Q1

8

Q2

7 (ignoring case): `{'eighty', 'seventh', 'provides', 'anywhere', 'm', 'breakfast', 'connecting'}`

Q3

733 (ignoring case, otherwise 752)

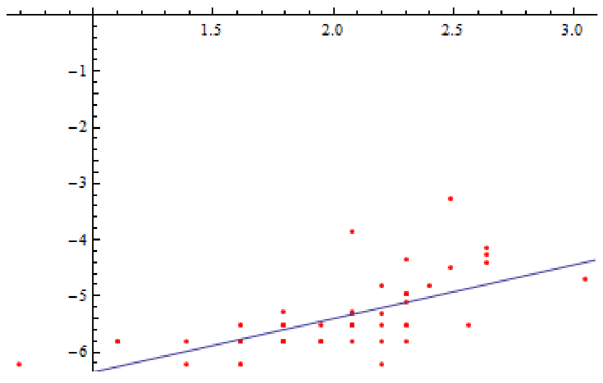
Q4

In my implementation, rules are indexed by both left and right side. Using this trick, the actual execution time largely depends on the tagging ambiguity of a specific sentence rather than its length. If we do linear regression on log(length)-log(time), we get $k < 1$, and it doesn't fit very well:

```
In[176]:= lm = LinearModelFit[Log[data], x, x]

Out[176]= FittedModel[ -7.31126 + 0.95462 x ]

In[177]:= Show[ListPlot[Log[data], PlotStyle → Red], Plot[lm[x], {x, 0, 30}]]
```



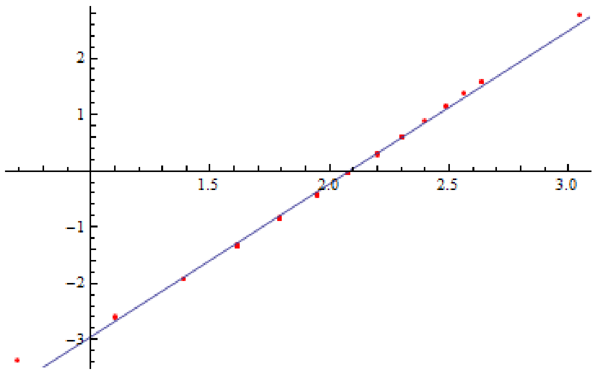
To eliminate the effect the of this optimization, we call `time.sleep(0.01)` each time before enumerating possible rules. By doing this, we get k=2.7, which is more consistent with the time complexity we know:

```
In[167]:= lm = LinearModelFit[Log[data], x, x]

Out[167]= FittedModel[ -5.66466 + 2.71695 x ]

In[174]:= Show[ListPlot[Log[data], PlotStyle → Red], Plot[lm[x], {x, 0, 30}]]

Out[174]=
```



Q5

```
./output/q5-dev.output 438 brackets
./data/dev.trees    474 brackets
matching    406 brackets
precision   0.9269406392694064
recall  0.8565400843881856
F1  0.8903508771929824
```

Q6

1. Handling parsing failure

Previously when we fail to parse a sentence, we simply print “0”. This gives us relatively high precision (since we tag only when we are confident) at the cost of low recall.

To improve recall, even if we cannot produce a parsing tree for an entire sentence, we should try to get credits for correctly parsing some parts of it. At the same time, we should still avoid bold guesses that seriously harms precision (and consequently F1 score).

Here’s what we do: when we fail to parse a sentence, we try to split it in to several parts, such that we can build a parsing tree for each part. Assuming the labels of the roots of these trees are $T_1, T_2, \ldots, T_m$, we add a new rule $TOP \rightarrow T_1, T_2, \ldots, T_m$ that has no cost. In other words, we assume the sub-trees are independently generated and we try to maximize the joint probability.

By doing this, we considerably improved recall without harming precision very much. As a result, F1 score is also improved:

```
./output/dev.output464 brackets
./data/dev.trees    474 brackets
matching    426 brackets
```

```
precision   0.9181034482758621
recall  0.8987341772151899
F1  0.908315565031983
```

2. Differentiating between <unk>s

We try to differentiate unknown words that:

a. Ends with "ing" / "s" / "ed"
b. Starts with capital letter / a non-letter character
c. Is a numeric string
d. Has length 1

by replacing them with different placeholders. It turns out that this doesn't help improving F1 score, if not making it worse. It seems that:

a. There are not many unknown words in `dev.strings`, the effect of differentiating between `<unk>`s is limited.
b. There are not many rare words in `train.trees` that are treated as unknown words; differentiating between `<unk>`s may overly limit the range of tags an unknown word can take.

3. Vertical markovization

We can perform simple vertical markovization by adding a "parent" tag to each non-terminal node. If we abandon the sentences that cannot be parsed, the resulting precision rise up to 0.968, but the recall plummets to 0.767, and F1 goes down to 0.856. If we try to construct trees anyway as we previous described, we get precision 0.925, recall 0.8607 and F1 0.8918. Clearly, smoothing is necessary.

We perform smoothing by backing off to the unmarkovized model with parameter $\alpha = 0.8$. This improves both precision and recall, and consequently improves F1 score:

```
./output/dev.output464 brackets
./data/dev.trees   474 brackets
matching   429 brackets
precision   0.9245689655172413
recall  0.9050632911392406
F1  0.9147121535181237
```

Q7

```
./output/test.output   462 brackets
./data/test.trees  471 brackets
matching   426 brackets
precision   0.922077922077922
recall  0.9044585987261147
F1  0.9131832797427651
```