Task

- ☑ Take any approach
- ☑ Explore more features
- ☑ Explain Improvements

Java files can be found in my hw1 file.

```
HMMTagger Output - Baseline
Training
Decoding
87.00 (122274/140551)
```

```
GreedyTagger Output – With additional features
Training
Decoding
95.27 (133900/140551)
```

From HW2Test.java

Approach: **Greedy Tagger** | Classifier: **Naïve Bayes** | Epsilon: **0.0000001**

For starters I ended up using a greedy tagger approach with Naïve Bayes as the classifier along with an epsilon of 0.0000001. I tried other taggers but I kept running out of memory. This makes sense as the data is huge and doing like an exhaustive search would hog my memory easily. Even in TopKtagger id run out of memory. I didn't look at the hidden markov model as I had troubles getting it to work so I left the default case (HMM tagger) to be representative of my attempt for that. As for the epsilon of 0.0000001, this came about midway through testing as I found lowering the constant improved accuracy to a degree. After becoming too small, epsilon would reduce accuracy because it would punish the lack of a feature existing rather than the quality of a feature existing. Quick example. Running the HMMTagger with 0.0000001 rather than 0.0001 would result in the accuracy becoming 93.08 as opposed to 87.

From AbstractTagger - Features Explored

Previous: **Tags** & **Words** & **Suffix** | Forward: **Words** | Current: **Word** & **Prefixes** & **Suffix** & **Length**

I can categorize my features into three main types. Previous looking, forward looking and current looking. Let's take a look at each of them briefly.

Previous Tags – 3 features

Here I made 3 features getting index -1, index -2 and index -3 tags. These turned out to be helpful at predicating the current tag of the index word. This makes sense as a sequence of tags could help predict the next tag. However when including index-4 tag, it hurt the accuracy of the program.

```
Example:  t = (index-1 < 0) ? null : tags.getTag(index-1);
          features.add(new StringFeature("f1", t));
```

Previous Words – 2 features

Similar in nature to the previous. Knowing the words of index-1 and index-2 helped the accuracy of decoding. I forgot to mention for this one and previous tags, that null is filled in to avoid null pointer exceptions.

```
Example:  t = (index-1 < 0) ? null : words.get(index-1);
          features.add(new StringFeature("f8", t));
```

Previous Suffix
This Helped a bit as knowing the previous suffix gives a clue to the next part of speech. I tried previous prefix and going more back to like index-2 but that didn't help accuracy.

```
Example:  if (index-1 >= 0)  {for(int w =1; w<=3; w++)
           {t = (PrevWordLen-w >= 0) ?  PrevWord.substring(PrevWordLen-w, PrevWordLen) : null;
              features.add(new StringFeature("f15", t))     }}
```

Forward Words – 2 features
Using forward words helped in decoding the current word. This makes sense as it gives context to the current word similar to how using previous words do. I used 2 words forward because when using 3 or more, the results were less accurate than just using 2.

```
Example:  t = (index+1 > words.size()-1) ? null : words.get(index+1);
              features.add(new StringFeature("f5", t));
```

Current Word
This was part of the base code given. Just like a prior it estimates the tagger based off the current word.

Current word's Prefix
This was a huge boost to accuracy. This makes sense as an ending like "ing" or "ed" or "s" does help predicted the part of speech of a given word. I played with the length of the prefix and concluded three was the best. This makes sense too as a 3 lettered suffix is most common.

```
Example:  for(int i =1; i<=3; i++)
              {t = (i <= wLength) ?  word.substring(0, i) : null;
               features.add(new StringFeature("f11", t));              }
```

Current word's Suffix
Similar to the current words prefix, this helps in determining the part of speech greatly. Similarly as before, I played with the length of the Suffix to conclude 5 was the best given the training data.

```
Example:  for(int j =1; j<=5; j++)
              {t = (wLength-j >= 0) ?  word.substring(wLength-j, wLength) : null;
              features.add(new StringFeature("f12", t));
```

Current Word Length – 3 features
These were funny in nature how I just guessed a bunch to come up with these. However they have some reason behind them as words with a length of 1 should be emphasized as well as of length 2. I theorized as well that long words shouldn't change part of speech much either so I should emphasize that too. After playing around with the numbers, it came out that 7 was the best as a cut off.

```
Example:  t = (wLength >=7) ?  words.get(index) : null;
              features.add(new StringFeature("f13", t));
```

Closing Thoughts

Compared to the HMM tagger that we went over in class, my abstract tagger does a better job at looking parts of each word as opposed to being strictly confined to observation likelihoods and transition likelihoods. My abstract tagger has some of that as it looks backward both on a word to word basis and on a tagger to tagger basis. However my abstract tagger also looks a bit forward as it takes into account the next two words. Given all that I've said, my programs main improvements and strengths over the class defaults come from parts of word features like prefixes and suffixes.