University of Chicago | Spring 2016
CMSC 12300: Computer Science with Applications
Christine Chung & Leith McIndewar

Campus as a Lab

Detecting Unusual Trends in Electricity Usage Data

June 6, 2016 | Repo for report, slides, & code: https://github.com/cszc/123_Electricity_Data

Introduction

The University of Chicago is one of the largest consumers of electricity in the Chicago area. While this is not surprising given the University's size and the scope of its work, it is important that energy consumption be reduced wherever and whenever possible. Efficiency gains serve to both save money from the budget for power and reduce the University's carbon footprint. This quarter, the Campus As A Lab initiative began with the

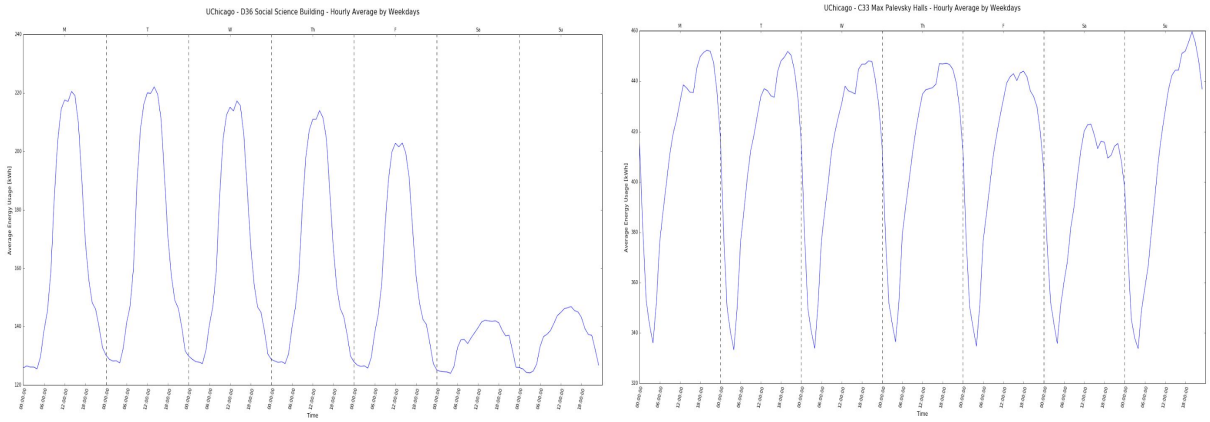
goal to involve students in the process of finding ways in which the University can reduce its resource use and energy consumption.

In 2015, the 160 campus buildings and grounds used $50M of energy (electricity + natural gas). This energy use accounts for approximately 70% of the University's greenhouse gas (GHG) emissions. Improving campus energy efficiency is the key driver to impacting GHG emissions and the University's environmental footprint. It is a complex problem to unravel, given the physical, economic and behavioral factors involved.

Dataset

We are using a dataset of the electrical meter readings for the University of Chicago from ComEd taken every 30 minutes for the last two years. The readings were taken from over 300 meters across more than 100 buildings.

Below: An example of two weekly patterns - the Social Science Building (left) and Hospital (right)



Dataset Size

The primary dataset we used is around 850 MB. We originally planned on bringing in additional temperature and weather data. However, after meeting with the facilities manager, we discovered that most buildings do not control their own temperature and so would not be strongly correlated with weather. (Many buildings do use window unit air

conditions, so there was typically greater use in the summer.) Other datasets that we looked into, but ultimately did not use, include physical building data (square footage, year built, etc. See: http://facilities.uchicago.edu/services/space-data/). We ingested geojson data into the ElasticSearch database, but did not use it for analysis. We also binned the meter readings using data from the UChicago academic and event calendars.

Observations

The dataset contains readings from over 300 campus meters taken in 30 minutes intervals from March 2014 - March 2016 and contains 9,454,879 rows. The dataset fields include: Type, Meter, Date, Start Time, Usage, Usage Unit, Temperature, Temperature Unit, Electric Usage

Hypothesis

After thinking about several different questions that could be answered with the electricity data, we settled on developing a way to detect if any building (or meter) is using an abnormal amount of electricity or exhibiting surprise behavior for a given time window. The University is on an hour ahead purchase system. Further, the University is taxed extra on the five highest-use energy days in the city. With many hundreds of meters to monitor, a system that can quickly detect anomalous usage patterns could be useful for responding to issues that could cost substantial amounts of money or cause a disruption in service.

Additionally, ComEd will eventually make more granular data available to its customers. As the frequency of the readings increases, the amount of data will also quickly grow. Our goal was to develop ways to search for anomalous usage that can scale well to greater amounts of data.

Environment

For k nearest neighbors, we used an EC2 instance with a PostgreSQL database. We used pandas to convert the results of the queries to data frames.

We also implemented an ElasticSearch database on a three-node distributed EMR cluster on Amazon Web Services. We thought ElasticSearch was well suited to this task because it is distributed, automatically organizes information in clusters of nodes and queries it with MapReduce, and can easily integrate with multiple EC2 instances on Amazon web services. Furthermore, ElasticSearch works well with real-time data. Since data is indexed, queries are extremely fast. Originally designed to work with logging data, it has many features built it to easily ingest, query for, and analyze time series data.

The installation involved sshing into each node (2 slaves and 1 master), installing ElasticSearch, and opening the correct ports and changing the configurations on each node so that each ElasticSearch instance was connected. We then ingested through the master node, which automatically sharded to the slave nodes. We also installed Kibana, an open source visualization tool that sits on top of ElasticSearch. With this, we could monitor ingestion in real-time by going to the master node's public dns at port 5601. Kibana also provides a visual interface to explore data in real time, which we used to initially explore the data.

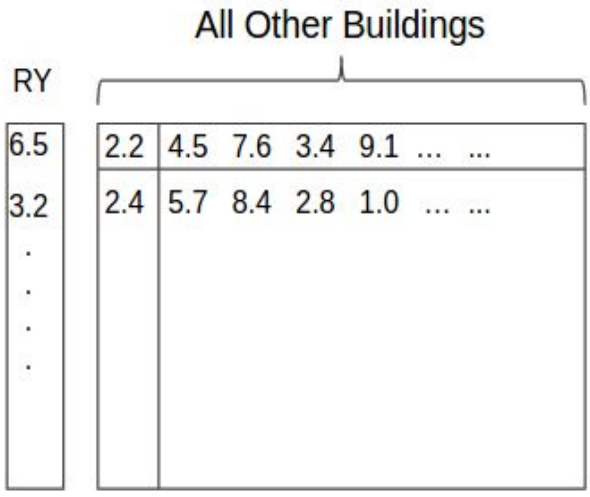
We used a python library, Elastic-Search-DSL, that provides a high level interface to Elastic Search's RESTful API. We ingested each field as both an analyzed and raw field. This means that each string along with its suffixes is indexed. We thought this feature could be useful in many cases. For example, using the analyzed strings, you could query for the energy usage of every building containing the word 'hospital' or 'library'. One major issue we encountered was that the indexing of the data caused the size of the database to balloon. Even though our original data was less than one gigabyte in csv format, after ingestion, our database was more than 15 GB. Our first attempt to ingest that data caused the server to run out of space after 9 hours of ingestion. We had to upgrade our EC2 instance to a larger size, and reingest the rest of our data.

Algorithms

K-Nearest Neighbor

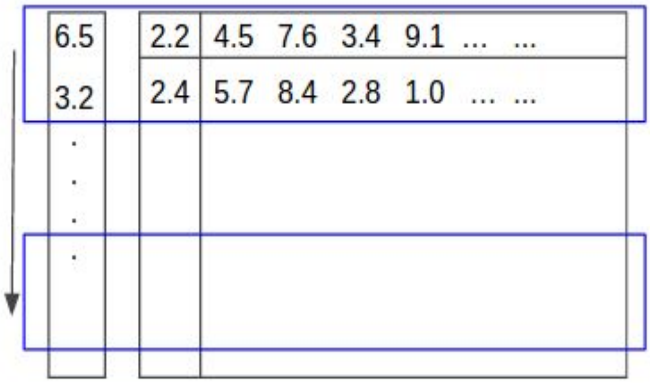
One approach we used was finding the k nearest neighbors to the current time window to identify anomalies in electricity useage. We started by defining the window size and k

used for the algorithm. To check if a building is using an abnormal amount of energy, we compared the readings from every other building (represented as a flattened matrix) during the most recent window to a sliding window across the dataset. Using the Euclidean distance between the matrices, we found the k nearest time windows from the past. We then looked at what the building of interest was doing during that time. Using these, we find the distance between the building of interest and its behavior during its k nearest neighbors.This idea assumes that other buildings are behaving normally or that any abnormal behavior from a single building will not significantly change the overall findings.



Right: An illustration of the KNN algorithm

Tarzan



The Tarzan algorithm is based on the paper, "Finding Surprising Patterns in a Time Series Database in Linear Time and Space", by Keogh et al. (2002)[1]. It was written as part of research sponsored by Facebook. In the paper, they lay out a method of finding surprising, anomalous patterns in time series data. Surprising is defined as *'if the frequency of its occurrence differs substantially from that expected by chance, given some previously seen data'.*

In sum, for a given window *w* in a time series *X*, you subtract the window's expected value from a reference time series *R* from its observed frequency in *X*. If you provide a reference time series that represents typical usage patterns (in the University of Chicago's case, a full quarter with no holidays), the algorithm should be able to predict unusual

[1] http://www.cs.ucr.edu/~eamonn/sigkdd_tarzan.pdf

patterns, such as holidays, outages, and more. In our implementation, we compare recent readings (for example, the last 24 hours or 48 readings), and look for periods of 4-hour windows that might be anomalous compared to a building or meter's entire history. However, we generalized the algorithm such that it can be used for anything. For example, it can be used to find surprising patterns in an individual building compared to all other buildings, another similar building, or a shortened window of its own history.

The first step of the algorithm is to discretize all time series. To do this, you slide a rolling window of a designated length over your time series of interest X and reference time series R. From each window the algorithm calculates a single real number. We use the slope of a one-degree line calculated using ordinary least squares. Then, you sort the slopes into equal quantile buckets. Each of these buckets is assigned a symbol. In our implementation, we use a maximum alphabet size of 52, using all lowercase and uppercase ASCII letters. Then, you go back to your original time series *X* and *R* and replace the values with the symbol from the bucket within which they fall.

The second step of the algorithm involves building suffix trees to quickly look up the probability of any suffix in a given window in *X.* After looking at many suffix tree and suffix array implementations, we discovered that we could use Python's built-in count method in the String class instead, which, when tested, offered similar performance to our suffix-tree implementation.

The third step involves stepping along the discretized time series *X* and for a given window, calculating its expected frequency in *R.* The algorithm relies on Markov Models to calculate the expected value. A score is then generated by subtracting the expected value from the observed frequency. Any score above a given threshold is flagged as 'surprising'. The details of our implementation can be found in the file Tarzan.py.

```
void TARZAN (time_series R, time_series X,
        int l1, int a, int l2, real c)
    let x = DISCRETIZE_TIME_SERIES (X, l1, a)
    let r = DISCRETIZE_TIME_SERIES (R, l1, a)
    let Tx = PREPROCESS (r, x)
    for i = 1, |x| − l2 + 1
        let w = x[i,i+l2−1]
        retrieve z(w) from Tx
        if |z(w)| > c then print i, z(w)
```

Table 4: Outline of the Tarzan algoritm: l_1 is the feature window length, a is the alphabet size for the discretization, l_2 is the scanning window length and c is the threshold

Big Data Approaches

MapReduce

For KNN, we used MapReduce to distribute the work of finding the nearest neighbors. Instead of the usual approach of reading and mapping data from a file, we used a file to define the parameters of a query for chunks of the data in our PostgreSQL database. Each line contained the arguments to the search that would yield equal parts of the data to each node. After searching the portion of the data, each mapper yielded the k nearest neighbors for that portion of the data. This was then passed on and reduced to the overall k nearest. The final results yielded the meter name and the distance between the most recent window of usage and the average of the k nearest neighbors. Each node looped over every building.

Python Multithreading

We used the Python multithreading class to implement the Tarzan algorithm. We initialized a queue with a list of dictionaries. Each dictionary provided the parameters to the Tarzan algorithm and specified which column or meter name within a pandas dataframe to calculate scores. We initialized n workers for the number of threads specified. Each worker would pull a dictionary from the queue and, when completed, would pull another item from the queue. If the item was *None*, the worker would join with the main thread. When all threads joined to the main thread, the program would end.

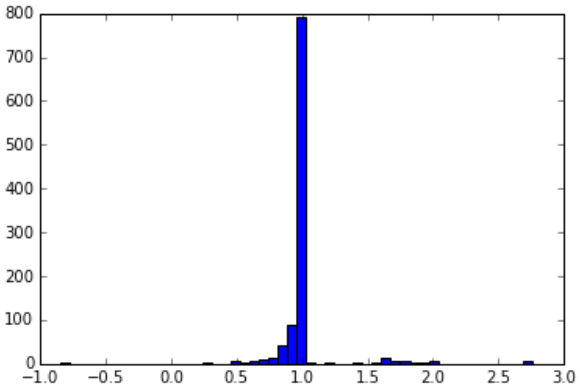
Results

Findings

KNN was sped up through the use of MapReduce, but would need additional tuning to perform well on larger datasets. In particular, it would be helpful use EMR on a larger dataset to gauge speed improvements. It may also be interesting to set up a pipeline to work with streaming data.

Both algorithms were able to find interesting anomalies. For example, Tarzan found several four hour windows during which the meter readings at Crerar Library remained

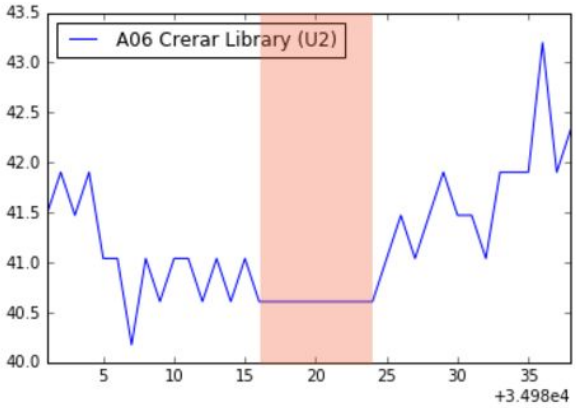
unchanged. Readings typically change every half hour, even if the change is slight. These windows could indicate periods during which the meter was malfunctioning. We also found that the Tarzan algorithm was sensitive to tuning, specifically with regard to the alphabet size, window size, and feature window size (used to discretize the strings). Best thresholds also varied from building to building and depended on the initialization of the parameters. For a baseline, we used an alphabet size of 16, window length of 8 (or 4 hours), and feature length of 4 (or 2 hours).



Right: Scores generated for Crerar (top), an anomalous period found for Crerar (middle), the time series found to be anomalous (second from bottom), and the original time series (bottom).

Run Times

When testing the k nearest neighbors approach, the local version (without MapReduce) took just over half an hour to run through the entire dataset with a window size of 10 (5 hours) and k equal to 5. The same approach running on the EC2 with the PostgreSQL database was around 5 minutes faster. We were not able to finish implementation to run on EMR. With more data, we expect that the time difference would be much larger.



	datetime	A06 Crerar Library (U2)
34995	2016-04-10 03:00:00	41.040
34996	2016-04-10 03:30:00	40.608
34997	2016-04-10 04:00:00	40.608
34998	2016-04-10 04:30:00	40.608
34999	2016-04-10 05:00:00	40.608
35000	2016-04-10 05:30:00	40.608
35001	2016-04-10 06:00:00	40.608
35002	2016-04-10 06:30:00	40.608
35003	2016-04-10 07:00:00	40.608
35004	2016-04-10 07:30:00	40.608

The basic version of Tarzan ran in 23 minutes and 29 seconds. The multithreaded version of Tarzan ran in 32 minutes and 15 seconds using three threads. Just discretizing the entire dataset took 23 minutes and 15 seconds. The multiprocessing version of Tarzan ran 9 minutes, but was unable to exit.

Challenges

We were unable to successfully use the Multiprocessing library from Python. The implementation using multi-processing was able to process all meters in 9 minutes, more than 3 times faster than the multithreading implementation and 2 times faster than the basic implementation. However, the program would never exit, and we unable to resolve this issue.

Further, even though we successfully implemented multi-threading with python, the implementation provided no performance benefit. This is because the use of the Global Interpreter Lock (GIL) prevents more than one thread from accessing the CPU. Therefore, Python code using multithreading does not scale across threads. In fact, the addition of context switching provided a significant decrease in performance.

We suspect that if we pre-processed the time series as strings and re-ingested them into the database, we could improve the processing time for Tarzan. Afterall, discretization accounts for a significant portion of the run time. As each new datapoint comes it, it can be discretized and appended to the end of the string. Unfortunately, we were unable to re-ingest the discretized strings as our EMR cluster had to be shut down before we could attempt this method. The two different json files for ingestion is included in our repo: discretization.json and tarzan.json. The discretizations.json file contains a mapping of building, window length, and datetime for each building. The tarzan.json file includes the former, along with the precomputed scores and anomalous windows.

Another point of pain was the set-up of the EMR ElasticSearch cluster. In order for the master node to shard to the slave nodes, each instance of ElasticSearch needed to be manually configured to the correct IP addresses and listening to the correct ports. However, it was difficult to know which IP address to use, as AWS provides both a public and private IP address and internal and external DNS host names. Learning to configure ElasticSearch properly represented a significant portion of our set-up time.