Algorithms for speech and natural language processing (MVA 2017/2018)
Homework 3

Rémi Lespinet

1 Full pipeline

Here's a description of the whole processing pipeline. I have used the NLTK library for tokenization as well as pos tagging, and I've implemented the rest myself.

1. **Loading the input as UTF-8**
2. **Unescape HTML characters**

   This replaces characters like `Mario &amp; Luigi` to `Mario & Luigi`
3. **Multi-line reconstruction (Dynamic programming)**

   Because tweet are separated by newline characters but tweets can contain such character, there an ambiguity. I use a dynamic programming approach that tries to minimize an energy to find the best tweet separation (detailed in appendix A)
4. **Remove last words if the tweet is cut**

   Often in the corpus, tweets are cut because they are too long. When this happens, I remove the last letters up to the point where i find a space character.
5. **Remove emojis, links and @names (regex)**
6. **Process hashtags**

   Because sometimes the hashtags have a meaning, as in

   `@HeralddeParis: City of #Paris turns off the lights at the Eiffel Tower`

   If the hashtag word contains only characters and does not contain a case change, I keep it. This is a heuristic that works in the majority of cases
7. **Clean non ascii characters**
8. **Lower case everything**

   For the rest of the processing, I lower case all the text, This changes

   `So DON'T BLAME THIS ON ALL` → `So don't blame this on all`
9. **Apply normalization and abbreviation dictionary**

   I use a normalization dictionary and an abbreviation dictionary constructed from this website. This takes care of the one to many dependency and correct mistakes such as

   ```
   ... plans of going to Paris omg  →  ... plans of going to Paris oh my god
   I dont know why people ...       →  I don't know why people ...
   ```
10. **Process english contractions**

    I use regex to transform constractions such as

    `... to honor those that've been lost` → `... to honor those that have been lost`

    I do not process *'s* as it can mean different things and it would require the context.

11. **Generate tokenization and post tagging (use NLTK Library)**
12. **Load contex2vec pretrained on ukwac**
13. **Cleanup the list of word used using NLTK dictionary**
14. **For each word in each tweet that is not a proper name**
    - Run context2vec on this word, we obtain a context affinity vector c of size 160000 (size of the dictionary).
    - Compute a list of candidate at edit distance less than 2 in the contex2vec dictionary using a precomputed tree structure (see appendix B)
    - For each candidate w, compute the Damerau-Levensthein distance d_w. The formal similarity s is computed as $s_w = f(d)$. I've implemented the Damerau-Levenshtein version where each substring is updated at most one because I think that in our case, the full version slows down the computation and doesn't really improve.
    - Calculate the likelihood using the context and the formal similarity as $l_w = s_w \cdot c_w$ for each candidate w
    - Take the candidate word with maximum l_w , if it's likelihood is below a threshold, keep the original word, otherwise use it instead.

2 Successful examples

Here's are some successful tweet outputs

```
I : Tbh I'm jealous my neighbor went to Paris with his gf!
O : to be honest i am jealous my neighbor went to Paris with his girlfriend !
```

```
I : @RT_com: #Paris txi drivers turned off their meters, took people home for
    free - reports https://t.co/UUjfMTCXsM https://t.co/1TicKMbNJy
O : paris taxi drivers turned off their meters , took people home for free - reports
```

Here's an example that I've constructed (note that the were is corrected to where):

```
I : idk were it'll be next week, i havent checkd atm
O : i do not know where it will be next week, i have not checked at the moment
```

3 Failing examples

1. **Parsing hashtags**

   The heuristic that I've introduced fail, which causes things such as

   ```
   I : @MalikRiaz: My name is Malik Riaz. I am a Muslim. I condemn the #ParisAttack.
   O : my name is malik riaz . i am a muslim . i condemn the .
   ```

   To solve the hashtag better we should try to decompose the hashtag into different words and use the context to see if it is to be removed or not.

2. **Bigger abbreviation dictionary**

   The dictionary of abbreviation is probably too small, we should use a bigger one, for example the tweet

   ```
   I : @RajaOmarFarooq: Prophet Muhammad pbuh only taught us Peace, Love and Respect
   O : prophet muhammad pbuh only taught us peace , love and respect
   ```

3. **Parsing date and time**

   There's a lot of date and time in tweet, and the system currently does not handle this.

4. **Pronunciation related mistakes**

   There are also pronunciation related mistakes such as homophones which are not correctly handled when the words have different spelling such as

```
I : Turn left, then write and you'll find it !
O : turn left , then write and you will find it !
```

We could use a homophone dictionary here and use contex2vec independently of the edit distance.

5. **Too many mistakes in the input**

   When there is too much mistakes, this becomes really hard because context2vec doesn't know the majority of the word in the context, which is why normalization and abbreviation dictionaries are very important. For example with the sentence in the previous example

   ```
   I : do u kno were i cld fnid it ?
   ```

   the system succeed to find the correct sentence :

   ```
   O : do you know where i could find it ?
   ```

   This is possible because *u* and *cld* are in the dictionary, if I replace *cld* by *cuold* we obtain

   ```
   I : do u kno were i cuold fnid it ?
   O : do you know where i cuold fnid it ?
   ```

4 Conclusion

There's a huge room for improvement, tweets are really hard to normalize. They can really contain anything : some use different languages, the hashtags sometimes play both the role of a word, emojis or smileys can also be used as words, etc...

In the end, I was very surprised about the quality that we can obtain with this "simple" model. I think we can get better results by training context2vec on a clean corpus of tweets.

A Multi-line tweets

In the input corpus, each tweet is separated with a newline character, but tweets can also contain such characters, which makes the task of reconstructing tweets ambiguous. Even though this is not the main point of the homework, I've implemented a heuristic algorithm that does this processing. It is based on the following observations

- Tweets in the corpus cannot be longer than 140 characters
- The string `RT` in a newline almost surely marks the end of a character
- The unicode character ... marks the end of a tweet (that has been cut because too long)

The algorithm tries to find tweet delimitations by minimizing an energy by dynamic programming

$$S(w_i, \dots w_j) = \begin{cases} \left(\frac{1}{K}\text{len}(w_1, \dots w_n) - \sum_{k=i}^{j} \text{len}(w_i)\right)^2 & \text{if } \sum_{k=i}^{j} \text{len}(w_i) < 140 \\ \min_{i \leq k \leq j} S(w_i, w_k) + S(w_{k+1}, w_j) & \text{otherwise} \end{cases}$$

Where K is the minimum number of tweets that fit the input.

B Fast lookup tree structure

If we want to correct each word based on the formal similarity and context similarity, we want to compute, for each word w of the sentence the edit distance with the contex2vec dictionary. This is very costly as it requires to process all the dictionary at each step. To solve this problem, I noticed that a word s can not be at edit distance less than 2 to a word t if t contains 2 or more character that s doesn't have.

To exploit this, I encode each word of the contex2vec dictionary as a bag of letter, which gives me a vector (whose size is the number of elements in the alphabet $\mathcal{A}$)

I look at the letter $\hat{\alpha}$ that differentiate the most the words (given a letter $\alpha \in \mathcal{A}$, I separate the dictionary in two sets, the words that have this letter X_α, and the words that doesn't $\bar{X}_\alpha$, and I compute

$$\hat{\alpha} = \arg\min_{\alpha \in \mathcal{A}} |X_\alpha - \bar{X}_\alpha|$$

I can then apply the same operation recursively on the subdictionaries $X_{\hat{\alpha}}$ and $\bar{X}_{\hat{\alpha}}$ with the modified alphabet $\mathcal{A} - \{\hat{\alpha}\}$. By doing so I construct a tree that can be used to get candidate words in the dictionary very fast by crawling the tree. A simplified view of the crawling part is given below

```
def CRAWL_TREE(T, word, edit=2):

    if edit < 0 or T is a leaf:
        return T.data

    if T.letter in word:
        left  = CRAWL_TREE(T.left, word, edit-1)
        right = CRAWL_TREE(T.right, word, edit)
    else:
        left  = CRAWL_TREE(T.left, word, edit)
        right = CRAWL_TREE(T.right, word, edit-1)

    return concatenate(left, right)
```

Using this structure it is guaranteed to retrieve the words at edit distance 2 or less, but of course the list returned also contains words at edit distance more than 2 in the general case. We could improve this structure by considering repetition of letters and splitting better the dictionary (right now the algorithm is greedy), but in practice the number of words retrieved is in the order of 10^2 while the number of words in the dictionary is in the order of 10^5