most representative top 5 features (e.g., bi-grams) for each author from your trained LMs?
- 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-F24-HW3-{Group Name}.{zip,pdf}**.

## Rubric (50 points + 10 bonus points)

- Code (35 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. preprocessing steps outlined in example code are followed) (+2)
    - Data is properly processed for discriminative model (i.e. dataloaders are created with correct labels) (+2)
    - Bigrams / trigrams are correctly created (+2)
    - Training/Dev set are correctly created and test flag is implemented as specified (+5)
  - Modeling (12 points)
  - Accuracy for generative model is on avg above random chance (i.e. 25%) (+5)
  - Correct implementation of at least three different smoothing algorithms (smoothing, interpolation, backoff, etc) (+5)
  - Correct implementation of Huggingface sequence classifier (+2)
- Report (15 points)
  - Includes appropriate references (+1)
  - Description of the encoding type, method of smoothing, interpolation, and other tweaks is clearly explained in report (+1)
  - Accuracy numbers for **both** models are reported (+2)
  - Report includes 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, theoretical differences, etc) (+3)
- **Bonus Points (Max: +10 points)**
  - Show top-5 features for each author. (+2)
  - Implement n-gram language models without using NLTK library (+3)
  - Show the best result (i.e., the lowest perplexity in the class, the highest accuracy) for each model (+1 to +2 points)
  - Show results and analyses on various n-gram models/smoothing methods (+1 to +3 points)