university of groningen / faculty of arts

Detecting local events in the Twitter stream

by

Chris Pool

Bachelor thesis
June 2, 2015

DETECTING LOCAL EVENTS IN THE TWITTER STREAM

CHRIS POOL

PREFACE

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1 INTRODUCTION

With the growing number of people using social media services like Twitter and the enormous amount of data available makes it interesting to use as a source of information about events that are taking place at a certain time and location. This real-time information in combination with user sentiment can be very informative, for example to provide news items with more information about the opinion of people present.

This thesis describes a method of finding local events based on tweets with geo-information and categorize them into 5 categories (No-event, Sport, Entertainment, Meeting and Incident) and to find relevant information within those tweets about the location and people involved. Local events are events that take place in a certain radius (Z) within a certain time interval (X). For this research $z = 0.5\mathrm{km}, x = 5\mathrm{hours}$.

My research question is: Is it possible to detect local events from the Twitter stream that take place in the Netherlands based on Tweets with geo-information, to see if you can detect what type of events take place and who the important people/organizations or topics are to make an interactive map of these events.

You can find in the first chapter the discussion of existing research. In the second chapter I describe the techniques that i use for my research and in the final chapter my experiment and results.

2 RELATED LITERATURE

Walter and Kaisser[1] describe in their paper Geo-spatial event detection in the Twitter stream a way to recognize geo-spatial events. They focus on small events that take place in a small area like fires and traffic jams. It differs from my research because I want to focus on locations where events of different sizes can take place but the features they use can be useful for my research. Especially how they calculate the clusterscore (cluster of tweets from a region) can be useful. Unfortunately they do not describe how they make clusters of locations.

Amineh Amini et al.: Density-Based Data Streams Clustering Survey[2] is a paper about different clustering techniques and is interesting to see which ones I can use to make clusters of the locations and the events. DBSCAN is one of the density-based clustering algorithms I maybe can use. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is developed for clustering large spatial databases with noise, based on connected regions with high density. The density of each point is defined based on the number of points close to that particular point called the point's neighborhood. The dense neighborhood is defined based on two user-specified parameters: the radius (of the neighborhood, and the number of the objects in the neighborhood

3 METHOD

In this chapter the used methods for the experiment are described.

3.1 Clustering

The first task is to cluster the tweets based on their location and the time the tweets are published. There are several approaches in location clustering that can be used as discussed in the previous chapter. Because the aim of the research was to make something that could work in real-time it was necessary to look for an approach that was quick. GeoHash is is a system invented by Gustavo Niemeyer [**?**] that converts latitude and longitude coordinates into a *geoHash*. The hash represents an area where in the location is present. The more digits used indicates a smaller area, and thus a higher precision. If two locations share the same prefix it indicates (not always) that the locations are nearby.

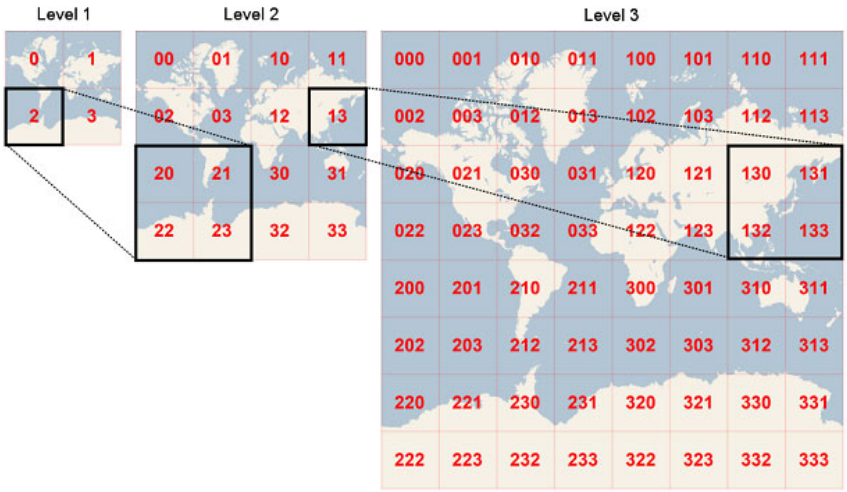

Figure 1: Geohash

Figure 1 illustrates how geoHash works. The first digit indicates which halve of the earth the area is on. The second digit divides that area in four areas and that continues resulting in the table below that show the different lengths of the *geoHash* and the accuracy of that length. Because the dimensions of the area are different at each latitude the values represent the worst-case scenario at the equator.

Table 1: Geohash Precision [**?**]

GeoHash length	Area width x height
1	5,009.4km x 4,992.6km
2	1,252.3km x 624.1km
3	156.5km x 156km
4	39.1km x 19.5km
5	4.9km x 4.9km
6	1.2km x 609.4m
7	152.9m x 152.4m
8	38.2m x 19m
9	4.8m x 4.8m
10	1.2m x 59.5cm
11	14.9cm x 14.9cm
12	3.7cm x 1.9cm

There are several implementations of this algorithm in Python. I used the GeoHash library written in 2009 by Hiroaki Kawai [1] because of the good documentation and excellent features. The function to calculate the *geoHash* of a location requires the desired accuracy, latitude and longitude of that location and returns the *geoHash* for that location.

```
1 lon = 52.348
2 lat = 6.2342
3 accuracy = 7
4
5 hash = geohash.encode(lon,lat,accuracy)
6 >>> print(hash)
7 u1k3y1c
```

The library also provides functions to calculate the neighboring geoHashes, this is useful because in a grid clustering approach border cases can occur. This functionality can be used to merge those border cases.

For clustering based on time the UNIX timestamp can be used. That is a system to represent time using the number of seconds since 1 January 1970. Python has build-in functions to deal with UNIX timestamps. The tweets are sorted on timestamp making it easy to make clusters of time series.

3.2 NLTK

Python has a lot of functions to use while doing NLP tasks but for more advanced tasks it is not powerful enough. Therefore I have chosen to use the Natural Language Tookit (NLTK) [2]NLTK has a lot of modules that can be used for a wide range of NLP tasks. It is also really easy to use and well documented. Another advantage is that NLTK is modular, meaning you can use small portions of the toolkit that work together with other libraries.

3.3 Classification

After forming the potential events (*Event Candidates*) the Event Candidates have to be classified as a specific event category or not an event. NLTK provides excellent documentation and functions for classifying.
NLTK enables you to use several classifiers:

- ConditionalExponentialClassifier

- DecisionTreeClassifier
- MaxentClassifier
- NaiveBayesClassifier
- WekaClassifier
- SVMClassifier (using scikit-learn)

In my experiment I have chosen to use the MaxEntClassifier, NaiveBayes and a SVM classifier. Training the different classifiers is easily done with NLTK. The format of the training file is the same for the different classifiers. Also functions to calculate accuracy, precision and recall are standard NLTK functions. I tested these three classifiers because their different approaches.

3.4 Named Entity Reconinigzion

For finding the important persons, organizations and locations I want to use Named Entity Reconinigzion(NER). NER is the proces of labeling data with corresponding categories like person, organization or location. For example:

> Klaar voor **Ajax(organization) - NAC(organization) @Amsterdam(location) @ArenAPoort(location)** in **Amsterdam(location)** Zuidoost, **Noord-Holland(location)**

Current NER tools were designed to process large texts and perform poorly with tweets due to the noisy and short style of tweets. One of the most used NER tools is Stanford NER. This Java implementation is also known as CRFClassifier. The software comes with 3 trained classifiers that perform particularly well for the three classes persons, organizations and locations. Some experiments are done with using labeled tweets as train data but mostly in English.

Because there is no classifier trained with Dutch tweets I want to train my own classifier that I describe in detail in the next chapter.

4 EXPERIMENT AND EVALUATION

For my experiment I worked together with David de Kleer in collecting the data and building the Python modules. The goal was to build a system that can detect events that take place at a specific location, for example concerts, football matches, fires and traffic jams and recognize the named entities in the tweets. Events that take place on a larger scale like elections and extreme weather conditions are ignored. The input for this system are tweets that have coordinates that we cluster based on proximity and time. These clusters of tweets are potential events (*eventCandidates*) that the system classifies using the following categories:

- **Other (OTH)** All events that are other then the following categories
- **Meeting (MEE)** All events that are meetings or conferences
- **Entertainment (ENT)** All events that have to do with concerts, movies or theater
- **No Event (NOE)** No Event
- **Sport (SPO)** All events that have to do with sport

4.1 System architecture

Python was used for this system because of the excellent tools available to build a machine learning system. NLTK and Scikit learn are used for classifying eventCandidates. GitHub was used as file repository. The system consists of five different modules each containing multiple scripts. In this section the main properties of the modules are described. In the following sections the modules are explained in detail.

EVENTCANDIDATES This is the module that generates the EventCandidates by clustering the tweets on geoHash and timestamp and saves the result as a JSON file.

ANNOTATER This module makes it possible for multiple judges to annotate the data created with the EventCandidates module.

CLASSIFIERCREATOR This module creates/trains the classifier(s) using the annotated eventCandidates.

EVENTDETECTIVE This module uses the trained classifier to classify eventCandidates and generate markers for the Google map.

NER This module uses the NER classifier that is explained in the Named Entity Recognition section of this chapter.

4.2 Data collection

For the experiment tweets are used that have coordinates. Using a simple Grep[1] command it was possible to retrieve only tweets with this information from the Karora machine[2] . For our system we need two datasets. One for training and developing our system that we call *devset* and one for testing our final system that we call *testset* The first dataset consisted of Dutch tweets from March 2015 that we have downloaded from the Karora machine.

1 Command-line utility for searching plain-text data sets for lines matching a regular expression
2 karora.rug.nl

In total there where 566.549 tweets with geo information. That is about 3% of the total number of tweets published in that month. The trainset consists of 165.848 tweets from the second half of April 2015. The tweets are stored in a tab seperated text file with the fields tweet, latitude, longitude, username and timestamp

> @jokiecroky heb ze terug gebracht naar de plek waar ik ze heb opgehaald... volgende dag werd ik gebeld dat ze samen een mooi huisje hadden 5.106646 52.06455 TonyJuniorLive 2015-04-15 00:01:37 CEST Wed

4.3 Creating Event Candidates

For clustering the tweets and preparing them for annotation and classification we use the EventCandidates module. This module processes all tweets from a given dataset and generates the *eventCandidates* using four scripts. TweetPreprocessor.py, ClusterCreator.py, ClusterMerger.py and EventCandidates.py

TWEETPREPROCESSOR.PY reads the file with tweets and tokenizes the tweets and removes the frequent occuring words from the tokens, convert the timestring to a Unix timestamp and converts the latitude en longitude in a geoHash. The output of this script is a list of dictionaries where each dictionary is a tweet with the following structure:

```
{'text': '@MrFrankLegs Daar gaan we. Een vicieuze cirkel ??',
'lat': 52.964285,
'geoHash': 'u1kjqcc',
'localTime': '2015-04-24 10:54:16',
'user': 'sandersierdsma',
'tokens': ['@mrfranklegs', 'gaan', 'vicieuze', 'cirkel'],
'lon': 5.923195,
'unixTime': 1429865656}
```

CLUSTERCREATOR.PY iterates over all tweets and adds the tweet to a new dictionary that uses two keys. The first is the geoHash and the second is the timestamp of the last added tweet.

```
clusters['u15y07t']['1429604082'] = [...list of tweets that have the same geoHash and have the same
```

We create this dictionary using the following function:

```

#setting used for this experiment
self.MINUTES = 60

def __createClusters(self):
        for tweet in self.tweetDicts:
            geoHash = tweet["geoHash"]
            tweetTime = tweet["unixTime"]
            foundTime = tweetTime

            if geoHash in self.clusters:
                for times in self.clusters[geoHash].keys():
                    if times <= tweetTime <= times + self.MINUTES * 60:
                        foundTime = times

            self.clusters[geoHash][foundTime].append(tweet)
            if tweetTime != foundTime:
```

```
18                # use new timestamp as key to keep event active
19                self.clusters[geoHash][tweetTime] = self.clusters[geoHash][foundTime]
20                # Remove old key
21                del self.clusters[geoHash][foundTime]
```

The output of this script is a dictionary with all clusters with the structure described in script x.

CLUSTERMERGER.PY checks if there are clusters that are neighbors and refer to the same event like the example below where a lot of tweets are published in the same time period but in two different geoHashes.

Figure 2: Geohash border case(Red dots are tweets)

In the output from the clusterCreator.py script we be in this case two different clusters. To merge events that refer to the same event but have different geohashes (and are neighbors) we use this script to merge these cases. We do this by iterating over all clusters and calculate the neighboring geohashes. For each neigboring geohash we check if there is an cluster within the same time period. If we find a cluster with a neighboring geoHash and with the same timestamp +/- 60 minutes we calculate the word overlap because it can also be a different event that is coincidental taking place at the same time and close to current geoHash. We calculate this wordoverlap by looking at the 10 most common word in both clusters. For each word that overlaps a score of one point is awarded. If a hashtag or username overlaps two points are awarded. If a score exceeds the threshold the function returns true. The threshold is determined by experimenting with a lot of different values.
We discussed the number of iterations needed because it is also possible to look at the neighbors of the neighbors but we concluded that one iteration was sufficient. The area you get when you look at the neighbors for one iterations is big enough and if you go bigger you get events that are not local.

```
1  def _calculateWordOverlap(self,clusterA, clusterB):
2      wordsClusterA = self._getImportantWords(10, clusterA)
3      wordsClusterB = self._getImportantWords(10, clusterB)
4      result = {}
5
```

```
    #intersect the two lists and adding the scores
    for wordA, scoreA in wordsClusterA:
        for wordB, scoreB in wordsClusterB:
            if wordA == wordB:
                result[wordA] = scoreA + scoreB
                if wordA[0] == '#':
                    result[wordA] *= 2
                if wordA[0] == '@':
                    result[wordA] *= 2

    if sum(result.values()) > self.THRESHOLD:
        return True
    else:
        return False
```

The output of this script is an updated dictionary where neighboring clusters about the same event will be merged.

EVENTCANDIDATES.PY is the wrapper class for this module. This script generates all eventCandidates using the scripts described above. This function requires two parameters. The first is the name of the text file with the unprocessed tweets. The second is the desired name of the dataset. This script saves a JSON file with all eventCandidates.

4.4 Annotation

We created two datasets. One for developing and one for testing. We annotated the testing dataset in the final period of our research. The data is annotated with an interactive tool by two judges. This tool also calculates the kappa score: was 0.79 for the development dataset and 0.8 for the test dataset. According to book IR is dat fair/good. We discarded the eventCandidates we did not agree on.

For the development dataset we annotated 1350 event candidates with these categories. In 87% of the cases the judges agreed.

Table 2: Confusion matrix development dataset annotation

	OTH	MEE	ENT	NOE	INC	SPO
OTH	<1>	.	.	2	1	1
MEE	21	<207>	7	25	.	2
ENT	3	.	<20>	5	.	.
NOE	14	27	18	<619>	9	9
INC	1	2	.	12	<178>	.
SPO	1	.	.	5	.	<60>

For the devset we annotated 500 event candidates with these categories. In 86% of the cases the judges agreed.

Table 3: Confusion matrix test dataset annotation

	OTH	MEE	ENT	NOE	INC	SPO
OTH	<.>	.	.	.	1	.
MEE	4	<110>	13	9	.	3
ENT	.	.	<8>	2	.	.
NOE	3	19	4	<199>	4	2
INC	1	.	.	.	<78>	.
SPO	.	1	2	2	.	<60>

4.5 Event detection

For the event detection I trained two classifiers using NLTK. The first classifier trains using the words (800 most common words with stopwords removed) as features. Several values were tried but 800 resulted in the best performance. Also several types of classifiers were tried. Naive Bayes produced the best result and was also the quickest. The output of that classifier (a category) was used as a feature for the second classifier that used more features as described in the next section.

4.5.1 *Feature selection*

The first classifier only uses the words as features. The second classifier uses the output of the first classifier as a feature. This feature is by far the most valuable for the system. In the table below the performance of the features are described. In appendix x the complete results can be found.

Table 4: Effect of features

	Accuracy
All features	0.84
Category	0.81
Location	0.51
WordOverlapSimple	0.63
WordOverlapUser	0.6
WordOverlapUser, Category	0.82

CATEGORY The result of the classifier that uses the most common words as features.

LOCATION The first 5 characters in the geoHash are used as a feature. It will be easier to detect events on locations were events often take place.

WORDOVERLAPSIMPLE This feature is a numeric value that represents how many tweets consist of the same words. The score is calculated by counting the occurrences of each type and dividing it by the number of tweets. Hashtags get a bonus score.

```
1  def _wordOverlapSimple(self, candidate):
2          types = Counter()
3          for tweet in candidate:
4              types.update(set(tweet['tokens']))
5          score = 0
6          for t in types:
7              if types[t] > 1:
8                  if t[0] == '#':
9                      score += (types[t] * 2)
10                 else:
11                     score += types[t]
12
13         return round((score / len(candidate)) * 2) / 2
```

WORDOVERLAPUSER This feature calculates the overlap of types among users. The score is the highest when all users use the same words. The result is the log of the score and rounded to 0.5.

```
def _wordOverlapUser(self, candidate):
        '''Calculate the overlap of features among users'''
        userTypes = defaultdict(list)
        types = Counter()

        for row in candidate:
            userTypes[row['user']].extend(row['tokens'])

        for user in userTypes:
            types.update(set(userTypes[user]))
        score = 0
        for t in types:
            if types[t] > 1: #ignore if only in one tweet
                if t[0] == '#':
                    score += (types[t] * 2)
                else:
                    score += types[t]

        if score > 1:
            s = log(float(score) * float(len(userTypes.keys()) ))
            #return round((score * 2) / len(candidate))
            return round((s / len(candidate) )* 2 ) /2
        else:
            return 0.0
```

4.5.2 *Results event detection*

The event classifier performs good. It is a few procent less then the Upper bound of 87%.

Table 5: Results using all features

	Naive Bayes			Maximum Entropy			SVM		
	Accuracy = 84%			Accuracy = 83%			Accuracy = 81%		
	P	R	F	P	R	F	P	R	F
NOE	0.85	0.90	0.88	0.83	0.93	0.88	0.85	0.83	0.84
SPO	0.77	0.49	0.60	0.77	0.49	0.60	0.77	0.49	0.60
ENT	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MEE	0.76	0.79	0.77	0.79	0.74	0.76	0.65	0.81	0.72
INC	0.97	0.97	0.97	0.94	0.94	0.94	1.00	0.97	0.99
OTH	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

The different classifiers perform almost the same. The

Table 6: Confusion Matrix Naive Bayes

	OTH	MEE	ENT	NOE	INC	SPO
OTH	<.>	.	.	1.	.	.
MEE	.	<82>	2	14	.	12
ENT	.	.	<.>	1	.	.
NOE	.	27	5	<179>	1	7
INC	.	.	.	2	<76>	.
SPO	.	1	1	2	1	<16>

Table 7: Confusion Matrix Maximum Entropy

	OTH	MEE	ENT	NOE	INC	SPO
OTH	<.>	1	.	.	.	.
MEE	.	<81>	2	7	2	11
ENT	.	.	<.>	.	1	.
NOE	.	26	5	<186>	1	7
INC	.	1	.	4	<73>	.
SPO	.	1	1	2	1	<16>

Table 8: Confusion Matrix SVM classifier

	OTH	MEE	ENT	NOE	INC	SPO
OTH	<.>	1	.	.	.	.
MEE	.	<89>	2	31		14
ENT	.	.	<.>	.		.
NOE	.	19	5	<166>	1	4
INC	.	.	.	.	<73>	.
SPO	.	1	1	2	1	<17>

4.6 Named Entity Recognition

The default NER tagger in NLTK is the Stanford NER classifier. This is according to many researches the best available. Unfortunately there are no classifiers trained using Dutch tweets and the performance of the available classifiers was so poor (see table x) i decided to train my own classifier. Because of limited time I decided to automatically annotate the training data creating a silver standard.

4.6.1 *Data collection*

I used the wordlists that are used in Alpino(LINK), that are four lists with dutch words categorized in (LOCATIONS, PERSONS, ORGANIZATIONS and MISC). I used those list to download tweets that at least contain one word from these lists. I then used the wordlists to annotate the tweets automatically in de the desired training data format. This resulted in a dataset of one million annotated tweets.

4.6.2 *Training*

For training the classifier the dataset was divided in a train set and a test set. The Stanford NER tagger trained with several features, NGR

4.7 Results

In this section the results of the two components, the classifier and the NER tagger are discussed.

4.7.1 *NER*

I compare my results with the standard Stanford 4 class classifier. This classifier is trained on xxx and uses the same classes as my classifier.

Table 9: NER Classifier trained with Tweets

Entity	P	R	F1	TP	FP	FN
LOCATION	0,7333	0,7333	0,7333	11	4	4
MISC	0,0909	0,500	0,1538	1	10	1
ORGANIZATION	0,7500	0,8824	0,8108	15	5	2
PERSON	0,6154	0,5517	0,5818	16	10	13
Totals	0,5972	0,6825	0,6370	43	29	20

5 CONCLUSION AND DISCUSSION

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

6 APPENDIX

	accuracy	geen_event	sport	entertainment	bijeenkomst	incident	anders
		P R F	P R F	P R F	P R F	P R F	P R F
All features	0.84	0.85 0.90 0.88	0.77 0.49 0.60	0.00 0.00 0.00	0.76 0.79 0.77	0.97 0.97 0.97	0.00 0.00 0.00
Category	0.81	0.85 0.82 0.84	0.73 0.46 0.56	0.00 0.00 0.00	0.66 0.84 0.74	0.99 0.97 0.98	0.00 0.00 0.00
Location	0.51	0.49 0.94 0.65	0.00 0.00 0.00	0.00 0.00 0.00	0.66 0.25 0.36	0.50 0.06 0.11	0.00 0.00 0.00
WordOverlapSimple	0.63	0.60 0.85 0.71	0.00 0.00 0.00	0.00 0.00 0.00	0.57 0.33 0.42	0.77 0.83 0.80	0.00 0.00 0.00
wordOverlapUser	0.6	0.57 0.93 0.71	0.00 0.00 0.00	0.00 0.00 0.00	0.00 0.00 0.00	0.66 0.90 0.76	0.00 0.00 0.00
wordOverlapUser, category	0.82	0.86 0.83 0.85	0.77 0.49 0.60	0.00 0.00 0.00	0.66 0.85 0.74	1.00 0.97 0.99	0.00 0.00 0.00

REFERENCES

[1] Hiroaki Kawai. Geohash - fast, accurate python geohashing library.

[2] NLTK. Natural language toolkit.