

# ASSIGNMENT 1

Due: Friday 09/27/2024 @ 11:59pm EST

## Disclaimer

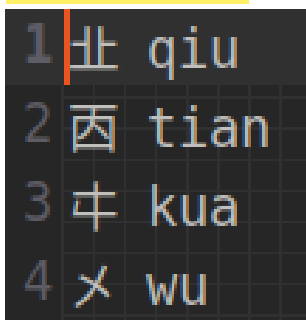
I encourage you to work together, I am a firm believer that we are at our best (and learn better) when we communicate with our peers. Perspective is incredibly important when it comes to solving problems, and sometimes it takes talking to other humans (or rubber ducks in the case of programmers) to gain a perspective we normally would not be able to achieve on our own. The only thing I ask is that you report who you work with: this is **not** to punish anyone, but instead will help me figure out what topics I need to spend extra time on/who to help.

## Setup: The Data

Some settings, like typing on a phone, writing Chinese/Japanese characters, etc. are slow (even by human standards). It can be helpful for the machine to guess what the next character(s) the user will type. In this assignment, you will build a language model at the character-level, test how well it performs, and use it to **build a system that will recommend Mandarin characters given the user-inputted phonetic spellings of that character**.

Included with this file are some directories and a few files. The first notable element is the **data** directory. This contains both data files and code files. Here are the data files:

- **data/english/train**: The english data training set
- **data/english/dev**: The english data development (validation) set
- **data/english/test**: The english data test set
- **data/mandarin/charmap**: A file containing mandarin character/pronunciation pairs. The format looks like this:



1	止	qiu
2	西	tian
3	中	kua
4	又	wu

Each line has exactly two whitespace-separated columns. **The first column is the character, and the second is the pronunciation.**

- **data/mandarin/train.han**: The mandarin training data
- **data/mandarin/dev.pin**: The mandarin development data, but only the inputs to the model
- **data/mandarin/dev.han**: The mandarin development data, but only the correct answers (i.e. what your model should have produced)
- **data/mandarin/test.pin**: The mandarin test data, but only the inputs to the model

- `data/mandarin/test.han`: The mandarin test data, but only the correct answers (i.e. what your model should have produced)

The code files look like this:

- `data/charloader.py`: This file contains some utility functions for loading data from files
- `data/mandarin.py`: This file contains a utility function for loading in a mandarin file and replacing pronunciation of characters with the character from the charmap.

## Section: Starter Code

In addition to the `data` directory, there are a few code files included with this pdf. These files are:

- `utils.py`: This file contains a few useful functions for reading/writing to files (some of this functionality is duplicated with the code files in the `data` directory), but most importantly it contains the `Vocab` class. This class acts as a set of tokens, and will allow you to convert between the raw token and its *index* into the set. This will become important in the future when we work with neural networks, but for now this functionality isn't strictly necessary but I want you to get used to seeing it.
- `unigram.py`: This file contains the `Unigram` class which follows a specific api:
  - `start() -> Sequence[str]`: This method returns the start state, which is an abstract sequence of tokens.
  - `step(q: Sequence[str], w: str) -> Tuple[Sequence[str], Mapping[str, float]]`: This method runs one step of the model, where `q` is the state the model is in before the step, and `w` is the token to be read by the model. This method returns a pair, the first item is the state of the model after reading token `w`, and the second item is a mapping from token to log-probability after having read in `w`. For instance, the code to compute the log-probability of `foo` would be (using model `m`):

```
q: Sequence[str] = m.start()
lp_foo: float = 0f

q, p = m.step(q, "<BOS>")
lp_foo += p["f"]

q, p = m.step(q, "f")
lp_foo += p["o"]

q, p = m.step(q, "o")
lp_foo += p["o"]

q, p = m.step(q, "o")
lp_foo += p["<EOS>"]
```
- `test.py`: This file contains the training/eval loop to train a (to be created later) bigram language model on the sentence “the cat sat on the mat” and also eval the bigram model on that same sentence. Please use this to help debug your language model.
- `predict.py`: This file will train (at the moment) a unigram model the training data and eval the unigram model to predict the next character you type into the console based on the characters you have types previously.

The files `predict.py` and `test.py` use the `argparse` module, so you will need to supply arguments to these files if you choose to run them.

## Baseline (10 points)

As mentioned previously, the file `unigram.py` implements a unigram language model (with the above interface) in the `Unigram` class:

- (5 points) Create a file called `baseline.py`. Write a function called `train_unigram` that will load in the training data (`data/english/train`) and uses `unigram.Unigram` to train a unigram model.
- (5 points) In your file `baseline.py`, write a function called `dev_unigram` that takes your trained `unigram.Unigram` model as an argument, loads in the development data (`data/english/dev`), and, for each character position in the development data (including EOS), predicts the most probable character given all previous *correct* characters. This function should return two integers, the number of correctly predicted characters, and the total number of characters. Your accuracy should be at least 15%.

## English-Character Language Modeling (50 points)

Now you will try to improve the quality of your character predictions:

- (20 points) Create a file called `ngram.py`. In this file you should implement a generic n-gram language model with the class `Ngram`. Your class should follow the same api as `unigram.Unigram`, and it should take the value `n` as input to the constructor. I highly suggest you use the `test.py` file to help debug your solution.
- (5 points) Create a file called `english.py`. This file will work similarly to `baseline.py`. Create two functions: `train_ngram` and `dev_ngram`. These methods should perform the same experiment you did with the `unigram.Unigram` model, but instead now use a 5-gram character-level language model. Your accuracy on the development set should be at least 50%.
- (20 points) Now try to make your language model better. In your report, briefly describe what modifications you tried, and for each modification, what accuracy on the development set was. You must try at least one modification, and the accuracy on the development set should be better than the accuracy of your unmodified model.
- (5 points) In your `english.py` file, write a function called `test_ngram`. This function should take a fully-trained instance of your best model (which `train_ngram` should now produce) as an argument, load in the test data (`data/english/test`), and, much like `dev_ngram`, should predict the test data and measure accuracy. To get full credit, you must score at least 60% on the test set.

## Mandarin-Character Suggestion (40 points)

Mandarin is written using a rather large set of characters (3-4k), which presents a challenge for typing. Most younger users type Mandarin using a standard QWERTY keyboard to type the pronunciation of each character in Pinyin. This became possible thanks to statistical models (like n-gram models) that can automatically guess what the right character is.

For this part, you will write a program that can read in pronunciations (simulating what a user might type) and predicts what the correct mandarin character(s) are. To do this, you will use the following files:

- `data/mandarin/train.han`: You will train your language model on this.
- `data/mandarin/charmap`: As described earlier, this contains a mapping from pronunciations to mandarin characters.

- `data/mandarin/{dev.pin,test.pin}`: These files contain the data that will be input to your model. Sentences in this file have the following format:

o f a n , <space> ni zai yong shen me v p s ya ?

which means “ofan, what vps [virtual private server] are you using?” This is a simulation of what a user might type. For each whitespace-separated token, you have to guess what character the user meant to type (given the correct previous tokens). Each token could be one of:

1. a pronunciation (like `ni`), which can convert to one of the mandarin characters listed in the `charmap` file.
2. a single character (like `o`), which can convert to itself, or
3. `<space>`, which converts to a space.

Here is how we’re going to solve this problem:

- (10 points) Create a file called `charpredictor.py`, and in this file, create a class called `CharPredictor`. This class should take three arguments as input to the constructor: `n`, the path to the `charmap` file, and the path to the `.han` training file. This constructor should keep two mappings as fields: one that maps from the pronunciation to a set of mandarin characters, and another that maps from a mandarin character to the english pronunciation. You should also instantiate a language model, train it on the training data, and save it as a field of this class.
- (10 points) Create a method in this class called `candidates(token: str) -> Sequence[str]`. This method will, given a token, produce all of the tokens that the input token could map to. The output sequence is the union of tokens from (1) (2) and (3) above.
- (5 points) Create a method in this class called `start() -> Sequence[str]`. This method should act like the `start()` method of the language models.
- (5 points) Create a method in this class called `step(q: Sequence[str], w: str) -> Tuple[Sequence[str], Mapping[str, float]]` which will perform differently than the language model api. Your model should only predict the log-probabilities of the tokens that appear in the `candidates(w)` set. The state transition will be the same.
- (5 points) Create a file called `mandarin.py`. This file should contain functions `train_model` which instantiates your `CharPredictor` class and trains it on the training data using a bigram language model. You should also have a `dev_model` which loads in the data from `data/mandarin/dev.pin`, and, for each token, uses your `step` method to predict the most probable token from the `candidates` set. Your function should produce the number of correct predictions as well as the total number of predictions. You should get an accuracy of at least 90%.
- (5 points) Create a function `test_model` which evaluates your trained character predictor on the test set and report the total number of correct predictions as well as the total number of predictions. For full credit, you must get an accuracy of at least an 87% accuracy.

## Submission

Please turn in all of the code files that you wrote to gradescope as well as a pdf report. In your report you should include console output of having run your code as well as any observations you made during your experiments. If an instruction in this prompt says to report on something, you should include those elements in your pdf. Please do not handwrite these: typeset pdfs only. There will be separate links on gradescope for your code and your pdf. Your code will be autograded, while your report will be graded by us.