Department of Informatics, Systems and Communication

An HMM-based Approach for Misspelling Correction

Probabilistic Models for Decisions

Giorgia Adorni
806787

Elia Cereda
807539

Nassim Habbash
808292

Academic Year 2018-2019

Contents

1 Introduction **4**

2 Problem Formulation **5**
2.1 Design Choices 5
2.2 Software 6

3 Dataset **7**
3.1 Sentences Dataset 7
3.2 Typos Dataset 7
3.3 Language Dataset 8
3.4 Perturbed Dataset 9

4 Models **10**
4.1 Noisy Channel Model 10
4.2 Hidden Markov Model 12
4.3 Most Likely State Sequence 14

5 Experiment and Results **16**
5.1 Evaluation Metrics 16
5.2 Experiments 18
5.2.1 Experiment 1 18
5.2.2 Experiment 2 20
5.2.3 Experiment 3 20

6 User Interface **23**

7 Conclusions **24**

References **26**

List of Figures

4.1 Diagram of the Noisy Channel Model 11
4.2 Illustration of Hidden Markov Model 13
4.3 Trellis example 14

List of Tables

5.1 Confusion matrix 17
5.2 Hmm model 18
5.3 Typos performance evaluation 19
5.4 Sentences performance evaluation 19
5.5 Hmm model 21
5.6 Typos performance evaluation 21
5.7 Sentences performance evaluation 22

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 for both individuals and businesses.

Nowadays we have advanced communication 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 a string of text, typically a phrase, we are interested in detecting and correcting misspelling errors in the typed phrase. This project aims to implement an Error Model, capable of offering corrected candidates to a single misspelled word, and a Hidden Markov Model, capable of finding the most likely sequence of candidates for each word in a sentence.

The two models work sequentially in a pipeline, but are not strictly dependant on each other, as the Error Model acts as a local corrector, and the HMM acts as a maximizer of the probability of the sequences of candidates.

2.1 Design Choices

We have chosen to implement this solution using the English language for various reasons. First of all, great majority of material in literature deals with this problem in the English language. 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 assume that a typed word only depends on the previous one for ease of implementation. As such we will implement a first order HMM. 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.

We also assume that, in our artificial perturbation of words, the number of errors in a word follows a binomial distribution depending on the length of the word.

2.2 Software

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

Chapter 3

Dataset

3.1 Sentences Dataset

We used two different datasets of English sentences to construct the transition probability matrix, following the procedure described in Section 4.2.
The first one is a concatenation of public domain book excerpts from Project Gutenberg[1], containing about a million words.

The second one has been extracted from the collection of the "Lord Of The Rings" books[2].

We then applied some preprocessing procedures to each corpus, in particular we divided them in lower-case sentences and removed special characters and punctuation, obtaining the `big_clean` and `lotr_clean` datasets.

3.2 Typos Dataset

The basic typos dataset was collected from the following resources:

- BIRKBECK[3] : contains 36 133 misspellings of 6136 words, taken from the native-speaker section (British and American) of the Birkbeck spelling error corpus.

- HOLBROOK[3]: contains 1791 misspellings of 1200 words, taken from the book "English for the Rejected" by David Holbrook (Cambridge University Press - 1964).

- ASPELL[3]: contains 531 misspellings of 450 words, taken from one assembled by Atkinson for testing the GNU Aspell spellchecker.

[1]`http://www.gutenberg.org/wiki/Main_Page`
[2]`https://www.kaggle.com/mokosan/lord-of-the-rings-character-data`
[3]`https://www.dcs.bbk.ac.uk/~ROGER/corpora.html`

- WIKIPEDIA[3]: contains 2455 misspellings of 1922 words, taken from the misspellings made by Wikipedia editors.
- URBAN-DICTIONARY-VARIANTS [4]: contains 716 variant spellings, taken from the text scraped from Urban Dictionary (in UK English).
- SPELL-SET[5]: contains 670 typos.
- TWEET-TYPO[6]: contains 39 172 typos, taken from Twitter.

All the datasets were cleaned and joined in a single one containing 79 677 rows, each with a typo and the corresponding correct word. This dataset was then divided into two corpora: 80% is used as a train set (63 679 rows) and 20% is used as a test set (15 998 rows).

To evaluate the performance of our model, we also created another dataset of typos starting from the
`lotr_clean` file. For each word contained in this corpus we generated five typos according to the algorithm that will be defined in Section 3.4. This dataset contains 62 759 rows, with the same structure described above. We then split it in train and test datasets, respectively containing 50 058 and 12 701 rows.

3.3 Language Dataset

A language model represents the frequency of words in a certain language. We used two different language model datasets.
The first one is a lists of most frequent words from Wiktionary and the British National Corpus. We use `frequency-alpha-gcide`, a smaller version derived from the original dataset Google's ngram corpora, that includes wordlists, cleaned up and limited to only the top 65 537 words.

We found some problems with this dataset, for example the lack of proper names, city names, countries, brands, etc. Moreover, most of the typical words of the language used in the sentence dataset were missing. For this reason, we decided to create a new language model `lotr_language_model`, based on the frequency of the 12 506 distinct words in the `lotr_clean` dataset.

Both of these datasets contain, for each word, the frequency it appears in the corresponding text corpus.

[4]`https://www.kaggle.com/rtatman/spelling-variation-on-urban-dictionary`
[5]`https://www.kaggle.com/bittlingmayer/spelling`
[6]`http://luululu.com/tweet`

3.4 Perturbed Dataset

In order to evaluate the performance of our algorithm for the Most Likely Sequence, we created new datasets of perturbed sentences, starting from the `big_clean` and `lotr_clean` datasets described in Section 3.1.

Estimates for the frequency of spelling errors in human-typed text vary greatly depending on the situation: from 1-2% when carefully retyping already printed text to 10-15% when writing web queries. For this reason, we generated four different texts from each dataset, introducing errors in respectively **FIXME** 10-15-20% of the words.

FIXME: The disturbance introduced presents an error dependent on the error model previously presented, with the difference that it is created starting from the typos belonging to the test datasets. **FIXME**:Analysis of spelling error data has shown that the majority of spelling errors consist of a single-letter change and so we often make the simplifying assumption that these candidates have an edit distance of 1 from the error model.

We implemented a perturbation algorithm, which for each line of our input file generates a new perturbed string, according to the following steps:

1. The probability that a word has an edit is computed by multiplying the value of p, coming from the error model, by the percentage of errors desired (10-15-20%).
2. For each word of length n, the number of edits to be introduced x is calculated according to the relation $x \sim \text{Bin}(n, p)$
3. The characters to be changed within each word are chosen randomly.
4. **FIXME**: The type of edit to be introduced within each word is chosen according to the probability of each type of error. The disturbance goes to alter the letters designated not in a random manner. In fact, we use four different probabilities to define whether a letter will be deleted from the index in question, if a new letter will be inserted after the actual character, or if the current character will be replaced with one of the possible letters according to the error model probability, or if the current character will be swapped with the next or the previous one.

Swap errors are only introduced if there are no further changes in the word. Cases of elimination of a whole word are excluded, as these would heavily influence the evaluation metrics as they are inconsistent with our model.

Chapter 4

Models

Typing sentences can be seen as a Markovian process. To correct sequences using HMMs, we can assume the hidden states to be representing the intended words, while the observations are the typed words. 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 model that implements this criterion is the Noisy Channel Model **??**. The second criterion estimates the most likely state sequence, or *trellis path*. The model and algorithm used to implement this criterion is the Hidden Markov Model and the **Viterbi** algorithm **??**.

4.1 Noisy Channel Model

The **error model** implemented in this project is called *Noisy Channel Model*. In this model we treat the original word (the misspelled word) as if a correctly spelled word had been "distorted" by passing through a noisy communication channel. This channel introduces "noise" in the form of substitutions or other changes to the letters, making it hard to recognise the "true" word.

Non-word errors are detected by looking for any word not found in a dictionary. To correct them we first generate **candidates**, according to a distance given as a parameter to the model (`edit_distance`), that are real words with a similar letter sequence to the error.

This noisy channel model is, therefore, a kind of Bayesian inference. Having an observation x (a misspelled word), we want to find the word w that generated this misspelled word (the intended word). Out of all possible words in our language model L we want to find the word $\hat{w}$ that maximised the probability $P(\hat{w}|x)$

$$\hat{w} = \arg\max_{w \in V} P(w|x). \tag{4.1}$$

Applying Bayes's rule, factoring out the denominator, since $P(x)$ doesn't change for each word because we are always asking about the most likely word for the same observed error x, and limiting L such that we're going to consider only a set of candidates C at a maximum edit distance for the word x, we get the formula:

$$\hat{w} = \arg\max_{w \in V} P(x|w)P(w). \tag{4.2}$$

The distance used to limit the set of candidates C is the **Levenshtein** edit distance **??**.

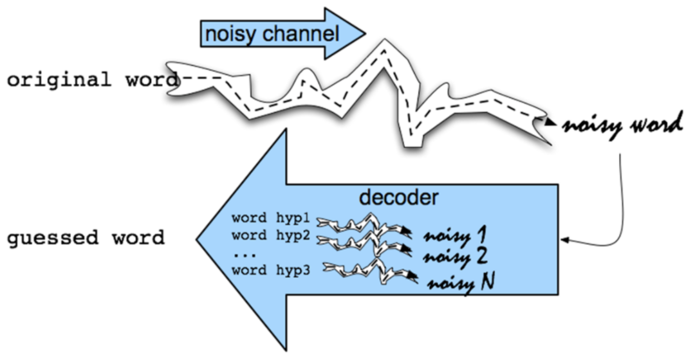

Figure 4.1: Diagram of the Noisy Channel Model

Once obtained a set of candidates, the formula 4.2 requires us to compute the two components, respectively, the prior probability of a hidden word w and the channel model (or likelihood). The prior probability $P(w)$ is given by the language model, that is obtained by counting the frequencies of each word in a corpus of text. The channel model $P(x|w)$ used in this project makes the assumption that $P(\text{balbo}|\text{bilbo}) = P(\text{a}|\text{i})$. The channel error model is trained on a corpus of spelling errors coming from different datasets. In particular, it is estimated just using the number of times that the a letter i was substituted for the letter a.

Considering each character of an alphabet A, generally, we'll have a confusion matrix $|A| \times |A|$ for each kind of channel model we're going to use. The following are the channel models used in this project:

- CHARACTER DELETIONS: $\text{del}[x, y] = \dfrac{\text{count (xy typed as x)}}{\text{count (xy)}}$
- CHARACTER INSERTIONS: $\text{ins}[x, y] = \dfrac{\text{count (x typed as xy)}}{\text{count (x)}}$
- SUBSTITUTION OF CHARACTERS: $\text{sub}[x, y] = \dfrac{\text{count (x typed as y)}}{\text{count (x)}}$

- TRANSPOSITION OF ADJACENT CHARACTERS: $\mathrm{swap}[x, y] = \dfrac{\text{count (xy typed as yx)}}{\text{count (xy)}}$

This model is appropriate for estimating the likelihood of **non-word spelling errors**, or errors were the misspelled word isn't in the vocabulary (e.g. writing *giraffe* as *graffe*). **FIXME**: When for a input typo we do not have any candidate ??

Real-word errors, or errors were the misspelled word is in the vocabulary (e.g. writing *work* as *worm*), need a slightly different approach.

We're still searching for the candidate that maximizes formula 4.2, but the channel model is treated differently. We need to assume, since the word is in the vocabulary, that the input word is not actually an error. We will call $P(w|w)$ as α. For this measure, we can make different assumptions about what should its value be, according to the writing task associated with the text. For example, we can say that professionally editing a text has an alpha of 0.99, while casually texting someone has an alpha of 0.80.

So, given a typed word x, let the channel model $P(x|w)$ be alpha when $x = w$, and then distribute $1 - \alpha$ evenly over all other candidate corrections of $C(x)$

$$P(x|w) = \begin{cases} \alpha & \text{if } x = w \\ \frac{1-\alpha}{|C(x)|} & \text{if } x \in C(x) \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

We'll then replace the edit probability of the various confusion matrices for non-word spelling errors with an equal distribution of $1 - \alpha$, while keeping the logic of the model intact.

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

4.2 Hidden Markov Model

A **Hidden Markov Model** (HMM) allows us to talk about both observed events, like misspelled words that we see in the input, and hidden events, like the intended words, that we think of as causal factors in our probabilistic model.

Our HMM is specified by the following components:

- $Q = q_1 q_2 \dots q_N$: a set of N **states**
- $A = a_{11} \dots a_{ij} \dots a_{NN}$: a **transition probability matrix** A.
  Each a_{ij} representing the probability of moving from state i to state j, such that $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$. In our case, the state transitions are given by the probability of one word given its predecessor, obtained by a certain corpus of text.

- $O = o_1 o_2 \dots o_T$: a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, \dots, v_V$
- $B = b_i(o_t)$: a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation to being generated from a state i. These are given by a corpus of misspelling errors.
- $\pi = \pi_1, \pi_2, \dots, \pi_N$: an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i.

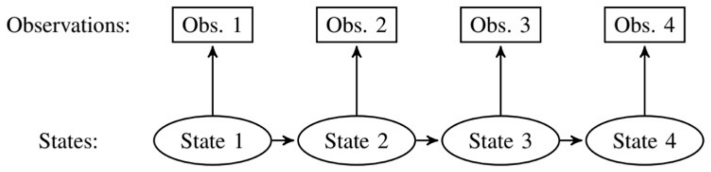

Figure 4.2: Illustration of Hidden Markov Model

We consider a *first-order* Hidden Markov Model, that instantiates two simplifying assumptions. First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state. Second, the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other states or any other observations.

For every observed word, we will consider a subset of the every possible word as its correct candidate, given by the error model described in the next section.

Typing being a sequential process, as the HMM proceeds from state to state, we will also have to limit the candidates generated by each observation with only those having an actual transition from state Q_{i-1} to Q_i.

In figure 4.3 we present an example of the evolution of an HMM for the sentence "*someone else alwais has to cary on the storry*".

FIXME

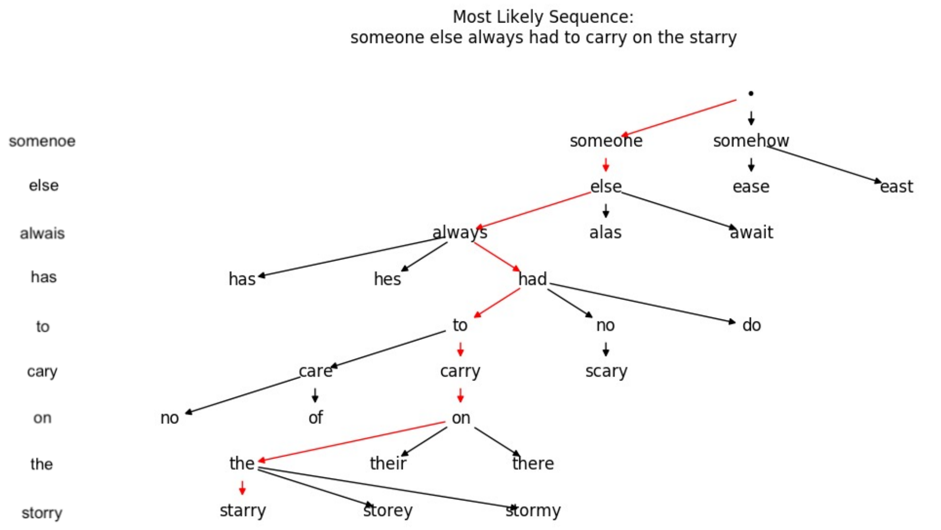

Figure 4.3: Trellis example

In this example, the most likely state sequence *someone else always had to carry on the starry*. In this case, the algorithm partially fails, because the intended sentence was *someone else always has to carry on the story*.

4.3 Most Likely State Sequence

The Viterbi algorithm calculates the most probable sequence of hidden states, the words intended. The Viterbi algorithm is a probabilistic extension of minimum edit distance. Instead of computing the "minimum edit distance" between two strings, Viterbi computes the "maximum probability alignment" of one string with another.

The initial probability of being in a state i, π_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:

$$\begin{aligned} V_{1,t+1} &= P(x_1, \ldots, x_t, x_{t+1}, y_1, \ldots, y_t, y_{t+1}) = \\ &= \arg\max_{x_{1:t}} p(x_1, \ldots, x_t | y_1, \ldots, y_t) = \\ &= \alpha \cdot p(y_{t+1}|x_{t+1}) \cdot \max_{x_t} \Big(p(x_{t+1}|x_t) \max p(x_1, \ldots, x_t | y_1, \ldots, y_t) \Big) \end{aligned} \tag{4.4}$$

The initial state probabilities π 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, **FIXME**: come vengono prese 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 **FIXME**: locally/-globally best state.

FIXME: (pi la probabilità iniziale del prima stato non lo abbiamo, non lo facciamo perchè dal nostro dataset non abbiamo sempre la prima parola. . .) **NOTA**: in Articolo.pdf del progetto non parla mai di probabilita' iniziale ma considera sempre solo il language model, a sto punto possiamo mettere la nota sopra tra i possibili miglioramenti nelle conclusioni

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.

Chapter 5

Experiment and Results

FIXME comparison **FIXME** performance

5.1 Evaluation Metrics

We evaluate the local model performances, or that carried out on individual typos, through various measures of accuracy. We compute the TOP-1 ACCURACY comparing the misspelt word and the best candidate predicted by the model. Than we also compute the TOP-3 ACCURACY and TOP-5 ACCURACY, comparing the misspelt word with the first N candidates, in this case 3 and 5, given from the model.

As regards the performance evaluation of the entire sequences, after every training and prediction we save a csv file containing the sentences with the perturbations, the sentences that we want to predict (hidden truth) and the sentences provided by the model.

The idea to evaluate the performance of the model is very simple: we scroll through the lines containing the perturbed, the predicted and the original (intended) text and, for each word we verify if it was disturbed or not and if it corresponds to the original truth. Therefore, for each word, the following cases can occur:

1. *Perturbed word not correctly provided*
   (a) the word was not the subject of attempted correction by the model
   (b) the word has been the subject of attempted correction by the model but without success
2. *Perturbed word correctly provided*
3. *Unperturbed word not correctly provided*

4. *Unperturbed word correctly provided*

We can therefore represent the confusion matrix of the data in output through table 5.1:

		Model Prediction	
		TRUE	FALSE
Hidden Truth	TRUE	True Positive Case 2.	False Negative Case 1(a).
	FALSE	False Positive Case 3. & 1(b).	True Negative Case 4.

Table 5.1: Confusion matrix

From the confusion matrix defined in the table above, it is possible to calculate the following performance metrics in a simple way:

- RATE OF PERTURBED WORDS CORRECTLY PREDICTED:

$$\frac{\text{True Positive}}{\sum \text{Perturbed}} = \frac{\text{Case 2.}}{\text{Case 2.} + \text{Case 1.}}$$

- RATE OF UNPERTURBED WORDS NOT CORRECTLY PREDICTED:

$$\frac{\text{True Negative}}{\sum \text{Unperturbed}} = \frac{\text{Case 2.}}{\text{Case 3.} + \text{Case 4.}}$$

- ACCURACY: measures the goodness of the model among the positive correction results obtained

$$\frac{\text{True Positive} + \text{True Negative}}{\sum \text{All}} = \frac{\text{Case 2.} + \text{Case 4.}}{\text{Case 1.} + \text{Case 2.} + \text{Case 3.} + \text{Case 4.}}$$

- PRECISION: ratio of the corrected predictions with respect to the significant values returned by the model (the corrections made)

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} = \frac{\text{Case 2.}}{\text{Case 1(b).} + \text{Case 2.} + \text{Case 3.}}$$

- RECALL: proportion of all the correct results

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} = \frac{\text{Case 2.}}{\text{Case 1(a).} + \text{Case 2.}}$$

- F-MEASURE: weighted average (between 0 and 1) with respect to precision and recall
$$2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Based on these metrics, it is then possible to define whether the models have behaved more or less correctly.

5.2 Experiments

We performed three different types of experiments.

The first one using as a transition model the dataset `big_clean`, the associate perturbed dataset and test error models, and the language model `frequency-alpha-gcide`.

A second one using the same datasets but introducing a lemmatisation consisting in a simple dictionary lookup.

The last one using as transition model the dataset `lotr_clean`, the associate perturbed dataset and test error models, and the language model `lotr_language_model`.

In all the experiments to come, reference will be made to the following variables:

- p: the probability that a word has an edit
- ins: the probability that a word has a letter insertion
- del: the probability that a word has a letter deletion
- sub: the probability that a word has a letter substitution
- swap: the probability that a word has a swap between two letters

5.2.1 Experiment 1

The first experiment was carried out on two different hmm that differ in the choice of the `max_edits` parameter. In the first case, in fact, the edit distance considered was 1, while in the second 2. In both cases the hmm is structured as follows:

max states	language model	sentence ds	train typo ds	test typo ds
5	`big_language_model`	`big_clean`	`big_train`	`big_test`

Table 5.2: Hmm model

In the two tables to follow are shown the results obtained as regards the evaluation of the *local correction* of our model on the typos dataset and the correction of the entire sequence with the *Viterbi algorithm*.

	Edit Distance 1		Edit Distance 2	
	Train	Test	Train	Test
Num. observation	63 759	15 918	63 759	15 918
Time (sec)	53	14	1648	408
Accuracy Top1	35.01%	35.60%	36.09%	36.84%
Accuracy Top3	46.24%	46.62%	48.61%	48.99%
Accuracy Top5	50.19%	50.56%	52.86%	53.16%

Table 5.3: Typos performance evaluation

	Edit Distance 1			
Dataset Perturbation	10%	20%	30%	40%
Time (sec)	201	199	191	186
Perturbed correct	64.06%	60.16%	57.30%	52.81%
Unperturbed not correct	41.70%	43.36%	44.84%	46.58%
Exact match	2.60%	2.32%	1.56%	1.42%
Accuracy	58.82%	57.47%	55.73%	42.60%
Precision	16.00%	25.12%	30.74%	33.75%
Recall	79.66%	77.36%	74.30%	70.40%
F-Measure	46.28%	48.49%	49.65%	49.82%

...

	Edit Distance 2			
Dataset Perturbation	10%	20%	30%	40%
Time (sec)	1110	1121	1116	1121
Perturbed correct	54.96%	55.47%	54.34%	51.76%
Unperturbed not correct	55.19%	56.31%	57.52%	58.43%
Exact match	0.90%	1.00%	0.76%	0.80%
Accuracy	45.81%	46.00%	45.85%	45.14%
Precision	10.67%	18.45%	23.7%	27.22%
Recall	89.91%	90.94%	90.69%	88.21%
F-Measure	38.77%	41.55%	44.37%	46.18%

Table 5.4: Sentences performance evaluation

5.2.2 Experiment 2

The second experiment was carried out using the same model described in the experiment above, with the difference that candidates are generated with lemmatisation.

Language models can vary widely in content, depending on different factors such as datasets, preprocessing, and so on. An ideal scenario would be having every different conjugation, inflection and mood of a word in a language model, to assess its frequency in a certain language. But this is not always the case, and as our experimentation progressed, we found time and time again that a certain word, independently from its rarity, would be missing from our language model, and as such finding its inherent probability, $P(\text{word})$, would either be solved by defaulting to a base probability, or finding another solution.

The different solution came in the form of lemmatisation. Lemmatisation takes into consideration the morphological analysis of the words. To do so, it is necessary to have detailed dictionaries which the algorithm can look through to link the form back to its lemma. Lemmatisation is more powerful than stemming because it doesn't consist only of a stemming algorithm, but also a dictionary to lookup the correct originating lemma.

We use one of the most popular packages for Natural Language Processing in Python, the **Natural Language Toolkit** (NLTK). We use the **WordNet** lexical database for lemmatisation.

Introducing lemmatisation gave us an improvement of about 2% overall accuracy on the models, as, for example, `run` and `running` would both end up looking for $P(\text{run})$, and words that did not have a corresponding term in the language model most probably had the originating lemma in it. But we also found that in some cases the lemmatisation algorithm used a somewhat heuristic approach, for example considering both `books` and `bookses` to be valid plural forms of the `book` lemma. This meant that we could not assume that a word was correct if it resulted in a correct lemma contained in the language model, defeating our intended purpose for lemmatisation. Moreover the resulting models would lose the nuance among different words derived from a single lemma, which may have a different frequency of use.

The biggest issue we found was a large time overhead introduced by lemmatisation in the process of generating candidates, making the process last from an original 0.03 seconds to more than 0.33 seconds per word, which we deemed unreasonable for the real-time usage we had in mind for the project, thus abandoning the idea of lemmatisation.

5.2.3 Experiment 3

As for the previous cases, the third experiment too was carried out on two different HMMs with the same edit distances as the previous experiments. The parameters on which the structure is based are shown in the table below:

max states	language model	sentence ds	train typo ds	test typo ds
5	`lotr_language_model`	`lotr_clean`	`lotr_train`	`lotr_test`

Table 5.5: Hmm model

In the two tables to follow are shown the results obtained as regards the evaluation of the *local correction* of our model on the typos dataset and the correction of the entire sequence with the *Viterbi algorithm*.

	Edit Distance 1		Edit Distance 2	
	Train	Test	Train	Test
Num. observation	49 959	12 570	49 959	12 570
Time (sec)	32	9	1207	309
Accuracy Top1	39.75%	40.47%	63.71%	63.87%
Accuracy Top3	44.74%	45.34%	76.44%	76.75%
Accuracy Top5	46.25%	46.92%	80.86%	80.80%

Table 5.6: Typos performance evaluation

	EDIT DISTANCE 1			
Dataset Perturbation	5%	10%	15%	20%
Time (sec)	116	114	114	112
Perturbed correct	79.97%	76.55%	73.74%	71.76%
Unperturbed not correct	13.68%	14.46%	14.86%	15.69%
Exact match	30.89%	27.51%	25.23%	22.66%
Accuracy	85.99%	84.62%	83.32%	81.73%
Precision	28.99%	41.97%	48.90%	52.85%
Recall	91.19%	88.30%	86.09%	84.31%
F-Measure	74.19%	72.87%	72.44%	71.97%

...

	EDIT DISTANCE 2			
Dataset Perturbation	5%	10%	15%	20%
Time (sec)	866	869	854	849
Perturbed correct	73.26%	72.05%	71.10%	69.90%
Unperturbed not correct	18.47%	18.86%	19.46%	20.16%
Exact match	22.74%	21.26%	19.52%	18.38%
Accuracy	81.07%	80.17%	79.06%	77.79%
Precision	22.40%	34.49%	41.35%	45.37%
Recall	94.31%	94.16%	93.63%	93.43%
F-Measure	68.76%	68.60%	68.53%	68.58%

Table 5.7: Sentences performance evaluation

Chapter 6

User Interface

Chapter 7

Conclusions

The approach we presented allows us to obtain acceptable performance, but it is strongly linked to the quality and selection of the datasets of both models. In particular it is important to use larger Language Models.

Our experimentation with different datasets has made us come to the conclusion that, for the models to work optimally, each dataset must share the same language. That is, the datasets must share most of the vocabulary used in them. We have noticed that when the datasets don't share the same vocabulary, and the distribution of words isn't aligned between them, the error model might produce candidates that it found most probable for the language model he's trained for, but in the HMM, if trained on a transitions dataset not sharing the same language as the language model, might discard these candidates for less-likely ones that have a higher transition probability.

In the future, we can generalised some assumptions and made some improvements,

- update the various dataset: in particular we can use a new frequency dataset consistent with the language model one.
- Compute the initial probabilities π from the language model
- Consider additional type of error in the model
- improving the noisy channel error model to consider neighbouring characters - Use other algorithms, such as the forward-backward (smooothing) instead Viterbi
- Optimise the code, in particular the lemmatisation/stemming
- Use recurrent neural network instead the hidden markov models.
- We can also improve the performance of the noisy channel model by changing how the prior and the likelihood are combined. In the standard model they are just multiplied together.

The fact that words are generally more frequent than their misspellings can be used in candidate suggestion, by building a set of words and spelling variations that have similar contexts, sorting by frequency, treating the most frequent variant as the source, and learning an error model from the difference

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. 15).

[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.

[4] James H Martin and Daniel Jurafsky. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition.* Pearson/Prentice Hall Upper Saddle River, 2009.