COSI 114a HW 4 2025

Due Date

Due **Tuesday, November 4th, 10pm** Eastern Standard Time (UTC-5). Need to turn it in later? Follow the late homework policy on Moodle. Please email lignos@brandeis.edu if you need an extension for religious observance or another situation that cannot be addressed by using 1-2 late days.

Introduction

In this assignment, you will implement an HMM part of speech tagger. This assignment must be completed in order. Each problem builds on previous problems.

Release history

This assignment will receive updates based on student questions. We advise you to always use a live copy rather than saving this to a PDF or printing it out. A new version will be noted and an announcement will be made for any changes larger than minor clarifications and correcting typos/style.

10/17/2025 (1.0.0): First release

Collaboration Policy

Do not discuss the contents of your solution with anyone but the course staff. Do not show any portion of your solution to another person inside or outside of the class, and do not allow anyone to look at your solution. You may discuss the assignment with other students to help clarify what the assignment is asking you to do. You can also ask other students (or StackOverflow, Google, etc.) narrowly-focused Python questions ("how do I remove a character from the end of a string?"), but not questions about how to complete the assignment ("how should I parse each line into a sentence of tokens?"). You may not use generative AI (ChatGPT, etc.) or code generation tools (GitHub Copilot, etc.) in any way when working on this assignment. If these are enabled in your IDE, you must turn them off when working on this

assignment, including PyCharm's "Enable local full line completion suggestions" feature. It is okay to use the output of the Google "AI Overview" when searching, since it is essentially impossible to avoid and usually gives correct Python guidance. You should never cut and paste code from any source, but you may follow the practices laid out in basic Python tutorials, Real Python, GeeksforGeeks, etc.

Completing this assignment

All code you write should be contained in a single file with the filename `hw4.py`. You are provided a `hw4.py` file with the function definitions that you need to fill in.

Testing your solution

You will need the `cosi114a_hw4.zip` file posted on the Moodle homework page. This contains a file `test_hw4.py` and a helper module called `grader.py`. To test your solution, run `python test_hw4.py` in the terminal, which will give you a grade out of 50 points. These public tests will determine 50/100 points of your grade for this assignment. For more information on how to run the tests, see this guide. The private grading tests will determine the remaining 50 points.

Getting help

If you have questions about what the assignment is asking you to do, ask them on the HW 4 Questions forum on Moodle. For help debugging your solution, bring it to office hours. If you have a very simple Python question (one that does not require debugging your code) or think something is wrong with the tests, please email `cosi114ahelp-group@brandeis.edu`. Please paste (never attach!) the **full contents** of your homework file at the bottom of the email when emailing the help address. **If you attach Python code to an email, it will never be received!**

Submitting your solution

Please submit your assignment using the HW submission instructions (steps 0-3, do not run steps A-D again).

General notes

1. You can always assume that your code will be called with arguments of the correct type by the tests.
2. You should remove the `pass` statement from the function definitions when you implement them.

3. You can write as many helper functions as you like, as long as you include them in the single file you submit. Do not change the declarations or type annotations of the functions we provide.

Background

Before you begin

This assignment makes significant use of classes. If you are not comfortable with concepts like class, object, subclass, parent class, or inheritance, be sure to solidify your understanding before proceeding too far in the assignment. Relevant resources for learning about classes are in the resources guide for the class. You can also read Python's own tutorial on classes, which can be very helpful, even if a bit too detailed for this course. You are also welcome to come to office hours for help.

Data

The data for this assignment is a section of the Penn Treebank.

The tests take care of loading the data for you, but if you want to write your own tests, you can use `load_pos_data("test_data/train_pos.txt")` to load the training data. This function (located in `test_hw4.py`) will return a generator of sentences which are each represented as `list[TaggedToken]`. Note that since it is in the `test_hw4` module, which depends on `hw4`, you cannot use that function in `hw4.py`. Make a new file (for example, `debug_hw4.py`), and import what you need to from `hw4` and `test_hw4` there.

The `TaggedToken` class has already been implemented for you, and serves as a container that holds each token's text and part-of-speech tag. This way, you have a single object with an explicit reference to both the text and POS tag of a given token, which frees you from having to use a tuple and trying to remember whether indices 0 and 1 correspond to the token text and POS tag, etc.

The following snippet illustrates how to work with the generator returned by `load_pos_data`, as well as with the `TaggedToken` class and its `.text` and `.tag` instance attributes:

```
>>> from test_hw4 import load_pos_data
```

```
>>> tagged_sents = load_pos_data("test_data/train_pos.txt")
>>> for sent in tagged_sents:
...     for token_tag in sent:
...         print(token_tag.text, token_tag.tag)
It PRP
was VBD
Richard NNP
Nixon NNP
```

The Tagger Class

`Tagger` is an abstract class, which means it implements some methods common to all taggers, but leaves other methods to be implemented by its subclasses. Because all taggers will all have a uniform `test` and `tag_sentences` method, by putting them in `Tagger`, we can implement them just once in the base class and use them in all the subclasses.

The other tagger classes, such as `UnigramTagger` in the example below, will *inherit* from the `Tagger` base class. This means that they are subclasses of `Tagger` and will have its methods available to them as well.

```
class UnigramTagger(Tagger):
    ...
```

This says that `UnigramTagger` will inherit from `Tagger`, so it has the same `tag_sentences` and `test` methods which it inherited.

`BigramTagger` is a bit more complex. It inherits from `Tagger` and `ABC`. ABC stands for abstract base class. In Python, inheriting from ABC is necessary to create an abstract class, one which has methods that are not implemented because their child classes are meant to implement those methods in different ways.

For example, `tag_sentence` is defined in `Tagger` with the `@abstractmethod` decorator; this means you'll implement different versions of it in each of the taggers you create. The same goes for `train`, which is only implemented in `BigramTagger`;

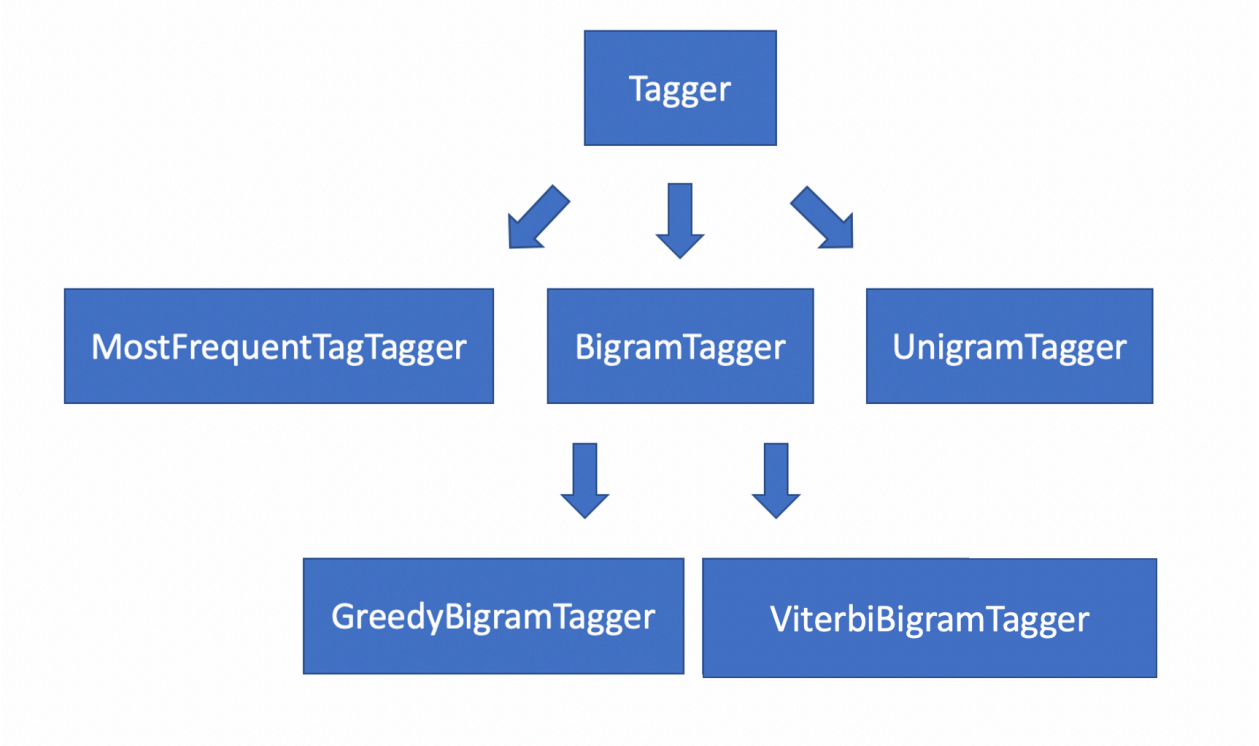
you will be responsible for implementing it for `MostFrequentTagTagger` and `UnigramTagger`.

Similarly, both `GreedyBigramTagger` and `ViterbiBigramTagger` will inherit from `BigramTagger`, which means they will use the `train` method implemented in their parent class. So all you'll have to implement for the `GreedyBigramTagger` and `ViterbiBigramTagger` is how to tag an individual sentence—all other functionality comes "for free" from the parent classes `BigramTagger` and `Tagger`.

This may seem complicated, but trust us, it saves you from having to write the same functions multiple times and also will make your code easier to read and debug since there's less redundancy.

The most important thing is that you follow the instructions regarding what to implement and always obey comments like `# DO NOT MODIFY THIS CLASS` and `# DO NOT IMPLEMENT THIS METHOD HERE`.

To recap, below is a visual representation of how the different tagger classes in this assignment relate to each other. An arrow of the form `X → Y` denotes that the class `X` is the parent of class `Y`, i.e. that `Y` inherits from `X` . Note that inheritance is transitive: if `X → Y` and `Y → Z` both hold, then this implies that `X → Z` holds as well. Concretely, since `BigramTagger` inherits from `Tagger`, and `GreedyBigramTagger` inherits from `BigramTagger`, we know that `GreedyBigramTagger` also inherits from `Tagger` as well.



Visualization of the class hierarchy for this assignment.

Provided functions

These are some helpful methods that we've provided for you and that you should make use of in completing this assignment.

```
def safe_log(n: float) -> float:
```

We've provided the function `safe_log` for you. It will return negative infinity if given 0 and log(n) otherwise. Use this method instead of `log` so you don't have to worry about handling log(0) which is undefined.

```
>>> safe_log(4)
1.3862943611198906
>>> safe_log(0)
-inf
```

```
def max_item(scores: Dict[str, float]) -> Tuple[str, float]:
```

We've provided you with `max_item` which takes a dictionary of tag strings mapped to their scores (which can be any integers or floats) and will return the `(max tag, max value)` tuple:

```
>>> d = {"NN": .78, "VB": .23, "DT": .98}
>>> max_item(d)
('DT', 0.98)
```

```
def most_frequent_item(counts: Counter[str]) -> str:
```

We've provided another useful function that gets the most frequent key from a counter:

```
>>> _most_frequent_item(Counter(["a", "b", "a", "c"]))
'a'
```

Imports

The following imports (and nothing else) are allowed:

- You should import `log` from `math` (we have already done so in the stub file we provide). We provide a wrapper function called `safe_log`, so you should not call `log` directly.
- You should import containers from the `collections` standard library module (e.g., `defaultdict`, `Counter`). The stub file does this for you.
- You may import whatever you need from the `typing` module to support type annotations.
- You may import anything from `operator`, `itertools`, and `functools`.
- **You cannot import `numpy`**. From vast experience, we can tell you that most students who try to use numpy only end up making their code slower, harder to read, and harder to debug than the approach we recommend (which is based around dictionaries). For non-vectorized operations, numpy array access is often slower than Python lists, and very little of HMMs can be vectorized easily when smoothing is involved.
- The code we provide you imports from `abc` for abstract base classes.

You should freely use the `defaultdict` and `Counter` classes in this assignment; the restrictions from previous assignments do not apply to this one.

Assumptions

Throughout the entire assignment, you can assume the following:

1. Every sentence passed as an argument contains at least one token.

2. Every iterable of sentences passed as an argument contains at least one sentence.
3. For your `SentenceCounter` class, the `train` method is always called before any of the methods that return probabilities or the tagset.
4. For your taggers, `tag_sentence` is never called without `train` being called first.

1. Most Frequent Tag Tagger

First, you'll implement a simple baseline tagger. This tagger will figure out the most frequent tag in the training data and tag every single token with the most common tag in the entire dataset. Needless to say, it's going to do very poorly overall.

Here's what the methods should do:

```
def train(self, sentences: Iterable[Sequence[TaggedToken]]) -> None:
```

- Count all the tags in sentences and store the most common tag in an instance attribute. Use `most_frequent_item` to get the most common tag.

```
def tag_sentence(self, sentence: Sequence[str]) -> list[str]:
```

- For the sentence provided, use the most frequent tag to tag each token. The sentence is a `Sequence[str]` where each string is the text of each token. You will return the a list of tags that is as long as the original list of tokens.
- In the example below, assume that the most frequent tag in the training data was `"X"`, (which is not a tag in the real data).

```
>>> tagger.tag_sentence(["I", "like", "apple", "cider"])
["X", "X", "X", "X"]
```

Bad implementation penalties

1. Storing anything in an instance attribute other than the most frequent tag, for example storing counts for all tags, storing any of the original tokens or sentences, etc. (-5 points)
2. Computing the most frequent tag inside the `tag_sentence` method (-5 points). Perform this in `train`. In addition to the bad implementation penalty, doing this

incorrectly can cause you to fail timed tests.

2. Unigram Tagger

The unigram tagger provides a much stronger baseline. Rather than assigning all tokens the same tag, it will assign every token the most frequent tag associated with its type in the training data. For example, if the type "attack" is seen as "NN" 10 times and "VB" 9 times in the training data, the unigram tagger will tag all tokens of "attack" as "NN" every time, regardless of context. Note that here and everywhere else in the assignment, your handling of strings should be case-sensitive; "attack" and "Attack" are two completely different, unrelated types.

Here's what the methods should do:

```
def train(self, sentences: Iterable[Sequence[TaggedToken]]) -> None:
```

- Count all the tags in the sentence and store what you need for the `tag_sentence` method. You will want to store a mapping from each type to the tag to use with that type (i.e. the most frequent tag seen with it). You will also want to keep the most frequent tag overall across all tokens. You do not need to (and should not) keep anything else. Use `most_frequent_item` to get the most common tag and most common tag per type.

```
def tag_sentence(self, sentence: Sequence[str]) -> List[str]:
```

- For each token in the sentence, **tag it with the tag that appears most often in the training data for that type**. You must have determined what the most frequent tag for every type is **in advance in the `train` method.**
- For types not seen in training, **tag them with the most frequent tag seen over all the tokens seen during training**. Hint: this is one of the few times in this entire course that using the `get` method on a dictionary can be used to make things easier.

Bad implementation penalties

1. Storing anything in an instance attribute other than the most frequent tag for each word type and the most frequent tag across all tokens, for example storing

the counts of all tags for each tokens, storing any of the original tokens or sentences, etc. (-5 points)
2. Computing the most frequent tag for each type inside the `tag_sentence` method (-5 points). Perform this in `train`. In addition to the bad implementation penalty, doing this incorrectly can cause you to fail timed tests.

3. SentenceCounter

Very similarly to HW 3, this class will store all the needed counts needed to calculate the transition, emission, and initial probabilities for an HMM tagger. Since you will use Lidstone smoothing for emission probabilities (but not transition probabilities), the class will have `k` as a parameter in the `__init__` method. Your bigram taggers will each have an instance of this class and will call the `count_sentences` method to train.

All of the probabilities returned by this class will be standard probabilities, not log probabilities.

Methods

You need to implement the following methods:

```
def count_sentences(self, sentences: Iterable[Sequence[TaggedTok
en]]) -> None:
```

Iterate through all the sentences and count all relevant tokens and tags. All computationally-expensive work should be done here. Do not store the original sentences or any `TaggedToken` objects.

```
def emission_prob(self, tag: str, word: str) -> float:
```

Return the Lidstone-smoothed emission probability for the tag and token specified. This corresponds to B in an HMM. Compute Lidstone smoothing as follows:

- *N* is the total number of times the tag was seen in training.
- *V* is the vocabulary for **this tag**, the number of unique words it was seen with in training. **Note that this is very different than the implementation of smoothing for the naive Bayes assignment, where the vocabulary is shared across classes.**

(Footnote for nitpickers: Given that the emission probability can be computed using an unbounded vocabulary (unlike the restricted feature set in the naive Bayes assignment), you might have noticed that this is actually a degenerate probability distribution. The total probability over the potentially infinite vocabulary sums to greater than one. One way to reconcile this is to pretend every word not seen with a given tag during training is actually the special zero-count word "UNK" (unknown), and add one to the vocabulary to account for it. Empirically, doing so has no effect on tagger performance, so we left adding one to the vocabulary it out of the assignment. A better approach would be to estimate the pseudo-count for UNK using held-out data, but that requires even more effort. Regardless, just do what the assignment tells you, and don't call the probability distribution police on the course staff.)

```
def transition_prob(self, tag1: str, tag2: str) -> float:
```

Return the (unsmoothed) probability of transitioning from `tag1` to `tag2`. This is the A matrix of the HMM.

```
def initial_prob(self, tag: str) -> float:
```

Return the (unsmoothed) probability of a tag being the initial tag in a sequence. This is the π vector of the HMM.

```
def unique_tags(self) -> list[str]:
```

Return a sorted list of all the unique tags. Because this method is key to the deterministic behavior of your tagger, we have implemented the sorting method for you. You need to call `items_descending_value` with a Counter that has counted all of the tags encountered in the input to the `count_sentences` method. For example, `sorted_tags = items_descending_value(tag_counts)`. **You must compute this value in the `count_sentences` method and return it in the `unique_tags` method.** If you perform this sort every time `unique_tags` is called, you will fail timed tests.

Implementation details

You need to avoid zero division errors by checking if the denominator is zero, and if it is returning 0.0 for `emission_prob` and `transition_prob`. `initial_prob` cannot cause a `ZeroDivisionError` because you can assume you saw at least one non-empty sequence during training. You should ensure that no matter what strings are provided for the tags or tokens (even if they were never observed in the training

data), these three functions will never crash due to a `ZeroDivisionError` or a `KeyError` due to a failed dictionary lookup.

All of the `{emission, transition, initial}_prob` methods as well as `unique_tags` **must be implemented in constant (O(1)) time!** This means that they should not contain **any** loops or calls to the `sum` function or `total` method, or any conversions like `list`. As in past assignments, you should compute whatever you need inside the call to `count_sentences` so that you only have to compute it once. Note that if you do not implement these methods in constant time, you will fail the `test_efficient_implementation` test, which is a timed test, and may fail additional grading tests.

Bad implementation penalties

1. Using any loops or calling any functions that loop in your `{emission, transition, initial}_prob` or `unique_tags` (-3 points per method).
2. Storing lists of all of the original sentences' tags, tokens, or both to be counted in a second pass in `count_sentences` (-5 points). Instead, just compute all the counts you need in a single pass over the sentences. Note that this does not preclude you from doing work after the loop over sentences on any counting data structures you create, such as computing the totals to be used as a denominator, etc. This penalty is for storing the data in uncounted form in a way that is memory-inefficient when it could have been counted in the main loop.

4. Bigram Taggers

You will implement two bigram HMM taggers: `GreedyBigramTagger` and `ViterbiBigramTagger`.

For all calculations in your bigram taggers, you must work in the log space for numerical stability, adding log probabilities rather than multiplying probabilities. Use `safe_log` which was provided to you as it handles `log(0)` for you (by returning negative infinity) and uses the correct base for the log (base e). **Never multiply probabilities in any of the methods of these classes**.

The `__init__` method

Since both `GreedyBigramTagger` and `ViterbiBigramTagger` inherit from `BigramTagger`, the `__init__` method has already been implemented for you in the `BigramTagger` class. You should not modify that implementation in any way nor implement your own `__init__` functions in either of `GreedyBigramTagger` or `ViterbiBigramTagger`. Simply leave the already implemented `__init__` as-is and use it in the subclasses.

When the tests create `GreedyBigramTagger`/`ViterbiBigramTagger` objects, they pass in as an argument a value `k`:

```
>>> tagger = GreedyBigramTagger(0.0)
>>> tagger = ViterbiBigramTagger(1.0)
```

The `__init__` method will then take this `k` value and pass it to the `SentenceCounter` that the tagger uses internally. Note that your code must work when a zero value has been provided for `k`.

Training, counts and probabilities
As mentioned above, the `train` method of `BigramTagger` has already been implemented for you, and simply consists of calling `count_sentences` which collects the counts needed for calculating probabilities. Do not change this, and do not call `train` or `count_sentences` directly in any of the other methods.

Instead, in your implementations of `tag_sentence` for both bigram taggers, you should use the probability methods that you implemented in `SentenceCounter`. You can access these probability methods like so:

```
self.counter.transition_prob(prev_tag, tag)
self.counter.emission_prob(tag, token)
```

Determinism Tests
When decoding a tag sequence using greedy or Viterbi decoding, you should make sure your solution is *deterministic* in the event of a tie between scores.

Deterministic means that you will get the same results if you run multiple times. But more than just being deterministic, your code should predictably resolve ties such

that if multiple tags have the same score, the one that is sorted first by `unique_tags` will be chosen.

Doing this is actually extremely simple:

- Whenever you iterate over all possible states, ensure they are in the sorted order specified by `SentenceCounter`'s `unique_tags` method.
- Make sure that when you insert scores into a dict or list, you do so in the order of the sorted labels. If you use `max_item`, it will pick the first element with the top score, so if you have inserted into the dictionary in the right order, everything will work out well.
- If you implement your solution well, all this is zero extra work.

The public tests will not check if your code is behaving deterministically.

Sequence Probability

In this section, your task is to implement `sequence_probability` in `BigramTagger` which serves as the parent class of both `GreedyBigramTagger` and `ViterbiBigramTagger`:

```
def sequence_probability(
    self, sentence: Sequence[str], tags: Sequence[str]
) -> float:
```

The `sequence_probability` method should take a sentence (a sequence of token strings) and its tags (a sequence of tag strings) as arguments and compute the log probability of the entire sequence of the tags given the tokens. You can assume that `sentence` and `tags` are the same length and that length is greater than or equal to one.

Make sure to use `safe_log`, which will ensure that if the probability is zero, negative infinity (`float("-inf")`) is returned. (Technically, the log of a zero probability is undefined, but negative infinity is much easier to work with.)

Below is an example for finding the sequence probability of the sentence “foo bar” where “foo” has tag “A” and “bar” has tag “B”:

```
>>> tagger.sequence_probability(["foo", "bar"], ["A", "B"])
```

```
-3.2188758248682006
```

Since `BigramTagger` is an abstract class, this function can only be called from an instance of `GreedyBigramTagger` or `ViterbiBigramTagger`. If you want to write your own tests for this method, instantiate one of those classes. You will get an error if you try to create an instance of `BigramTagger`, as it is an abstract class and can't be instantiated.

To compute the sequence probability, use the equations discussed in class (HMMs I lecture). You should be able to get all values needed from `self.counter`. You can assume that the `train` method has been called before `sequence_probability`.

Greedy Bigram Tagger

This tagger is greedy in the sense that it makes a decision for which tag to output at the current point without considering the whole tag sequence the way Viterbi will. For this tagger, you just need to implement its `tag_sentence` method.

```
def tag_sentence(self, sentence: Sequence[str]) -> List[str]:
```

For each token, choose the tag with the max probability based on the transition probability from the previous token's best tag and emission probability. Like other `tag_sentence` methods, you should return the tag sequence as a list of string tags. Implement greedy prediction as discussed in class.

Note: if you do not follow the instructions and instead write an inefficient `tag_sentence` method, you will fail timed grading tests.

Viterbi Bigram Tagger

For this tagger you also just need to implement `tag_sentence`, which will be the Viterbi algorithm:

```
def tag_sentence(self, sentence: Sequence[str]) -> List[str]:
```

The exact data structures are up to you, but you'll want to set up a way of tracking the scores for each state of each token and also a method of tracking back pointers.

For more on Viterbi decoding, see the relevant lecture slides, textbook reading, and the video posted on Moodle.

Hints

1. Viterbi decoding is very difficult to debug. It's much easier to spend a lot of time working it out on paper and then writing correct code, as opposed to not working it out, writing incorrect code, and trying to figure out what's wrong with it.
2. The algorithm described in the textbook (and elsewhere) for Viterbi decoding uses arrays for all data structures. This requires turning tags into indices, which makes debugging harder. Instead, we recommend that you focus on storing scores in dictionaries that go from tags to their scores, keeping a list of dictionaries (one per sequence position).
3. Note that the `max_item` helper is useful for both the inductive and termination steps. Do not write your own max function!
4. In the termination step, it is often easier to build the list of tags backwards (beginning with the final tag) and then reverse it.
5. If you aren't sure whether your Viterbi implementation is computing the scores correctly, check them using your `sequence_probability` function.

Note: if you do not follow the instructions and instead write an inefficient `tag_sentence` method, you will fail grading tests.

Bad implementation penalties

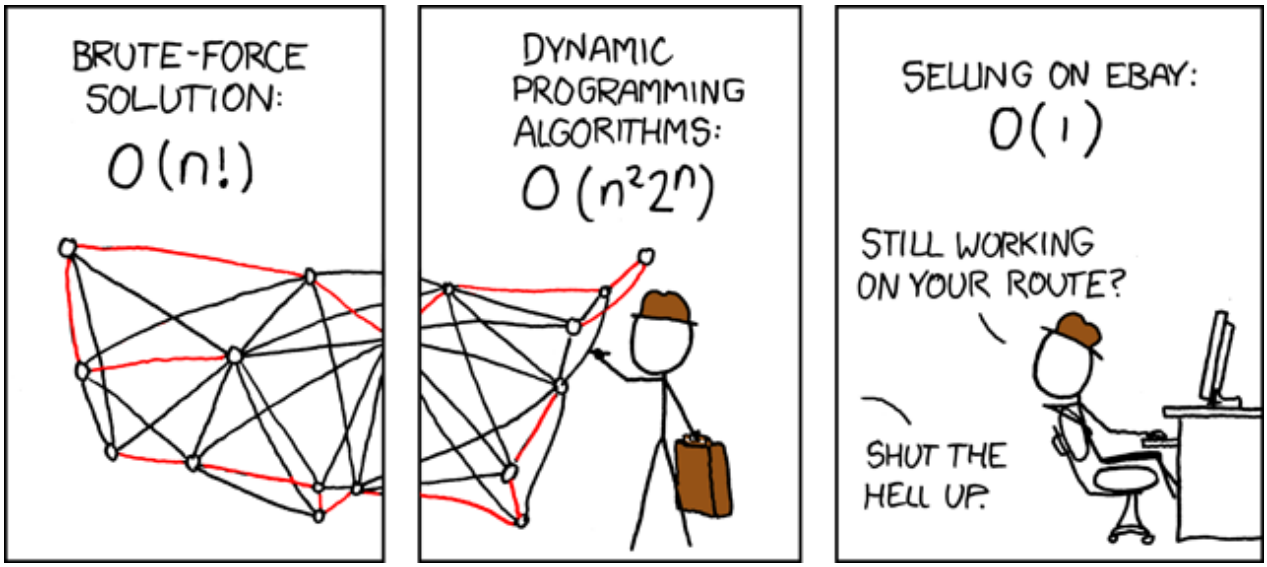
1. Storing any data other than a `SentenceCounter` in the classes you implement. (-5 points) The `BigramTagger.__init__` method is already implemented for you and you should not modify it.

FAQ

Can I write my own custom `.py` files?

Your final submission must be contained in a **single file** with the filename `hw4.py`. That being said, you *can* write your own `.py` files for debugging purposes if you wish. This is better than putting random testing code in your main file, which you then have to remember to remove. However, don't ever attempt to write parts of the solution in anything but your main file.

Obligatory closing meme



Travelling Salesman Problem