Introduction to Computational Linguistics
Session 3: Writing Aids

Stephen Bodnar

Universität Tübingen

November 15, 2023

Assignment 1

- How is it going ? Questions ?
- Please upload submission on Moodle before November 22, 2023, 16:10
- Submit answers in a single .pdf file
- In case of questions: please use the moodle discussion forum
- Don't wait too long to start!

Reading Assignment

- Language & Computers - Chapter 2

Plan

1. Types of Writing Errors
2. A first practical application: Automatic Spell Checking
3. References

Introduction

"I don't see any use in having a uniform and arbitrary way of spelling words. We might as well make all clothes alike and cook all dishes alike."
- Mark Twain (http://www.twainquotes.com/Spelling.html)

Spelling is a convention we all usually follow, but not always

- Regional differences: *colour* vs *color*
- Literature:
  - "...there wos other genlmen come down" (Charles Dickens' Bleak House)
- Text messages: " I dunno, cos the bus is l8?"

Question

Why is standardized spelling useful?

Standardised Spelling's Usefulness

Some reasons:

- Communicating across groups speaking different pronunciations / dialects / regional differences
- Easy lookup of words in dictionaries (e.g. itinerary vs. aitinerary vs. etinerary)
- Meaning more directly accessible through the 'shape' of words (vocalisation step unnecessary)
- Facilitates automatic processing, e.g. optical character recognition, search engines

Non-word errors

Non-word errors

- Example: “hte” instead of “the”
- String of characters that does not exist as a word in the language
- Different reasons:
  - typographical error: error in typing, person knows how to type it correctly
  - spelling confusion: person lacks knowledge for correct spelling
- Different keyboard layouts

Non-word Errors and Intransparent Writing Systems

Intransparent Writing Systems: Often no direct correspondence between phonological and character representation, e.g.

- silent characters (“knight”)
- similar-sounding characters (“s”/‘c”/“z”)
- long/short vowels (“receive”)
- doubled consonants (“application”)

Other Non-word Error Types

- false friend/transfer error
  e.g. “brave” (EN) vs. “brav” (DE)
- split errors (“auto matic”)
- run-on errors (“machinelearning”)
- space in wrong place (“atoll way” vs. “a toll way”)

$\rightarrow$ exact cause of error difficult to determine

Describing Non-Word Errors

Spelling errors resulting in non-words can be described in terms of operations:

- insertion (“applicaition”)
- deletion (“brekfast”)
- substitution (“artificiel”)
- transposition (“autoamtic”)

Single vs. Multi-Error Misspellings

Can distinguish between:

- single-error misspelling
- multi-error misspelling (multiple spelling errors in one word)

Trends

- around 80% single-error misspellings
- most misspellings differ in length from target by at most two characters
- first letter most often spelled/typed correctly

Real-Word Errors (only a glance)

- local syntactic error
  Their is a problem.
- long-distance syntactic error
  The next group of three hundred Tour de France cyclists are about to begin the race.
- semantic errors
  For the preparation of his studies, he usually borrows cooks from the library.
- repetitions
  This this is something you can often see in a text.

Introduction to Automatic Spell Checking

Aim: Detect and correct non-word errors

Examples: Microsoft Word, but also Linux's *ispell* program

Key steps:

1. error detection
2. generation of candidate corrections
3. ranking of candidate corrections

Information Sources For Automatic Spell Checking

Different tasks (detection, generation, ranking) employ different information sources:

- Knowledge-based approaches (hand-crafted)
- Statistical approaches (data-driven)

Examples of errors

- *I sat down and opened hte book.*
- *That factory's speciality is wooden boets; they are nice but they cost a lot.*

When would each approach be useful ?

Detecting Non-Word Errors

Fairly straightforward

- Search for user input in a trusted word list, e.g., aspell http://wordlist.aspell.net/dicts/

Generating Candidate Corrections

Use rule-based approaches to suggest corrections:

- Modeling of frequent errors: “hte” instead of “the”
  e.g. LanguageTool

Search through well-formed word list for similar character sequences

- Relies on *similarity measures*: function that quantifies similarity between any two strings.

Defining similarity

- Imagine we want to build a spell checker, and that we need to invent a similarity measure
- Let's look at three proposals ...

Defining word similarity - Proposal 1

What happens if we define word similarity as word length?

- low difference in word length => word is similar

Can try out some examples

1. calcDifInLen('fyr', 'fry') = 0
   low size difference => similar?
2. calcDifInLen('firetruck', 'fry') = 4
   large size difference => dissimilar
3. calcDifInLen('zzz', 'fry') = 0
   low size difference => similar???

OK, so that definition doesn't work, but let's try another.

Defining word similarity - Proposal 2

What happens if we define word similarity as number of differences?

- loop over characters in the two words; count positions with unmatching characters
- more mismatches => less similar

Examples

1. countDifChars('fyr', 'fry') = 2
2. countDifChars('zzz', 'fry') = 3
3. countDifChars('firetruck', 'fry') = 8
4. countDifChars('fretruck', 'firetruck') = 8
5. countDifChars('ffiretruck', 'firetruck') = 9

Results:

- 1) and 3) are fine, 2) is an improvement over length definition
- BUT 4) and 5) have many differences due to shift, poor result

Conclusion: Not a good measure either.

Observations from proposed definitions

Requirements for similarity measure

- should reflect the similarity of contents
- needs to be robust against slight misalignments

Let's now look at the popular Minimum Edit Distance measure.

Defining word similarity - Minimum Edit Distance

Define similarity as the number of edit operations needed to transform a non-word into a dictionary word.

- more edit operations needed $=>$ less similar
- seems flexible, like it would align with how humans judge word similarity
- but how to program? not as straightforward as previous proposals

Minimum Edit Distance - Programming Ideas

How to program edit distance calculation?

- For inspiration, can look at the fine-grained steps a human typist would perform.
- Example: non-word 'fyr' and dictionary word 'fry'
- Typist A might
  1. put the cursor between the 'f' and the 'y' and type an 'r' key (insert)
  2. move to cursor to the end of the word and remove the 'r' character (delete)
- Typist B might
  1. delete first 'y' character
  2. insert a 'y' at the end

Minimum Edit Distance - Many Possibilities

In fact, many many possible sequences for transforming ‘fyr’ into ’fry’:

- Another (longer) possibility:
  1. placing cursor after ’f’ character,
  2. delete two characters
  3. insert ’r’ and ’y’
- A long but important sequence:
  1. delete all characters in ’fyr’
  2. insert all characters in ‘fry’

As the textbook points out, can even imagine absurd sequences, like placing the cursor at ’f’, then inserting ’f’ followed by deleting ’f’ 100 times.

Summarising so far

Key points:

- multiple ways to transform 'fyr' into 'fry'; some require more edit operations than others
- of the many possible ways, we are interested in the shortest sequence of operations that yields 'fry'
- to find shortest sequence, we need to explore all reasonable sequences of edit operations that lead to 'fry'
- also need to avoid exploring ridiculous sequences

Need way to set up problem that guides search behaviour.

Directed Acyclic Graphs - A Useful Tool

Directed Acyclic Graphs (DAGs) are a useful tool for thinking about problems with sequences

- Nodes represent the current state of a sequence
- The arcs represent actions

Example: a simple todo list

- Before starting -> Important email sent -> Programming task unfinished -> Relaxing at home
- First '->' arc represents action of writing of an email

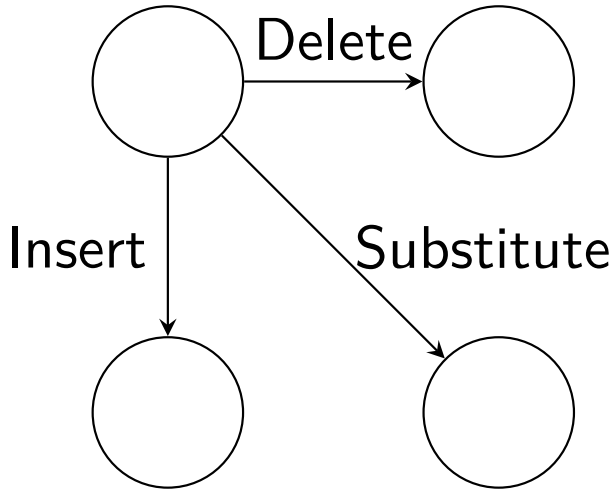
Important: DAGs can be effectively translated into computer programs

Slides that follow based on Dickinson, Brew, & Meurers (2013)

Applying DAGs to edit distance calculation

We can represent a sequence of edit operations

- Each node is the state of the word at a particular point during transformation
- Each arc is an edit operation (delete, insert, substitute) with cost = 1
  - Exception: substituting same character costs 0.



Representing full edit distance calculation with a DAG

Use example of transforming ‘fyre’ into ‘fry’

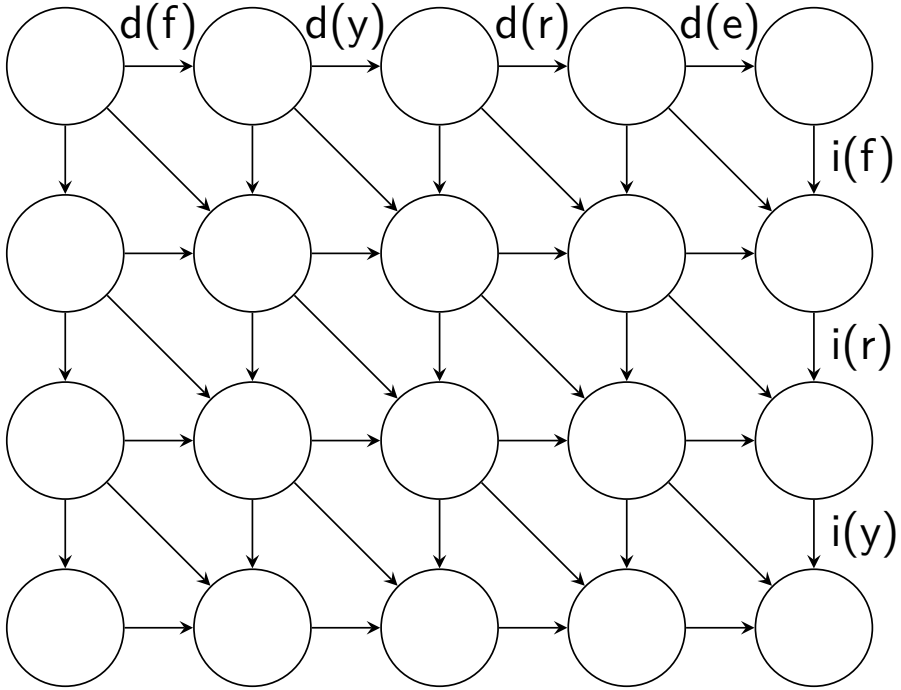
- First, add start node representing state ‘fyre’ (top left)
- Next, add end node representing state ‘fry’ (bottom right)
- Then, add all possible paths from start to end.
  - Can begin by adding the ‘important’ sequence from above:
    four deletes followed by three inserts
    fyre -> yre -> re -> e -> ‘’ -> f -> fr -> fry
  - Can follow by adding three inserts followed by four deletes
    fyre -> ffyre -> frfyre -> fryfyre -> fryyre -> fryre -> frye -> fry
  - Question:
    What edit operation hasn’t been featured in any of our paths yet?

Representing full edit distance calculation with a DAG - cont.

- Paths added so far
  1. Four deletes followed by three inserts
     fyre -> yre -> re -> e -> '' -> f -> fr -> fry
  2. Three inserts followed by four deletes
     fyre -> ffyre -> frfyre -> fryfyre -> fryyre -> fryre -> frye -> fry
  3. Three substitutions followed by a delete
     fyre -> fyre -> frre -> frye -> fry
  4. Can continue adding arcs in to include all possible edit sequences

Computing edit distances - An example

- Four deletes followed by three inserts for transforming *fyre* to *fry*



Computing the minimum edit distance

Very first solution to this problem was recursive one

```
1  public int calc_cost(String userInput, String dictionaryCandidate) {
2
3    if(userInput.length == 0){
4      return dictionaryCandidate.length
5    }
6    else if(dictionaryCandidate.length == 0){
7      return userInput.length
8    }
9    else {
10     var costToReachNewNode =
11             0 if userInput[0] == dictionaryCandidate[0] else 1
12
13     return costToReachNewNode + minimum(
14         // explore delete operation
15         calc_cost(userInput[1:]), dictionaryCandidate),
16
17         // explore insert operation
18         calc_cost(userInput, dictionaryCandidate[1:])),
19
20         // explore substitution operation
21         calc_cost(userInput[1:], dictionaryCandidate[1:])
22     )
23   }
24 }
```

Computing the minimum edit distance - Naive

- Traverses all paths in DAG, calculates cost of shortest path
- Re-computes same paths multiple times => inefficient
- Can memoize recursive implementation for more efficiciency (see accompanying file)
- Exists also an iterative function to traverse the DAG (for those interested)

Steps involved in spell checking

1. error detection
2. generation of candidate corrections
3. $=>$ ranking of candidate corrections

Spell Checkers - Ranking Stage

Goal: rank good candidates by how well they fit the context

- context means adjacent language units
- today we look at word context
- in ranking stage we rely on statistical knowledge sources

Candidate Ranking - Word Context

Example

- *That factory's speciality is wooden boets; they are nice but they cost a lot.*
- What suggestions should our spell checker offer for *boets*?
- How to decide between these candidates ?
- We need a resource that we can ask questions like "Given this sentence, which of these N words fits best?"

Candidate Ranking - Language Model

- Problem can be solved to some degree with a Language Model
  - Models the likelihood of word sequences
    - Likelihood(*That factory's speciality is wooden* ***boots****; they are nice but they cost a lot.*) = relatively low
    - Likelihood(*That factory's speciality is wooden* ***boats****; they are nice but they cost a lot.*) = relatively high
  - Very often implemented with N-grams
  - More later in the course!

References and Acknowledgments

These slides are based on slides for Intro to CL, 2021 by Dr. Björn Rudzewitz.

The contents and structure of the slides are based on the book "Language and Computers" by Chris Brew, Detmar Meurers, and Markus Dickinson.