haptik

NOV 3, 2018 6:44:00 PM / CHIRAG JAIN

EXTRACT SPELLING MISTAKES USING FASTTEXT

TAG

MACHINE LEARNING

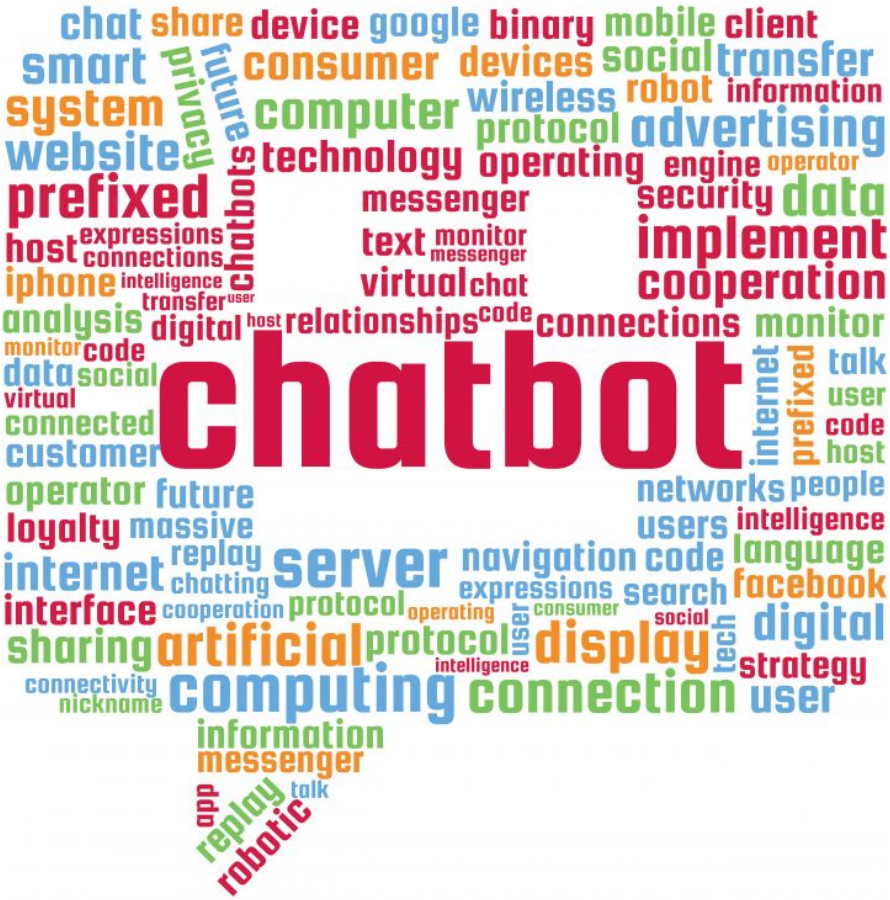
At Haptik, we deal with a lot of noisy conversational data. This conversational data is generated from our 100+ chatbots that are live across various platforms like mobile apps, web-SDKs etc.

Even though **12.5% of India** can speak English, **it ranks at 27th on the English Proficiency Index**. This means any kind of textual data obtained from Internet users like ours should not only expect multilingual data but also a lot of noise in the form of grammatical errors, internet slang, spelling mistakes, etc. Noise can create significant problems both during training and testing bots. During training time, a lot of noise can prevent machine learning algorithms from learning quickly, while at test time unseen noise can lead to wrong predictions.

Rare words like spelling mistakes often end up with not very useful vectors during the training process. This is because they occur so rarely that their randomly assigned vector hardly moves. Usually, rare words are marked as unknown words and discarded from the training process. If we can

correct spellings accurately, we can keep more of our training data as well as reduce the total number of unique words which impact the training time and memory.

In this post, we will try tackling **spelling errors**. Given that we have a moderately sized corpus of text data, we will extract a dictionary of spelling mistakes and their possible corrections from the data in an unsupervised way.



Although we can simply borrow a list of correctly spelled words from Internet dictionaries and use something like edit distance coupled to precompute all spelling errors, we wanted to experiment with a completely data-driven approach. Also, as mentioned earlier, our data also contains a fair amount of ‘Hinglish’ (romanized Hindi) messages. In India, romanizing or transliterating words from native language and changing the language back and forth in the same sentence is quite common, especially when it comes to messaging and social media. This is referred to as code mixing or code-switching.

This is mostly due to native languages being cumbersome to type on regular layout keyboards. Although such romanized words weakly follow some (unspoken) rules, there are no consistent guidelines to do so and can vary wildly. This means such sentences can have words which are not technically spelling mistakes but would be considered as such because they don't belong in any standard dictionary. We would also like to extract such words and their variants throughout the corpus.

For any natural language corpus, the vocabulary roughly follows a Zipfian distribution.

It states that *the frequency of any word is inversely proportional to its rank in the frequency table.* Put simply, a small subset of all words occur very frequently and the rest very rarely. Working on a small subset of our chatbots data with **1.8M lines and 11M total words, there are 182K unique words and about 80% of the unique words occur 5 times or less.**

1. A quick look at the vocabulary tells us that such words are either named entities (mostly proper nouns) or spelling mistakes.

2. Another interesting observation is that some spelling mistakes are more common than others and end up occurring even 100+ times across the corpus. An example is a word 'mony' (spelling error for 'money') that occurs 300+ times. Commonly used internet slangs and SMS shortened versions of words ('know' -> 'knw', 'you' -> 'u') also belong to this category.

3. There are also spelling mistakes like 'remainder', which even though is a legit English word, in our corpus it is often used in place of 'reminder'. Such contextual mistakes are a bit harder to correct with an unsupervised method and out of the scope of this post.

To tackle our problem we will use fastText. **FastText** is a way to obtain dense vector space representations for words. It modifies the Skip-gram algorithm from word2vec by including character level sub-word information.

So first for any word, say “hello” it would break it down into character n-grams. FastText asks for a min_n and max_n for character n-grams. So for example, min_n = 3, max_n = 4, “hello” would be broken down into

[“##h”, “#he”, “hel”, “ell”, “llo”, “lo#”, “o##”, “###h”, “##he”, “#hel”, “hell”, “ello”, “llo#”, “lo##”, “o###”]

where ‘#’ represents the padding character. (Note it doesn’t necessarily have to be ‘#’. FastText takes care of padding on its own)

Now, each of these character n-grams is assigned a vector instead of the main word itself. The vector for the main word itself is defined as the sum vector of all of its char n-grams. These vectors improve over the course of training via the skip-gram algorithm. Now, since we are considering char n-grams as input tokens, we can end up with a larger input space than our original vocabulary size. (In the example of min_n = 3, max_n = 4, above 27 ^ 3 + 27 ^ 4 = 551,124). To work around this problem, fastText uses hashing trick to hash these character n-grams to a fixed number of buckets. This way items hashed to the same bucket are assigned the same vector. **This keeps the memory bounded without affecting the performance severely.**

Coming back to the task, since fastText uses sub-word level information, any two words which have a similar set of character n-grams, can be expected to have their vectors nearby in the vector space. Since most spelling mistakes are just one or two characters wrong (edit distance <= 2) such words will have vectors close enough.

Getting started

Okay, so let’s first read the corpus and make the word to frequency dictionary. Since I took care of preprocessing the corpus beforehand, I am simply tokenizing on whitespace:

```
import io
import collections
import matplotlib.pyplot as plt
import nltk
import enchant
```

```
words = []
with io.open('corpus.txt', 'r',
encoding='utf-8') as f:
    for line in f:
        line = line.strip()
        words.extend(line.split())

vocab = collections.Counter(words)
vocab.most_common(10)
```

We get:

```
[('i', 174639),
('to', 127111),
('my', 84886),
('is', 69504),
('me', 67741),
('the', 63488),
('not', 51194),
('you', 50830),
('for', 47846),
('?', 45599)]
```

Inspecting the other end of the Counter

```
list(reversed(vocab.most_common
()[-10:]))
```

Unsurprisingly, we find words from other languages. In fact, about 3000 words at the bottom are in the non-Latin script. We will ignore these words later in the script:

```
[('酒店在haridwar',
1),
  ('谢谢', 1),
  ('谈', 1),
  ('看不懂', 1),
  ('的人##', 1),
  ('现在呢', 1),
  ('王建', 1),
  ('火大金一女', 1),
  ('李雙鈺', 1),
  ('拜拜', 1)]
```

Okay, now let's train a fastText model on the corpus:

```
$ fasttext skipgram -input corpus.txt -output model -minCount
```

```
1 -minn 3 -maxn 6 -lr 0.01 -dim 100 -ws 3 -epoch 10 -neg 20
```

I am keeping **minCount** 1 to try and learn a vector for all words, **ws** controls the window size hyperparameter in the skip-gram algorithm, 3 means for every word we will try to predict 3 words to its left and right in the given corpus. Changing **ws** won't have much dramatic effects on our task.

The main parameters are **minn** and **maxn** which control the size of character n-grams as explained above. We settled with 3 to 6, larger values would mean words would need to have longer substrings in common.

Okay, now we will load this model with Gensim and check some nearest neighbours:

```
from gensim.fasttext import FastText

model = FastText.load_fasttext_format('model')

print(model.wv.most_similar('recharge', topn=5))
print(model.wv.most_similar('reminder', topn=5))
print(model.wv.most_similar('thanks', topn=5))
```

This gives,

```
[('rechargecharge', 0.9973811507225037),
 ('rechargea', 0.9964320063591003),
 ('rechargedd', 0.9945225715637207),
 ('erecharge', 0.9935820698738098),
 ('rechargw', 0.9932199716567993)]

 [("reminder'⏰", 0.992865264415741),
 ('sk-reminder', 0.9927705526351929),
 ('myreminder', 0.992688775062561),
 ('reminderw', 0.9921447038650513),
 ('ofreminder', 0.992128312587738)]

 [('thanksd', 0.996020495891571),
 ('thanksll', 0.9954444169998169),
 ('thankseuy', 0.9953703880310059),
 ('thankss', 0.9946843385696411),
 ('thanksb', 0.9942326545715332)]
```

Okay, we seem to be getting spelling mistakes of our desired words very close to them (similarity scores are **> 0.99**).

Now, we will walk through our vocabulary, query the fastText model for each word's nearest neighbours and check for some conditions on each neighbour. If the neighbor passes these conditions, we will include the neighbour as a spelling mistake for the word. These conditions ensure that whatever we get at the end has less false positives:

```
word_to_mistakes = collections.defaultdict(list)
nonalphabetic = re.compile(r'[^a-zA-Z]')

for word, freq in vocab.items():
    if freq < 500 or len(word) <= 3 or
nonalphabetic.search(word) is not None:
        #  To keep this task simple, we will not try finding
        #  spelling mistakes for words that occur less than
500 times
        #  or have length less than equal to 3 characters
        #  or have anything other than English alphabets
        continue

    # Query the fasttext model for 50 closest neighbors to
the word
    similar_words = model.wv.most_similar(word, topn=50)
    for similar_word in results:
        if include_spell_mistake(word, similar_word,
similarity_score):
            word_to_mistakes[word].append(similar_word)
```

Here are the rules we use to include something like a spelling mistake for a word:

```
enchant_us = enchant.Dict('en_US')
spell_mistake_min_frequency = 5
fasttext_min_similarity = 0.96
def include_spell_mistake(word, similar_word, score):
    """
    Check if similar word passes some rules to be considered
a spelling mistake

    Rules:
       1. Similarity score should be greater than a threshold
       2. Length of the word with spelling error should be
greater than 3.
       3. spelling mistake must occur at least some N times
in the corpus
       4. Must not be a correct English word.
       5. First character of both correct spelling and wrong
spelling should be same.
       6. Has edit distance less than 2
    """
    edit_distance_threshold = 1 if len(word) <= 4 else 2
    return (score > fasttext_min_similarity
            and len(similar_word) > 3
            and vocab[similar_word] >=
spell_mistake_min_frequency
            and not enchant_us.check(similar_word)
            and word[0] == similar_word[0]
            and nltk.edit_distance(word, similar_word) <=
edit_distance_threshold)
```

Some rules are straightforward:

- Spelling mistake word vector must have high vector similarity with correct word's vector,
- Spelling mistake word must occur at least 5 times in our corpus,
- It must have more than three characters
- It should not be a legit English word (we use Enchant which has a convenient dictionary check function).

Other rules are based on observations like:

- The first character in a word with spelling mistake is usually correct so we can add a constraint that both correct and wrong spellings should have the same first character.

- Since most spelling errors lie within 2 edits of the correct word, we will ignore words that are more than 2 edits away.
- We can customize these rules according to the desired results. We can increase the edit distance threshold or ignore the first character same rule to be more lenient.

At this point, most of our work is done, let's check **word_to_mistakes**:

```
print(list(word_to_mistakes.items())[:10])

[
 ('want', ['wann', 'wanto', 'wanr', 'wany']),
 ('have', ['havea', 'havr']),
 ('this', ['thiss', 'thise']),
 ('please', ['pleasee', 'pleasr', 'pleasw', 'pleaseee',
'pleae', 'pleaae']),
 ('number', ['numbe', 'numbet', 'numbee', 'numbr']),
 ('call', ['calll']),
 ('will', ['willl', 'wiill']),
 ('account', ['aaccount', 'acccount', 'accouny', 'accoun',
'acount', 'accout', 'acoount']),
 ('match', ['matche', 'matchs', 'matchh', 'matcj', 'matcg',
'matc', 'matcha']), ('recharge', ['rechargr', 'recharg',
'rechage', 'recharege', 'recharje', 'recharhe', 'rechare'])
 ]
```

Nice! Now as a final step let's create an inverted index for fast lookup:

```
inverted_index = {}
for word, mistakes in
word_to_mistakes.items():
    for mistake in mistakes:
        if mistake != word:
            inverted_index[mistake] =
word
```

Now, this inverted_index dict can be used for quick lookup and correction.

One extra thing I did, that might not necessarily be needed, was merging transitive links in this index. What I noticed

was some word ‘A’ corrects to ‘B’ and in another entry ‘B’ corrects to ‘C’. In such cases, I chose to correct all ‘A’, ‘B’ and ‘C’ to whichever occurs the most in the corpus. I used the connected components algorithm to mark clusters and merge them.

That’s it. However, this method is not entirely accurate.

1. Very common proper nouns can still slip through the rules and end up being corrected when they shouldn’t be.
2. A manual inspection must still be done once to remove errors.
3. Another drawback is spelling mistakes that never occurred in the corpus will not have a correction in the index.

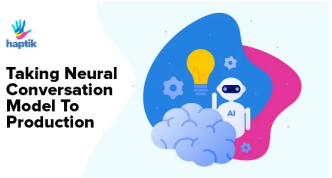
Nevertheless, this was a fun experiment.

If you already have a list of correct spellings with you (say top 10k most frequent words from a corpus like above), you can use something like Symspell which will generate all possible mistakes within 2 deletion operations for each word and then apply same deletion operations on spelling mistakes and use specialized index for fast lookup and correction.

Do let us know in comments about how you’re handling such use cases.

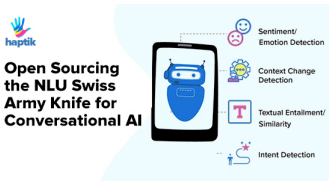
Posted by
Chirag Jain
on Nov 3, 2018 6:44:00 PM
Find me on:

RELATED BLOGS



Taking Neural Conversation Model to Production

Spello - The Spell Correction library



Open-sourcing the NLU 'Swiss army knife' for conversational AI

Comments

First Name*

Last Name

Email*

Website

Comment*

protected by reCAPTCHA
Privacy - Terms

Submit Comment

HAPTIK TECH BLOG

AI That Puts Your Customers First