LING185A, Assignment #7

Due date: Wed. 2/28/2018

Download `ProbBigrams_Stub.hs` from the course website, and rename it to `ProbBigrams.hs`.[1] This file contains the code for a basic probabilistic bigram model. You will need to submit a modified version of this file. There are `undefined` stubs for each of the functions you need to write.

Also download the accompanying files `Corpus01.hs`, `Corpus02a.hs` and `Corpus02b.hs`, and save them in the same directory as `ProbBigrams.hs`. You should not modify these, and you do not need to submit them.

Background

Recall that, if the vocabulary of possible words is some set V, then a probabilistic bigram grammar specifies probabilities $\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x)$ for each $x \in V$ and $y \in V$. So there are approximately $|V|^2$ probabilities to be specified.[2] Using these component probabilities, we can compute a probability for any full sentence, like this:

$$\begin{aligned}\text{probability of 'John saw Mary'} = &\Pr(W_{\text{next}} = \text{John} \mid W_{\text{last}} = \texttt{<s>}) \\ &\times \Pr(W_{\text{next}} = \text{saw} \mid W_{\text{last}} = \text{John}) \\ &\times \Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw}) \\ &\times \Pr(W_{\text{next}} = \texttt{</s>} \mid W_{\text{last}} = \text{Mary})\end{aligned}$$

A game we can play now is the following. A "referee" finds a corpus of text somewhere, and gives us a large portion of it (maybe say 90%); this portion is called the *training corpus* or *training data.* We do our best to guess/estimate values for these $|V|^2$ probabilities that we need, on the basis of that training corpus. Then the referee gives us the rest of the corpus — this portion is called the *test corpus* — and we see what probability our chosen values assign to this test corpus. The aim of the game is to guess/estimate values of the probabilities which maximize the probability of the test corpus.

So how should we go about computing an estimate of probabilities like $\Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw})$, on the basis of some training corpus? We will consider four different ways.

Option #1: The simple bigram model

One straightforward idea is to define a function p_{S} (subscript '`S`' for "simple") as follows, for any two words $x, y \in V$:

$$p_{\text{S}}(y \mid x) = \frac{\text{count}_{\text{S}}(x, y, D)}{\text{count}_{\text{S}}(x, D)}$$

[1] Use this name exactly, please.

[2] I say "approximately" because the sum-to-one requirement means that if you know $|V| \times (|V| - 1)$ of these probabilities, you can work out the other $|V|$. And, maybe we should be adding 2 to $|V|$, depending on whether you include `<s>` and `</s>` in the set V. But life's too short to worry about these things, so "approximately $|V|^2$" will do just fine, and I'll make similar simplifications below.

where $\text{count}_S(x, y, D)$ is the number of times the bigram xy appeared in the training corpus D, and $\text{count}_S(x, D)$ is the number of times the word x appeared in the training corpus D. Then we might suppose that a good way to estimate probabilities like $\Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw})$ on the basis of some training data is via the assumption that

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x) = p_S(y \mid x)$$

Note that p_S is just a function, defined in a certain way, like any other. We write $p_S(y \mid x)$ instead of $p_S(y, x)$ as a sort of "reminder" of how we're using it, but it's not, strictly speaking, a probability. It's just a number that we calculate on the basis of the training data D.[3] This point may seem a bit obscure at first, but it should become a bit clearer when we try out a few other alternatives to p_S below.

A model that works this way is implemented in the file that you downloaded. We can "show" a fresh (untrained) model the two training bigrams 'a b' and 'a c', and then ask the resulting trained model for $p_S(\text{c} \mid \text{a})$, like this:

```
*ProbBigrams> bigramProbSimple (addBigramSimple ("a","c")
                                                (addBigramSimple ("a","b") freshSimpleModel))
                               ("a","c")
0.5
```

We can do the same thing[4] more concisely using `trainTestSimple`, which takes as arguments a training corpus and a test corpus (both lists of sentences), and computes the probability that a model trained on the training corpus assigns to the test corpus.

```
*ProbBigrams> trainTestSimple ["a b", "a c"] ["a c"]
0.5
```

At this point (or at least before you move on to the task of implementing Option #2 below) you might take a look at the implementations of the functions `addBigramSimple`, `bigramProbSimple`, `freshSimpleModel` and `trainTestSimple`, and the type `SimpleModel`. The `Simple` part of these names indicates that they implement the "simple" model p_S. These make use of some general helper functions that are defined in the top section of the file; in particular, notice `addOne`, `divisionHelper` and `trainModel`.

As we mentioned in class, this model works by maintaining one hashtable that maps words to their counts (corresponding to $\text{count}_S(x, D)$ from above), and one hashtable that maps word-pairs to their counts (corresponding to $\text{count}_S(x, y, D)$ from above). The parametrized type `Map.Map a b` represents hashtables mapping things of type `a` to things of type `b`, so these two individual hashtables have type `Map.Map String Int` and `Map.Map (String,String), Int`, respectively.

Here are the functions on these hashtable types that we are making use of. These should be enough for everything you will need to do below. The `Ord k` constraint on the types expresses the fact that the type `k` that we choose to use as our keys must be ordered/sortable. You should not need to worry about this detail.

- `Map.member :: (Ord k) => k -> Map.Map k a -> Bool`
  `Map.member x tbl` returns `True` iff there is an entry for the key `x` in the hashtable `tbl`.
- `Map.insert :: (Ord k) => k -> a -> Map.Map k a -> Map.Map k a`
  `Map.insert x v tbl` returns a new hashtable that is like `tbl` but in addition has the value `v` for the key `x`.
- `Map.findWithDefault :: (Ord k) => a -> k -> Map.Map k a -> a`
  `Map.findWithDefault def x tbl` returns the value associated with `x` in the hashtable `tbl`

[3] Perhaps things would be a bit clearer if we wrote $p_S(D)(y \mid x)$ instead, to make explicit the dependence on D.

[4] Actually it's not *quite* the same thing: when we use `trainTestSimple` like this, we're actually showing the model *six* training bigrams, not just the two from the previous example: `("<s>","a")`, `("a","b")`, `("b","</s>")`, `("<s>","a")`, `("a","c")` and `("c","</s>")`.

if there is one, or `def` if there is no entry for `x` in `tbl`.

- `Map.adjust :: (Ord k) => (a -> a) -> k -> Map.Map k a -> Map.Map k a`
  `Map.adjust f x tbl` returns a new hashtable that is like `tbl` but has been modified by applying the function `f` to the value associated with the key `x`.
- `Map.empty :: Map.Map k a`
  `Map.empty` is an empty hashtable. (It has the generic type `Map.Map k a` for the same reason that the empty list `[]` has type `[a]`.)

OK, let's give it a whirl! The imported module `Corpus01` contains a corpus[5] split into two sections, named `training` (397 sentences) and `test` (5 sentences) that we can use our new toy on like this:

```
*ProbBigrams> trainTestSimple Corpus01.training Corpus01.test
2.2041799923561194e-39
```

So given the estimates we made on the basis of those 397 sentences of training data, the probability of the 5 sentences of test data is about 2.2×10^{-39}. That's a fairly small number, but that's to be expected, if you think about it. This is the part where you would now go and compare results with rivals who chose something other than $p_{\text{S}}(y \mid x)$ as their way of estimating $\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x)$ on the basis of training data. If they got a number less than 2.2×10^{-39} on `Corpus01.training` and `Corpus01.test`, then you get bragging rights (and vice-versa).

Let's try it on another (larger) corpus, `Corpus02a`:

```
*ProbBigrams> trainTestSimple Corpus02a.training Corpus02a.test
0.0
*ProbBigrams> length Corpus02a.training
1250
*ProbBigrams> length Corpus02a.test
40
```

That's not a good result: our model says that the probability of these test sentences is zero. But the training sentences and the test sentences were drawn from the same source, and if you have a look at them (in `Corpus02a.hs`) it certainly seems reasonable to hope that the information we extract from the training sentences should lead us to assign a nonzero probability to the test sentences. The problem, more specifically, is that our model assigns zero probability to the sentence 'this echidna wins frequently'; and this, it turns out, is because our model never saw the two-word sequence 'wins frequently' in the training data. So $p_{\text{S}}(\text{frequently} \mid \text{wins}) = 0$, and since we decided to take

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x) = p_{\text{S}}(y \mid x)$$

our model ended up with the conclusion that $\Pr(W_{\text{next}} = \text{frequently} \mid W_{\text{last}} = \text{wins}) = 0$, which is apparently not true.

```
*ProbBigrams> bigramProbSimple (trainModel addBigramSimple Corpus02a.training freshSimpleModel)
                               ("wins","frequently")
0.0
```

So at least when it comes to `Corpus02a`, it seems that the function p_{S} is not a great way to estimate these probabilities after all. We need some way to end up assigning nonzero probabilities to at least *some* bigrams that do not appear in the training corpus, such as 'wins frequently'. How? The general idea is that even if we didn't see 'frequently' come after 'wins', we might decide that this has a nonzero probability of happening if we did see 'frequently' come after words that are *similar to* 'wins'. Of course all the action now is in deciding what counts as *similar to* what. In practice, some obvious things to do would include trying to guess at the syntactic category of a word (maybe on the basis of its morphology), and take two words to be similar

[5] If you're curious, this corpus was generated from one of the grammars used in this paper: http://onlinelibrary.wiley.com/doi/10.1207/s15516709cog0000_64/abstract.

iff they belong to the same syntactic category (e.g. they are both verbs). For our purposes, to keep things simple, we'll try a couple of artificial toy versions of this idea.

Option #2: Categorizing based on length

Let's suppose we think that words can be meaningfully classified according to their length: there are words of category LONG, which are those containing more than four characters, and words of category SHORT, which are those containing four or fewer characters. If we think there's some meaningful distinction between LONG words and SHORT words, then maybe there's a meaningful distinction between how likely a certain word (say, 'frequently') is to occur after a LONG word and how likely it is to occur after a SHORT word.

So, let's define a function p_{L} (subscript 'L' for "length") as follows, for any two words $x, y \in V$:

$$p_{\text{L}}(y \mid x) = \frac{\text{count}_{\text{L}}(\text{cat}_{\text{L}}(x), y, D)}{\text{count}_{\text{L}}(\text{cat}_{\text{L}}(x), D)}$$

where $\text{cat}_{\text{L}}(x)$ is the length-category of the word x (i.e. either LONG or SHORT), $\text{count}_{\text{L}}(c, y, D)$ is the number of times the word y appeared after a word of category c in the training corpus D, and $\text{count}_{\text{L}}(c, D)$ is the number of times a word of category c appeared in the training corpus D. Then we might suppose that a good way to estimate probabilities like $\Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw})$ on the basis of some training data is via the assumption that

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x) = p_{\text{L}}(y \mid x)$$

So although the task we set ourselves is estimating values for all of the approximately $|V|^2$ probabilities of the form

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x)$$

this strategy simplifies things by supposing that this can actually be done by estimating only approximately $2 \times |V|$ values: instead of separately estimating how likely 'frequently' is to appear after each of the $|V|$ different words in our vocabulary, we just estimate how likely it is to appear after each of the two categories, LONG and SHORT. Of course a consequence of this assumption is that if $\text{cat}_{\text{L}}(x_1) = \text{cat}_{\text{L}}(x_2)$, then

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x_1) = \Pr(W_{\text{next}} = y \mid W_{\text{last}} = x_2)$$

which is probably oversimplifying. But it makes it less likely that we will assign probabilities of zero in situations where we shouldn't: even if the training corpus is too "sparse" to provide good estimates of all the $|V|^2$ values required by p_{S}, it might still provide good estimates of the $2 \times |V|$ values required by p_{L}. We say that p_{S} has $|V|^2$ parameters, whereas p_{L} has only $2 \times |V|$ parameters.[6]

Your task here is to implement the p_{L} model by filling in the definitions of the functions `addBigramByLength`, `bigramProbByLength`, `freshLengthModel` and `trainTestLength`, and defining the type `LengthModel`. You can do this by following the pattern of the corresponding functions with `Simple` in their name, which implement the p_{S} model: if you copy and paste that code, you won't have to change much. Remember that you can find the length of a `String` using the predefined `length` function. (Of course if we were doing this properly we'd factor things out more nicely so that there wasn't so much code duplication — but if we did that here, I would just be directly pointing out to you which pieces you need to change and which pieces you don't!)

When you've done that, you should get these results:

```
*ProbBigrams> trainTestLength Corpus01.training Corpus01.test
6.170502271540673e-73
*ProbBigrams> trainTestLength Corpus02a.training Corpus02a.test
1.4757564967918652e-281
```

[6] But see footnote 2.

So the length-based model p_{L} does worse than the simple model p_{S} on `Corpus01`, since $6.2 \times 10^{-73} < 2.2 \times 10^{-39}$, but does better than the simple model on `Corpus02a`, since it managed to assign a non-zero probability to `Corpus02a.test`. In particular, when trained on `Corpus02a.training`, $p_{\text{L}}(\text{frequently} \mid \text{wins}) > 0$, even though 'wins frequently' does not appear in this training corpus.

```
*ProbBigrams> bigramProbByLength (trainModel addBigramByLength Corpus02a.training freshLengthModel)
                                 ("wins","frequently")
1.4423076923076924e-2
*ProbBigrams> bigramProbSimple (trainModel addBigramSimple Corpus02a.training freshSimpleModel)
                               ("wins","frequently")
0.0
```

Of course the sacrifice we have made is that we no longer distinguish between, for example, the bigrams 'wins frequently' and 'eats frequently', since we only see the categories of 'wins' and 'eats' (which are both SHORT). The lack of these sorts of fine-grained distinctions is the reason p_{L} does worse than p_{S} on `Corpus01`.

Option #3: Categorizing based on starting-with-a-vowel

Let's suppose we also think that words can be meaningfully classified according to whether their first letter is a vowel or not: there are words of category VOWEL, which are those that begin with a vowel ('a', 'e', 'i', 'o' or 'u'), and words of category CONS, which are those that do not begin with a vowel. We suppose that there's a meaningful distinction between how likely a certain word (say, 'frequently') is to occur after a VOWEL word (say, 'eats') and how likely it is to occur after a CONS word (say, 'wins').

So, let's define a function p_{V} (subscript 'V' for "vowel") as follows, for any two words $x, y \in V$:

$$p_{\text{V}}(y \mid x) = \frac{\text{count}_{\text{V}}(\text{cat}_{\text{V}}(x), y, D)}{\text{count}_{\text{V}}(\text{cat}_{\text{V}}(x), D)}$$

where $\text{cat}_{\text{V}}(x)$ is the vowel-category of the word x (i.e. either VOWEL or CONS), $\text{count}_{\text{V}}(c, y, D)$ is the number of times the word y appeared after a word of category c in the training corpus D, and $\text{count}_{\text{V}}(c, D)$ is the number of times a word of category c appeared in the training corpus D. Then we might suppose that a good way to estimate probabilities like $\Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw})$ on the basis of some training data is via the assumption that

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x) = p_{\text{V}}(y \mid x)$$

Like p_{L}, this model p_{V} has $2 \times |V|$ parameters, in contrast to p_{S} which has $|V|^2$ parameters.

Your task here is to implement the p_{V} model by filling in the definitions of the functions `addBigramByVowel`, `bigramProbByVowel`, `freshVowelModel` and `trainTestVowel`, and defining the type `VowelModel`. (Remember that a `String` is just a `[Char]`, so you can use a `case`-expression to split it into `c:cs` and `[]` cases.)

When you've done that, you should get these results:

```
*ProbBigrams> trainTestVowel Corpus01.training Corpus01.test
3.6185686680831194e-77
*ProbBigrams> trainTestVowel Corpus02a.training Corpus02a.test
5.5504756814248e-284
```

So this vowel-based model does slightly worse than the length-based model p_{L} on both `Corpus01` and `Corpus02a`. But it manages to assign a non-zero probability to bigrams like 'wins frequently' that did not appear in the training corpus, for the same reasons as p_{L} did.

```
*ProbBigrams> bigramProbByVowel (trainModel addBigramByVowel Corpus02a.training freshVowelModel)
                                ("wins","frequently")
4.327833378414931e-3
```

Option #4: Combining length-based and vowel-based information

If the length-category and the vowel-category of a word are both meaningful indicators of the ways in which that word can combine with other words, then ideally we would like to be able to make use of both kinds of information at once. A very simple idea would be to average the values from $p_{\texttt{L}}$ and $p_{\texttt{V}}$:

$$\tfrac{1}{2}p_{\texttt{L}}(y \mid x) + \tfrac{1}{2}p_{\texttt{V}}(y \mid x)$$

which gives both of the previous models equal "weight". More generally, we can mix them according to some chosen weighting $\lambda \in [0, 1]$.[7] So let's define a function $p_{\texttt{I}}$ (subscript '`I`' for "interpolated") as follows, for any two words $x, y \in V$:

$$p_{\texttt{I}}(y \mid x) = \lambda \times p_{\texttt{L}}(y \mid x) + (1 - \lambda) \times p_{\texttt{V}}(y \mid x)$$

Then we might suppose that a good way to estimate probabilities like $\Pr(W_{\text{next}} = \text{Mary} \mid W_{\text{last}} = \text{saw})$ on the basis of some training data is via the assumption that

$$\Pr(W_{\text{next}} = y \mid W_{\text{last}} = x) = p_{\texttt{I}}(y \mid x)$$

Notice that this model has a total of $4 \times |V| + 1$ parameters: we need to have values for the $2 \times |V|$ parameters from $p_{\texttt{L}}$, we need to have values for the $2 \times |V|$ from $p_{\texttt{V}}$, and we also need to have a value for the additional parameter λ. Notice that there is no "obvious" way to work out a good value for λ from any given training corpus.

Your task here is to implement the $p_{\texttt{I}}$ model by filling in the definition of the function `trainTestInterpolated`. Note that this function has an extra argument compared to the previous `trainTest` functions, which is the additional parameter λ. So you do not need to work out a value for λ on the basis of the training corpus; it's up to whoever calls `trainTestInterpolated` to choose a value for this parameter.

When you've done that, you should get these results, using 0.5 as the value for λ:

```
*ProbBigrams> trainTestInterpolated 0.5 Corpus01.training Corpus01.test
2.5720908858667294e-74
*ProbBigrams> trainTestInterpolated 0.5 Corpus02a.training Corpus02a.test
1.858648641983316e-278
```

So on `Corpus01`, this interpolated model's performance is in between that of $p_{\texttt{L}}$ (which scored 6.2×10^{-73}) and that of $p_{\texttt{V}}$ (which scored 3.6×10^{-77}). This is probably what one would expect by interpolating between two models that were each attuned to the "made up" categorizations we tried out, on a corpus where those categorizations aren't very meaningful.

But on `Corpus02a`, it *just so happens* (`<wink/>`) that the length-based and vowel-based categorizations we are trying out *are* in fact both independently meaningful predictors of what's likely to follow a given word — and so this interpolated model does better than either of the two "component" models on their own! And we got this result just by trying $\lambda = 0.5$ arbitrarily. Try a few other values for λ to see if you can get even better results. (In another course, this is where we would be breaking out the calculus and start doing some differentiation . . .)

[7] It's conventional to use λ for this — it has nothing to do with the lambda calculus!

Model	Probability for `Corpus02a.test`
$p_{\texttt{I}}$ with $\lambda = 0$, i.e. $p_{\texttt{V}}$	5.6×10^{-284}
⋮	⋮
⋮	⋮
$p_{\texttt{I}}$ with $\lambda = 0.5$	1.9×10^{-278}
⋮	⋮
⋮	⋮
$p_{\texttt{I}}$ with $\lambda = 1$, i.e. $p_{\texttt{L}}$	1.5×10^{-281}

The fact that the interpolated model outperforms its two components on `Corpus02a` and not `Corpus01` is not a coarse-grained consequence of the different kinds of sentences in the two corpora (i.e. it's not a fact about the difference between words like 'frequently', 'wins', 'antelope' and 'every' on the one hand and words like 'forget', 'matter', 'treat' and 'strange' on the other, or about the difference between simple four-word sentences and longer complex sentences). To see this, try it out with `Corpus02b`, which contains sentences constructed according to the same rules as the rules used for `Corpus02a` (i.e. both are generated by the same *non-probabilstic grammar*, one might say). This shows the same pattern as `Corpus01`.

```
*ProbBigrams> trainTestLength Corpus02b.training Corpus02b.test
2.5067136364904823e-272
*ProbBigrams> trainTestVowel Corpus02b.training Corpus02b.test
1.396042781915158e-264
*ProbBigrams> trainTestInterpolated 0.5 Corpus02b.training Corpus02b.test
9.746166134072484e-268
```

The difference is that `Corpus02b` was created by sampling randomly, whereas `Corpus02a` was (for the purposes of this exercise) sampled according to a distribution which makes length-category and vowel-category information both useful predictors of the following word.