1. P(c), the language model. We could create a better language model by collecting more data, and perhaps by using a little English morphology (such as adding "ility" or "able" to the end of a word). –> suffixes, 2–, 3– and 4–letter sequences
- by expanding big.txt with extra words, but the correction rate declines. Might because of overfitting.

2. P(w|c), the error model. So far, the error model has been trivial: the smaller the edit distance, the smaller the error. The intuition is that the two edits from "d" to "dd" and "s" to "ss" should both be fairly common, and have high probability, while the single edit from "d" to "c" should have low probability.
- Assign a higher weight to the words that delete or insert an identical alphabet to the alphabet next to it, ex: wrong_1: adress, wrong_2: addess, right: address, wrong_1 has a higher weight than wrong_2. Below is the modification part of my code.

```
from collections import defaultdict

def edits1(word, catagory, weight = 1.0):
    if catagory == 2:
        weight = weight * 0.09
    dict = defaultdict(lambda: [])
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    for L, R in splits:
        if R: #deletion
            if len(R)>1 and R[0] == R[1]:
                dict[L + R[1:]].append(weight*0.8)
            else:
                dict[L + R[1:]].append(weight*0.6)
            for c in letters:# replace
                dict[L + c + R[1:]].append(weight*0.6)
        if len(R)>1: #swap
            dict[L + R[1] + R[0] + R[2:]].append(weight*0.6)
        for c in letters:#insertion
            if (len(R)>1) and (c == R[0]):
                dict[L + c + R].append(weight*0.8)
            else:
                dict[L + c + R].append(weight*0.6)
    return_set = set()

    for key, value in dict.items():
        total = 0
        for weight in value:
            total = total+weight
        return_set.add((key, total/len(value)))
    return return_set
    #return set(deletes + transposes + replaces + inserts)
```

```
def correction(word):
    return max(candidates(word), key=lambda x: P(x[0])*x[1])
```

```
def candidates(word):
    mapping = dict(known(edits2(word, 2)))
    out = set()
    out = known(edits1(word,1))
    for a, b in known(edits1(word,1)):
        del mapping[a]
    for x, y in mapping.items():
        out.add((x, y))

    for a, b in known([(word, 1.0)]):
        out.add((a,(b*P(word)*0.2)))
    return (out or [(word, 1.0)])
```

```
def known(words):
    return set(w for w in words if w[0] in word_count)
```

```
def edits2(word, catagory):
    return (e2 for e1 in edits1(word, 1) for e2 in edits1(e1[0], 2, e1[1]))
```

3. For words beyond edit distance 2, we could consider extending the model by allowing a limited set of edits at edit distance 3. For example, allowing only the insertion of a vowel next to another vowel, or the replacement of a vowel for another vowel, or replacing close consonants like "c" to "s" would handle almost all these cases.

- Haven't done it.

4. As for speed, we could cache the results of computations so that we don't have to repeat them multiple times.

- Haven't done it.

```
#print(unit_tests())
#spelltest(Testset(open('development_set.txt'))) # Development set
spelltest(Testset(open('spell-testset1.txt'))) # Development set
#spelltest(Testset(open('spell-testset2.txt'))) # Final test set
```

```
67% of 270 correct (1% unknown) at 7 words per second
```

- The result of the modification bounces between 67% ~ 69% with different weights.