Task 1 & 2

- [x] Create class HerreraSegment
- [x] Handle unseen words
- [x] Test on tags
- [x] Tune program

See java file HerreraSegment.java and file HerreraSegmentTest.java for mostly commented code

Improved Results

[2, things, that, dont, mix] -12.29
[10, turn, ons] -18.45
[90s] -21.38
[100, things, about, me] -10.20
[all, i, want] -8.48
[annoying, bios] -16.04
[april, fools, jokes] -14.20
[a, really, good, ejob] -11.35
extras
[jinho, is, here] -12.93060480362456
[risk, it, for, the, biscuit] -7.600659555191378
[window, dressing] -9.576145828178902
[how, i, met, your, mother] -7.56615960305899

Original Results

[2thingsth, at, do, nt, mix] -5.51
[10, turn, on, s] -8.66
[9, 0, s] -13.48
[100things, a, b, out, me] -5.98
[all, i, want] -3.49
[an, no, yingbios] -6.03
[a, p, r, i, lfoolsjokes] -5.99
[are, ally, go, ode, job] -5.32

Task 3

- [x] Explain Improvements
- [x] Explain how tags handled
- [x] Submit all java files

Summary:

HerreraSegmentTest.java and improvement summery

Used Interpolation with unigramWeight = .005 && bigramWeight = .995

This gives room for unigrams to sway probabilities even if bigrams exist for the given words. You still get most of the benefit from before when using backoff && unigram = 0.01 as bigrams make up a majority of the probability and unigrams are punished harshly.

Bigrams (no smoothing)

I did experiment with implementing Discount/Laplace smoothing but given my setup with interpolation it didn't help as it would increase overall unseen wacky words way too much messing up my results.

Unigram (Discount Smoothing) : alpha = 0.9

We need a base probability for unseen words and this worked out before. It makes logical sense to punish the unseen word by a fraction of the minimum word. No real improvement from before.

Test Cases

Besides the final two test cases that were really long, I achieved to correctly segment all the other cases by adjusting probabilities within the segment class. I even added some extra cases to test the effectiveness of my program. I'm happy with the results as I was

able to fit in some dynamic programming via a hashmap to reduce the number of calculations used within the prior exhaustive search. With improvements on runtime and accuracy, the only thing left was to further increase efficiency.

Overall probabilities decreased from the original as my adjustments punished wrong segments and boosted likely characteristics of a good segment.

Accuracy was greatly improved upon from the original because of the probability adjusts that I'll go over in the next section.

Failures

I was unable to segment the final two cases. I tried developing a greedy search approach to skip calculations and comparisons so that it could at least complete but I failed miserably.

HerreraSegment.java and how tags are handled

Run Time Improvement

First a hashmap was created to store lists of strings that contained previously calculated probabilities so that we wouldn't have to recomput these numbers whenever a duplicate list would arise.

Sequence

Same as original

Class SegmentAux

Creates a list<String> Nestedlist to add to the hashmap later

Within the forloop when comparing word1 and word2 is where I filtered for segment characteristics to adjust probabilities. I'll review my code from a high level and go over what a specific layer allowed me to filter through subsequently that improved my code.

1st layer of characteristics | if (word1 == null)

I wanted to adjust probabilities (P) of the first word in the possible segment. This allowed me to boost P of a partition that consisted of or contained a number. Why? Because I wanted these partitions stand alone

2nd layer | if (word1.length() == 1)

This let me get rid of any letters not " i " or " a " from standing alone. This also allowed me to forcefully join any single digit in word1 together with word2 so that numbers like 10 wouldn't turn into 1, 0. In this same layer that I punish endings to numbers like "st", "rd" to single numbers so that "1 st" and " 3 rd" are more likely to be combined.

3rd layer | if (containsnumber(word2))

I made the method containsnumber to take into account if a segment contains a number, so that it could punish it. This leads to combing word1 and word2. This helps in situations when comparing “9” and “0s”.

4th layer | if (word2.length() == 1)

This layer helped combine a single digit in word2 if word1 was any real number. This made 100 possible by punishing the possibility of 10,0 appearing. This also assisted in making sure word2 would combine with word 1 if word2 wasn’t a or i. I also made the case for if “ I “ (capital i) was a partition, to boost the probability as its likely to be on its own.

5th layer | if (word2.equals("ing"))

This later made sure “ing” never stood on its own by punishing the probability.

6th layer | if word1 ends in “ing” and word2 does not begin with “s”

This helped huge with annoying bios as it boosts the probability that if word2 doesn’t begin with “s”, that the partition of word1 should end there where it is.

7th layer | if (word2.length() >= 2)

Here I helped endings like th,st,nd,rd to be combined with word1 if word1 was anynumber. In this same layer I tackled the issue of suffixes. I wanted these to be combined with whatever in word1. Here I also punished the consecutive appearance of words of length 2. This helped a bunch with making [dont] come together as opposed to [do , nt].

Final thoughts:

I believed I tackled some cases by two sides that helped the accuracy of my program: for example ruling out when word1 was a single value not a or i && when word2 was a single value not a or i. This got rid of other combinations that could be confusing I believe. When my characteristic filters were applied to other cases outside of the given examples it worked, so I’m confident that it should perform decently with other tags.

My failure to improve efficiency to handle longer tags amounts mostly because I didn’t start the homework early enough to leave time to continue meddling with the code.