IRE Assignment 1

Snehal Kumar

2019101003

Solution

Given an index, to create a spelling correction algorithm for any language, we can use an ensemble of algorithms to get a fairly robust result.

1. Tokenization

Given a string, we tokenize the string in an attempt to get maximum similar or matching results for each token to get the final result after the tokens are passed for processing through the remaining algorithms to maximize accuracy.

Pseudo code to get the best 4 corrections for a given query string:

```
vector<string> best_corrections(string query, int top = 4)
    // tokenize the string into words
    vector<string> tokens = tokenize(query)
    map<string,vector<string>> matches;
    for token in tokens:
        // best matches to token word in the vocab
        matches[token] = best_matches(token)
    // set of best corrections base on matches map
    return combination_set(matches, top)
```

2. Edit Distance

This gives us the minimum number of edits required to change one token to another and more importantly, a metric for measuring the difference between two strings and can thus be used for getting the closest matches to the tokens syntactically.

For more efficient approach corresponding to our requirement, we can modify the costs for insertions and substitutions owing to the frequency of users accidentally typing extra words and the distance between the characters on the qwerty keyboard.

Pseudo code given two token strings of length N and M respectively:

```
int edit_distance(string token1, string token2){
  int substitutionCost[26][26] ->  distance between characters on keyboard
  // d[i, j] -> edit distance between token1[1..i] and token2[1..j]
  d[i, j] := minimum(d[i-1, j] + 1,                   // deletion
                     d[i, j-1] + 2,                   // insertion
                     d[i-1, j-1] + substitutionCost[i, j])  // substitution
  return d[N, M];
```

Time Complexity: O(N*M)

3. n-gram

In order to find the best matches from the edit distances between the token and the vocabulary, we use n-gram analysis to get the best candidate words. The edit distance between these candidates are then computed to get efficient and accurate results whilst also saving on unnecessary computation for each word in the index.

We can convert the token into a set of n-grams and apply approximate matching to get our candidate words. Some of methods to get the most likely corrections includes using z-scores and modelling the probability of OOV by introducing <unk> token.

We can also apply n-gram overlap amongst the posting lists by using the Dice coefficient.

In our solution, we create a posting list of all bigrams (n=2) and calculate the Dice coefficient using the lists.

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|}$$

Given two posting lists A and B of bigrams for query token and correction word respectively. After n-gram analysis we get the candidate words for edit distance calculation. The following code iterates over the probable candidates and returns the best among them

Pseudo Code:

```
int Dice(int bgm_A, int bgm_B, int hits){
    return 2*hits/(bgm_A + bgm_B);
}

int count_bigrams(string word){
    // N - 1 bigrams for N length word
    return word.length() - 1;
}

set<pair<int,string>> n_gram_candidates(string query, int top = 100){
    string bigrams = query;
    // Common words
    map<string,int> hits;
    // top words that are similar with query
    set <pair<int,string>> candidates; // (dice_coeffient, word)
    for bigram in bigrams:
        for word in posting_list(bigram):
            old_Dice = Dice(count_bigrams(query), count_bigrams(word), hits[word]);
            hits[word]++;
            current_Dice = Dice(count_bigrams(query), count_bigrams(word), hits[word]);
            // update the dice if current word present in top set
            if {old_Dice, word} in candidates:
              candidates.update({current_Dice,word});
            // insert into candidates
            else if candidates.size() < top:
              candidates.insert({current_Dice, word});
            worst_word = candidates.begin();
            // update the candidates with word acc to current_dice value
            else if current_Dice > worst_word.first:
                candidates.remove(worst_word)
                candidates.insert({current_Dice, word})
  return candidates;
}
```

From the set of generated candidate words, we compute the edit distances and get the best matches:

```
vector<string> best_matches(string token, top=20){
    candidates = n_gram_candidates(token);
    // max heap of best corrections for token (edit dist,word)
    best_matches= priority_queue<pair<int,string>,vector<int,string>>;
    for word in candidates:
        dist = edit_distance(word, token);
        if best_matches.size() < top:
            best_matches.push({dist, word});
        // Update worst word in heap
        else if dist < best_matches.top().first:
            best_matches.pop();
            best_matches.push({dist, word});
```

```
    return best_matches
}
```

4. Context Sensitization

In order to get a better match, we also need to include the context of the words. This approach adds a semantic matching in addition to the syntactic match. We do this by including correlation among the words of the query string.

For a query of 20 words and top 100 best matches, there are a total of 100^{20} words to calculate. Hence, to save computation we store the frequency map of tokens to get the best correction of length {1 ... i } and repeat till query length.

Pseudo Code:

```
vector<string> combination_set(vector<string> token_groups, int top){
  // stores the most frequent queries of length idx from first idx tokens
    vector<string> query_word_set;
    for token in token_groups:
        frequency_map<string,int> new_word_set;
        for cand in token:
            for word in query_word_set:
                new_word_set.insert(word + cand);
        // closest to index in argument
        query_word_set = new_word_set.closest(top);
  return query_word_set;
}
```