
Department of Informatics, Systems and Communication

# An HMM-based approach for misspelling correction

## Probabilistic Modeles for Decision

Giorgia Adorni 806787

Elia Cereda 807539

Nassim Habbash 808292

**Academic Year 2018-2019**

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In recent decades, technology has had a strong impact on everyone's life. It plays an important role in the communication process, simplifying different activities both individuals and businesses.

Nowadays we have advanced communication technology tools available, such as smartphones, tablets and computers that have simplified the way humans communicate.
Companies can write an e-mail and deliver it to all their consumers in a few minutes. People can message their friends at every moment and share an interest with new friends from different countries.

This advancement in communication technology has made it necessary to equip our technological tools with a series of programs and software that control and correct automatically the misspelt words typed.

In this project, we propose and evaluate an automatic spelling correction algorithm, modelling the typing process as an *Hidden Markov Model* (HMM).

# Chapter 2

# Problem formulation

Given an observation sequence, typically a phrase, and the model parameters, we are interested in detect and correct errors estimating the optimal state sequence.

In our HMM, the hidden states represent the intended words and the observations are the typed words.

The initial state probabilities $\pi$ are given by the word frequencies and the state transitions $A_{ij}$, that is the probability of one word given its predecessor obtained from ??

The emission probabilities, $B_{ij}$, represent the probability of a typed word given the intended one, and depend on the confusion probabilities.

## 2.1 Purpose

## 2.2 Design choices

We have chosen the English language for different reasons. First of all, the great majority of material in literature deals with the problem in question in the English languag. Moreover it is a simple language, both from a grammatical and a lexical point of view: it lacks in certain symbols, like accents and apostrophes, and genres. Furthermore all punctuation and special character symbols were not considered, but only letters and sometimes numbers.

We considered that a typed word only depends on the previous one, being in the framework of Markov chains. If we know the probability of a word given its predecessor, the frequency of each word, and the probability to type word x when word y is intended, we have all the necessary ingredients to use Hidden Markov Models.

## 2.3 Software

We have developed the project in **Python**.
The interface is a native macOS application written in **Swift**.

# Chapter 3

# Dataset

## 3.1 Dataset acquisition

### 3.1.1 Error Model

This dataset was collected from the following resources [1] [2] [3] [4]:

- BIRKBECK: contains 36 133 misspellings of 6136 words, taken from the native-speaker section (British and American) of the Birkbeck spelling error corpus.
- HOLBROOK: contains 1791 misspellings of 1200 words, taken from the book "English for the Rejected" by David Holbrook (Cambridge University Press - 1964).
- ASPELL: contains 531 misspellings of 450 words, taken from one assembled by Atkinson for testing the GNU Aspell spellchecker.
- WIKIPEDIA: contains 2455 misspellings of 1922 words, taken from the misspellings made by Wikipedia editors.
- URBAN-DICTIONARY-VARIANTS: contains 716 variant spellings, taken from the text scraped from Urban Dictionary (in UK English).
- SPELL-SET: contains 670 typos.
- TWEET-TYPO: contains 39 172 typos, taken from Twitter.

[1] https://www.dcs.bbk.ac.uk/~ROGER/corpora.html
[2] https://www.kaggle.com/rtatman/spelling-variation-on-urban-dictionary
[3] https://www.kaggle.com/bittlingmayer/spelling
[4] http://luululu.com/tweet

All the datasets are joined and cleaned. After that, it contains 39 888 row, in each of which there is the typo and the correct word. The dataset obtained is divided into two corpora: 80% is used as a train set (31 808 row) and 20% is used as a test set (8080 row).

### 3.1.2 Language Model

The language model dataset is a concatenation of public domain book excerpts from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus.

This dataset contains about a million word. It is use to estimate the probability of a word by counting the number of times each word appears in this dataset.

### 3.1.3 Frequency Model

The words frequency dataset includes wordlists derived from the Google's ngram corpora. In particular we use the dataset `frequency-alpha-gcide.txt`, a smaller version of the original dataset, cleaned up and limited to only the top 65 538 words.

Each row of the corpus contains the ranking of the word, the word itself, the count of the occurrences of the word, a percentage of how often each word was being used and a cumulative percentage.

With the following dataset we found some problems due to the lack of proper names, city names, countries, brands etc.

### 3.1.4 Perturbated Dataset

In order to evaluate our algorithm on whole sentences, we create a new perturbed dataset starting from the dataset `big` described in the section 3.1.2.

The disturbance introduced presents an error partly dependent on the error model previously presented, with the difference that it is created starting from the typos belonging to the test dataset.

Three different texts have been generated, each of which has a percentage of errors in the text of 10-15- 20% respectively.

We implemented a perturbation algorithm, which for each line of our input file generates a new perturbed string.
The input text is perturbed accordingly the following steps:

1. Give us from the error model the value of $p$, the probability that a word has an edit, for each word of length $n$, the number of edits to be introduced $x$ is calculated according to the relation $x \sim \text{Bin}(n, p)$

2. FIXME
3. FIXME
4. The disturbance goes to alter the letters designated not in a random manner. In fact, we use three different probabilities to define whether a letter will be deleted from the index in question, if a new letter will be hung (FIXME: come), or if the current character will be replaced with one of the possible letters (coming from the error model).

Cases of elimination of a whole word are excluded, as these would heavily influence the evaluation metrics as they are inconsistent with our model.

```
def perturb():
        # Create a model for the test set
        hmm = HMM(1, max_edits=2, max_states=3)
        hmm.train(words_ds="../data/word_freq/frequency-alpha-
            gcide.txt",
        sentences_ds="../data/texts/big_clean.txt",
        typo_ds="../data/typo/new/test.csv")

        cleaned = open("../data/texts/big_clean.txt", "r")

        if not os.path.exists("../data/texts/perturbated/"):
        os.makedirs("../data/texts/perturbated/")

        perturbed = open("../data/texts/perturbated/big_perturbed
            .txt", "w")

        # probability that a word has an edit
        p = 0.10

        for line in cleaned:
                line_words = line.split()

                for i, word in enumerate(line_words):
                        n = len(word)
                        x = np.random.binomial(n, p)        # x ~
                            Bin(p, n)  number of errors to
                           introduce in the word

                        # choose two letter to change
                        indices = np.random.choice(n, x, replace=
                           False)
```

```
                indices = -np.sort(-indices)

                for j in range(x):
                        r = np.random.random()  #FIXME
                            now choose an edit randomly

                        def substitute(word):
                        l = list(word)
                        if not l[indices[j]] in hmm.
                            error_model["sub"]:
                                l[indices[j]] = random.
                                    choice(string.
                                    ascii_letters).lower()
                        else:
                                l[indices[j]] = np.random
                                    .choice(list(hmm.
                                    error_model["sub"][l[
                                    indices[j]]].keys()))
                        return "".join(l)

                        # insert a letter in a random
                            position (after idx)
                        if r < 0.33:
                                new_letter = random.
                                    choice(string.
                                    ascii_letters)
                                word = word[0:indices[j]]
                                    + new_letter + word[
                                    indices[j] + 1:]

                        # delete a letter
                        elif r > 0.66:
                                if len(word) == 1:
                                        # if the word is
                                            1 char, don't
                                            delete the
                                            word but
                                            substitute it
                                            with another
                                            one
                                        word = substitute
                                            (word)
                                else:
```

```
                                word = word[0:indices[j]]
                                    + word[indices[j] +
                                    1:]

                            # substitute a letter
                            else:
                                word = substitute(word)

                        line_words[i] = word

                line = " ".join(line_words)
                perturbed.write(line)

        perturbed.close()
        cleaned.close()
```

Listing 3.1: Text perturbation algorithm

# Chapter 4

# Models

We will consider two optimality criteria. The first one chooses the states that are individually most likely and maximizes the expected number of correct individual states. The second criterion estimates the most likely state sequence, or *trellis path*. The algorithm used to implement these criteria is the **Viterbi** algorithm.

## 4.1 Noisy channel model

This model is a sort of a local corrector. It consists of two components: a source model, corresponding to $P(\text{candidate})$, and a channel model, that is $P(\text{typo}|\text{candidate})$.

Given an input typo, for example *adventhre*, the first stage generates a set of candidate corrections for the word, for example *adventure*, *adventurer*, *adventured*... Each candidate is scored by $P(\text{candidate})P(\text{typo}|\text{candidate})$ and then normalised by the sum of the scores for all proposed candidates.

(How is estimated the prior? How are computed the conditional probabilities?)

We choose the most likely candidate, the one with the highest probability.

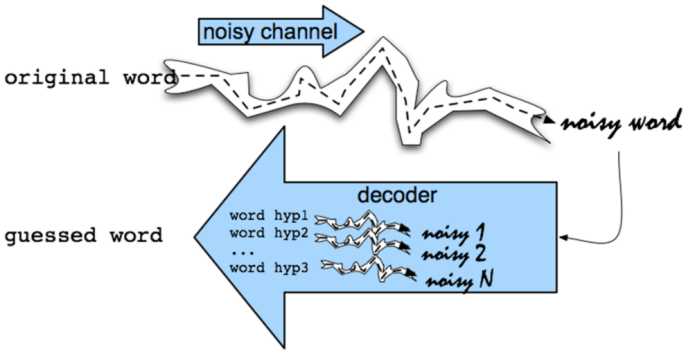


Figure 4.1: Noisy channel

When for a input typo we do not have any candidate ??

### 4.1.1 Common Errors

The following type of errors were considered:

- OMITTED DIGIT
- ADDITIONAL DIGIT
- SUBSTITUTED DIGIT: this is the probability to type a character $i$ when the character $j$ was intended ($P(i|j)$). This probability was determined experimentally.

## 4.2 Most likely state sequence

The Viterbi algorithm calculates the most probable sequence of hidden states, the words intended.

The initial probability of being in a state $i$, $\pi_i$, in our case the probability of intend a word $i$, and the transition probabilities $A_{ij}$, or the transition from the word $i$ to the next word $j$, are given. Since we have observed the output $y_1, y_2, \ldots, y_t$, that is the sentence written with typos, it is possible to computed the most likely state sequence $x_1, x_2, \ldots, x_t$, the sentence intended, starting from the following expression:

$$\begin{aligned} V_{1,t+1} &= P(x_1, \ldots, x_t, x_{t+1}, y_1, \ldots, y_t, y_{t+1}) = \\ &= \arg\max_{x_{1:t}} p(x_1, \ldots, x_t | y_1, \ldots, y_t) = \\ &= \alpha \cdot p(y_{t+1}|x_{t+1}) \cdot \max_{x_t} \Big( p(x_{t+1}|x_t) \max p(x_1, \ldots, x_t | y_1, \ldots, y_t) \Big) \end{aligned} \tag{4.1}$$

The initial state probabilities $\pi$ are actually the word frequencies (? We don't estimate it in a proper way), the state transition probabilities are given by the probability of a word given its predecessor, and the emission probabilities are the probabilities to type word $i$ when word $j$ was intended.

In our implementation, we construct the *trellis* choosing the locally/globally best state. As we can see in the picture below, we display an example of the trellis drawing only the best predecessor of each state.

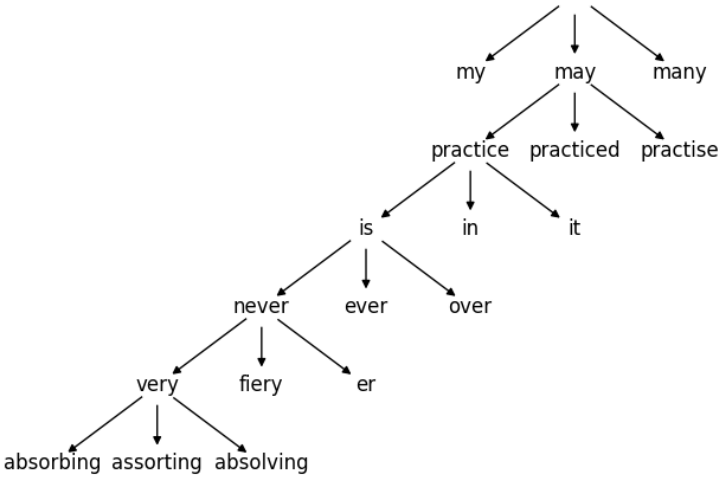


Figure 4.2: Trellis example

In this example, we observed the sentence *my practice iu never iery absorcing* and our algorithm return the most likely state sequence *may practice is never very absorbing*. In this case, the algorithm partially fails, because the intended sentence was *my practice is never very absorbing*.

We decide to implement the **Viterbi** based algorithm instead the Forward-Backward algorithm relying on the experiments carried out in the literature. The HMM-Based Error Correction Mechanism for Five-Key Chording Keyboards article [1] explains that the Forward-Backward algorithm estimates the most likely state for each observation, but the resulting state sequence may not be a valid succession of words in natural language (or a very unlikely word sequence) and produce inferior results.

## 4.3 Myers' Algorithm

## 4.4 Hidden Markov Model

# Chapter 5

# Experimental results

# Chapter 6

# Conclusions

## 6.1 Future works

# Bibliography

[1] Adrian Tarniceriu, Bixio Rimoldi, and Pierre Dillenbourg. "HMM-based error correction mechanism for five-key chording keyboards". In: *2015 International Symposium on Signals, Circuits and Systems (ISSCS)*. IEEE. 2015, pp. 1–4 (cit. on p. 14).

[2] Yanen Li, Huizhong Duan, and ChengXiang Zhai. "A generalized hidden markov model with discriminative training for query spelling correction". In: *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2012, pp. 611–620.

[3] Grzegorz Szymanski and Zygmunt Ciota. "Hidden Markov models suitable for text generation". In: *WSEAS International Conference on Signal, Speech and Image Processing (WSEAS ICOSSIP 2002)*, pp. 3081–3084.