Geo Query by Example

Sotera Defense Systems

August 12, 2015

This document describes the code base for the back end of the Geo-Query by Example tool. Geo-Query by Example is a set of analytics which aims to leverage social media data to identify events of interest based on a set of like examples. The document will first cover the driving principal of the tool, then dive into the code base to explain the packages and libraries.

Introduction

One result of the increasingly connected world is a rich set of publicly available data coming from social media. In particular, websites such as Twitter and Instagram are driven by serendipitous thought. As a result, the text associated with their posts often reflects sentiment with respect to the nature of the surrounding area. When combined with the fact that a good portion of users enable geo location information to be associated with their posts, one can build up a set of language which is representative of either a type of area[1], or events[2]. This type of language can then be used to create an algorithm to determine the similarity of other areas.

[1] e.g. hospitals, airports

[2] e.g. marathons, riots

To be clear, this application has well understood limits. This type of analysis will only be useful in identifying area which ellicit stream of consciousness thought. As such, a buildings typical occupants (i.e. a person going to their place of work), will be unlikely to be as effective. Similarly, places that have a wide variety of uses such as parks will have too wide of variety of topics to create an effective discriminator. On the other side of the spectrum, landmarks such as the statue of liberty will typically cause a discriminator too focused to be used to find anything but itself. None the less, there are many locations which do create sufficiently specific discriminators for the tool to be of general interest.

Data and Infrastructure

Raw Data

N.B., if viewing this document as a PDF, all margin URLs should be clickable.

This tool is general purpose and can be applied to any type of data that fulfills two conditions; first, that the data contains text reflective of the surrounding environment, and second, that latitude and longitude information associated with each post. The data that we have most often leveraged come from Twitter[3] and Instagram[4]. In both cases, the type of data that is provided is JSON formatted text

[3] `https://dev.twitter.com`

[4] `https://instagram.com/developer/?hl=en`

files. We do implicitly assume the raw data is in Hadoop Data File System[5], and that it is line delimited, so custom preprocessing may need to occur to accommodate different formats. Additionally, one can add new formats (given they are line delimited) by modifying the file 'lib/to_parquet.py', where we define multiple input sources.

[5] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

Processing Infrastructure

The backend code is written in Python, and is intended to leverage Apache Spark [6]. Spark is an open source software framework for distributed processing of very large data sets. "In contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's in-memory primitives provide performance up to 100 times faster for certain applications."[7] Spark's high throughput is extremely useful for processing large volumes of data, and further efficiency gains are made by preprocessing data over the course of several steps, which we will detail later.

[6] https://spark.apache.org/

[7] https://en.wikipedia.org/wiki/Apache_Spark

Within spark, the Spark SQL[8] library is used to make use of the efficiency of parquet files[9] (vertically aligned data formats), and spark's machine learning library, MLlib [10], allows us to use powerful machine learning techniques that scale to train on large scale feature vectors. Leveraging the complete suite of big data analysis tools provided through Apache, we can create an analytic that is capable of processing large volumes of data with robust techniques, and high throughput.

[8] http://spark.apache.org/docs/latest/sql-programming-guide.html

[9] https://parquet.apache.org/

[10] http://spark.apache.org/docs/latest/mllib-guide.html

Control Flow

The program has been separated into two separate portions, starting with Extract, Transfer, Load (ETL). For our purposes, ETL includes all steps that are common to every operation, and hence it is used as a way of increasing processing time efficiency by reducing redundant computation. The process is depicted diagrammatically in figure 1.

The next portion of the program is the main analytic. There are three distinct programs which perform similar functions, but the general flow of the program is to create or load an analytic to be used to rank area based on their similarity to the areas of interest upon which the analytic is trained, and write these points out a file.

We will use two terms to refer to geographically bound regions. The first is 'Region of interest', and will refer to a larger scale region which defines a dataset (usually the size of a city or larger). We reference this type of area exclusively during the ETL process. The other type of area is an 'Area of interest', which is used to discriminate between data used to train a model, and the data the model is applied to.

Extract, Transfer, Load

As mentioned previously, the data from our raw sources typically comes in JSON formatted text files, which are unique to the input data source. The first step of the process is to identify if the posts are usable. The typical stream from Twitter is called the "1% stream", because it represents one percent of all tweets created. This is indiscriminate of the content of the posts, as well as if the post has geo-tagging enabled. Our first program, 'filterData.py' reads the data in from various data sources, and begins by identifying that it has geo-tagging information, as well as scans the text for words which can be scored[11]. We choose to focus on smaller localized datasets as processing time scales with the amount of data we examine, as well the fact that we will typically be zeroing in on a particular area of interest. We then proceed to filter the data to be within the bounding box which defines the region of interest, and finally, write the data out in the parquet file format.

[11] This requirement merely enforces that it is not part of our stop word list, that it is not a url, or username (as identified by the @ symbol proceeding the name).

The next program, 'precomputeIDF.py', runs over multiple datasets from filterData, and defines the document frequency[12] across every data set. Here, we define a document to be a single post. This number is needed to perform TF-IDF ranking of terms which are used as inputs for the machine learning algorithm later in the process. Another important utility of this process is to define a common feature vector which spans all data sets. In doing so, we enforce that any term appears in 1/1,000,000 documents; this restriction is useful in speeding up the training of the machine learning analytic, and also removes spurious words such as names or uncommon misspellings. The importance of a feature vector which spans data sets will become more clear when we discuss saving models in the Main Analytic section.

[12] `http://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html`

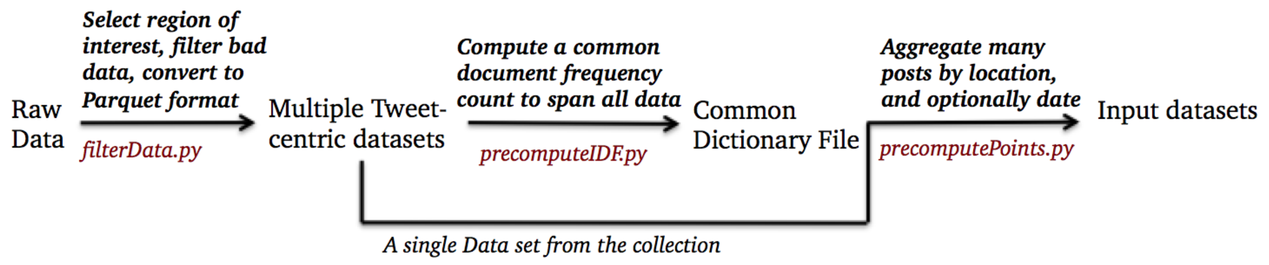
Raw Data
Select region of interest, filter bad data, convert to Parquet format
filterData.py
Multiple Tweet-centric datasets
Compute a common document frequency count to span all data
precomputeIDF.py
Common Dictionary File
Aggregate many posts by location, and optionally date
precomputePoints.py
Input datasets
A single Data set from the collection

Figure 1: A diagrammatic representation of the ETL process. Above the lines as physical descriptions of the manipulation to the data, and below in red are the files which execute the programs.

The last step in the process is to run the aggregation process for individual data sets. If we imagine an abstract flow for the program, one can imagine one of two ways proceeding; the first is one which scores individual tweets and then applies clustering techniques atop of the output to find the regions of high correlation, and the second is to aggregate points before they are input into the ML algorithm, and score regions instead of individual tweets. Both paradigms have been tested, with the later proving to be more effective at suppressing false positives in high frequency post regions (typically the downtown regions of major cities).

As the content of individual points will not very from job to job, we can speed up the analytic by running aggregation ahead of time. To do so we create bins of latitude and longitude by limiting the precision to a few decimal places (thereby collapsing a distribution of points to a single tweet). The tradeoff for this efficiency is a loss of flexibility in changing bin sizes from job to job, however, if the bin sizes are created intelligently, it's best to stick with a single 'one size fits all' approach.

The job of aggregating by point is fairly straightforward. After binning the data, a word count for each point is performed based on the collection of tweets for that point. Then a TF-IDF score is calculated for each term that appears in the point, and create a sparse vector based on the order of the DF document created in the previous step. The final product is a set of points represented by a latitude, longitude, a sparse vector representing the complete language variety, and optionally a date.

Main Analytic

At this point, ETL has already taken of the majority of heavy lifting as far as getting data ready for use in training and applying a classification algorithm[13]. The three main analytic programs are findSimilarPlaces, findSimilarEvents, and refindSimilarPlaces. The first two programs are represented diagrammatically in figures 2 and 3. The names reflect the purpose of the programs themselves. Abstractly, when we attempt to find a place, we return areas in space who's content is similar to the content in the areas designated as training areas. When we find an event, we return a set of places that are also segmented by date.

[13] Close inspection of the primary analysis code, one will note that we actually employ a regression algorithm, rank points based on their score, and return the top N entries.

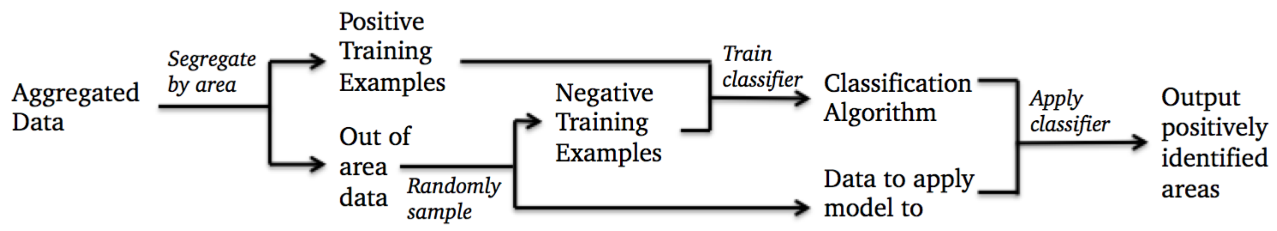

Figure 2: A diagrammatic representation of the findSimilarPlaces program.

To begin, we will examine findSimilarPlaces. The main inputs to findSimilarPlaces are the input data set, and a file which defines an area of interest. The input area file is a JSON document which describes polygons with a list of latitudes, longitudes, and date ranges.

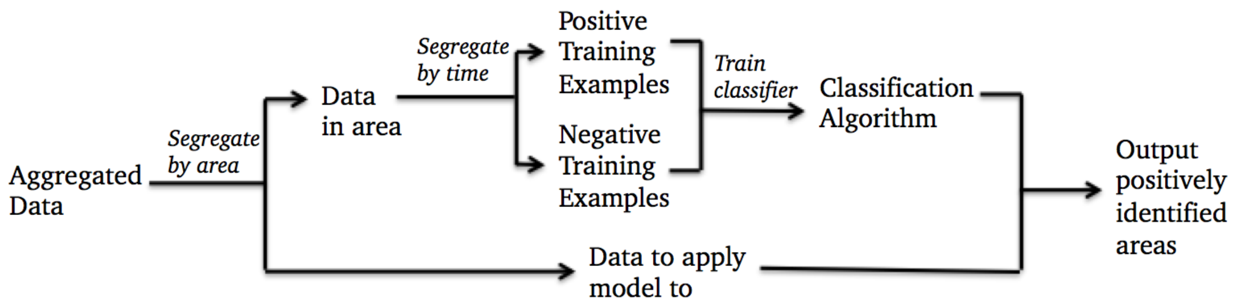

Figure 3: A diagrammatic representation of the findSimilarEvents program.

In the instance that the event is a single date, the same min and max date is provided. Higher granularity than one date is difficult, as data becomes too sparse to construct an effective ensemble. We typically also give the site a label to make it easy to identify different areas in the same file. The format for the area of interest file is of the following format:

The format for the input area shape files.

```
{
    "sites" : [
        {
            "name":"site of interest",
            "lats":[
                30.2669,
                ...
            ],
            "lons":[
                45.4535,
                ...
            ],
            "dates":[
                {
                    "min":"2013-02-17",
                    "max":"2013-02-17"
                },
                ...
            ]
        }
    ]
}
```

The control flow for findSimilarPlaces is straightforward. We filter points based on being inside or out of the region of interest. We label those entries in the training region as positive training examples, and take a small subsample of the outside region to be used as negative training examples. These are then used to train a random forest model[14]. The resulting model can optionally be saved for reuse later.

[14] `http://spark.apache.org/docs/latest/mllib-ensembles.html`

We then proceed to apply the model to the remaining data. As stated previously, the type of random forest created is a regressor, so a floating point score is assigned to the remaining points. These points are sorted on this regression score, and a variable number of entries are returned (which is controlled by an input variable).

The control flow for findSimilarEvents is very similar. The key differences are that the input data is segregated by date (handled in ETL). The other key difference is the source of the training data. The data is first segregated by area, as is done before, however, all the data outside the area of interest can have the model applied to it. Instead, the area of interest is now segregated as being inside or outside the time window of interest. Those events inside the time window are the positive training examples, and a portion of those outside the window are used as the negative training examples. Once the model is trained, the remainder of the process is the same, save for a small difference to the output format.

The last mode of operation is to apply a previously trained model to a new area. As was mentioned when discussing the findSimilarPlaces control flow, it is possible to store a trained model by providing an input variable.[15] It is similarly possible to save an event model. The purpose of refindSimilarPlaces is apply a model of either type to a new data set. As an example, a model trained on the Austin marathon can be applied on a dataset in New York to find the New York Marathon. The control flow for this program is extremely simple, where you read in the model, and apply it to the existing data. The output for the two programs vary's only slightly, as we attempt to take on formatting issues on the back end for convenience to the UI. The format is show on the following pages.

[15] These models are stored to HDFS, so it is not possible to overwrite a model with a new one, and attempting to do so will cause the program to exit in an error state.

The output format for events.

```
{
    "type": "event",
    "modelDict": [
        "love",
        ...
    ],
    "dates": {
        "2014-12-15": {
            "clusters": [
                {
                    "nTotal": 20,
                    "lat": 29.3025,
                    "lon": -98.4895,
                    "poly": [
                        [
                            29.303,
                            -98.489
                        ],
                        ...
                    ],
                    "score": 0.72,
                    "dict": [
                        "week",
                        ...
                    ]
                },
                ...
            ]
        }
    }
}
```

As mentioned, the output if JSON formatted, and in this case has a type of “event" (in contrast to the place based output which is of type "place"). The other top level key common to both is the 'modelDict', which represents the most important words in the training regions, as ranked by TF-IDF. The key which only appears when you are looking at the "event", data type is "dates", which is another dictionary indexed by the dates on which matches exist. Each date has a "cluster" object, which is a list of clusters. The place type simply has the clusters as it's top level object.

The output format for places.

```
{
    "type": "place",
    "modelDict": [
        "love",
        ...
    ],
    "clusters": [
        {
            "nTotal": 20,
            "lat": 29.3025,
            "lon": -98.4895,
            "poly": [
                [
                    29.303,
                    -98.489
                ],
                ...
            ],
            "score": 0.72,
            "dict": [
                "week",
                ...
            ]
        },
        ...
    ]
}
```