30 min

PyCon 2017

Portland, Oregon

- we are going to talk about fuzzy search.
- It's Approximate string matching
- Why? Sometime we don't really know what we are looking for

Fuzzy Search
Approximate string matching

- Why
    - Abbreviation: what does "LGTM" mean?
    - Misspellings: "I sware" or "I swear"?
    - Lack of whitespaces: BEDBATHANDBEYOND
    - Cut-off words
    - etc.

- Both US and USA point to the same entity: United States
- Fuzzy search usually based on some sort of string distance

- Applications
    - Entity resolution
        - "US" and "USA"
    - Search Engines
    - Auto correct
- Search based on String distance
    - Quantify qualitative data for analytical purpose

Soundex

- phonetic algorithm
- index by sound as pronounced in English
- assigns a soundex coding
- ideal for spelling inconsistencies

American Soundex Coding

http://www.archives.gov/research/census/soundex.html
(http://www.archives.gov/research/census/soundex.html)

every soundex code is a letter and three numbers

Number	Letter
1	B,F,P,V
2	C,G,J,K,Q,S,X,Z
3	D,T
4	L
5	M,N
6	R

Ignore A,E,I,O,U,H,W,Y

In [ ]:

```
import jellyfish as j

a=j.soundex('PORTLAND')
print(a)
b=j.soundex('PRTLAND')
print(b)
```

In [ ]:

```
# Numbers and Letters
a=j.soundex('Word #123')
print(a)
b=j.soundex('Word')
print(b)
```

In [ ]:

```
a=j.soundex('accept')
print(a)
a2=j.soundex('except')
print(a2)
```

In [ ]:

```
#homophones
a=j.soundex('ado')
print(a)
a2=j.soundex('adieu')
print(a2)
```

In [ ]:

```
a=j.soundex('forth')
print(a)
a2=j.soundex('fourth')
print(a2)
```

Soundex with PostgreSQL

https://www.postgresql.org/docs/9.1/static/fuzzystrmatch.html (https://www.postgresql.org/docs/9.1/static/fuzzystrmatch.html)

```
CREATE EXTENSION fuzzystrmatch;
```

In [ ]:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://jiaqi@localhost/pycon')
connection = engine.connect()
```

In [ ]:

```
from sqlalchemy.sql import text

query = "select soundex('Anne'), soundex('Ann'), difference('Anne','Ann')"
res = engine.execute(text(query))
res.fetchall()
```

Create Table & Load Data

```
Create table pypi
(packages varchar, description varchar);

COPY pypi from 'pypi.csv' delimiter ',' csv;
```

In [ ]:

```
query = """
        SELECT    packages
        FROM      pypi
        WHERE     soundex(packages) = soundex('fuzzysearch')"""
res = engine.execute(text(query))
res.fetchall()
```

In [ ]:

```
query = """
        SELECT    description
        FROM      pypi
        WHERE     soundex(description) = soundex('fuzzysearch')"""
res = engine.execute(text(query))
res.fetchall()
```

In [ ]:

```
query = """
        SELECT    description
        FROM      pypi
        WHERE     difference(description, 'fuzzysearch') > 2 limit 10"""
res = engine.execute(text(query))
res.fetchall()
```

Soundex

- Soundex is pretty easy to implement
- Computationally fast
- only works on ASCII characters (no foreign languages)
- How do you calculate distance

Levenshtein distance

- also call edit distance
- accounts for how many characters you have to change to have the same string
- computationally fast (can handle real time processing)

- pairwise comparison

In [ ]:

```
import Levenshtein as l

l.distance('SMYTHE', 'SMITH')
```

In [ ]:

```
l.distance('pypi', 'pypy')
```

Pitfall: Comparing Addresses

In [ ]:

```
str99 = '99 Broadway'
str100 = '100 Broadway'
str999 = '999 Broadway'

l.distance(str99, str100)
```

In [ ]:

```
l.distance(str99, str999)
```

In [ ]:

```
l.distance(str999, str100)
```

Longer Strings

In [ ]:

```
str1='Mike\'s New York Deli and co'
str2='Sam\'s New York Deli and co'
l.distance(str1,str2)
```

In [ ]:

```
import Levenshtein as l
str1='Mike\'s Deli'
str2='Sam\'s Deli'
l.distance(str1,str2)
```

Levenshtein

- counting raw edits penalizes long strings: use a ratio of edits to length
- weighing numbers differently from letters

N-grams

In [ ]:

```
def ngram(tokens, n):
    grams =[tokens[i:i+n] for i in range(len(tokens)-(n-1))]
    return grams
```

In [ ]:

```
sentence_gram = "The quick brown fox jumped over a lazy dog".split()
grams = ngram(sentence_gram, 3)

for gram in grams:
    print(gram)
```

In [ ]:

```
word_gram = "pycon2017"
grams = ngram(word_gram, 3)

for gram in grams:
    print(gram)
```

Scoring Similarity: Jaccard similarity

intersection over the union

In [ ]:

```
def get_sim(a_tri,b_tri):
    intersect = len(set(a_tri) & set(b_tri))
    union = len(set(a_tri) | set(b_tri))
    return float(intersect)/(union)
```

In [ ]:

```
print(grams)
get_sim(grams, grams)
```

In [ ]:

```
a_gram = ngram('aabcccdeeeffgghhh', 3)
b_gram = ngram('abcccdeeffgghhh', 3)
get_sim(a_gram, b_gram)
```

Trigam Search with Postgres

https://www.postgresql.org/docs/9.1/static/pgtrgm.html (https://www.postgresql.org/docs/9.1/static/pgtrgm.html)

```
create extension pg_trgm;
```

In [ ]:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://jiaqi@localhost/pycon')
connection = engine.connect()
```

In [ ]:

```
query_des="""
    SELECT
                a.description,
                similarity(lower(a.description), :descript) as similarity
    FROM        pypi as a
    WHERE       lower(a.description) % :descript
    ORDER BY    similarity DESC"""
```

In [ ]:

```
from sqlalchemy.sql import text

description = 'fuzzy search'
res = engine.execute(text(query_des), descript=description)
```

In [ ]:

```
descriptions = res.fetchall()
descriptions
```

In [ ]:

```
query_package="""
    SELECT
                a.packages,
                similarity(lower(a.packages), :p) as similarity
    FROM        pypi as a
    WHERE       lower(a.packages) % :p
    ORDER BY    similarity DESC"""
```

In [ ]:

```
pk = "fuzzysearch"
res = engine.execute(text(query_package), p=pk)
packages = res.fetchall()
packages
```

In [ ]:

```
for elem in packages:
    if elem[1]>=0.5:
        print(elem)
```

heuristics = for both fields over heuristics - well different situations have differnet codes - statistical model leveraging the trigram score as a feature (back to slides here)

Other Similarity Metrics

- NLTK: wordnet
- Word2Vec: uses cosine distance
  - cosine distance between two vectors

In [ ]:

```
from nltk.corpus import wordnet
```

In [ ]:

```
word1 = wordnet.synsets("blue")
word2 = wordnet.synsets("green")
```

In [ ]:

```
word1[0].wup_similarity(word2[0])
```

In [ ]:

```
#sample data set from: http://mattmahoney.net/dc/text8.zip
import word2vec
word2vec.word2phrase('text/text8', 'text/text8-phrases', verbose=True)
word2vec.word2vec('text/text8-phrases', 'text/text8.bin', size=100, verbose=
```

...

In [ ]:

```
import word2vec
model = word2vec.load('text/text8.bin')
model['coffee']
```

In [ ]:

```
def get_similar_words(word):
    indexes, metrics = model.cosine(word)
    return model.generate_response(indexes, metrics).tolist()
```

In [ ]:

```
get_similar_words('coffee')
```

In [ ]:
```
indexes, metrics = model.analogy(pos=["coffee", "night"], neg=["day"], n=10)
model.generate_response(indexes, metrics).tolist()
```

In [ ]:
```
#analogy
#sun + cold - warm
indexes, metrics = model.analogy(pos=["sun", "cold"], neg=["warm"], n=10)
```

In [ ]:
```
model.generate_response(indexes, metrics).tolist()
```