Working with Twitter

A short lecture

Agenda

- Introduction
- Data Acquisition
- Geospatial Data
- Working with JSON
- Density and Distribution
- Text Mining from Twitter Data
- Summary

Setup your computer

In order to follow this tutorial interactively, you need to setup your computer a bit. You will need

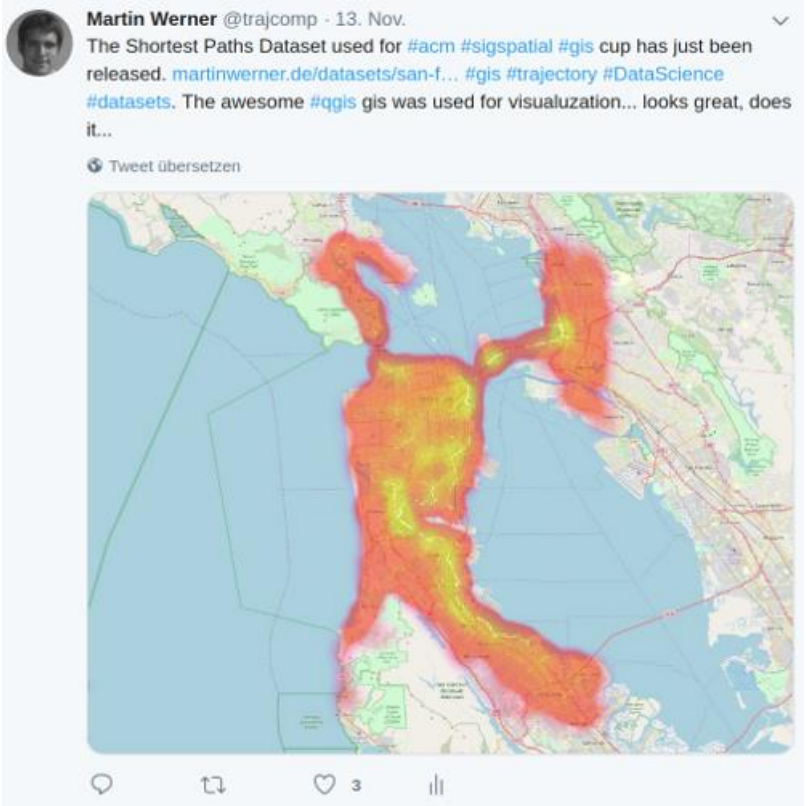
- a copy of jq in your path https://stedolan.github.io/jq/download/
- most preferably a bash shell and basic unix tools (mingw or git shell for Windows)
- the following data files downloaded into a folder of your computer sample-tweet.json and tweets.json.
- optional: QGIS if you have downloaded twitter data to visualize it geographically
- optional: python, pip, and tweepy if you want to stream twitter data yourself

Introduction

A tweet

Twitter in a nutshell:

- A **tweet** is a short message
- A **hashtag** is a word starting with a #. It is used to assign a topic to a tweet.
- A **mention** is a word starting with @ and is used to address a (public) message to a person or company
- A **follower** is someone who subscribed for updates from you
- A **like** is when someone clicks the heart below the tweet.
- A **retweet** is when a (possibly commented) copy of the tweet is send out



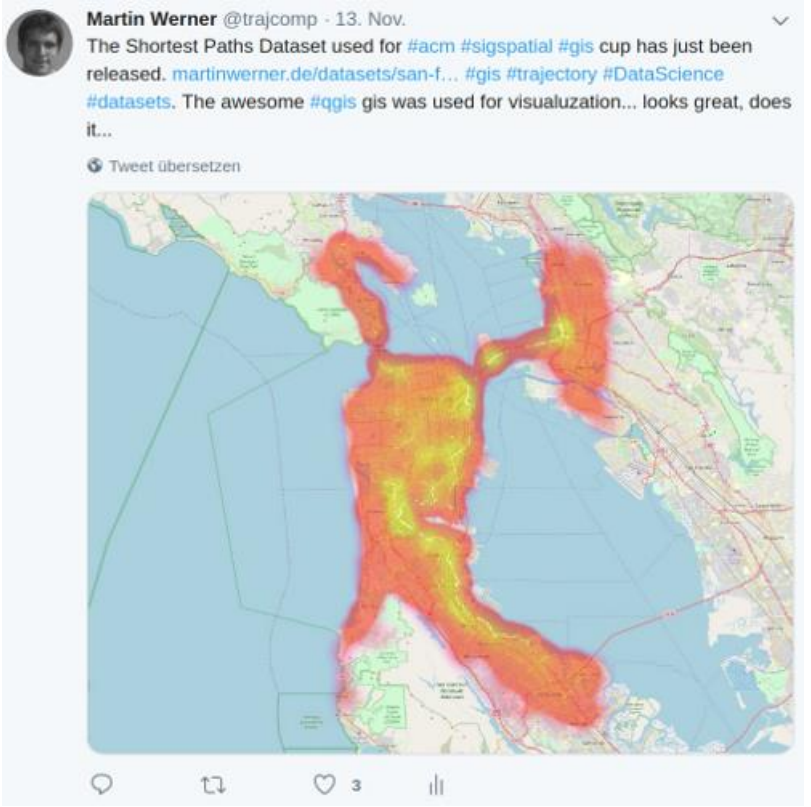
The network

A tweet object contains all of this information (redundantly at the time of API access).

You get it as a JSON object from the API

In other words as a **nested key-value data** structure

This **data structure** contains all information needed to render the tweet including lots of information on the account of the author.



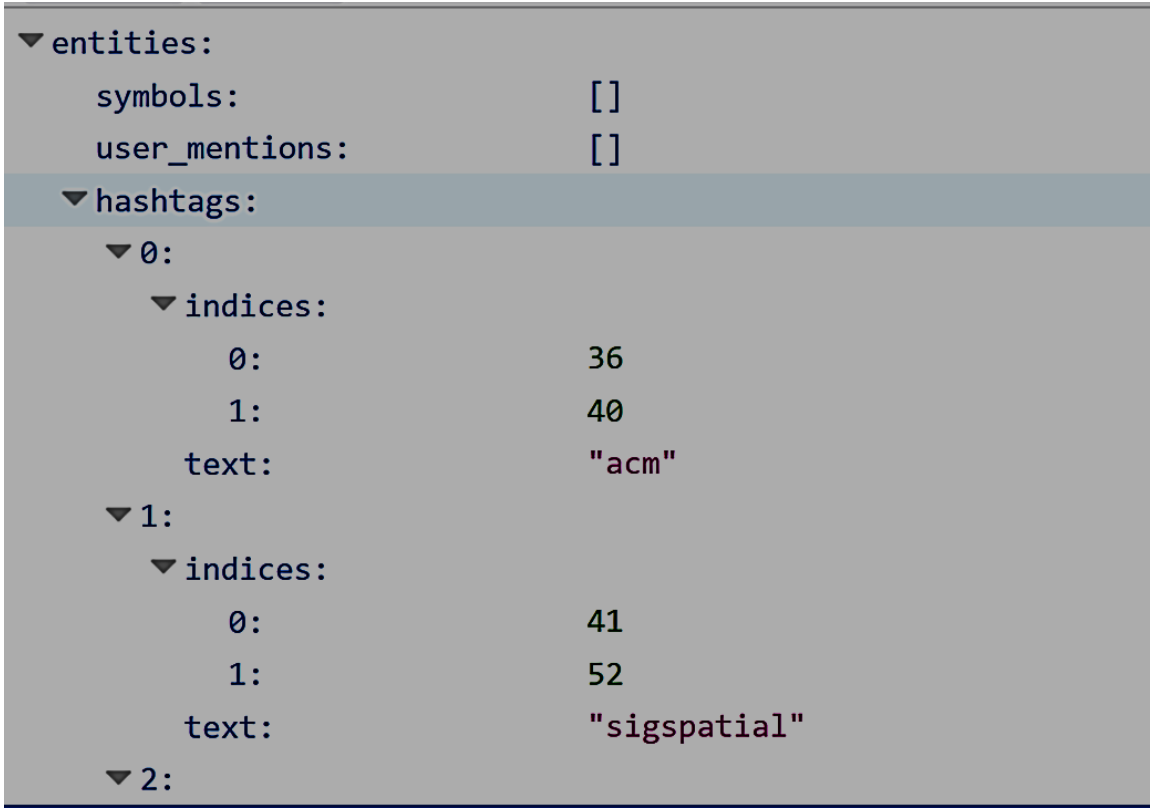
Twitter Data Objects

Key	Value
contributors	null
truncated	true
text	"The Shortest Paths Dataset used for #acm #sigspatial #gis cup has just been released. https://t.co/pzeEleBfu9 #gis… https://t.co/IF7z1WnUDk"
is_quote_status	false
in_reply_to_status_id	null
id	1062405858712272900
favorite_count	3
source	"Twitter for Android"
retweeted	false
coordinates	null
entities	{…}
in_reply_to_screen_name	null

Twitter Data Objects

Key	Value
in_reply_to_user_id	null
retweet_count	0
id_str	"1062405858712272898"
favorited	false
user	{…}
geo	null
in_reply_to_user_id_str	null
possibly_sensitive	false
lang	"en"
created_at	"Tue Nov 13 18:04:29 +0000 2018"
in_reply_to_status_id_str	null
place	null

Entities


entities:
symbols: []
user_mentions: []
hashtags:
0:
indices:
0: 36
1: 40
text: "acm"
1:
indices:
0: 41
1: 52
text: "sigspatial"
2:

Twitter Data Remarks

- each tweet has a **unique 64 bit unsinged ID** given as an integer (field id) and as a string (field id_str)
- each tweet has a **timestamp created_at** and though I created this tweet in Germany (GMT+1), it is stored in UTC (GMT+0) time zone. All tweets share **this timezone**. In this way, it is very easy to relate tweets to each other on a global scale, but more difficult to relate a tweet to the local time of day.
- the **language is estimated** by twitter
- Some **user account information** is embedded into the tweet. This is highly redundant, but very useful for web performance: A tweet object is designed to be sufficient to render the tweet with Javascript (e.g., create the view shown above).
- **hashtags** are isolated
- a field **truncated** has been introduced for compatibilty: when Twitter changed away from the short 140 character tweets to longer tweets, they made all APIs return a truncated version of all tweets that is short enough for the old API guarantee. If it is truncated, the field truncated tells us. In addition, the tweet might contain an additional field full_text, however, with different API options prooving that my client was aware of this new feature.

Let‘s get some tweets

Data Acquisition

Preparing for API Access

Twitter provides a nice and clean API and the first thing you will need is, well, a Twitter account.

- Then, as of July 2018, you must apply for a Twitter developer account and give some information on how you want to use the Twitter API
- Then, you need to create an app which provides you with credentials to use the API.

As this process is changing over time, just find it on Twitters web pages.

Keys and Tokens

Setting up the app gives you

- The Consumer Key (API Key)
- The associated Consumer Secret (API Secret)
- An Access Token
- An associated Access Token Secret

Each of those is an alphanumeric string.

Tip: Record in secret.env

Create a file **secret.env** similar to

```
#Access Token
TWITTER_KEY=274[...]M9b
#Access Token Secret
TWITTER_SECRET=WKS[...]1oI
#Consumer Key (API Key)
TWITTER_APP_KEY=8Co[...]Plt
#Consumer Secret (API Secret)
TWITTER_APP_SECRET=cEI[...]net
```

- Then you can easily access them from your programs and inside your containers, but they don‘t end up in the source code!

Streaming Twitter Data

Twitter provides two ways of accessing data

Query

- Ask for a certain hashtag, location, or object and get back a certain result set

Stream

- Create a filter (specification of what you are interested in) and get one tweet after another

Stream is better? Probably, but not for all…

Advantage of Streaming:

For spatial applications, I love hanging on the stream, because you get a temporal sample of the data which is not skewed towards temporal hotspots.

Downside of Streaming:

You need to operate a reliable system for getting the data (interruptions lead to missing time intervals in your sample)

Streaming in practice

You can rely on the tweepy library to manage the Twitter API from within python. It is simple and actively maintained. However, it is not ultimately stable…

You can as well develop your own API client using the documentation offered by Twitter, this can (could) be very stable…

Streaming Framework


```
auth = tweepy.OAuthHandler(os.environ['TWITTER_APP_KEY'],os.environ['TWITTER_APP_SECRET'])
auth.set_access_token(os.environ['TWITTER_KEY'], os.environ['TWITTER_SECRET'])
api = tweepy.API(auth)
stream_listener = StreamListener()
stream = tweepy.Stream(auth=api.auth, listener=stream_listener)

stream.filter(locations=[-180.0,-90.0,180.0,90.0])
```

This attaches you to the stream. You receive information in the way that the library will call certain functions on your object StreamListener, which you have to implement yourself.

Running on Linux looks like

```
$> source secret.env
$> python my-streamer.py
```

Streaming Details – A StreamListener


```
class StreamListener(tweepy.StreamListener):
    def __init__(self):
        self.outfile = open('tweets.json',"a+")
        tweepy.StreamListener.__init__(self);

    def on_status(self, status):
        tweet=json.dumps(status._json)
        print(tweet, file=self.outfile)


    def on_error(self, status_code):
              ... add proper error handling (like throwing an uncaught exception ;-)...
```

This very simple listener (not production ready!) opens a single file **tweets.json** for appending data and writes each tweet into this file.

Note that it does not contain any error checking
(which might or might not be a good idea)

Wrapup

- We have now three components
  - Secret.env with all the API details
  - Main.py implementing the tweepy client and StreamListener class
  - Hopefully a tweets.json to work with (downloaded from the API)

The real world

- **Problem:** Inevitable Faults
  - Library errors: We can‘t handle (while running)
  - API errors: We can‘t handle (while running)
  - Host errors: We can partially handle (but do we catch all)
  - Network errors: We could handle easily, but why?
- **Solution:**
  - Fail fast: throw exceptions (don‘t catch them) all over the place and restart your script quickly (but make sure, that you keep friendly with repeated fails – otherwise Twitter might ban you)
    - Rely on systemd, docker (with restart policy), or your own „shell“ to restart the script and take appropriate actions and delays (e.g., exponential delay in case of repeated error)

Let us manage tweets

Working with JSON

JSON

- JSON stands for JavaScript Object Notation
- JSON has become one of the central data representations on the Internet.
  - extendible,
  - human-readable
  - easy to write.
- It can be read by all major programming languages and has been around for a long time in the context of **RESTful services**.

Handling JSON

- The downside of JSON is the complexity of things you can model with it (including tweets).
- In contrast to traditional SQL or XML, tweets don‘t follow a specific schema
- Working with tweets can be done from any good programming language
  - Writing programs for simple operations is over-complicated
  - JSON can be complicated

JQ

Luckily, this problem has converged to a very nice query langauge and a command line tool called JSON Query Processor (JQ)

JQ Basics

JQ can be used for querying and pretty-printing JSON collections (that is files containing multiple JSON objects)

The most basic query matches everything and is expressed as“.”

- jq . tweets.json

- cat tweets.json | jq .

Where are the colors

JQ Expressing values

icaml$ **cat sample-tweet.json | jq true**

true

icaml$ **cat sample-tweet.json | jq false**

false

icaml$ **cat sample-tweet.json |jq 1.42**

1.42

icaml$ **cat sample-tweet.json | jq '"this is a string"'**

"this is astring"

Objects and Arrays

- Basically, JSON has two higher-order datatypes:
  - Objects
    - icaml$ cat sample-tweet.json | jq '{"key1":42,"key2":"a string"},
      {
        "key1": 42,
       "key2": "a string "
      }
  - Arrays
    - icaml$ cat sample-tweet.json |jq '[1,2,3,4],
      [
        1,
        2,
        3,
        4
      ]

Combined


```
icaml$ cat sample-tweet.json | jq '{"array":[1,2,4,8],"2d array":[[1,2],[3,4]],"nested
objects":{"key":"value"}}'
{
  "array": [
    1,
    2,
    4,
    8
  ],
  "2d array": [
    [
      1,
      2
    ],
    [
      3,
      4
    ]
  ],
  "nested objects": {
    "key": "value"
  }
}
```

Extracting Fields

- The dot operator (prepending a field name) selects elements from an object

```
icaml$ cat tweets.json |jq '.id_str'
"1062406263932444672"
"1062405858712272898"
"1036898465270444032"
"1034516701235372032"
"1027811999529529344"
[...]
```

Chaining…

- You can chain this operator. The second in the chain is applied to the result of the first
- .A.B is actually SELECT(B, SELECT(A,…))

```
icaml$ cat sample-tweet.json |jq .user.entities.url.urls
[
  {
    "url": "https://t.co/74ySSExk6l",
    "indices": [
      0,
      23
    ],
    "expanded_url": "http://www.martinwerner.de",
    "display_url": "martinwerner.de"
  }
]
```

Arrays and Brackets

- Bracket expressions are used to access arrays

```
icaml$ echo "[[1,2],[3,4]]" | jq '.[0]'
[
  1,
  2
]
icaml$ echo "[[1,2],[3,4]]" | jq '.[0][1]'
1
icaml$ echo "[[1,2],[3,4]]" | jq '.[1][0]'
2
icaml$ echo "[[1,2],[3,4]]" | jq '.[0][1]'
3
icaml$ echo "[[1,2],[3,4]]" | jq '.[1][1]'
4
```

Arrays and Brackets

- Unspecific brackets loop over the elements

```
icaml$ echo "[[1,2],[3,4],[5,6]]" | jq '.[][0]'
1
3
5
icaml$
```

Now with real tweets…

Applying what we learnt (and more)

Extract Hashtags

- Let us extract hashtags from a tweet object:
  - Loop over all hashtags with an unspecific bracket operation:

```
icaml$ cat sample-tweet.json |jq '.entities.hashtags[].text'
"acm"
"sigspatial"
"gis"
"gis"
```

Now with multiple tweets


```
icaml$ cat tweets.json | jq '.entities.hashtags[].text'
"acm"
"sigspatial"
"gis"
"gis"
"MyData2018"
"SpatialComputing"
"GISChat"
"DataScience"
"tutorial"
"Spark"
"AWS"
"Docker"
"spatial"
"analytics"
"DataScience"
```

Problem: A set of tweets results in a concatenation of the sets of hashtags each tweet contains. This might not be what we wanted.

Solution: Create a sequence of object instead of a sequence of strings!

Maintain the structure


```
icaml$ cat sample-tweet.json | jq '{"id":.id_str, "hashtag": .entities.hashtags[].text}'
{
  "id": "1062405858712272898",
  "hashtag": "acm"
}
{
  "id": "1062405858712272898",
  "hashtag": "sigspatial"
}
{
  "id": "1062405858712272898",
  "hashtag": "gis"
}
{
  "id": "1062405858712272898",
  "hashtag": "gis"
}
```

Calculating with JQ

- Of course, you can calculate with JQ (as with most query languages)

```
icaml$ echo "[]" | jq 1+2
3
icaml$ echo "[]" | jq '"hello " + "world!"'
"hello world!"
icaml$ echo "[]" | jq '[1,2]+[3]'
[
  1,
  2,
  3
]
icaml$ echo "[]" | jq '{"key":"value"}+{"key2":"value2"}'
{
  "key": "value",
  "key2": "value2"
}
icaml$
```

Warning

- But it is not always what you expect:

```
icaml$ echo "[]" | jq '{"key":"value"}+{"key":"value for duplicate key"}'
{
  "key": "value for duplicate key"
}
```

Brackets (Rounded ones)

- Sometimes, you need to scope operations into an explicit expression. This is done using round brackets (as in math)

```
icaml$ echo "[]" | jq '"x"+"y"*2'
"xyy"
icaml$ echo "[]" | jq '("x"+"y")*2'
"xyxy"
```

The , operator

- If you want to run several queries, you can create a sequence of results using the , operator:

```
icaml$ cat sample-tweet.json |jq '.id_str, .text'
"1062405858712272898"
"The Shortest Paths Dataset used for #acm #sigspatial #gis cup has just been released. https://t.co/pzeEleBfu9 #gis… https://t.co/IF7z1WnUDk"
icaml$
```

Remark:

- Actually, the generation of arrays we have seen

  [1,2,3]

  is a combination of the [] operator creating an array from a set and the , operator creating a sequence, and the values 1,2, and 3.

Piping

- Similar to chaining for the . operator, we can pipe expressions meaning that the result of the left expression is made the input of the right expression.

```
icaml$ cat sample-tweet.json |jq '.user | .name'
```

JQ Functions

- Finally, JQ provides many functions you will want to have (basically all you can think of and more)
  - cat sample-tweet.json |jq '. | keys,
  - echo [1,2,3,4] | jq 'map(.+1)' results in [2, 3, 4, 5]
  - echo '{"key":"value","key2":"value2"}' | jq 'map_values(.+"_")‘
  - echo '{"key":"value","key2":"value2"}' | jq 'to_entries‘

See the JQ manual for more functions and their explanations.

Finally

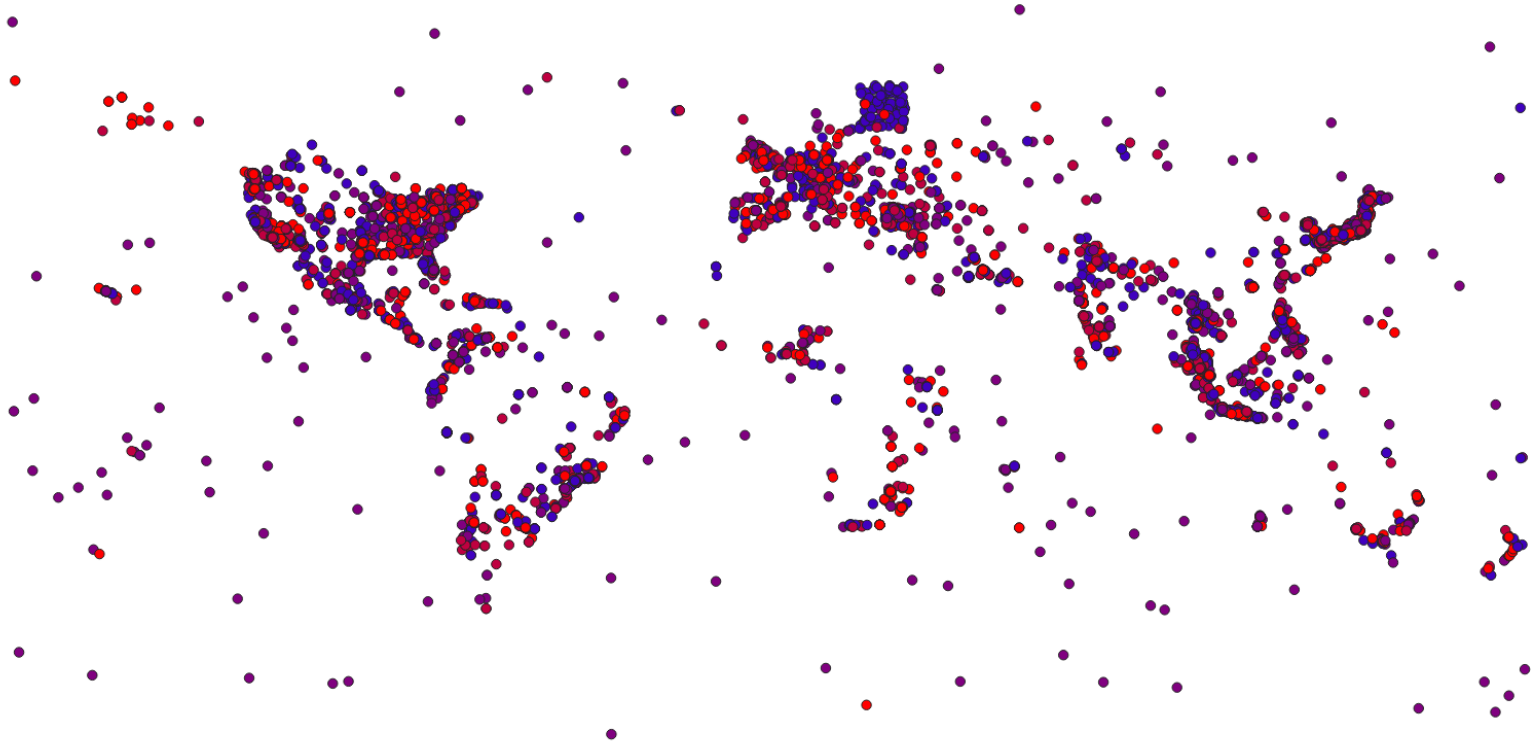
Extracting Geo-Located Tweets

Using JQ, WKT, and CSV

- A tweet is precisely geolocated, when the field location is defined.
- Query:
  - First all that have geography
  - Then, extract coordinates and, for example, follower_count into an array
  - Turn this array into a CSV and write it

```
cat <file> | jq –r ‘select(.coordinates != null)
|
[.coordinates.coordinates[0],.coordinates.coordin
ates[1],.user.followers_count] | @csv‘ > geo-
follower.txt
```

QGIS…



Result for 200k tweets:

Density and Distribution

Lets start the analysis

- Apply Inverse Distance Weighted Interpolation
  - Create a raster in which every pixel is set up from the inversely weighted neighbors
  - Parameters 2.0 and 300x300 are fast (for a quick check)

Text Mining from Twitter Data

Text Mining Basics

Text Mining and Natural Language Processing

- Extract Knowledge from (natural, spoken) language

Abstract Techniques:

- Text Preprocessing
- Feature Extraction
- Model Building / Machine Learning

Text Preprocessing

- **Stemming**
  Playing, Played, Plays => play, but not Plain
- **Remove Stop Words**
  Words that are too frequent to transmit task-specific information
  and, or, today, the they
- **Remove Corpus-Specific Stop Words**
  Words that are too frequent don‘t gain information, words that are too rare cannot be learned. Remove both categories (e.g., the top 10% of the most frequent and most rare words.
- **Casing**
  Turn all letters to lower-case, translate complex literals like ä in German to ae. Idea: reduce dimensionality by restricting to 26 letters + space.
- **Punctation and Numbers**
  Punctation and numbers are similar to stopwords in that they do not transmit task-specific information unless the text mining exploits grammar.
- **White-Space Removal**
  In some cases, white space can be removed. This has positive and negative effects on text mining.
- and many more…

Warning

- Classical text mining systems do not work without pre-processing
  - Dimensionality too high
  - Language too complex to learn
  - Datasets too small
- The best preprocessing is, however, not to preprocess
  - Preprocessing always introduces information loss (often grammar, sometimes important words like **not**)
  - All of the preprocessing tasks themselves are language-dependent and difficult.

Sparse Representation

In Text Mining, it is customary to create a vocabulary of things (containing a few thousand „words“) and to represent a document (sentence) with a sparse vector in which every „word“ that occurs in „document“ implies a one in a certain location.

Given a corpus C, we create a matrix in which a row represents a document and a column represents whether a word is in the document.

Term-Document-Matrix

Document (Sentence)	Inaccuracy	Explain*	Text	…
A little inaccuracy saves a lot of explanation	1	1	0	…
Explaining text mining is difficult and often inaccurate	0	1	1	…

The resulting matrix is **very sparse**:
- Every row contains exactly as many ones as the Document contains words.
- Every column contains exactly as many ones as the word appears in documents

Words are bad, what about n-grams?

- N-grams are sequences of N neighboring things
  - Character n-grams (n=3):
    Explanation => Exp, xpl, pla, lan, ana, nat, ati, tio, ion
  - Word n-grams (n=3)
    A little inaccuracy saves a lot of explanation =>
    A little inaccuracy, little inaccuracy saves, inaccuracy saves a, saves a lot, a lot of, lot of explanation

It is known (since the advent of information theory) that language approximations with character or word n-grams capture a lot of syntactical and grammatical structure. (see **Shannon: A Mathematical Theory of Communication**; the author in which Shannon entropy is introduced)

Character n-grams (1948, Shannon)

1. Zero-order approximation (symbols independent and equiprobable).

XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZL-HJQD.

2. First-order approximation (symbols independent but with frequencies of English text).

OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL.

3. Second-order approximation (digram structure as in English).

ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TU-COOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE.

4. Third-order approximation (trigram structure as in English).

IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME OF DEMONS-TURES OF THE REPTAGIN IS REGOACTIONA OF CRE.

Word n-grams (Shannon, 1948)

5. First-order word approximation. Rather than continue with tetragram,

REPRESENTING AND SPEEDILY IS AN GOOD APT OR COME CAN DIFFERENT NATURAL HERE HE THE A IN CAME THE TO OF TO EXPERT GRAY COME TO FURNISHES THE LINE MESSAGE HAD BE THESE.

6. Second-order word approximation. The word transition probabilities are correct but no further structure is included.

THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER TOLD THE PROBLEM FOR AN UNEXPECTED.

TF-IDF

Rationale: Rare Words carry more information.

TF-IDF assigns a score (weight) to a word in a document giving higher waits to unexpected words:

Term Frequency:

$$TF(w) = \frac{Number\ of\ times\ w\ is\ in\ D}{Number\ of\ words\ in\ D}$$

Inverse Document Frequency:

$$\mathrm{IDF(w)} = \log \frac{Number\ of\ documents\ in\ D}{Number\ of\ Documents\ containing\ w}$$

TF-IDF

- TF-IDF is the product of term frequency and inverse document frequency:

$$TFIDF(w, d, C) = TF(w, d, C) \cdot IDF(w, d, C)$$

- This scheme is **largely used in ranking keyword searches** in databases, though more advanced techniques are used for search as well.

- Example: MySQL (after creating a fulltext index)
  - SELECT COUNT(*) FROM table WHERE MATCH(<column>) AGAINST(<word>);

Machine Learning

- Consider the tutorial from scikit-learn to learn
  - It will show an example of how to build n-grams, etc.
  - It contains links to advanced setups
  - It links to Latent Semantic Analysis / Topic Extraction
    - A topic is kind of a joint probability of vectors and the result of topic extraction is a probability of how likely a document is covering a certain topic. Topics can be mined from a training set or completely unsupervised.

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

Text Embedding

- Sparse Vectors are good, but
  - They are often too high-dimensional (one column for each word?)
  - It is not easy to deal with them
- Idea
  - Create a feature vector for each word such that the linear algebra operation „+“ gets a meaning
  - This is called text embedding and FastText is a simple and efficient implementation of this regime.

How does it work?

- Use character-level n-grams
- Add special characters to beginning and end of word, concretely
  - Matter ➔ <ma, mat, att, tte, ter, er>
- Use deep learning (stochastic gradient decent) with a single hidden layer of chosen dimensionality (the embedding layer, e.g., 100d)
- Optimize

$$\sum_{t=1}^{T} \left[\sum_{c \in \mathcal{C}_t} \ell(s(w_t,\ w_c)) + \sum_{n \in \mathcal{N}_{t,c}} \ell(-s(w_t,\ n)) \right]$$

How does it work?

$$\sum_{t=1}^{T}\left[\sum_{c\in\mathcal{C}_t}\ell(s(w_t,\ w_c))+\sum_{n\in\mathcal{N}_{t,c}}\ell(-s(w_t,\ n))\right]$$

In this formula:

- t ranges over all words in the corpus
- The square bracket thing contains two loss terms
  - One positive (left)
  - One negative (right)
- s is a scoring function comparing two vectors. In this case, it is just the scalar product
- l is the logistic loss $\ell : x \mapsto \log(1 + e^{-x}),$

How does it work?

$$\sum_{t=1}^{T} \left[\sum_{c \in \mathcal{C}_t} \ell(s(w_t,\ w_c)) + \sum_{n \in \mathcal{N}_{t,c}} \ell(-s(w_t,\ n)) \right]$$

This translates to the following optimization:

Find vectors w_t for words such that the vectors of words that appear near each other ($c \in C_t$) are similar while vectors for random words (negative samples, $c \in N_{t,c}$) are dissimilar

and with a few tricks (see fasttext.cc), this gives models that are quite good…

Additive Structure

To some extent, relations like the following hold:

$$King - Man + Woman \approx Queen$$
$$Paris - France + Germany \approx Berlin$$

This depends on the choice of the scalar product and the nature of the corpus. It highlights that **complex information can be mined** from text in a unsupervised setting.

Text Classification with FastText

- The embeddings generated in this way can be used to train classifiers (fasttext has a simple classifier already)
  - Apply LSTM

- With more text, more complicated models are possible
  - Transformer Models (check out BERT)

Task for Today

Step 1: Download Dataset (temporarily available !)

Step 2: Create and Visualize spatial aspects (JQ + GIS of your choice)

Step 3: Label Dataset for Fasttext

Step 4: Download pretrained weights, create your own weights, etc.

Some ideas:

Land vs. Water: Predict if a tweet comes from land or water

Predict Language of tweet

Predict user category?

Predict tourist (e.g., time zone != home time zone)

Thanks!