IIT Madras

Assignment 1 - CS6370

Pranali Yawalkar - CS11B046
Rishitha Dega - CS11B035

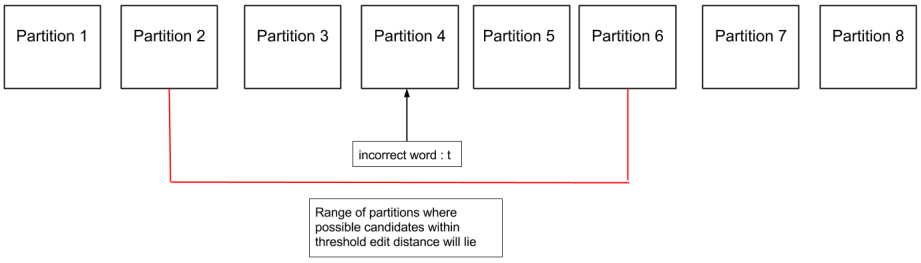
September 28, 2014

1 Candidate generation

- Aim : To find a set of correct words C from the dictionary containing words close to the incorrect word t.
- Closeness : small edit distance (with cost of insertion, deletion, substitution and transposition = 1)
- Choosing the threshold tolerable edit distance such that $\forall$ c $\in$ C, edit distance between c and t is lesser than the threshold
- Finding edit distance with all neighbours is a time consuming process . The partitions created in the dictionary come for rescue here. The threshold (decided by tweaking the parameters while training) is defined as :

$$Threshold = \begin{cases} 1: & \text{if } t.length() \leq 3 \\ 2: & \text{if } t.length() \leq 5 \\ 3: & \text{if } t.length() \leq 7 \\ 4: & \text{if } t.length() \leq 12 \\ 5: & \text{if } t.length() \leq 20 \\ 6: & otherwise \end{cases}$$

Now all the words which are edit distance less than the threshold will lie in the partitions corresponding to string sizes within threshold size difference from partition corresponding to the partition of the query word.



- Lower Bound for edit dist : Finding the vector code for the incorrect word t.

$$\boxed{LB(t,c) = NumBits(VectorCode(t) \oplus \text{VectorCode (c) })}$$

Where (a $\oplus$ b) is defined as the absolute difference between a and b. (Remember a and b are frequencies of occurrence of letter)

CLAIM : LB / 2 is the lower bound for the edit distance

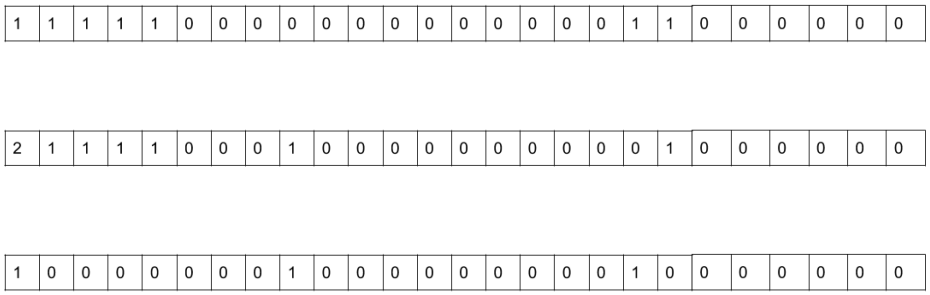
- Proof :
Consider the following cases :

1. Insertion : LB changes by atmost 1 and edit distance changes by atmost 1. EditDist (t , c) $\geq$ LB
2. Deletion : LB changes by atmost 1 and edit distance changes by atmost 1. EditDist (t , c) $\geq$ LB
3. Substitution : LB changes by atmost 2 and edit distance changes by 1. EditDist (t , c) $\geq$ EditDist (t , c) $\geq$ LB
4. Transposition : no change in LB and edit distance changes by 1. EditDist (t , c) $\geq$ LB

Thus the most loose lower bound for the editDist is:

$$\boxed{EditDist(t,c) \geq \frac{LB}{2}}$$

Example : Let VC (x) = VectorCode (x)
Let t = "abdcste" and c = "abdicate"



$\implies$ LB (t, c) = 3
$\implies$ EditDistance (t, c) $\geq$ 1

- Prune away all the words from the partitions with LB (t, c) $>$ 2 $\times$ Threshold.
- From the ones remaining, prune the ones with editDist (t, c) $>$ Threshold
- Summarising the candidate generation with pruning :
  1. Find threshold using the formula mentioned
  2. Find the potential partitions
  3. In those partitions, find words with LB(t, c) $\leq$ 2 $\times$ Threshold
  4. Among these words, those eligible for the final candidate set are the ones with EditDist(t, c) $\leq$ Threshold

1.1 Other models explored

We tested BK trees for candidate generation model. For the same set of candidates produced by BK trees and our model, the average time taken by BK trees is almost double of ours. Considering that editDistance is a metric distance, BK tree tries to prune away subtrees when the threshold is crossed. However, the distribution of words is such that all of them are at an p $<$ edit distance $<$ q from each other where p and q are integers and not very far from each other (p = 1 and q = 15). This small range of edit distance does not facilitate significant pruning and the search spans a significant portion (around 50%) of the tree in most cases. The overhead of building the tree and search rather adds to the time taken.

---

2 Word Spell Check

Generating a standalone model where erroneous words are given and the model suggests corrections.

2.1 Noisy channel model

The goal is to find the intended word given a word where the letters have been scrambled in some manner. For a given correct word c, and an incorrect word t, where c is a potential candidate present in our dictionary, the noisy channel is the matrix :

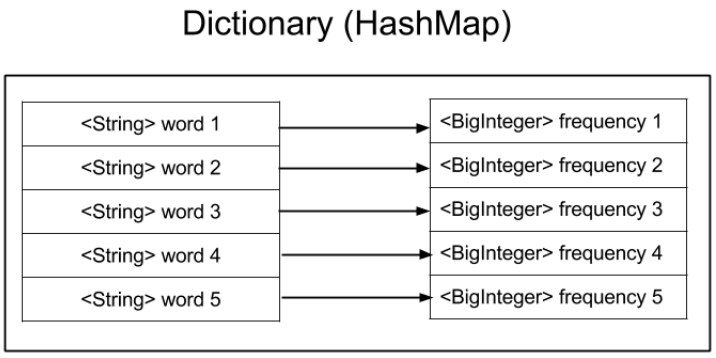
$$\boxed{\Gamma_{tc} = Pr(t|c)}$$

The kind of spelling mistakes being considered are :

1. Insertion of characters :
   Eg c="abdicate" and t="abdicaste"
2. Deletion of characters :
   Eg c="abdicate" and t="abdicte"
3. Substitution of characters :
   Eg c = "abdicate" and t = "abdicste"
4. Transposition of characters :
   Eg c = "abdicate" and t = "abdciate"
5. Pronunication : Eg c = "throughout" and t = "thruout"
   Here more than the structure of the correct and the incorrect word, the pronunciation being similar is valued more.

2.2 Offline Index structure on training data

- Dictionary :
  - Used for storing frequencies of the words of the dictionary under consideration. Size of the dictionary =

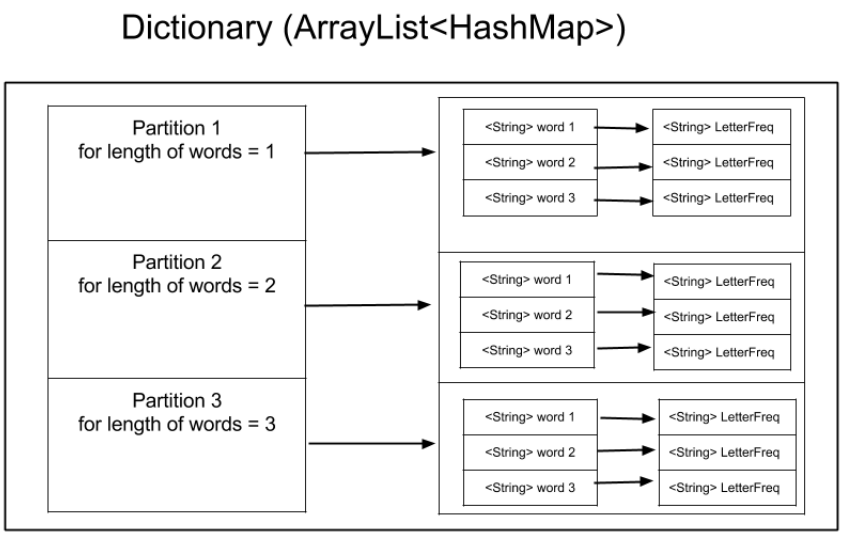


- Confusion Matrix : Building the confusion matrix for the errors of type : insertion, deletion, substitution and transposition of characters. The confusion matrix data is obtained from the Norvig sources. They are stored in matrix and used to evaluate the probabilities. Usage will be discussed later. Generally, these matrices

are computed with a bootstrapping procedure. Initially a uniform distribution is assumed over the possible confusions. Then the program is run over the training set to find corrections for the words that spell rejects. These corrections are used to update the confusion matrices, recursively.

- Pre-processing :
  - To build the dictionary : The Norvig dictionary (enable1.txt) contains a list of all valid words and the Norvig data of frequencies (count_1w.txt) contains frequencies of all words present in the corpus including proper nouns and dialectical words. An intersection of these two would give us our dictionary with valid words occurring in English dictionary and their frequencies in the corpus.
  - Partitions for candidate generation : For generating the potential candidates for an incorrect word, the dictionary is partitioned for faster access based on the length of the words of the dictionary. All words of length i go to partition i, and so on. The words in each partition are arranged such that the vector code of the word is stored against it in the partition hashmap.
    Constant sized vector code : Frequency of each letter in the alphabet occuring in the word.
    Eg vector code of "abdicate" =

2	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0



2.3 Algorithm

- Query : incorrect word t
- Offline Processing : Dictionary, confusion matrix for insertion, deletion, substitution and transposition
- Online Processing : Generated set of candidates
- Model used to find best candidates : Bayesian Model.
- The bayesian model form :

$$\boxed{\text{P (A | B)} = \frac{((P(B|A) \times P(A))}{P(B)}}$$

- Applying this to our model, we want to estimate P (c | t) which is the probability that the correct word is c given that the observed word is t.

$$\boxed{\text{P (c | t)} = \frac{((P(t|c) \times P(c))}{P(t)}}$$

- While comparing the candidates in C set, we want the one with highest P (c | t). We can eliminate considering P(t) as it will be a constant denominator for all the candidates.

$$\boxed{\text{P (c | t)} \propto (P(t|c) \times P(c)}$$

- P (c) is generalized in a simplistic way of frequency of occurence as follows :

$$\boxed{\text{P (c)} = \frac{(freq(c))}{\text{total number of words in corpus}}}$$

$$\boxed{\text{P (c)} \propto (freq(c))}$$

- As seen in the first section, P (t | c) is our noisy channel model and these probabilities are computed from the confusion matrices. The confusion matrices are built based on the commonly made typing or cognitive errors leading to insertion, deletion, substitution or transposition of characters. Characteristics of the matrices :

- confusion matrix for insertion :
  Matrix_ins [x][y] = number of times "x" was written as "xy" i.e insertion of y after x
- confusion matrix for deletion :
  Matrix_del [x][y] = number of times "xy" was written as "x" i.e deletion of y after x
- confusion matrix for substitution :
  Matrix_subs [x][y] = number of times "y" was written as "x" i.e substitution of y by x
- confusion matrix for transposition :
  Matrix_trans [x][y] = number of times "xy" was written as "yx"
- digrams[x][y] = number of times the words in the corpus contained the substring "xy"
- unigrams[x] = number of times x occured in all the words in the corpus

- Estimating P(t|c) [nosiy channel : probability of generating the wrong word given the correct one]

$$P(t|c) = \begin{cases} \frac{Matrix\_ins[c_{p-1}][t_p]}{unigrams[c_{p-1}]+1} : & \text{if } insertion \\ \frac{Matrix\_del[c_{p-1}][c_p]}{digrams[c_{p-1}][c_p]+1} : & \text{if } deletion \\ \frac{Matrix\_subs[t_p][c_p]}{unigrams[c_p]+1} : & \text{if } substitution \\ \frac{Matrix\_trans[c_{p-1}][c_p]}{digrams[c_{p-1}][c_p]+1} : & \text{if } transposition \end{cases}$$

- Handling multiple errors :
  We are considering that the errors committed in a word are independent of each other. Therefore :

$$\boxed{P(nerrors) = \prod_{i=1}^{i=n} \text{P } (\text{i}_{th} \text{ error })}$$

- To avoid division by zeros issues, all the denominators in the above formulae are added 1 extra

- Normalization : Because the order of the Probabilities obtained is often around 10^{-15} owing to the huge denominators (unigrams and digrams), we have come up with a smoothing technique :

1. Get the exponent value to be positive by multiplying the probabilities with 10^{50} (worst case).
2. For each candidate c $\in$ C, set P (c|t) to the $\log_{10}$ of probability P(c|t).

- Penalising the higher edit distanced candidates : dividing the probabilities by their corresponding edit distance
- Considering Phonetics : Like the example given above, "thruout" is more similar to "throughout" than "thrust" though the editDist("thruout", "throughout") is more than the editDist("thruout", "thrust"). We are using Metaphone external library to find the phonetic representation of t and c. The final probabilities are divided by the edit distance between these phonetics (added 0.5 to avoid divide by zero error).

  So the final formula for the probabilities considering multiple errors is :

$$\boxed{\text{P (c | t)} \propto (P(t|c) \times P(c)}$$

$$\boxed{\text{P (n errors)} = \prod_{i=1}^{i=n} \text{P (i}_{th} \text{ error)}}$$

$$\boxed{\text{P (n errors)} = P(nerrors) \times 10^{50}}$$

$$\boxed{\text{P (n errors)} = log_{10}(\text{P (n errors)})}$$

$$\boxed{\text{P (n errors)} = \frac{\text{P (n errors)}}{(editDist(t,c))^2 + 1}}$$

$$\boxed{\text{P (n errors)} = \frac{\text{P (n errors)}}{\text{editDist (phonetics(t), phonetics(c)) + 0.5}}}$$

2.4 Complexity

- Storage :
  1. Dictionary size : ($\|D\| \times$ (Size of String + Size of BigInteger))
  2. Confusion matrices : (26 $\times$ 26 $\times$ 4 $\times$ Size of int)
  3. Partitions : ($\|D\| \times$ (Size of String + Size of String))
  4. Diagrams : (26 $\times$ 26 $\times$ Size of BigInteger)
  5. Unigrams :(26 $\times$ Size of BigInteger)
- Time :
  1. Processing time :
     - Reading dictionary : $\mathcal{O}(\|D\|)$

- Reading confusion matrices : $\mathcal{O}$(Confusion Matrix size) = $\mathcal{O}(1)$

- Generating and storing Paritions : $\mathcal{O}(\|D\|\times$ Size of longest string in D) = $\mathcal{O}(\|D\|)$

2. Run time :
   - Vector Code calculation of t : $\mathcal{O}$(size of t)
   - Generating candidates : Assuming a very large threshold on editDistance tolerance, in the worst case we would visit all the strings in the dictionary. $\mathcal{O}(\|D\|\times 26)$ for xor computation.
     $\mathcal{O}$(size of Longest string in D $\times$ $\|D\|\times$ size of t) for edit distance computation . However, because the threshold is of the order fraction of length of t, the strings pruned away in the algorithm of candidate generation is significant to bring down the run time. Though the number of strings pruned away before editDistance calculation depends on the incorrect string and the distribution of the dictionary, fixing the threshold to be of the order of fraction of length of t prunes away the rest of the partitions as explained before (refer figure above).
   - Say that the candidate set is C, the computation for the final probabilities top-5 :
     * Edit distance computation to calculate insertion, deletion, substitution and transposition cost as well : P (t | c) :
       $\mathcal{O}$(size of Longest string in C $\times$ $\|C\|\times$ size of t)
     * P (c) : assuming Hashmap storage and search complexity = $\prime(1)$
     * Sorting the probabilities to get the top-5 (comparator sort) : $\mathcal{O}(\|C\|)$

2.5 Results

Test Set 1: http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines
Test Set 2: Test cases from moodle

	Number of words N	Accuracy (MRR)	Preprocessing time	Avg Candidate gen time	Avg total time
Test Case 1	4010	0.928	520ms	46.1ms	52.15ms
Test Case 2	15	0.74	479ms	49.6ms	76.73ms

2.6 References and acknowledgements

- http://en.wikipedia.org/wiki/Noisy_channel_model
- Candidate generation : "Generating Candidate Spelling Corrections" presented by Dustin Boswell
- A Spelling Correction Program Based on a Noisy Channel Model by Mark D. Kemighan, Kenneth W. Church, William A. Gale
- Norvig data for dictionary and confusion matrices : http://norvig.com/ngrams/

---

3 Phrase and Sentence Spell Check

Words present in the phrases and sentences need to be checked for spelling errors and corrected.

3.1 Models

3.1.1 n gram model

Taking into account the context of the phrase, the spell correction uses the n gram model to find the probability of the correct word given that context words around it.

- Probabilistic Language Model : Let a phrase W = w_1, w_2, w_3, ..w_m

$$\boxed{P(W) = P(w_1, w_2, w_3, ..w_m)}$$

- Applying chain rule :

$$\boxed{P(W) = P(w_1) \times P(w_2 \mid w_1) \times P(w_3 \mid w_1, w_2) \times ..P(w_m \mid w_1, w_2, w_3, ..w_{m-1})}$$

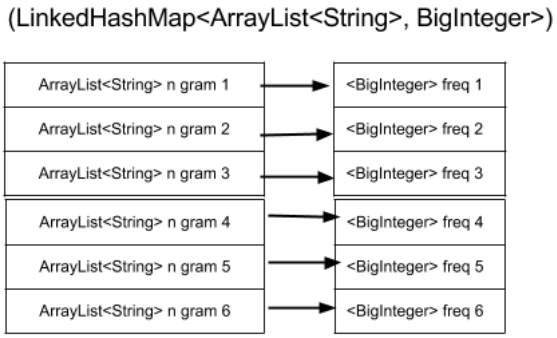
- Markov assumption for n gram model : the probability of a word depends on the n-1 words seen before that, and the n-1 words after the word which will get affected by its position at the i_{th} location.

3.1.2 POS Tagging and Exact Matching

- To account for context, a window of length ≤ 4 around the wrong word is considered.
- POS Tagging: This part of sentence (within the window) is tagged and searched for in the training data with POS Tags to obtain probability (say P_1).
- The part of the sentence within window is searched (Exact Match) in the training data set to get probability(say P_2)
- Finally, the sentence that maximizes the probability P is choosed, where P is the weighted maximum probability from the two methods (POS tagging and Exact Matching).

3.2 Offline Index structure on training data for Ngrams

Dictionary is the same as used in Word Spell Check part. Here the n grams (for n = 1, n = 2, n = 3, n = 4, n = 5) are taken from the frequent n-grams data based on the corpus of Contemporary English. Instances in the training data follow this format :
<frequency, $word_1$, $word_2$...$word_n$>
Example : < 35 whereas those who > as an instance of 3 gram data.
The n-gram frequencies are stored as follows :

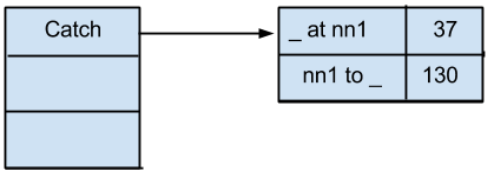


3.3 Offline Index structure on training data for POS and Exact Matching

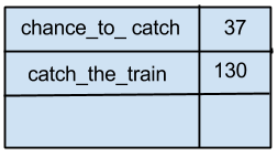
- Training data is taken from frequent n-grams (POS tagged) data based on corpus of Contemporary American English. Instances in training data follow this format: <frequency word1 word2 word3 POSTag1 POSTag2 POSTag3>.
  Example of an instance in 3-gram training data:

  - 37 catch the train vvi at nn1

- Two index structures are constructed, one for POS Tagged sentences and other for exact matching using ngrams.
- Index for POS tagged sentences: For each word in the corpus a list of sentences in which it has occurred is stored.
  - The following sentences training data are stored as shown in the figure. 1. 130 chance to catch nn1 to vvi
    2. 37 catch the train vvi at nn1



- Index for n-grams: N-grams are stored along with their frequency as shown in the figure.



3.4 Algorithm : Ngrams

- Using the Markov assumption, and given that the incorrect word occurs at position i in the phrase.
  W = w_1, w_2, w_3, ..w_i...w_m be th given phrase. We can formulate the n gram model with Markov assumption as follows : for now suppose that n = 3. This model is extended to n ranging from 1 to 5 inclusive.

$$\boxed{P(W) = P(w_1) \times \mathrm{P}(\mathrm{w}_2 \mid \mathrm{w}_1) \times \mathrm{P}(\mathrm{w}_i \mid \mathrm{w}_{i-2}, \mathrm{w}_{i-1}) \,..\mathrm{P}(\mathrm{w}_m \mid \mathrm{w}_{m-2}, \mathrm{w}_{m-1})}$$

  Considering the above model,

$$\boxed{P(W) = c\times \mathrm{P}(\mathrm{w}_i \mid \mathrm{w}_{i-2}, \mathrm{w}_{i-1}) \times \mathrm{P}(\mathrm{w}_{i+1} \mid \mathrm{w}_{i-1}, \mathrm{w}_i) \times \mathrm{P}(\mathrm{w}_{i+2} \mid \mathrm{w}_i, \mathrm{w}_{i+1})}$$

  where

$$\boxed{c = P(w_1) \times \mathrm{P}(\mathrm{w}_2 \mid \mathrm{w}_1) \times..\mathrm{P}(\mathrm{w}_{i-2} \mid \mathrm{w}_{i-4}, \mathrm{w}_{i-3}) \times \mathrm{P}(\mathrm{w}_{i+3} \mid \mathrm{w}_{i+1}, \mathrm{w}_{i+2})..\times \mathrm{P}(\mathrm{w}_m \mid \mathrm{w}_{m-2}, \mathrm{w}_{m-1})}$$

  $\Longrightarrow$

$$\boxed{P(W) \propto \mathrm{P}(\mathrm{w}_i \mid \mathrm{w}_{i-2}, \mathrm{w}_{i-1}) \times \mathrm{P}(\mathrm{w}_{i+1} \mid \mathrm{w}_{i-1}, \mathrm{w}_i) \times \mathrm{P}(\mathrm{w}_{i+2} \mid \mathrm{w}_i, \mathrm{w}_{i+1})}$$

- The 3-gram frequencies from the index structure are used to find the above probabilities for each candidate by replacing the position i with every c ∈ C.
- The candidates corresponding to the top-5 highest frequencies are returned as the top suggestions.

3.5 Algorithm : POS tagging + Exact matching

- Query: Sentence with incorrect words.
- For each incorrect word generate set of potential candidates it can be replaced with.
- For all combinations of these candidates, generate sentences. For example, query = There will be no pece in this houe.
  Assuming candidate set of pece=peace, piece, houe=house, the following sentences are generated
  - There will be no peace in this house
  - There will be no piece in this house
- Exact matching with N-grams and POS Tags are computed for the generated sentences and checked in the index structure to get their frequencies.
  - N-grams: no_peace_in, peace_in_this, no_piece_in, in_this_house etc.
  - POS Tags: uh _ ii, _ ii dd1 etc
- The probability of each sentence is calculated and then the sentence with maximum probability is chosen as the correct answer.

---

4 Implementation

For single errors in the phrase or sentence, we are using the POS + exact matching to rank the candidates. However, for multiple errors we are using both the models : N grams and POS + Exact match. Ngrams is used to get the ranking of the tuples of the candidates for the errors and POS + Exact match is used to get the individual rankings for each incorrect word (not in a tuple.)

4.1 Results

test1 : synthetic data set prepared with number of phrases/sentences with 1 error = , 2 errors = and 3 errors =
test2 : the test cases given on moodle

	# One error sentences	# two error sentences	Accuracy (MRR)	Preprocessing time	Avg total time
Test Case 1	34	16	0.69	52.781s	83.94ms
Test Case 2	26	3	0.862	50s	114.65ms

4.2 Possible Extensions

Word errors (cognitive errors). Consider the example of "World Piece Orgranisation". Though "Piece"is a correct word in dictionary, the right word in this context should be "Peace". We had considered a set of homophone candidates for a correct word which could fit in the context properly.

4.3 Other models tried

Because of the sparsity of the data available in the n-grams (esp the higher n grams), we tried to integrate Wordnet dictionary with the model to enhance the strengths of the ngrams. Eg, say the input phrase is "to local capitalsts" and C contains "capitalists" as one of the candidates. Now our n gram is "to local capitalists". Because of sparsity, it may happen that "to local capitalists" may not be present in the n gram data giving us a zero count (or very less, after smoothing) of the candidate. The wordnet integration model tries to enhance this by using the frequencies of those present in the ngram and are related to the words in wordnet. Firstly it finds the POS tag of the phrase. It then compares the POS tag of all the trigrams present (the POS tagged trigrams are available from the same source as the trigram) with that of our phrase. It finds the POS tag which is most frequent in the trigrams and having minimum difference in the POS tags of the words with respect to the our trigram ("to local capitalists"). The positions where this found tag and our tag differ are noted. Wordnet is queried to give the hypernyms, homonyms and synonyms of these words as proxy words. A set of all trigrams is generated by replacing the word with its related words found from the wordnet and their frequencies are found from the trigrams table. The trigram score of "to

local capitalists" is enhanced using linear combination of the scores of the trigrams generated using wordnet replacement for the differently tagged word. This model would not only take care of missing trigrams and their frequencies, but also enhance the score of the ones already existing by using the trigrams having wordnet similarity with it and existing in the trigrams.

Wordnet integration was successfully done and Stanford Wordnet API was used to get hypernyms, homonyms and synonmyms of a given word. The problem faced during implementation was when we could not get a uniform tagging of the phrases. We used the stanford POS tagger which not only takes a lot of time to tag a phrase (around 8 - 9 secs per phrase), but also produces tags that are different from the ones available with us in the n gram data. We, hence, could not implement this model with our existing n gram model because of lack of coherence in the tags.

4.4 References and Acknowledgements

- http://en.wikipedia.org/wiki/N-gram
- n gram lecture (Language Models) covered in class
- Norvig data for dictionary and confusion matrices : http://norvig.com/ngrams/
- Free frequent n-grams data(to be downloaded oine) based on Corpus of Contemporary American English. http://www.ngrams.info/
- Wordnet Integration : Incorporation of WordNet Features to N-Gram Features in a Language Modeler by Kathleen L. Go and Solomon L. See

---