Energy Informatics

System Design — Data Analysis

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

11 Feb 2016

Second application
Text analysis

Second application

Statistical analysis on public texts

- Obtain a public domain text
  - Gutenberg project
  - Wikipedia (very large)
  - public corpora (e.g., `https://en.wikipedia.org/wiki/Brown_Corpus`)
- Possible tasks
  - Which language?
  - Which genre?
  - Which author?

Possible tasks

Which Language?

- Every language has a characteristic letter frequency
- https://en.wikipedia.org/wiki/Letter_frequency
- Also bigrams, trigrams, and quadgrams may be analyzed
- http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/

Possible tasks

Which Language?

- Every language has a characteristic letter frequency
- `https://en.wikipedia.org/wiki/Letter_frequency`
- Also bigrams, trigrams, and quadgrams may be analyzed
- `http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/`

Which genre / author?

- Analyze usage patterns of common words
- `https://en.wikipedia.org/wiki/Most_common_words_in_English`

On letter frequency and cryptanalysis

Background: substitution cipher

- Plain text and cipher text (after encryption) are drawn from the same set of symbols
- A (monoalphabetic) **substitution cipher** is a one-to-one mapping between symbols
- Particularly simple example: Caesar's cipher, which rotates letters by 13 (how would you decrypt?)

Example: Caesar's cipher

Caesar's substitution

symbols	abcdefghijklmnopqrstuvwxyz
substitutes	nopqrstuvwxyzabcdefghijklm

Example: Caesar's cipher

Caesar's substitution

symbols	`abcdefghijklmnopqrstuvwxyz`
substitutes	`nopqrstuvwxyzabcdefghijklm`

Application

plain text	`we had goldfish and they circled around`
cipher text	`jr unq tbyqsvfu naq gurl pvepyrq nebhaq`

On letter frequency and cryptanalysis

Breaking a substitution cipher

- Assumptions:
  - language is known
  - cipher text is sufficiently long
- Analyze letter frequency
- Match with letter frequency table for the language
- Compute inverse substitution

Which substitution is the best match?

- To assess different substitutions, we need to compute the distance to the language's letter frequency.

Which substitution is the best match?

- To assess different substitutions, we need to compute the distance to the language's letter frequency.
- The standard distance function for vectors $\bar{x}$ and $\bar{y}$ computes the square root of the squares of the differences:

$$d(\bar{x}, \bar{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$

Which substitution is the best match?

- To assess different substitutions, we need to compute the distance to the language's letter frequency.
- The standard distance function for vectors $\bar{x}$ and $\bar{y}$ computes the square root of the squares of the differences:

$$d(\bar{x}, \bar{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$

- The best solution/substitution is the one with smallest distance.

Vector distance in Python

Code

```
def distance(xs, ys):
    s = 0
    for x, y in zip (xs, ys):
        s += (x - y) * (x - y)
    return math.sqrt(s)
```

Explanation

- `zip (xs, ys)` creates a list of pairs `(x,y)` of corresponding entries from the lists `xs` and `ys`
- `for x, y in sequence`
  loops over the entries in `sequence`, which must be pairs, and binds `x` and `y` to the first and second component of each pair, respectively

Fitness

Alternatively

We could compute a fitness function that multiplies the probabilities of the actually occurring characters (or bigrams, or trigrams, etc).

Fitness

Alternatively

We could compute a fitness function that multiplies the probabilities of the actually occurring characters (or bigrams, or trigrams, etc).

A practical caveat

- All probabilities are significantly less than one.
- Probabilities for single letters are betwen 0.07% (z) and and 12.7% (e) with a geometric mean of 2.07%
- Multiplying many very small numbers leads to underflow!

Products of very small numbers

```
>>> # for a text with 10 letters
>>> 0.02 ** 10
1.0240000000000003e-17
>>> # for a text with 100 letters
>>> 0.02 ** 100
1.267650600228232e-170
>>> 0.02 ** 150
1.4272476927059644e-255
>>> 0.02 ** 180
1.5324955408658946e-306
>>> 0.02 ** 190
1.5e-323
>>> 0.02 ** 200
0.0
>>> # oops!
```

On floating point numbers

Range of floating point numbers

A double precision binary floating-point number has a coefficient of 53 bits, an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10^{-308} to 10^{308}, because the range of the exponent is $[-1022, 1023]$ and 308 is approximately $\log_{10}(21023)$.
`https://en.wikipedia.org/wiki/Floating_point`

Workaround: Use logarithms!

Recall

- $\log(a \cdot b) = \log(a) + \log(b)$
- If $0 < a < 1$, then $\log(a) < 0$.
- $\log(a) < \log(b) \Leftrightarrow a < b$

Logarithmic fitness function

- Compute the **sum** of the logarithms of the probabilities of all letters (bigrams, trigrams, etc).
- Get a result < 0.
- Highest log-probability corresponds to the highest probability!

Useful Python I/O idioms

Python IO

Reading a file naively

```
# prepare to 'r'ead from file 'filename'
f = open('filename', 'r')
s = f.read()
# process s = content of file
f.close()
```

- Reads all of a file named "`filename`" into the string `s`
- Then work with `s`

UNI FREIBURG

Reading a file naively

```
# prepare to 'r'ead from file 'filename'
f = open('filename', 'r')
s = f.read()
# process s = content of file
f.close()
```

- Reads all of a file named “`filename`” into the string `s`
- Then work with `s`
- Problems:
  - This will consume **a lot** of memory if the file is big
  - It’s easy to forget to close the file
  - No error handling

More robust file handling

Reading a file (recommended)

```
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```

More robust file handling

Reading a file (recommended)

```
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```

Advantages

- No memory issues as file is read line-by-line
- Automatic close when leaving `with`
- (Hidden) error handling if there is a problem with the file

More robust file handling

Reading a file (recommended)

```
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```

Advantages

- No memory issues as file is read line-by-line
- Automatic close when leaving `with`
- (Hidden) error handling if there is a problem with the file

Disadvantage

Have to deal with file contents one line at a time

Example: the word count utility

```
# wc counts lines, words, and characters in a file
def exe(name):
    # initialization
    lcount = 0     # line count
    wcount = 0     # word count
    ccount = 0     # character count
    with open (name, 'r') as f:
        for line in f:
            # process one line
            lcount += 1
            ccount += len(line)
            for words in line.split():
                wcount += 1
    return (lcount, wcount, ccount)
```

First application extended
Analysis of **voting patterns**

Scaling up

- We just looked at one ballot

Scaling up

- We just looked at one ballot
- Can hardly be called a voting pattern!

Scaling up

- We just looked at one ballot
- Can hardly be called a voting pattern!
- Using a spreadsheet is ok to analyze one data set

Scaling up

- We just looked at one ballot
- Can hardly be called a voting pattern!
- Using a spreadsheet is ok to analyze one data set
- But in 2015, there were 56 such ballots

Scaling up

- We just looked at one ballot
- Can hardly be called a voting pattern!
- Using a spreadsheet is ok to analyze one data set
- But in 2015, there were 56 such ballots
- Solution: program!

Reading xls in Python

Install Python module `xlrd` and `pandas`

- (from shell)
- `pip install xlrd`
- `pip install pandas`

Analyzing xls in Python

Prepare for processing with pandas

```
import pandas
import xlrd
wb = xlrd.open_workbook (
          '20160128_2_xls-data.xls')
sheet = pandas.read_excel(book, engine="xlrd")
```

Analyzing xls in Python

Prepare for processing with pandas

```
import pandas
import xlrd
wb = xlrd.open_workbook (
          '20160128_2_xls-data.xls')
sheet = pandas.read_excel(book, engine="xlrd")
```

Dataframe

- `sheet` is a pandas *dataframe*
- `len (sheet)` get # rows
- Columns are named according to first row in xls file
- `sheet.columns` lists columns
- `sheet.index` lists row indexes

Consistency
for a single row

```
def check_consistency(row):
    ja   = row['ja']
    nein = row['nein']
    enth = row['Enthaltung']
    ung  = row[u'ungueltig']  # should be u umlaut
    na   = row['nichtabgegeben']
    valid = ((ja == 0 or ja == 1) and
             (nein == 0 or nein == 1) and
             (enth == 0 or enth == 1) and
             (ung == 0 or ung == 1) and
             (na == 0 or na == 1) and
             (ja + nein + enth + ung + na == 1))
    return valid
```

Consistency
for all rows

Pattern for processing dataframe

```
for idx, row in sheet.iterrows():
    # idx - row number
    # row - representation of the row
    #       indexed by column names
```

Consistency

for all rows

Pattern for processing dataframe

```
for idx, row in sheet.iterrows():
    # idx - row number
    # row - representation of the row
    #       indexed by column names
```

Example (continued)

```
invalid_set = set()
for idx, row in sheet.iterrows():
    if not check_consistency(row):
        invalid_set.add(idx)

print("invalid␣entries:␣" +
      str (len (invalid_set)))
print(invalid_set)
```

End Part IV