AIWIR PROJECT

SPELLCHECKER

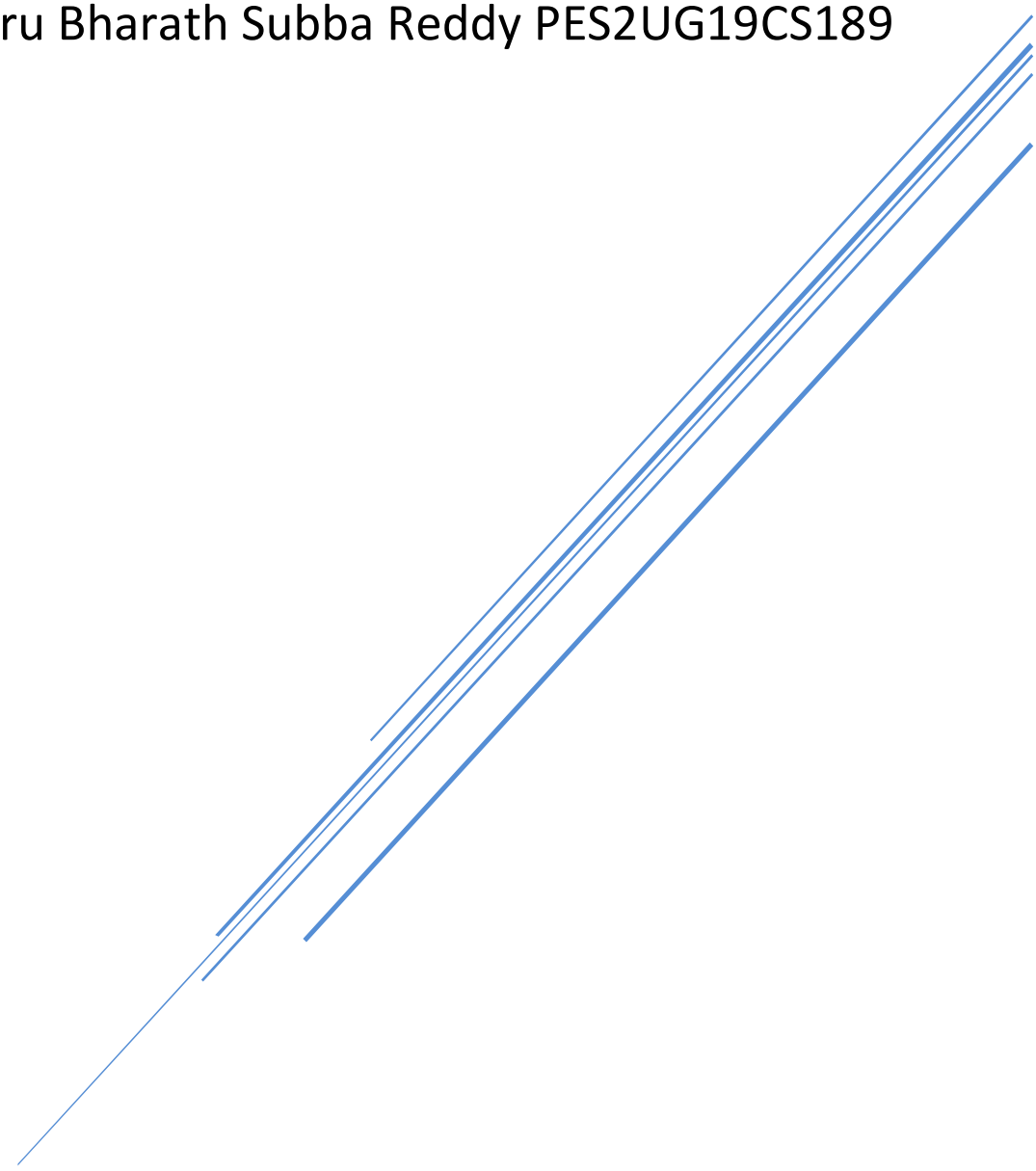
Batch-5

Group members

Achyut Jagini PES2UG19CS013

Aparna Kalla PES2UG19CS059

Koduru Bharath Subba Reddy PES2UG19CS189



Automatic Spelling Correction Techniques With Probabilistic Methods

I. Overview

This assignment builds a probabilistic spelling corrector to automatically correct errors in queries. More formally, given a (possibly corrupt) raw query R, the goal is to find the intended query Q which maximizes the probability $P(Q \mid R)$. That is, the program makes an estimate of the query which the user probably meant to submit. By Bayes' Theorem we have

$$P(Q \mid R) = \frac{P(R \mid Q)P(Q)}{P(R)} \propto P(R \mid Q)P(Q).$$

Since the goal is to find the value of Q which maximizes $P(Q \mid R)$, this shows it is sufficient to maximize $P(R \mid Q)P(Q)$. With the above formulation in mind, the code builds a probabilistic spelling corrector consisting of 4 parts:

1. **Language Model.** Estimates the prior distribution of unigrams and bigrams, allowing us to estimate $P(Q)$. We will use maximum-likelihood estimation, which counts the occurrences of token unigrams and bigrams in the training corpus in order to determine their prior probabilities.
2. **Edit Probability Model.** Estimates the likelihood of errors that may occur in a query, which allows us to estimate $P(R \mid Q)$. In particular, this component estimates the probability of characters being mistakenly deleted, inserted, substituted, or transposed in a query term.
3. **Candidate Generator.** Takes a raw query R submitted by the user, and generates candidates for Q.
4. **Candidate Scorer.** Combines (1), (2), and (3) to compute $Q^* = \arg\max_Q P(Q \mid R)$. That is, for each Q generated by the candidate generator, the scorer uses the language model to estimate $P(Q)$ and uses the edit probability model to estimate $P(R \mid Q)$, and finally chooses Q which maximizes $P(Q)P(R \mid Q)$.

A Note on Numerical Stability

Many of the probabilities we will encounter in this assignment are very small. When we multiply many small numbers together, there is a risk of underflow (https://en.wikipedia.org/wiki/Arithmetic_underflow). Therefore, it is common practice to perform this type of probability calculation in log space. Recall that:

1. The log function is monotonically increasing, therefore $\arg\max p = \arg\max \log p$.
2. We have $\log(pq) = \log p + \log q$, and by extension $\log\left(\prod_i p_i\right) = \sum_i \log p_i$.

As a result, if we want to maximize $P(\mathbf{x}) = P(x_1)P(x_2)\cdots P(x_n)$, we can equivalently maximize $\log P(\mathbf{x}) = \log P(x_1) + \log P(x_2) + \cdots + \log P(x_n)$. **For numerical stability, log-space formulation is used throughout the assignment.**

In [8]:

```
# Import modules
import math
import os
import urllib.request
import zipfile
from collections import Counter
from tqdm import tqdm
# from numpy import argmax
```

Approach 1: Spelling Correction with Uniform Edit Costs

1. Language Model

The language model estimates $P(Q)$ from the training corpus. We will treat Q as a sequence of terms $(w_1, \ldots, w_n)$ whose probability is computed as

$$P(w_1, \ldots, w_n) = P(w_1)P(w_2 \mid w_1) \cdots P(w_n \mid w_{n-1}),$$

where $P(w_1)$ is the unigram probability of term w_1, and $P(w_i \mid w_{i-1})$ is the bigram probability of (w_{i-1}, w_i) for $i \in \{2, \ldots, n\}$.

a. Calculating Unigram and Bigram Probabilities

The language model will use the maximum likelihood estimates (MLE) for both probabilities, which turn out to be their observed frequencies:

$$P_{\text{MLE}}(w_i) = \frac{\texttt{count}(w_i)}{T}, \qquad P_{\text{MLE}}(w_i \mid w_{i-1}) = \frac{\texttt{count}((w_i, w_{i-1}))}{\texttt{count}(w_{i-1})},$$

where T is the total number of tokens in our corpus, and where `count` simply counts occurrences of unigrams or bigrams in the corpus. In summary, computing unigram probabilities $P(w_i)$ and bigram probabilities $P(w_i \mid w_{i-1})$ is a simple matter of counting the unigrams and bigrams that appear throughout the corpus.

In [9]:

```
# Models prior probability of unigrams and bigrams.
class LanguageModel:

    def __init__(self, corpus_dir='pa2-data/corpus', lambda_=0.1):

        self.lambda_ = lambda_              # Interpolation Factor for smoothing by unigra
        self.total_num_tokens = 0           # Counts total number of tokens in the corpus

        self.unigram_counts = {}            # Dictionary to maintain unigram counts
        self.bigram_counts ={}              # Dictionary to maintain bigram counts

        for i in range(10):
            file = corpus_dir + '/' + str(i) + '.txt'
            with open(file, 'r') as fp:
                doc = fp.read()
                doc = doc.split()
                self.total_num_tokens += len(doc)
                for tok_id in range(len(doc)):
                    try:
                        self.unigram_counts[doc[tok_id]]+=1
                    except:
                        self.unigram_counts[doc[tok_id]]=1
                    try:
                        self.bigram_counts[doc[tok_id]+ " " + doc[tok_id+1]]+=1
                    except:
                        if(tok_id!=len(doc)-1):
                            self.bigram_counts[doc[tok_id]+ " " + doc[tok_id+1]]=1
```

With the unigram and bigram counts calculated, the query probabilities can now be computec. But bigrams that never occur in the corpus need to be interpolated.

b. Smoothing by Interpolation

The unigram probability model also serves as the vocabulary, since the corrector assumes that the query language is derived from the document corpus. However, even if the two query terms are both members of the query language, there is no guarantee that their corresponding *bigram* appears in the training corpus. To handle this data sparsity problem, the corrector *interpolates* unigram and bigram probabilities to get the final conditional probability estimates:

$$P(w_2 \mid w_1) = \lambda P_{\text{MLE}}(w_2) + (1 - \lambda) P_{\text{MLE}}(w_2 \mid w_1).$$

λ is set to a small value (say, 0.1) in the beginning, and experimented later with by varying this parameter to see if you better correction accuracies can be obtained on the development dataset. However, be careful not to overfit your development dataset.

In [10]:

```
# An extension of the Language Model Class
class LanguageModel(LanguageModel):
    def get_unigram_logp(self, unigram):

        try:
            return self.unigram_counts[unigram] / self.total_num_tokens
        except:
            # This block of code will never be entered since a candidate with OOV will neve
            return 0.00000000000000001

    def get_bigram_logp(self, w_1, w_2):

        try:
            return math.log(self.lambda_*self.get_unigram_logp(w_2) + (1 - self.lambda_)*(s

        except:
            # Need to return the interpolated value instead
            try:
                #print("\nBIGRAM DOES NOT EXIST: ", w_1+ " " + w_2)
                return math.log(self.lambda_*self.get_unigram_logp(w_2), 10)
            except:
                print("\nUNIGRAM DOES NOT EXIST: ", w_2)
                print("\n------------------------------")
                return -18

    def get_query_logp(self, query):

        query = query.split()

        # P(Q) = P(w1)*P(w2|w1)...P(wn|wn-1)
        # log(P(Q)) = log(P(w1) + log(P(w2|w1))+ ... + log(P(w2|w1))
        probability_product = 0
        for i in range(1,len(query)):
            probability_product = probability_product + self.get_bigram_logp(query[i - 1],
        probability_product = probability_product + math.log(self.get_unigram_logp(query[0]
        return probability_product
```

In [11]:

```
#lm.get_bigram_logp("quade", "quad")
#+lm.get_bigram_logp("quad", "xontroller")
#+ math.log(lm.get_unigram_logp("quade"),10)
```

In [12]:

```
#lm.get_bigram_logp("quad", "quad")
#+lm.get_bigram_logp("quad", "controller")
#+ math.log(lm.get_unigram_logp("quad"),10)
```

In [13]:

```
# Sanity Checks
lm = LanguageModel()

assert len(lm.unigram_counts) == 347071, 'Invalid num. unigrams: {}'.format(len(lm.unigram_
assert len(lm.bigram_counts) == 4497257, 'Invalid num. bigrams: {}'.format(len(lm.bigram_co
assert lm.total_num_tokens == 25498340, 'Invalid num. tokens: {}'.format(lm.total_num_token

# Test a reasonable query with and without typos (you should try your own)!
query_wo_typo = "stanford university"
query_w_typo = "stanfrod universit"

p_wo_typo = math.exp(lm.get_query_logp(query_wo_typo))                    # WHY exp?
p_w_typo = math.exp(lm.get_query_logp(query_w_typo))
print('P("{}") == {}'.format(query_wo_typo, p_wo_typo))
print('P("{}") == {}'.format(query_w_typo, p_w_typo))
if p_wo_typo <= p_w_typo:
    print('\nAre you sure "{}" should be assigned higher probability than "{}"?'
          .format(query_w_typo, query_wo_typo))
print('All tests passed!')
```

```
P("stanford university") == 0.08400910983345951
P("stanfrod universit") == 8.478900200669143e-07
All tests passed!
```

2. Edit Probability Model

The edit probability model attempts to estimate $P(R \mid Q)$. That is, for a fixed candidate query Q, the edit probability model estimates the probability that a (possibly corrupt) raw query R was submitted. The corrector quantifies the distance between the candidate query Q and the actual input R using the Damerau-Levenshtein distance (https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance). In Damerau-Levenshtein distance, the possible edits are **insertion**, **deletion**, **substitution**, and **transposition**, each involving single characters as operands.

In [14]:

```
class BaseEditProbabilityModel:
    def get_edit_logp(self, edited, original):
        # Returns the log probability of editing the 'original' to arrive at the 'edited'
        # where 'original'= true query and 'edited' = the raw query

        raise NotImplementedError  # Force subclass to implement this method
```

a. Uniform-Cost Edit Model

The *uniform-cost edit model* simplifies the computation of the edit probability by assuming that every individual edit in the Damerau-Levenshtein distance has the same probability. The uniform edit probability cost is obtained by a trial and error method.

The edit probability model is used to rank candidates for query corrections. The candidate generator (described in the next section) will make one edit at a time, and it will call the edit probability model each time it makes a single edit to a term, summing log-probabilities for multi-edit changes. Therefore, all this class does is calculate the probability of `edited` given that it is **at most one edit from `original`.**

In [15]:

```
# %%tee submission/uniform_edit_probability_model.py

class UniformEditProbabilityModel(BaseEditProbabilityModel):
    def __init__(self, edit_prob=0.01):

        self.edit_prob = edit_prob

    def get_edit_logp(self, edited, original):
        #print("RAW: ", edited, "| Candidate: ", original)
        prob = 0.0
        if edited == original:
            prob = 1 - 0.01 # Fixed probablity
        else:
            prob = 0.01
        return math.log(prob, 10)
```

In [16]:

```
# Sanity Checks
EDIT_PROB = 0.01
epm = UniformEditProbabilityModel(edit_prob=EDIT_PROB)

# Test a basic edit
edited, original = 'stanfrod', 'stanford'
assert math.isclose(epm.get_edit_logp(edited, original), math.log(EDIT_PROB, 10))

# Test a non-edit
assert math.isclose(epm.get_edit_logp(original, original), math.log(1. - EDIT_PROB, 10))

print('All tests passed!')
```

```
All tests passed!
```

3. Candidate Generator

The candidate generator takes a raw query R submitted by the user, and generates candidates for the intended query Q. Since more than 97% of spelling errors are found within an edit distance of 2 from the user's intended query, we encourage you to consider possible query corrections that are within distance 2 of R. This is the approach taken by Peter Norvig in his essay on spelling correction (http://norvig.com/spell-correct.html). However, it is not tractable to use a pure "brute force" generator that produces all possible strings within distance 2 of R, because for any R of non-trivial length, the number of candidates would be enormous. Thus we would have to evaluate the language and edit probability models on a huge number of candidates.

a. Candidate Generator with Restricted Search Space

The naïve approach can be made tractable by aggressively narrowing down the search space while generating candidates. There are many valid approaches to efficient candidate generation, but here are a few basic ideas:

- Begin by looking at *each individual term* in the query string R, and consider all possible edits that are distance 1 from that term.

- Remember that you might consider hyphens and/or spaces as elements of your character set. This will allow you to consider some relatively common errors, like when a space is accidentally inserted in a word, or two terms in the query were mistakenly separated by a space when they should actually be joined.
- Each time you generate an edit to a term, make sure that the edited term appears in the dictionary. (Remember that we have assumed that all words in a valid candidate query will be found in our training corpus, as mentioned above in Section IV.1.2 above).
- If you have generated possible edits to multiple individual terms, take the Cartesian product over these terms to produce a complete candidate query that includes edits to multiple terms. (But remember that you probably shouldn't go beyond a total edit distance of 2 for the query overall).

Again, there are many possible extensions and variations on the strategies mentioned here. We encourage you to explore some different options, and then describe in your written report the strategies that you ultimately used, and how you optimized their performance. Note that **solutions that exhaustively generate and score all possible query candidates at edit distances 1 and 2 will run too slowly and will not receive full credit.**

In [17]:

```
class CandidateGenerator:
    # Alphabet to use for insertion and substitution
    alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
                'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
                '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                ' ', ',', '.', '-']

    def __init__(self, lm, epm):

        self.lm = lm
        self.epm = epm
        self.vocab = set(lm.unigram_counts.keys())

    def get_num_oov(self, query):
        # Returns the number of out-of-vocabulary (OOV) words in `query`.
        return sum(1 for w in query.strip().split()
                   if w not in self.lm.unigram_counts)

    def filter_and_yield(self, query, lp):
        if query.strip() and self.get_num_oov(query) == 0:
            yield query, lp

    def in_vocab(self, words):
        return set(word for word in words if word in self.vocab)

    def edit_distance_one(self, word):
        splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
        deletes    = [L + R[1:]               for L, R in splits if R]
        transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
        replaces   = [L + c + R[1:]           for L, R in splits if R for c in self.alphabe
        inserts    = [L + c + R               for L, R in splits for c in self.alphabet]

        return set(deletes + transposes + replaces + inserts)

    def edit_distance_two(self, word):
        return set((e2, e1) for e1 in self.edit_distance_one(word) for e2 in self.edit_dist

    def get_candidates(self, query):
        #Generates candidates to the query at edit_distance<=2

        terms = query.strip().split()                   # List of terms in the query

        distance_one = []                               # Stores one edit distance terms  [
        distance_two = []                               # Stores two edit distance terms  [

        pos = 0

        term_index = [[terms[i], i] for i in range(len(terms))]    # {[term, index_in_query

        for key, value in term_index:
            temp = self.edit_distance_one(key)

            distance_one.append([temp, value])
            distance_two.append([self.edit_distance_two(key), value])
            #distance_two.append([self.edit_distance_two(key).difference(temp), value])

#         print("DISTANCE_ONE: ", distance_one)
#         print("DISTANCE_TWO: ", distance_two)
```

```
        # OPTIMIZATION 2 : Remove terms not in vocab
        accepted1 = {}                                  # Stores accepted 1-edit distance te
        for edited, index in distance_one:
            for j in edited:
                if ((j in self.vocab) or (len(j.split())>1 and j.split()[0] in self.vocab a
                    try:
                        accepted1[index].add((j, self.epm.get_edit_logp(j, terms[index])))
                    except:
                        accepted1[index] = {(j, self.epm.get_edit_logp(j, terms[index])),}
            try:
                temp2 = accepted1[index]
            except:
                accepted1[index] = {(terms[index], self.epm.get_edit_logp(terms[index], ter

        #print("ACCEPTED_1: \n", accepted1)

        accepted2 = {}                                  # Stores accepted 2-edit distance ter
        for edited, index in distance_two:
            for j2, j1 in edited:
                if ((j2 in self.vocab) or (len(j2.split())==2 and j2.split()[0] in self.voc
                    try:
                        accepted2[index].add((j2, self.epm.get_edit_logp(j2, j1) + self.epm
                    except:
                        accepted2[index] = {(j2, self.epm.get_edit_logp(j2, j1) + self.epm.
            try:
                temp2 = accepted2[index]
            except:
                accepted2[index] = {(terms[index], self.epm.get_edit_logp(terms[index], ter

        #print("\nACCEPTED_2: \n", accepted2)


        # Generate Candidate Queries with one-edit and zero-edit distance replacements

        candidate_queries_1 = []                    # Final candidate query list of one-edi
        cq = []                                     # Intermediate list of candidate querie
        candidate = ["", 0]                         # Intermediate candidate
        for i in range(len(term_index)):
            i_word, i_index = term_index[i][0], term_index[i][1]

            for tempi in range(i_index):             # Query remains unedited upto i_index.
                candidate[0]=candidate[0].strip()
                candidate[0] += ' ' + terms[tempi]   # Since, candidate query = query[:i_in
                candidate[1] += self.epm.get_edit_logp(terms[tempi], terms[tempi])  # Compu

            '''
            # Candidate includes all words upto the i_index'th word as-is. i_index'th word
            '''

            for edited, edit_prob in accepted1[i_index]:
                if candidate[0]!="":                                        # IF to handle
                    cq.append([candidate[0].strip() + " " + edited, candidate[1]+edit_prob]
                else:
                    cq.append([edited, edit_prob])

            '''
            # cq holds all possible queries with correction on the i_index'th term (and upt
            # Next step: Generate all possible queries hereforth
            '''
```

```
            if candidate[0]!="":                                        # Include the c
                cq.append([candidate[0].strip() + " " + i_word, candidate[1] + self.epm.get
            else:
                cq.append([i_word, self.epm.get_edit_logp(i_word, i_word)])

            '''
            # Case 1: Edit Distance of Query = 1
            # => Append the rest of the query [i_index+1:] to complete the final candidate
            '''

            cq2 = []
            for interm, edit_prob in cq:
                cdt = interm
                cdt_ep = edit_prob
                for tempi in range(i_index+1, len(terms)):
                    cdt = cdt.strip()
                    cdt += " " + terms[tempi]
                    cdt_ep += self.epm.get_edit_logp(terms[tempi], terms[tempi])
                cq2.append([cdt, cdt_ep])
                #cq2.append([ind + " " + ' '.join(query_terms[i_index + 1:]), edit_prob])
            candidate_queries_1 += cq2                                  # Add complete

            '''
            # Case 2: Edit Distance of Query = 2. With 2 single edit replacements.
            # => Iterate through the remainder of the query ([i_index+1:]) and edit one oth
            '''

            j = i_index+1
            candidate = ["", 0]     # Candidate query-subpart

            while(j<len(terms)):

                j_word, j_index = term_index[j][0], term_index[j][1]   # Second edit and in
                cdt = ["", 0]

                for tempj in range(i_index+1, j_index):                # Query remains unea
                    cdt[0] = cdt[0].strip()
                    cdt[0] += " " + terms[tempj]
                    cdt[1] += self.epm.get_edit_logp(terms[tempj], terms[tempj])
                    # candidate = [' '.join(query_terms[i_index+1:j_index]), 0]         # A

                cq2 = []                                                # Final Candia
                for edited, edit_prob in accepted1[j_index]:
                    for interm_ind in range(len(cq)):                   # Append corre

                        if cdt[0]!="":
                            c = [cq[interm_ind][0].strip() + " " + cdt[0].strip() + " " + e

                            for tempj in range(j_index+1, len(terms)):  # Append the r
                                c[0] = c[0].strip()
                                c[0] += " " + terms[tempj]
                                c[1] += self.epm.get_edit_logp(terms[tempj], terms[tempj])

                            cq2.append(c)
                        else:
                            c = [cq[interm_ind][0].strip() + " "  + edited, cq[interm_ind][

                            for tempj in range(j_index+1, len(terms)):  # Append the restof
                                c[0] = c[0].strip()
                                c[0] += " " + terms[tempj]
                                c[1] += self.epm.get_edit_logp(terms[tempj], terms[tempj])
```

```
                        cq2.append(c)
                j+=1                                                    # Next first e
                candidate_queries_1 += cq2                              # Add complete

            cq = []                                                     # Re-initializ

        '''
        print("\n--------------------SINGLE EDIT DISTANCES-------------------------\n")
        print(candidate_queries_1)
        print("\n---------------------                    -------------------------\n")
        '''

        # Generate Candidate Queries with a Single two-edit replacement

        pos = 0
        candidate_queries_2 = []
        candidate = ["", 0]
        for term, value in term_index:

            for i, edit_prob in accepted2[pos]:
                c[0] = candidate[0] + i
                c[1] = candidate[1] + edit_prob

                for tempi in range(pos+1, len(terms)):
                    c[0] = c[0].strip()
                    c[0] += " " + terms[tempi]
                    c[1] += self.epm.get_edit_logp(terms[tempi], terms[tempi])

                candidate_queries_2.append([c[0], c[1]])

            candidate[0] += term + " "                                  # Exclude correct
            candidate[1] += self.epm.get_edit_logp(term, term)
            pos += 1

        '''
        print("\n--------------------TWO EDIT DISTANCES-------------------------\n")
        print(candidate_queries_2)
        print("\n---------------------                  -------------------------\n")
        '''

        candidates = candidate_queries_1 + candidate_queries_2
        res= []
        for edited_query, log_edit_prob in candidates:
            res.append([edited_query.strip(), log_edit_prob])

        #print("\n\n\n",res)
        return res


model = CandidateGenerator(LanguageModel(), UniformEditProbabilityModel(BaseEditProbability
#model.get_candidates("did you go to stranford on unversit at stranforde")
#model.get_candidates('stranford unviersity')
#model.get_candidates('stnaford')
#model.get_candidates('quade quad xontroller')
```

In [18]:

```
# Sanity Checks
cg = CandidateGenerator(lm, epm)
query = 'stanford university'
num_candidates = 0
did_generate_original = False
for candidate, candidate_logp in cg.get_candidates(query):
    num_candidates += 1
    if candidate == query:
        did_generate_original = True

    assert cg.get_num_oov(query) == 0, \
        "You should not generate queries with out-of-vocab terms ('{}' has OOV terms)".form

assert 1e2 <= num_candidates <= 1e4, \
    "You should generate between 100 and 10,000 terms (generated {})".format(num_candidates

assert did_generate_original, "You should generate the original query ({})".format(query)

print('All tests passed!')
```

```
All tests passed!
```

4. Candidate Scorer

The candidate scorer's job is to find the most likely query Q given the raw query R. It does this by combining the language model for $P(Q)$, the edit probability model for $P(R \mid Q)$, and the candidate generator (to get candidates for Q). Formally, given raw query R, the candidate scorer outputs

$$Q^* = \arg\max_{Q_i} P(Q_i \mid R) = \arg\max_{Q_i} P(R \mid Q_i)P(Q_i),$$

where the max is taken over candidate queries $Q_i \in \{Q_1, \ldots, Q_n\}$ produced by the candidate generator given R.

a. Candidate Scorer with Weighting

When combining probabilities from the language model and the edit probability model, we can use a parameter to weight the two models differently:

$$P(Q \mid R) \propto P(R \mid Q)P(Q)^{\mu}.$$

Start out with $\mu = 1$, and then experiment later with different values of μ to see which one gives you the best spelling correction accuracy. Again, be careful not to overfit your development dataset.

In [19]:

```
# Returns the most likely query Q given a raw query R using the previously defined classes
class CandidateScorer:

    def __init__(self, lm, cg, mu=1.):

        self.lm = lm
        self.cg = cg
        self.mu = mu

    def get_score(self, query, log_edit_prob):
        # Uses the language model and `log_edit_prob` to compute the final score for a cand
        # Uses `mu` as weighting exponent for P(Q).

        p_q = self.lm.get_query_logp(query)
        try:
            return log_edit_prob+p_q
        except:
            return -100 # Why are we returning 100 here?

    def correct_spelling(self, r):
        # Calls the candidate generation function to obtain possible intended queries.

        candidates = self.cg.get_candidates(r)

        min_index = 0
        for i in range(len(candidates)):
            candidates[i].append(self.get_score(candidates[i][0],candidates[i][1]))     # C

        candidates.sort(key=lambda x:x[2], reverse=True)
        '''
        for i in range(len(candidates)):
            print(candidates[i][0], candidates[i][1], candidates[i][2])
        print("\n##############################################################################
        print(candidates[0][0], "\t", candidates[0][2])
        print("\n##############################################################################
        '''
        return candidates[0][0]
```

In [20]:

```
# Assumes LanguageModel lm was already built above
print('Building edit probability model...')
epm = UniformEditProbabilityModel()
print('Building candidate generator...')
cg = CandidateGenerator(lm, epm)
print('Building candidate scorer model...')
cs = CandidateScorer(lm, cg, mu=1.0)
print('Running spelling corrector...')

# Testing with our own queries
queries = [('quade quad xontroller', 'quad quad controller'),
           ('stranford unviersity', 'stanford university'),
           ('stanford unviersity', 'stanford university'),
           ('sanford university', 'stanford university'),
           ('stnaford university', 'stanford university')]

for query, expected in queries:
    corrected = cs.correct_spelling(query)
    print("\t'{}' corrected to '{}'".format(query, corrected))
    assert corrected == expected, "Expected '{}', got '{}'".format(expected, corrected)
print('All tests passed!')
```

```
Building edit probability model...
Building candidate generator...
Building candidate scorer model...
Running spelling corrector...
        'quade quad xontroller' corrected to 'quad quad controller'
        'stranford unviersity' corrected to 'stanford university'
        'stanford unviersity' corrected to 'stanford university'
        'sanford university' corrected to 'stanford university'
        'stnaford university' corrected to 'stanford university'
All tests passed!
```

b. Dev Set Evaluation (Uniform)

Now that we have constructed a basic spelling corrector, we will evaluate its performance on the held-out dev set. Recall that the dev set is stored across the files in `pa2-data/dev_set/` :

- `queries.txt` : One raw query R per line.
- `google.txt` : Google's corrected queries Q (one per line, same order as `queries.txt`).
- `gold.txt` : Ground-truth queries Q (again, one per line, same order).

Run the following cells to evaluate your spelling corrector on the dev set using your uniform edit probability model. We will also evaluate your model on a private test set after submission. For full credit, your spelling corrector with uniform edit probability model should achieve accuracy within 1% of the staff implementation *on the test set.* **We do not provide test set queries, but as a guideline for performance, the staff implementation gets 82.42% accuracy on the dev set.**

In [21]:

```
def dev_eval(candidate_scorer, verbose=False):
    """Evaluate `candidate_scorer` on the dev set."""
    query_num = 1
    yours_correct = 0
    google_correct = 0
    total_count = 0
    # Read originals, ground-truths, Google's predictions
    dev_dir = 'pa2-data/dev_set/'
    with tqdm(total=455, unit=' queries') as pbar, \
            open(os.path.join(dev_dir, 'queries.txt'), 'r') as query_fh, \
            open(os.path.join(dev_dir, 'gold.txt'), 'r') as gold_fh, \
            open(os.path.join(dev_dir, 'google.txt'), 'r') as google_fh:
        while True:
            # Read one line
            query = query_fh.readline().rstrip('\n')
#             print("Query = ", query)
            if not query:
                # Finished all queries
                break
            corrected = candidate_scorer.correct_spelling(query)
            corrected = ' '.join(corrected.split())  # Squash multiple spaces
            gold = gold_fh.readline().rstrip('\n')
            google = google_fh.readline().rstrip('\n')

            # Count whether correct
            if corrected == gold:
                yours_correct += 1
            if google == gold:
                google_correct += 1

            # Print running stats
            yours_accuracy = yours_correct / query_num * 100
            google_accuracy = google_correct / query_num * 100
            if verbose:
                print('QUERY {:03d}'.format(query_num))
                print('---------')
                print('(original):      {}'.format(query))
                print('(corrected):     {}'.format(corrected))
                print('(google):        {}'.format(google))
                print('(gold):          {}'.format(gold))
                print('Google accuracy: {}/{} ({:5.2f}%)\n'
                      .format(google_correct, query_num, google_accuracy))
                print('Your accuracy:   {}/{} ({:5.2f}%)'
                      .format(yours_correct, query_num, yours_accuracy))

            pbar.set_postfix(google='{:5.2f}%'.format(google_accuracy),
                             yours='{:5.2f}%'.format(yours_accuracy))
            pbar.update()
            query_num += 1
```

In [ ]:

```
# Set verbose=True for debugging output
dev_eval(cs, verbose=True)
```

```
  0%|          | 1/455 [00:03<25:42,  3.40s/ queries, google=100.00%, your
s=100.00%]

QUERY 001
---------
(original):      quade quad cache xontroller
(corrected):     quad quad cache controller
(google):        quad quad cache controller
(gold):          quad quad cache controller
Google accuracy: 1/1 (100.00%)

Your accuracy:   1/1 (100.00%)

  0%|          | 2/455 [00:04<13:21,  1.77s/ queries, google=50.00%, yours
=50.00%]

QUERY 002
---------
(original):      co2 in
(corrected):     co2 in
```

In [ ]:

```
class Edit:
    """Represents a single edit in Damerau-Levenshtein distance.
    We use this class to count occurrences of different edits in the training data.
    """
    INSERTION = 1
    DELETION = 2
    TRANSPOSITION = 3
    SUBSTITUTION = 4

    def __init__(self, edit_type, c1=None, c2=None):
        """
        Members:
            edit_type (int): One of Edit.{NO_EDIT, INSERTION, DELETION,
                TRANSPOSITION, SUBSTITUTION}.
            c1 (str): First (in original) char involved in the edit.
            c2 (str): Second (in original) char involved in the edit.
        """
        self.edit_type = edit_type
        self.c1 = c1
        self.c2 = c2


class EmpiricalEditProbabilityModel(BaseEditProbabilityModel):

    START_CHAR = ''      # Used to indicate start-of-query
    NO_EDIT_PROB = 0.1  # Hyperparameter for probability assigned to no-edit

    def __init__(self, training_set_path='pa2-data/training_set/edit1s.txt'):
        """Builds the necessary data structures to compute log-probabilities of
        distance-1 edits in constant time. In particular, counts the unigrams
        (single characters), bigrams (of 2 characters), alphabet size, and
        edit count for insertions, deletions, substitutions, and transpositions.

        Hint: Use the `Edit` class above. It may be easier to write the `get_edit`
        function first, since you should call that function here.

        Note: We suggest using tqdm with the size of the training set (819722) to track
        the initializers progress when parsing the training set file.

        Args:
            training_set_path (str): Path to training set of empirical error data.
        """
        '''
        # Your code needs to initialize all four of these data structures
        self.unigram_counts = Counter()  # Maps chars c1 -> count(c1)
        self.bigram_counts = Counter()   # Maps tuples (c1, c2) -> count((c1, c2))
        self.alphabet_size = 0           # Counts all possible characters
        '''

        self.unigram_counts = {}
        self.bigram_counts = {}
        self.alphabet_size = 0
        self.total_count = 0
        corpus_dir = 'pa2-data/corpus'
        for i in range(10):
            file = corpus_dir + '/' + str(i) + '.txt'
            with open(file, 'r') as fp:
                doc = fp.read()
```

```
                doc = list(doc)
                for char_id in range(len(doc)):
                    self.total_count+=1
                    try:
                        self.unigram_counts[doc[char_id]]+=1
                    except:
                        self.unigram_counts[doc[char_id]]=1
                    try:
                        self.bigram_counts[doc[char_id] + doc[char_id+1]]+=1
                    except:
                        if(char_id!=len(doc)-1):
                            self.bigram_counts[doc[char_id] + doc[char_id+1]]=1
        self.alphabet_size = len(self.unigram_counts.keys())

        # Maps edit-types -> dict mapping tuples (c1, c2) -> count(edit[c1, c2])
        # Example usage:
        #   > e = Edit(Edit.SUBSTITUTION, 'a', 'b')
        #   > edit_count = self.edit_counts[e.edit_type][(e.c1, e.c2)]
        self.edit_counts = {edit_type: Counter()
                            for edit_type in (Edit.INSERTION, Edit.DELETION,
                                              Edit.TRANSPOSITION, Edit.SUBSTITUTION)}

        with open(training_set_path, 'r') as training_set:
            for example in tqdm(training_set, total=819722):
                edited, original = example.strip().split('\t')
                e = self.get_edit(edited, original)
                if(e!=None):
                    try:
                        self.edit_counts[e.edit_type][(e.c1, e.c2)]+=1
                    except:
                        self.edit_counts[e.edit_type][(e.c1, e.c2)]=1


    def get_edit(self, edited, original):
        """Gets an `Edit` object describing the type of edit performed on `original`
        to produce `edited`.

        Note: Only edits with an edit distance of at most 1 are valid inputs.

        Args:
            edited (str): Raw query, which contains exactly one edit from `original`.
            original (str): True query. Want to find the edit which turns this into `edited

        Returns:
            edit (Edit): `Edit` object representing the edit to apply to `original` to get
                If `edited == original`, returns None.
        """
        ### Begin your code
        if edited == original:
            return None
        else:
            i = 0
            while(i<len(edited) and i<len(original) and original[i]==edited[i]):
                i+=1

            if(len(edited)-len(original)>=1):                                    # INSERTION
                if(i!=0):
                    c1 = original[i-1]
                else:
                    c1 = self.START_CHAR                        # What to do?
                c2 = edited[i]
```

```
                edit_type = 1

            elif(len(edited)-len(original)<=-1):                                    # DELETION
                c2 = original[i]
                if(i!=0):
                    c1 = original[i-1]
                else:
                    c1 = self.START_CHAR                        # What to do?
                edit_type = 2

            else:                                                                   # TRANSPOSI
                try:
                    c1 = original[i]                                                    # Fir
                    c3 = edited[i]
                    j = i
                    i+=1
                    while(i<len(edited) and original[i]==edited[i]):
                        i+=1

                    if(i<len(edited)): # Second Mismatch Found = TRANSPOSITION
                        c2 = original[i]
                        edit_type = 3

                    else:                                                               #
                        c1 = c3
                        c2 = original[j]
                        edit_type = 4
                except:
                    print(">>>>", edited, " | ", original)

            #print("{", original, "}-->{", edited, "}\t", edit_type, "\t|", c1, "\t|", c2,
            return Edit(edit_type, c1, c2)



    def get_edit_logp(self, edited, original):
        """Gets the log-probability of editing `original` to arrive at `edited`.
        The `original` and `edited` arguments are both single terms that are at
        most one edit apart.

        Note: The order of the arguments is chosen so that it reads like an
        assignment expression:
            > edited := EDIT_FUNCTION(original)
        or, alternatively, you can think of it as a (unnormalized) conditional probability:
            > log P(edited | original)

# WHICH IS THE RAW QUERY? -> Edited (most likely)
# WHICH IS THE CANDIDATE QUERY? -> Original (most likely)

        Args:
            edited (str): Edited term.
            original (str): Original term.

        Returns:
            logp (float): Log-probability of `edited` given `original`
                under this `EditProbabilityModel`.
        """

# w = correct word / original word
# x = misspelt word
```

```
# In insertion
# P(x|w) = ins[wi-1, xi] / count[wi-1]

# In deletion
# P(x|w) = del[wi-1, wi] / count[wi-1 wi]

        edit = self.get_edit(edited,original)
        denom = self.total_count

        if(edit!=None):

            try:
                count_edit = self.edit_counts[edit.edit_type][(edit.c1, edit.c2)]
            except:
                count_edit = 0

            if(edit.edit_type==1):                          # Insertion: c1 = '' or origina
                try:
                    denom = self.unigram_counts[edit.c1]
                except:
                    pass
            elif(edit.edit_type==2):                        # Deletion: c1 = '' or edited[i-
                try:
                    denom = self.bigram_counts[edit.c1+edit.c2]
                except:
                    pass
            elif(edit.edit_type==3):                        # Transposition: c1 = edited[i],
                try:
                    denom = self.bigram_counts[edit.c1+edit.c2]
                except:
                    pass
            else:                                           # Substitution: c1 = original[i]
                try:
                    denom = self.unigram_counts[edit.c2]
                except:
                    pass

            edit_prob = (count_edit+1)/(denom+self.alphabet_size)         # If denom 0, pro
#             print("Edited: {}\tOriginal: {}\tProb:{}\n".format(edited, original, edit_pro
        else:
            edit_prob = self.NO_EDIT_PROB
        return math.log(edit_prob, 10)
```

In [ ]:

```
print('Building the language model...')
lm = LanguageModel()
print('Building edit probability model...')
epm = EmpiricalEditProbabilityModel()
print('Building candidate generator...')
cg = CandidateGenerator(lm, epm)
print('Building candidate scorer model...')
cs = CandidateScorer(lm, cg, mu=1.0)
print('Running spelling corrector...')

# Testing with our own queries
#queries = [('stranford unviersity', 'stanford university')]
queries = [('quade quad xontroller', 'quad quad controller'),
           ('stranford unviersity', 'stanford university'),
           ('stanford unviersity', 'stanford university'),
           ('sanford university', 'stanford university'),
           ('stnaford university', 'stanford university')]

for query, expected in queries:
    corrected = cs.correct_spelling(query)
    print("\t'{}' corrected to '{}'".format(query, corrected))
    assert corrected == expected, "Expected '{}', got '{}'".format(expected, corrected)
print('All tests passed!')
```

In [ ]:

```
# Set verbose=True for debugging output
dev_eval(cs, verbose=True)
```

Results and Conclusions

As elaborated above, we have primarily implemented 2 probabilistic models: (a) Uniform Cost Model (b) Empirical Edit Cost Model The Uniform Cost Model - based on Bayes Theorem (which gives it a strong theoretical foundation) - gives us an accuracy of 87.25%, as opposed to Google's state-of-the-art model, which performs with an accuracy of 83.08%. The Empirical Edit Cost Model also has a higher accuracy than Google, of 87.25%, same as that of the Uniform Cost Model. When we compare these results with those of Stanford however, we see that while the accuracy obtained by us for the Uniform Cost Method beats Stanford's accuracy of 82.42%, their implementation of the Empirical Edit Cost Method results in an accuracy of 87.91%. The Uniform Cost Model naively assigns an equal cost to all edits. Practically, however, performing some edits is more expensive than others. Consequently, we assigned a different cost to each edit - insertion, deletion, transposition, and substitution, based on the frequency of occurrence as gleaned from a training dataset of spelling errors. Our initial hypothesis was based on the varying practical costs. The Uniform Edit Cost Model performed the same as the Empirical Cost Model, while the latter shoul. This is likely due to the distinction made with the type of operations where every operation uses a different formula to calculate the value of P(R Q), which could hinder the accuracy - even though the Empirical model is theoretically stronger than the naive Uniform model.