Correction orthographique par apprentissage Bayésien

Edouard Leurent
Bastien Debras

7 juin 2013

Cours d'apprentissage artificiel

Table des matières

1 Introduction

1.1 Démarche

Comment réagir face à une faute d'orthographe ? Une première idée pourrait être de regarder dans un **dictionnaire**. Si l'utilisateur tape un mot qui ne fait pas partie du dictionnaire, on le remplace simplement par le mot du dictionnaire le plus proche (en un sens à préciser).

Cependant, on se rend vite compte que cette définition du mot "le plus proche" est ambigüe : que faire si un grand nombre de mots sont tous également candidats à être le mot le plus proche ?

Dans cette situation, notre démarche pour lever l'ambiguïté a été de choisir mot le plus **probable**, c'est à dire le plus fréquent, celui qui apparait le plus dans l'usage. Pour apprendre ces fréquences, il faut donc disposer non pas d'un simple dictionnaire, mais plutôt d'un **corpus** de textes.

2 Formalisation

Nous allons ici préciser nos problèmes, et expliquer notre choix d'algorithmes pour les traiter

2.1 Les problèmes

Nous allons voir que le problème de correction orthographique peut se ramener à un problème de Machine Learning.

- On considère un mot tapé par l'utilisateur comme une **observation** du mot réel qu'il a voulu écrire. Les mots d'une langue sont alors interprétées comme des **classes**. Corriger une faute d'orthographe revient donc à trouver à quelle classe appartient une observation donnée. Nous avons un **premier problème de classification**, très fortement multiclasse (par exemple, la langue française contient environ 100 000 mots).
- Pour rester général, nous avons décider de ne pas considérer uniquement le français mais de traiter le cas d'un **langage quelconque**. Une fois plusieurs langages appris, corriger une phrase donnée nécessite de savoir dans quel langage il faut raisonner. On doit donc être capable d'identifier dans quelle langue est écrite une phrase. La détection de la langue constitue alors notre **deuxième problème de classification**.

Pour nos exemples, nous avons choisi d'utiliser le français et l'anglais.

2.2 Méthode Bayésienne

On a choisit de considérer les mots les plus probables, et on a donc adopté une approche probabiliste. Une méthode adéquate est donc d'utiliser un **classifieur naïf de Bayes**.

Il est formalisé de la manière suivante :

Étant donné l'ensemble C des corrections possibles d'un mot donné, on s'intéresse à la probabilité qu'une correction particulière soit la bonne.

On note m le mot tapé par l'utilisateur
c une correction possible de ce mot
$P(c|m)$ la probabilité que le mot lu m soit une observation du mot correct c

On a alors, d'après la **formule de Bayes**

$$P(c|m) = \frac{P(m|c) \bullet P(c)}{P(m)} \tag{1}$$

On essaie de maximiser cette quantité. Le dénominateur $P(m)$ n'est donc qu'un facteur normalisateur qui est constant sur l'ensemble C des corrections, et on peut donc choisir la convention $P(m) = 1$

On cherche donc

$$\max_{c \in C} P(m|c) \bullet P(c) \tag{2}$$

On constate alors que notre fonction à maximiser est constituée de deux termes

- $P(m|c)$ la probabilité d'avoir fait la faute m en voulant taper le mot c. C'est le **modèle d'erreur**.

– $P(c)$ la probabilité que l'utilisateur ait voulu taper le mot c. C'est le **modèle de langage**.

Cette séparation en deux termes est très intuitive. Comment corriger la faute de frappe *histre* par exemple ? On peut proposer les deux corrections *bistre* et *histoire* par exemple. Laquelle est la meilleure ?

– D'un coté le mot *bistre* est plus proche de *histre* que le mot *histoire*, car une seule modification est nécessaire contre deux pour *histoire*. Il est plus probable que l'utilisateur ait faite une seule faute plutôt que deux fautes dans un même mot.
– D'un autre coté, le mot *histoire* est bien plus courant que le mot *bistre*, et il est donc plus probable que l'utilisateur ait voulu écrire ce mot.

Il faut donc bien considérer ces deux critères du modèle de langage et du modèle d'erreur pour pouvoir lever l'ambiguïté et choisir la meilleure correction.

2.3 Le modèle de langage

Il nous faut établir un modèle permettant d'évaluer la probabilité $P(c)$ qu'un mot c soit utilisé.

Pour cela, on aborder une approche pragmatique : plus on rencontre un mot dans le corpus des textes, plus sa probabilité sera grande. La probabilité du mot est donc déterminée par sa fréquence. En fait, on assimile directement la probabilité d'un mot au nombre de ses occurrences. On ne divise pas par le nombre de mots du corpus, qui ne constitue qu'un facteur multiplicatif et ne permet pas de différencier les différentes corrections.

On pose donc simplement

$$P(c) = \#\{\text{occurences de } c\} \tag{3}$$

2.4 Le modèle d'erreur

On doit mettre en place un modèle permettant de calculer $P(m|c)$, la probabilité qu'en voulant écrire c l'utilisateur fasse une ou plusieurs fautes et aboutisse au mot m.

Nous avons choisit un modèle simple, basé sur la notion d'**opération élémentaire**.

Une opération élémentaire sur un mot est définie comme étant au choix :

– La suppression d'une lettre du mot
– Le remplacement d'une lettre du mot
– L'insertion du lettre dans le mot
– La transposition de deux lettres du mot

On interprète alors une faute de frappe comme une opération élémentaire, à laquelle on attribue une certaine probabilité d'erreur P_e. Cette probabilité est un paramètre de l'algorithme, choisi ici à $1/20^{\text{ème}}$.

On calcule ensuite la distance d'édition d entre deux mots, c'est à dire le nombre minimal d'opérations élémentaires pour passer d'un mot à l'autre.

On pose finalement le modèle d'erreur suivant :

$$P(m|c) = {P_e}^{d(m,c)} \tag{4}$$

Chaque opération élémentaire effectuée diminue donc la probabilité d'un facteur P_e.

2.5 Détection de la langue

On cherche la probabilité $P(l|m_1m_2m_3...m_n)$ qu'une phrase formée de plusieurs mots m_1 m_2 m_3 ... m_n soit écrite en une langue l.

Cette probabilité s'exprime à nouveau par la **formule de Bayes**

$$P(l|m_1m_2m_3...m_n) = \frac{P(m_1m_2m_3...m_n|l)P(l)}{P(m_1m_2m_3...m_n)} \tag{5}$$

Examinons chacun de ces termes

– $P(m_1m_2m_3...m_n)$ est la probabilité de la phrase. Ce terme est difficile à évaluer et est un facteur multiplicatif constant indépendant de les langues qu'on est en train de comparer. On peut donc sans perte de généralité le choisir comme égal à 1.
– $P(l)$ représente la probabilité de la langue. On suppose ici que l'utilisateur tape une phrase en français ou en anglais de manière équivalente, et on suppose alors toutes les langues équiprobables. A nouveau, ce terme est choisi comme étant égal à 1.

– $P(m_1 m_2 m_3 ... m_n|l)$ est la probabilité de la phrase sachant qu'elle est écrite en français. Pour simplifier le calcul, on prend une **hypothèse très forte** : on suppose les mots **indépendants** deux a deux. Ce n'est pas du tout le cas en pratique, ne serait-ce qu'à cause des règles de grammaire et des expressions idiomatiques, mais cela est suffisant pour avoir une bonne approximation. En effet, ce terme s'exprime alors

$$P(m_1 m_2 m_3 ... m_n|l) = P(m_1|l) \times P(m_2|l) \times P(m_3|l) \times ... \times P(m_n|l) = \prod_{i=1}^{n} P(m_i|l) \quad (6)$$

Or, les termes $P(m_i|l)$ peuvent être obtenus de la même manière que $P(c)$ précédemment, directement par le nombre d'occurrences dans le modèle de langage.

On a donc finalement l'équation suivante :

$$P(l|m_1 m_2 m_3 ... m_n) = \prod_{i=1}^{n} P(m_i|l) \quad (7)$$

3 Implémentation

Nous avons implémenté le modèle formel présenté ci-dessus dans un programme en Java. Nous décrirons chaque partie de notre code-source en suivant le plan du modèle.

3.1 Modèle de langage

3.1.1 La structure de dictionnaire

On veut stocker le nombre d'occurrences de chaque mot des œuvres du corpus. La structure de données que nous avons choisi est une table de hachage, implémentée en Java selon l'interface `Map` et la classe `HashMap`.

On se pose alors la question suivante : que faire si l'on rencontre un mot tel que *tyrannosaure*, parfaitement français mais jamais lu dans notre corpus d'apprentissage ? Il ne serait pas raisonnable de lui affecter une probabilité nulle uniquement parce qu'on ne l'a jamais vu auparavant. Il y a plusieurs approches possibles de ce problème, et nous avons décidé d'appliquer la plus simple d'entre elle : traiter un nouveau mot comme si on l'avait déjà vu une seule fois. Ce procédé s'appelle le **lissage**, parce qu'on lisse la distribution de probabilités qui aurait été nulle, l'arrondissant au plus petit dénombrement possible.

Pour cela, nous avons modifié légèrement la classe `HashMap` de la manière suivante : quand on fait un appel à la fonction `get()` et que la clé requise n'est pas présente dans la table de hachage, on renvoi la valeur 1 par défaut.

Dictionnaire.java

```java
import java.util.HashMap;
import java.util.Set;

/**
 * Table de hashage avec une valeur par defaut
 *
 */
public class Dictionnaire extends HashMap<String,Integer> {

        /**
         * Valeur par defaut
         */
        protected Integer valeurParDefaut;

        /**
         * Constructeur
         * @param defaultValue
         */
  public Dictionnaire(Integer defaultValue) {
    this.valeurParDefaut = defaultValue;
  }

  /**
```

```
    * Obtenir la valeur d'une cle
    */
   @Override
   public Integer get(Object k) {
     Integer v = super.get(k);
     return ((v == null) && !this.containsKey(k)) ? this.valeurParDefaut : v;
   }
}
```

3.1.2 Phase d'apprentissage

On doit maintenant parcourir un corpus de fichiers textes en comptant les occurrences de chaque mot pour remplir le dictionnaire de notre modèle.

Pour cela, on commence par convertir notre fichier texte en `String`.

```
/**
 * Lis un fichier texte
 * @param chemin
 * @return
 */
public static String lireFichier(String chemin) {
    String texte = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(chemin));
        StringBuilder sb = new StringBuilder();
        String line = br.readLine();
        while (line != null) {
            sb.append(line);
            sb.append("\n");
            line = br.readLine();
        }
        texte = sb.toString();
        br.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return texte;
}
```

Une fois qu'on a notre texte en String, on peut parcourir ses mots et compter leurs occurences en les ajoutant au dictionnaire.

```
/**
 * Decoupe un texte en mots
 * @param texte une chaine de caracteres
 * @return les mots du texte
 */
public static String[] mots(String texte) {
    return texte.split("[^\\p{L}]+");
}

/**
 * Apprentissage des frequences d'un ensemble mots dans le modele de langage
 * @param features les mots a apprendre
 */
public void apprendre(String [] features){
    for (String f : features)
```

```
            modele.put(f.toLowerCase(), modele.get(f.toLowerCase())+1);
        }
```

En lisant ainsi un ensemble de textes, nous pouvons alors établir une bonne distribution des probabilités de chacun des mots du modèle de langage.

On choisit les œuvres suivantes pour notre corpus d'apprentissage :

Français	Anglais
Émile Zola, *L'argent*	William Shakespeare, *Henry VI*
Émile Zola, *L'assommoir*	William Shakespeare, *Hamlet*
Émile Zola, *Germinal*	William Shakespeare, *MacBeth*
Victor Hugo, *Les Misérables*	Lewis Caroll, *Alice in Wonderland*
Marcel Proust, *Du côté de chez Swann*	Sir Arthur Conan Doyle, *Sherlock Holmes*
Stendhal, *Le rouge est le noir*	Herman Melville, *Moby Dick; or The Whale*
Alexandre Dumas, *Les trois mousquetaires*	Mary Shelley, *Frankenstein or The Modern Prometheus*
Gustave Flaubert, *Madame Bovary*	Charles Dickens, *Great Expectations*

Et pour le corpus de test, nous utilisons les œuvres suivantes :

Français	Anglais
Victor Hugo, *Notre dame de Paris*	Oscar Wilde, *The Picture of Dorian Gray*

3.1.3 Détection de la langue

A ce stade du développement, le modèle de langage implémenté nous permet directement de détecter la langue. En effet, il nous suffit de reprendre simplement l'expression de la probabilité d'une langue selon la phrase.

$$P(l|m_1 m_2 m_3 ... m_n) = \prod_{i=1}^{n} P(m_i|l) \tag{8}$$

Notre implémentation est la suivante :

```
/**
 * Calcule la probabilite d'une phrase
 * @param phrase
 * @return
 */
public double probabilitePhrase(String phrase) {
    double produit = 1;
    String[] mots = mots(phrase);
    for (String mot:mots)
        produit *= modele.get(mot);
    return produit;
}

/**
 * Determine le langage le plus probable
 * @param phrase
 * @param langages
 * @return le langage le plus probable
 */
public static Langage predireLangage(String phrase, Langage[] langages) {
    Langage langageMax = null;
    double probaMax = 0;
    for (Langage l:langages) {
```

```
24            if (l.probabilitePhrase(phrase) > probaMax) {
25                probaMax = l.probabilitePhrase(phrase);
26                langageMax = l;
27            }
28        }
29        return langageMax;
30    }
```

3.2 Modèle d'erreur

On veut maintenant pouvoir évaluer $P(c|m)$.

3.2.1 Génération des corrections

Pour calculer la correction la plus probable, il faut d'abord générer toutes les corrections possibles. Étant donné la densité des mots français dans l'ensemble des mots possibles, on décide de ne générer que les mots à distance 1 ou 2 du mot original. Parmi cet ensemble, l'un d'entre eux au moins sera un mot français, et sera surement plus probable qu'un mot de distance 3 ou supérieure (on ne fait que très rarement plus de deux fautes dans le même mot).

Si malgré tout on rencontre un mot qui n'est proche d'aucun mot français à une distance de 1 ou 2, on considérera que ce n'est pas une faute d'orthographe, mais simplement un mot du français que nous n'avons jamais rencontré. C'est le cas par exemple d'un mot rare comme *tyrannosaure*, comme on l'avait évoqué plus tôt en **??**.

Ainsi, on commence par générer toutes les séparations du mot original en deux mots, qu'on stocke dans un tableau `separations`.

On note ici $m = m_1.m_2$, où . est la concaténation.

On génère ensuite les différentes transformations du mot de la manière suivante :

- Suppression : On concatène m_1 et un suffixe de m_2 dans lequel on a supprimé sa première lettre
- Transposition : On concatène m_1 à la deuxième lettre de m_2, la première lettre de m_2, et la suite de m_2.
- Mutation : On concatène m_1 avec m_2 dans lequel la première lettre est remplacée par une lettre de l'alphabet Σ
- Insertion : On concatène m_1, une lettre du langage, et m_2

Notre code source est le suivant :

```
1   /**
2    * Genere l'ensemble des mots a distance 1 d'un mot donne
3    * @param mot
4    * @return
5    */
6   public Set<String> modifications1(String mot){
7           Set<String> modifications = new HashSet<String>();
8           String[][] separations =  new String[mot.length() + 1][2];
9           for (int i=0; i<mot.length()+1; i++){
10                  separations[i][0] = mot.substring(0, i);
11                  separations[i][1] = mot.substring(i, mot.length());
12          }
13          for (int i=0; i<separations.length; i++) {
14                  // Supressions
15                  if (!separations[i][1].isEmpty())
16                          modifications.add(separations[i][0] + separations[i][1].
                                substring(1));
17                  // Transpositions
18                  if (separations[i][1].length() > 1)
19                          modifications.add(separations[i][0] + separations[i][1].
                                charAt(1) + separations[i][1].charAt(0) + separations
                                [i][1].substring(2));
20                  // Mutations
```

```
                if (!separations[i][1].isEmpty())
                        for (int j=0; j<alphabet.length(); j++)
                                modifications.add(separations[i][0] + alphabet.
                                    charAt(j) + separations[i][1].substring(1));
                // Insertions
                for (int j=0; j<alphabet.length(); j++)
                        modifications.add(separations[i][0] + alphabet.charAt(j)
                            + separations[i][1]);
        }
    return modifications;
}
```

Remarque : On constate qu'on a besoin d'un alphabet Σ, qui n'est pas le même selon le langage. En effet, le français par exemple est une langue accentuée et contient des caractères spéciaux tels que le ç, contrairement à l'anglais qui se contente de lettres de A à Z. On doit donc enregistrer l'ensemble des lettres du langage dans une chaine de caractères, que l'on transmet lors du constructeur du langage.

```
public class Langage {
    /**
     * Le nom du langage
     */
    public String nom;

    /**
     * La liste des caracteres qui composent l'alphabet du langage
     */
    public String alphabet;

    /**
     * Le dictionnaire de frequence des mots
     */
    public Dictionnaire modele = new Dictionnaire(1);

    /**
     * Cree un nouveau langage par choix d'un alphabet et apprentissage d'un
         corpus
     * @param nom
     * @param alphabet
     * @param corpus
     */
    public Langage(String nom, String alphabet, String[] corpus) {
        this.nom = nom;
        this.alphabet = alphabet;
        for (String oeuvre:corpus)
            apprendre(mots(lireFichier(oeuvre)));
    }

    [...]

    /**
     * Integration du corpus francais
     */
    public static Langage francais = new Langage("Francais", "
        abcdefghijklmnopqrstuvwxyz[caracteres_accentues]",
            new String[]{"corpus/fr/dictionnaire.txt",
                         "corpus/fr/miserables1.txt",
                         "corpus/fr/miserables2.txt",
                         "corpus/fr/miserables3.txt",
                         "corpus/fr/miserables4.txt",
                         "corpus/fr/miserables5.txt",
```

```
                                "corpus/fr/assomoir.txt",
                                "corpus/fr/germinal.txt",
                                "corpus/fr/largent.txt",
                                "corpus/fr/swann.txt",
                                "corpus/fr/rougeetnoir.txt"
                                "corpus/fr/madamebovary.txt"});
    /**
     * Integration du corpus anglais
     */
    public static Langage anglais = new Langage("Anglais", "
        abcdefghijklmnopqrstuvwxyz",
                new String[]{"corpus/en/dictionary.txt",
                                "corpus/en/henriVI.txt",
                                "corpus/en/hamlet.txt",
                                "corpus/en/macbeth.txt",
                                "corpus/en/alice.txt",
                                "corpus/en/sherlock.txt",
                                "corpus/en/frankenstein.txt",
                                "corpus/en/mobydick.txt",
                                "corpus/en/greatexpectations.txt"});
}
```

On génère alors les corrections de distance 2 en itérant deux fois l'algorithme de modifications :

```
/**
 * Genere l'ensemble des mots a distance 2 d'un mot donne
 * @param mot
 * @return
 */
public Set<String> modifications2Connues(String mot) {
    Set<String> modifications = new HashSet<String>();
    for (String edit1:modifications1(mot))
        for (String edit2:modifications1(edit1))
            if (modele.containsKey(edit2))
                modifications.add(edit2);
    return modifications;
}
```

On ne conserve que les corrections potentielles qui sont connues de notre modèle.

3.2.2 Sélection des meilleurs corrections

Une fois les corrections générées, on calcule leurs probabilités selon la formule

$$P(c|m) = P(m|c)P(c) = {P_e}^{d(m,c)} \times \#\{\text{occurences de } c\} \tag{9}$$

Ces couples (correction,probabilité) sont représentés dans une classe `Suggestion`. Elles implémentent l'interface `Comparable` qui compare leurs probabilités.

Suggestion.java

```
/**
 * Mots suggeres pour correction avec leur probabilite
 *
 */
class Suggestion implements Comparable {

        /**
         * La suggestion
         */
        private final String mot;

```

```java
        /**
         * Probabilite de la suggestion
         */
    private double probabilite;

    /**
     * Constructeur
     * @param mot
     * @param probabilite
     */
    public Suggestion(String mot, double probabilite) {
        this.mot = mot;
        this.probabilite = probabilite;
    }

    /**
     * Renvoie la suggestion
     * @return mot
     */
    public String getMot() {
        return mot;
    }

    /**
     * Renvoie la probabilite de la suggestion
     * @return probabilite
     */
    public double getProbabilite() {
        return probabilite;
    }

    /**
     * Change la probabilite donnee
     * @param probabilite
     */
    public void setProbabilite(double probabilite) {
        this.probabilite = probabilite;
    }

    /**
     * Compare la probabilite de notre suggestion a celle d'une autre suggestion
     */
        @Override
        public int compareTo(Object o) {
                double autre = ((Suggestion)o).probabilite;
                if (probabilite > autre)
                        return 1;
                else if (probabilite < autre)
                        return -1;
                else
                        return 0;
        }

}
```

On veut ensuite pouvoir conserver les N meilleurs suggestions, celles qui ont les probabilités les plus importantes. On choisit pour cela une **structure de donnée adaptée** à ce problème : un **tas minimal** (MinHeap) de taille fixe N. On insère une par une les suggestions potentielles dans le tas, qu'on met à jour pour qu'il ne conserve que N meilleurs suggestions déjà lues.

L'algorithme est simple et optimal :

– On parcourt les suggestions. Pour chaque suggestion,
  1. On récupère le minimum du tas (en temps constant, c'est la racine)
  2. Si la nouvelle suggestion a une probabilité plus grande que le minimum, elle doit être insérée dans le tas.
     (a) On supprime le minimum, pour libérer une place dans le tas (temps logarithmique)
     (b) On insère la nouvelle suggestion dans le tas (temps logarithmique)

Les insertions et suppressions sont logarithmiques car on ne fait que descendre ou remonter le long des niveaux de l'arbre, en permutant les éléments pour maintenir la propriété de tas minimal (algorithme *bubble-up*).

Voici le code source de notre structure de donnée MinHeap

MinHeap.java

```
import java.util.*;

/**
 * Tas minimal de suggestions maximales
 *
 */
public class MinHeap {

        /**
         * Liste des suggestions dans le tas
         */
  List<Suggestion> h = new ArrayList<Suggestion>();

  /**
   * Taille du tas minimal
   */
  int taille = 5;

  /**
   * Constructeur
   */
  public MinHeap() {
  }

  /**
   * Constructeur avec une liste de suggestions
   */
  public MinHeap(Suggestion[] suggestions) {
    for (Suggestion suggestion : suggestions) {
      h.add(suggestion);
    }
    for (int k = h.size() / 2 - 1; k >= 0; k--) {
      faireDescendre(k, h.get(k));
    }
  }

  /**
   * Insertion d'une nouvelle suggestion
   * @param noeud
   */
  public void add(Suggestion noeud) {
        if (h.size() == taille) {
                if (min().compareTo(noeud) < 0)
                        remove();
                else
                        return;
        }
```

```
48
49          //Mise a jour de la probabilite d'une suggestion si le mot est deja
                present
50          Suggestion p = get(noeud.getMot());
51          if (p != null) {
52                  if (p.compareTo(noeud) < 0)
53                          p.setProbabilite(noeud.getProbabilite());
54                  return;
55          }
56
57        h.add(null);
58        int k = h.size() - 1;
59        while (k > 0) {
60          int parent = (k - 1) / 2;
61          p = h.get(parent);
62          if (noeud.getProbabilite() >= p.getProbabilite()) {
63            break;
64          }
65          h.set(k, p);
66          k = parent;
67        }
68        h.set(k, noeud);
69      }
70
71      /**
72       * Renvoie la Suggestion associee au mot cherche
73       * @param mot
74       * @return Suggestion depuis le tas associee au mot
75       */
76      public Suggestion get(String mot) {
77              for (Suggestion noeud : h) {
78                  if (noeud.getMot().equals(mot))
79                      return noeud;
80              }
81              return null;
82      }
83
84      /**
85       * Supprime le noeud min
86       * @return noeud enleve
87       */
88      public Suggestion remove() {
89              Suggestion removedNode = h.get(0);
90              Suggestion lastNode = h.remove(h.size() - 1);
91        faireDescendre(0, lastNode);
92        return removedNode;
93      }
94
95      /**
96       * Renvoie la suggestion avec proba minimale
97       * @return
98       */
99      public Suggestion min() {
100       return h.get(0);
101     }
102
103     /**
104      * Test de vacuite du tas
105      * @return
106      */
107     public boolean isEmpty() {
```

```
108       return h.isEmpty();
109     }
110
111     /**
112      * Compare un noeud avec ceux d'en-dessous pour le faire descendre s'il est
            plus grand
113      * @param k
114      * @param node
115      */
116     void faireDescendre(int k, Suggestion node) {
117       if (h.isEmpty()) {
118         return;
119       }
120       while (k < h.size() / 2) {
121         int fils = 2 * k + 1;
122         if (fils < h.size() - 1 && h.get(fils).compareTo(h.get(fils + 1))  > 0) {
123           fils++;
124         }
125         if (node.compareTo(h.get(fils)) <= 0) {
126           break;
127         }
128         h.set(k, h.get(fils));
129         k = fils;
130       }
131       h.set(k, node);
132     }
133
134     /**
135      * Test unitaire
136      * @param args
137      */
138     public static void main(String[] args) {
139       MinHeap heap = new MinHeap();
140       Suggestion noeud;
141       for (int i=0; i<10; i++) {
142           noeud = new Suggestion("noeud"+(int)(10*Math.random()), Math.random());
143           System.out.println(noeud.getMot() + "␣:␣" + noeud.getProbabilite());
144           heap.add(noeud);
145       }
146
147       System.out.println("------------------");
148
149       while (!heap.isEmpty()) {
150           noeud = heap.remove();
151           System.out.println(noeud.getMot() + "␣:␣" + noeud.getProbabilite());
152       }
153     }
154 }
```

Le code source de la sélection des meilleurs suggestions et de la correction finale est alors le suivant :

```
1  /**
2   * Determine tous les mots probables a partir d'un mot mal orthographie
3   * @param mot
4   * @return la liste des suggestions
5   */
6  public List<Suggestion> suggestions(String mot) {
7      double probabiliteTypo = 20.;
8      mot = mot.toLowerCase();
9      MinHeap heap = new MinHeap();
```

```
        heap.add(new Suggestion(mot, modele.get(mot)/1.));
        for (String edit1:connus(modifications1(mot))) {
            heap.add(new Suggestion(edit1, modele.get(edit1)/probabiliteTypo));
        }
        for (String edit2:modifications2Connues(mot)) {
            heap.add(new Suggestion(edit2, modele.get(edit2)/Math.pow(
                probabiliteTypo,2)));
        }
        Collections.sort(heap.h, Collections.reverseOrder());
        if (heap.isEmpty())
            heap.add(new Suggestion(mot, 0.));
        return heap.h;
    }

    /**
     * Renvoie la correction la plus probable
     * @param mot
     * @return la meilleure correction
     */
    public String corriger(String mot) {
        return suggestions(mot).get(0).getMot();
    }
```

Pour corriger une phrase entière en conservant la ponctuation, on utilise les expressions régulière pour détecter les mots et les corriger à la volée.

```
    /**
     * Corrige toute la phrase, en conservant la ponctuation
     * @param phrase
     * @return la phrase corrigee mot par mot
     */
    public String corrigerPhrase(String phrase) {
        Pattern pattern = Pattern.compile("[\\p{L}]+");
        Matcher m = pattern.matcher(phrase);
        StringBuffer sb = new StringBuffer();
        while (m.find())
        {
          m.appendReplacement(sb, "");
          sb.append(corriger(m.group()));
        }
        m.appendTail(sb);
        return sb.toString();
    }
```

3.3 L'interface homme-machine

Pour utiliser plus aisément notre modèle, nous avons décidé de développer une petite interface simpliste.

L'implémentation de cette interface n'étant pas l'objet de notre rapport, on se contentera d'en expliquer les grandes lignes.

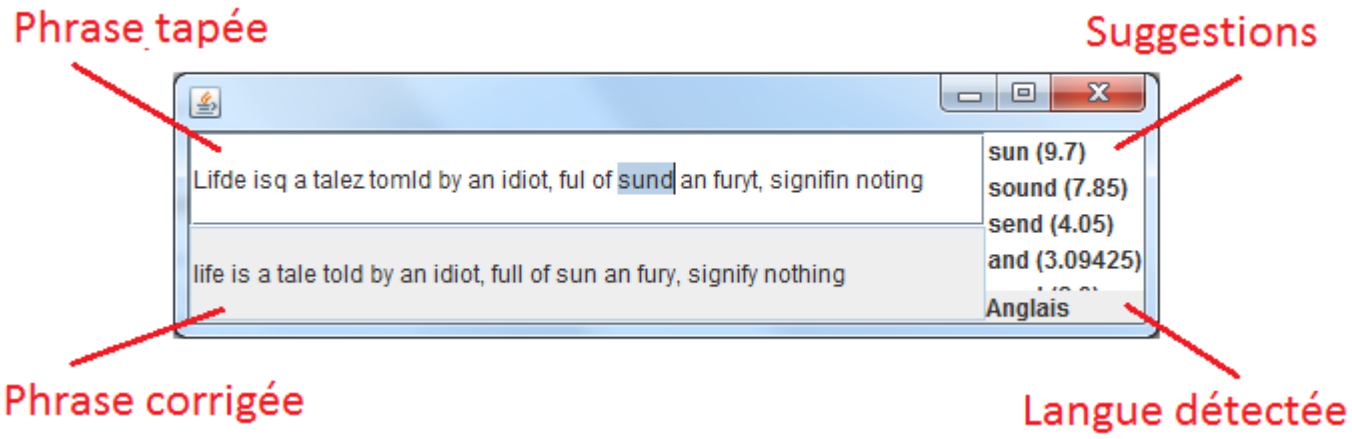

FIGURE 1 – L'interface graphique montrant la correction d'une célèbre locution

La fenetre est consituée de 4 zones :

- Un champ de texte dans lequel on peut taper une phrase
- Un champ de texte qui affiche la phrase corrigée selon notre algorithme
- Un composant de liste, qui affiche les différentes suggestions pour le mot sélectionné et leurs probabilités
- Un label qui affiche la langue détectée

Pour ne pas perturber la réactivité de l'interface, tous les calculs sont réalisés à intervalles réguliers dans un `Thread` séparé. Dès qu'une modification du texte est détectée, l'ordre est donné au `Thread` de mettre à jour la correction lors de sa prochaine itération.

Cela permet de visualiser les corrections en temps réel en conservant un sentiment de fluidité.

Le code source de l'interface graphique est le suivant :

Fenetre.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Dimension;
3  import java.awt.GridLayout;
4  import java.awt.event.KeyEvent;
5  import java.awt.event.KeyListener;
6  import java.awt.event.MouseEvent;
7  import java.awt.event.MouseListener;
8  import java.util.List;
9  import java.util.Vector;
10
11 import javax.swing.DefaultListModel;
12 import javax.swing.JComboBox;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JList;
16 import javax.swing.JPanel;
17 import javax.swing.JTextField;
18 import javax.swing.ListModel;
19
20 /**
21  * Fenetre d'affichage
22  * @author user
23  *
24  */
25 public class Fenetre extends JFrame implements KeyListener, MouseListener{
26         /**
27          * Panel d'affichage des phrases tapees et corrigees
28          */
```

```
29          JPanel panneauPhrases;
30
31          /**
32           * Panel des suggestions et de la langue
33           */
34          JPanel panneauOptions;
35
36          /**
37           * TextField de la phrase a corriger
38           */
39          JTextField input;
40
41          /**
42           * Affiche la langue la plus probable
43           */
44          JLabel nomLangagePredit;
45
46          /**
47           * Affiche la phrase corrigee
48           */
49          JTextField output;
50
51          /**
52           * Le langage le plus probable
53           */
54          Langage langagePredit;
55
56          /**
57           * Update l'affichage des phrases et langues
58           */
59          Updater updater;
60
61          /**
62           * Strings contenus dans les suggestions
63           */
64          Vector<String> choixModele;
65
66          /**
67           * Affichage des strings
68           */
69          JList<String> choixCorrection;
70
71          /**
72           * Cree la fenetre d'affichage
73           */
74          public Fenetre() {
75                  panneauPhrases = new JPanel();
76                  panneauOptions = new JPanel();
77                  panneauOptions.setLayout(new BorderLayout());
78                  panneauPhrases.setLayout(new GridLayout(2,1));
79                  input = new JTextField("Lifde␣isq␣a␣talez␣tomld␣by␣an␣idiot,␣ful
                        ␣of␣sund␣an␣furyt,␣signifin␣noting");
80                  input.addKeyListener(this);
81                  input.addMouseListener(this);
82                  output = new JTextField();
83                  output.setEditable(false);
84                  nomLangagePredit = new JLabel("???");
85                  choixModele = new Vector<String>();
86                  choixCorrection = new JList<String>(choixModele);
87                  setLayout(new BorderLayout());
88                  panneauPhrases.add(input);
```

```
89                  panneauPhrases.add(output);
90                  panneauOptions.add(choixCorrection, BorderLayout.CENTER);
91                  panneauOptions.add(nomLangagePredit, BorderLayout.SOUTH);
92                  add(panneauPhrases, BorderLayout.CENTER);
93                  add(panneauOptions, BorderLayout.EAST);
94                  corrigerPhrase();
95                  updater = new Updater(this);
96                  updater.start();
97                  setSize(new Dimension(700,170));
98                  setVisible(true);
99                  setDefaultCloseOperation(EXIT_ON_CLOSE);
100         }
101         /**
102          * Corrige la phrase tapee et l'affiche
103          */
104         public void corrigerPhrase() {
105                 langagePredit = Langage.predireLangage(input.getText(), new
                        Langage[]{Langage.anglais, Langage.francais});
106                 nomLangagePredit.setText(langagePredit.nom);
107                 output.setText(langagePredit.corrigerPhrase(input.getText()));
108                 afficherSuggestions();
109         }
110         /**
111          * Affiche les suggestions
112          */
113         public void afficherSuggestions() {
114                 int indice = input.getCaretPosition();
115                 String[] avant = input.getText().substring(0,indice).split("[^\\
                        p{L}]+");
116                 String mot =  avant[avant.length-1] + input.getText().substring(
                        indice).split("[^\\p{L}]+")[0];
117                 List<Suggestion> suggestions = langagePredit.suggestions(mot);
118                 choixModele.clear();
119                 for(Suggestion suggestion:suggestions) {
120                         choixModele.add(suggestion.getMot() + "_(" + suggestion.
                                getProbabilite() + ")");
121                 }
122                 choixCorrection.setListData(choixModele);
123         }
124
125         /**
126          * Main principal
127          * @param args
128          */
129         public static void main(String[] args) {
130                 Fenetre fenetre = new Fenetre();
131         }
132         /**
133          * Met a jour la phrase corrigee et le langage detecte a intervalle
                regulier
134          * @author user
135          *
136          */
137         class Updater extends Thread {
138                 Fenetre fenetre;
139                 boolean MAJRequise;
140                 public Updater(Fenetre fenetre) {
141                         this.fenetre = fenetre;
142                         MAJRequise = true;
143                 }
144                 @Override
```

```
145                 public void run() {
146                         while(true) {
147                                 if (MAJRequise) {
148                                         fenetre.corrigerPhrase();
149                                         MAJRequise = false;
150                                 }
151                                 try {
152                                         Thread.sleep(200);
153                                 } catch (InterruptedException e) {
154                                         e.printStackTrace();
155                                 }
156                         }
157                 }
158         }
159
160         @Override
161         public void mousePressed(MouseEvent arg0) {
162                 updater.MAJRequise = true;
163         }
164
165         @Override
166         public void keyPressed(KeyEvent arg0) {
167                 updater.MAJRequise = true;
168         }
169 }
```

4 Résultats

4.1 Tests simples

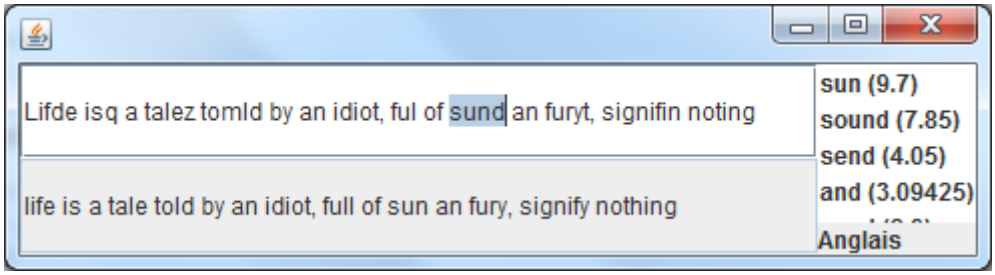

FIGURE 2 – Premier test, issu d'une locution bien connue

On teste notre programme avec la célèbre phrase suivante :

Lifde isq a talez tomld by an idiot, ful of sund an furyt, signifin noting.

Notre programme détecte la langue anglaise, et corrige la phrase de cette manière :

Life is a tale told by an idipt, full of sun an fury, signify nothing.

On constate que trois mots seulement sont mal corrigés.

– Le mot *sund* est corrigé en *sun* au lieu de *sound*.
  Examinons les suggestions et leurs probabilités :
  1. sun (9.7)
  2. sound (7.85)
  3. send (4.05)
  4. and (3.09)
  5. sund (2.0)

On observe que *sun*, *sound* et *send* sont tous trois à une distance de 1 de *sund*. Ils sont donc classés par nombre d'occurrences, et le mot correct *sound* n'apparait qu'en deuxième car il a été rencontré moins souvent.
De plus, la correction *and* est proposée malgré le fait qu'il nécessite 2 opérations, du fait de sa très grande fréquence en anglais. Sa probabilité est du même ordre de grandeur que celles des première suggestions, mais c'est cette fois le terme du modèle de langage qui est prépondérant et non plus celui du modèle d'erreur.
Enfin, le mot *sund* a été rencontré une fois seulement dans le corpus de texte. Comme c'est le mot original, cela suffit pour qu'il conserve une probabilité assez élevée.

– Le mot *an* est corrigé en *an* au lieu de *and*.
Examinons les suggestions et leurs probabilités
  1. an (2047.0)
  2. and (1237.0)
  3. a (843.5)
  4. in (640.0)
  5. as (306.3)

  Le mot original *an* est un mot existant, et qui plus est ayant une forte fréquence. La suggestion (correcte) *and* est à nouveau proposée en deuxième, du fait de sa forte fréquence. Les mots *a*, *in* et *as* sont tous trois également à une distance de 1, et sont donc classés par fréquence.

– Le mot *signifin* est corrigé en *signify* au lieu de *signifying*.
Examinons les suggestions et leurs probabilités
  1. signify (0.00175)
  2. signifying (7.5E-4)
  3. signifies (6.25E-4)
  4. signified (6.25E-4)
  5. signifier (2.5E-4)

  Le mot original *signify* sort en premier car il est plus proche du mot original. La bonne correction *signifying* est encore une fois en deuxième position, suivi par trois autres propositions toutes à une distance de 2.

Examinons un autre exemple, en français cette fois :

FIGURE 3 – Un deuxième test, qui illustre un phénomène intéressant

La phrase entrée est

Il faut tumours aller en cours d'apprentissage artificiel.

Notre programme la corrige correctement ainsi :

Il faut toujours aller en cours d'apprentissage artificiel.

Cet exemple est intéressant car la correction *toujours* est choisie alors qu'elle est à une distance de 2, alors que la correction *tumeurs* est mise de coté malgré le fait qu'elle soit plus proche du mot original, avec une distance de 1 seulement. C'est le modèle de langage qui a permis, non pas de lever une ambiguïté entre deux mots aussi proches, mais même de prendre la décision de **favoriser un mot moins proche mais beaucoup plus fréquent**. Ce comportement est plus efficace que de simplement choisir la suggestion la plus proche.

4.2 Tests à grande échelle

4.2.1 L'algorithme de test

Nous allons généraliser ce procédé de test en générant automatiquement des fautes d'orthographe via une fonction `modifier` et en les corrigeant. On observe alors le taux de corrections correctes avec la fonction `evaluer`.

Le code-source du test est le suivant :

```
/**
 * Effectue une modification sur un mot, pour generer une faute
 * @param mot
 * @return
 */
public String modifier(String mot) {
    int i = (int)(Math.random()*mot.length());
    String debut = mot.substring(0, i);
    String fin = mot.substring(i, mot.length());
    int lettre = (int)(Math.random()*alphabet.length());

    switch((int)(Math.random()*4)) {
    case 0: // Supression
        return debut + fin.substring(1);
    case 1: // Transposition
        if (fin.length() > 1)
            return debut + fin.charAt(1) + fin.charAt(0) + fin.substring(2);
        else return modifier(mot);
    case 2: // Mutation
        return debut + alphabet.charAt(lettre) + fin.substring(1);
    case 3: // Insertion
        return debut + alphabet.charAt(lettre) + fin;
    default:
        return mot;
    }
}

/**
 * Evalue la performance de l'algorithme de correction
 * @param texte Un texte
 * @param nombreMotsMax Le nombre de mots a traiter au maximum
 * @param biais Le nombre de fois qu'on considere avoir deja rencontre un mot
    nouveau
 */
public void evaluer(String texte, int nombreMotsMax, int biais) {
    int n = 0, fautes = 0, inconnus=0;
    for (String mot:mots(texte)) {
        if (mot.length() > 4) {
            mot = mot.toLowerCase();
            n++;
            if (biais >0)
                modele.put(mot, modele.get(mot)+biais);
            String faute = modifier(mot);
            String correction = corriger(faute);
            if (!correction.equals(mot)) {
                fautes++;
                if (!modele.containsKey(mot)){
                    inconnus++;
                    //System.out.println(mot + " inconnu !");
                }
            }
            if (n>=nombreMotsMax)
```

```
52                      break;
53                  //System.out.println(faute + " ->" + correction + "(" + modele.get(
                        correction) + "), " + mot + " attendu (" + modele.get(mot) + ")")
                        ;
54              }
55          }
56          System.out.println(fautes + " fautes sur " + n + " mots, avec " + inconnus +
                " inconnus (" + (100*fautes)/n + "%, " + (100*(fautes-inconnus))/(n-
                inconnus) + "%)");
57      }
```

4.2.2 Tests en français

Nous essayons d'abord sur une œuvre du corpus d'apprentissage, *L'argent* de Émile Zola.
On obtient le résultat suivant, sur les 1000 premiers mots :

```
120 fautes sur 1000 mots, avec 0 inconnus (12%)
```

On a donc **88% d'efficacité**, ce qui est assez satisfaisant. Cependant, cela est biaisé par le fait qu'on utilise la base dont on s'est servi pour l'apprentissage. En particulier, tous les mots lus étaient déjà connus du modèle de langage (forcément...)
Essayons avec un test plus sérieux sur une oeuvre n'appartenant pas au corpus d'apprentissage, *Notre Dame de Paris*, de Victor Hugo.

```
174 fautes sur 1000 mots, avec 34 inconnus (17%)
```

On tombe à **83% d'efficacité**, ce qui est logique. En effet, de nombreux mots du texte n'avaient jamais été rencontrés auparavant. En particulier, on retrouve bien sûr un grand nombre de noms propres, de lieux et de personnages. Mais pas seulement, des mots courants du français n'avaient également jamais été vus auparavant.

Le problème semble donc provenir du modèle de langage, qui est incomplet. Pour vérifier cette hypothèse, on ajoute un **biais** au programme, afin que lorsqu'on rencontre un mot inconnu, on simule l'avoir déjà vu plusieurs fois (100 fois par exemple). Ce procédé donne le résultat suivant :

```
111 fautes sur 1000 mots, avec 0 inconnus (11%)
```

Le taux de réussite remonte à **89% d'efficacité** et on observe des performances semblables au test sur une œuvre du corpus, ce qui est logique puisqu'on simule le fait que tous les mots soient déjà connus.

4.2.3 Tests en anglais

On essaie enfin sur des œuvres anglaises.
D'abord sur une œuvre du corpus : *Frankenstein or The Modern Prometheus*, de Mary Shelley.

```
129 fautes sur 1000 mots, avec 0 inconnus (13%)
```

On a un résultat de **87% d'efficacité**, proche de l'exemple du corpus français. On tente ensuite une nouvelle œuvre, *The Picture of Dorian Gray* de Oscar Wilde.

```
172 fautes sur 1000 mots, avec 41 inconnus (17%)
```

On obtient **83% d'efficacité**, à nouveau les performances baissent. Comme précédemment, certains mots rencontrés étaient inconnus (des noms propres, pour la plupart), et on améliore virtuellement le modèle de langage en simulant qu'on connait tous les mots avec un biais.

```
53 fautes sur 1000 mots, avec 0 inconnus (5%)
```

On obtient alors **95% d'efficacité** ce qui est très élevé, mais ce résultat est bien entendu faussé par le biais. En effet, même si l'on améliorait le modèle de langage pour qu'il connaisse plus de mots, cela augmenterait également le nombre occurrences des mauvaises corrections, ce que ne fait pas notre système de biais.

5 Conclusion

Ce projet nous a permis de constater que les classifieurs naïf de Bayes, bien que très simples en théorie, font preuve d'une efficacité redoutables pour traiter certains problèmes d'apparence compliquée.

Notre programme a un taux d'efficacité proche de supérieur à 85% en moyenne, ce qui est plutôt satisfaisant compte tenu des nombreuses hypothèses réductrices posées dans notre étude.

De nombreuses pistes d'améliorations semblent envisageable :

- Le modèle d'erreur est plutôt simpliste, et il y a de nombreux moyen de l'améliorer, par exemple pour prendre compte de la distance entre les touches du clavier. Ainsi, il est plus probable qu'un E soit remplacé par une touche voisine comme le Z plutôt que par un M qui est à l'opposé du clavier.
- Nous aurions aussi pu améliorer notre modèle pour qu'il prenne en compte le contexte d'un mot. En effet, en lisant le mot *desert* on hésite entre les corrections *désert* et *dessert*. L'incertitude peut être levée si on peut apprendre à repérer dans le contexte de la phrase des mots tels que *sucré*, *délicieux* ou bien *sable*, *chaleur* et en déduire des corrélations par des tests du ξ_2
- Le modèle actuel ne prend pas en compte les règles de grammaire, qu'il serait très utile d'implémenter. Il paraît même envisageable, quoique compliqué, d'utiliser des algorithmes d'apprentissage pour déduire de la lecture d'un corpus de texte certaines règles de grammaire