Rule-based Marketing Platform to Manage Call Detail Record

Chong Hu (ch3467), Wenjie Chen (wc2685), Yanchen Liu (yl4189), Jiajing Sun (js5504)
Electrical Engineering
Columbia University
Email: {ch3467,wc2685,yl4189,js5504}@columbia.edu

Abstract

Our project implement a rule-based marketing platform to manage Call Detail Record(CDR) [1]. We develop a Data Generator to simulate real-time CDR data. Streaming is achieved by message queue based on Redis and the queue manager, which is used to simulate subscriber mode. Five templates are developed based on Spark streaming to analyzed the streaming data. We choose Django framework to integrate the front-end and back-end and interact with the front-end UI. The front-end UI provides visualization of our results and the control of the back-end. Optimization and streaming algorithms are implemented to improve the performance of our system. We discuss how the size of data stream would affect the CPU utilization.

Index Terms

CDR, Call Details Records, Spark Streaming, Redis, Django, Data Generator, Template, Stream Processing

I. Introduction

Consider a phone company, Snooping Phone Services, whose management software gets a Call Detail Record (CDR) with information about each call placed using its network. Each CDR includes the caller and called numbers as well as the time and duration of a phone call. Snooping's management software should have access to a database where phone numbers are categorized by the type of business such as banking and insurance institutions, electric and gas utilities, cable and Internet services, travel services and others.

Our project aim to develop a marketing platform to allow customers to opt-in to receive offers for services that match their calling profile, possibly in return for monthly discounts in their phone bill. For instance, if a customer has recently called an insurance company, the marketing plat- form can send additional insurance offers to him or her. Similarly, if a customer is placing multiple international calls, Our marketing platform could send an offer with a special service package including long-distance calling deals.

We are implementing a rule-based marketing platform to manage the Call Detail Record. The Call Detail Record include the users ID, calling numbers, start time and duration of the call, as well as the region and business type. The first 3 templates show the total duration, business type and international region of the call records. The other 2 templates classify the tag of the individual and analyze the possibly available time for the customer to pick up the phone call.

II. System Architecture

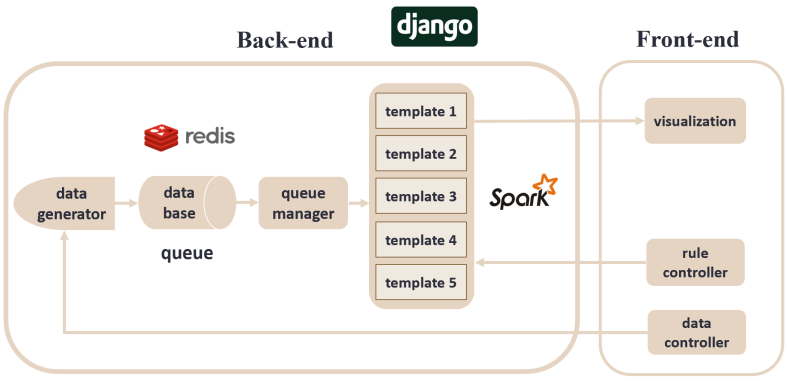

Fig. 1. System Architecture

The Call Detail Record came from the data generator, and then stored in a cache. We choose Redis as the cache of our system. After data was popped from the Redis port, they could be processed by spark streaming. We developed a queue manager to achieve the data flow. The code of Spark streaming is a part of the template. We choose Django to do the front-end/back-end interaction, as well as the integration of our whole system.

A. data generator

Each line of Call Detail Record (CDR) consists of the following information: ID of individual, calling number, called number, start time of the call, phone call duration, charge and result of the phone call. Since every individual could call many times, our data generator contains two main parts, which are single line CDR and individual generating CDR.

1) random generator: In our data generator, some of the CDR information need to be designed carefully, while some only need simple random choice, such as id, charge, result, call times. For every people and every CDR, we set different uuid to ensure the uniqueness respectively. For charge, we set it randomly. For result, we have only "answered" and "busy" two choice. We set the ratio 9:1. For call times, every people may make 10 to 50 calls one month. So these are the randomly set CDR information.

2) telephone number: The form of telephone number is set the same: "+[]-[]-[]-[]". The first bracket is the area code ranging from 1 to 300. For example, the area code for US is 1. If there are no country under this random code, we generate a new one again. For example, 11 represents no country. And we give the front end interface to control the rate of our place distribution. By default, the probability is 0.7 for the United States, 0.3 range from 300 region codes. The second bracket is three number ranging from 100 to 999. The third and the forth is similar created. Then we check the validation of every number and some prohibited cases, such as 911 doesn't appear in the number.

3) calling time: The difficulty of this part is that we need to generate call time data with order but also show randomness to analyze streaming data in sliding time window. We choose mapping function that map unix time to a virtual time in the past. In our design, every second in real world map to one month in 2016, where the call time data could be random chosen from the one-month time sliding window. Also, we give the front end an interface to control the time mode (four modes including midnight mode, morning mode, afternoon mode, evening mode) as shown in the presentation. In this way, our calling time is rich in information and changeable.

4) calling duration: The duration of all calling can set to satisfy three different type including Poisson distribution, Binomial distribution, Exponential distribution as shown in the presentation. In this part, we use the numpy-random package.

5) calling type: This part isn't shown in our original CDR. We set the type of each number and put it in the type database. There are 16 types under 6 categories. The pick rate of each type can be controlled by the front end.

6) output: Redis serve as cache and type database in the project. We build a Redis database to store the CDR type. A type and output key value are set into the database every time a CDR is generated. We use list type of Redis to add order into the streaming data. Queue manager could fetch data from Redis and send it to the specific port and IP of our templates, which will be introduced in the following subsection.

B. queue manager

After we start the data generator thread by click the start button in the web front-end, which will be introduced later, the call detail record will save into the Redis cache in a list type variable. Since we set a maximum memory limit for the Redis, we need to process the data based on the generating rate and build a dynamic balance between the data generator and the template. What we ought to do is set a back-pressure mechanism here, but based on the fact that we don't know how the company like T-mobile store their data (store same person's data in one database or distributed database), we can't simulate the situation when the processors' ability is lower than expected. So we just set a fixed sleep time for each generating cycle to guarantee the cache won't overflow.

Since we choose spark streaming as the processors' input API, we need to transform the queue data flow generated by Redis cache's pop. One of the accepted form is based on socket and bind the spark streaming API on the port. The method we apply here is to build a long-live tcp connection between the template process and the fixed port on the localhost. Once the connection is built, we don't need to change it until the system shut down.

C. template

All five templates are implemented with `spark streaming` and the main data structure used in `spark streaming` is `DStream`. To provide streaming data to `DStream`, we utilize `queue manager` to fetch data in the Redis cache and provide data source with `TCP` and `HTTP` protocol.

The `batchDuration` to acquire batches of data in continuous time is 5 seconds. All the time duration mentioned in this part is related with real world time. All five templates can be divided into two main categories. Template 0, 1, 3 are dealing with a cumulative result. For those templates, the `windowDuration` and the `windowSize` is set to the exactly same as `batchDuration`. The key functions applied here are `reduceByKey` and `updateStateByKey`. The former calculates the amount or the sum of each window; the latter calculates the cumulative result based on different keys. Template 2, 5 are dealing with the result inside each window. The window size is set as 60 seconds. The key function applied here is `reduceByKeyAndWindow`. The `windowDuration` is set to 60 secondes and `slideDuration` is set to 5 as we sliding window with each batch. The streaming can obtain some users' feature inside one time window. When new batch comes in, it will count the new batch and drop the elder batch's result. Therefore, the templates can always maintain the result of current time window.

1) First template: In template 1, CDR data is read from port 9000. We map the `DStream` by splitting the required information from CDR data, which are ID, call time, duration. We employ `reduceByKey` to further extract the information from rdd in `DStream`. `updateStateByKey` is used to sum all the call duration. In this way, we could sum the call time duration of every hour in 24 hours. The results are saved as json file. Optimization algorithms such as operator reordering are implemented to improve the system efficiency, which will be explained in later section.

2) Second template: Template 2 is basically the same with template 1. CDR data is read from port 9001. Besides the functions of spark streaming, template 2 connect to the type database the get the type of the call number for each rdd. We could get the sum of total call times of each type as the output. Optimization algorithms such as operator reordering and redundancy elimination ensure the high performance of this template.

3) Third template: Template 3 is similar to the first two templates. CDR data is read from port 9002. In this template, we count the total call times of different regions for the international call. Since 0.7 of the call numbers are from the United States, we split out the US in visualization results. We use operator reordering to firstly `reduceByKey` and map then in this template.

4) Forth template: Template 4 labels each user by call type and tag, which is the output of this template. Since user-based and product-based are the two main modes in the marketing system, we would like to analyze the similarity of call types made by user. We implement k-mean clustering algorithm to analyze the call types and generate a specific tag for the user, such as Business, Health, Education. For example, Bob is classified as Business tag and make the most calls for the housekeeping and property management type. We could then give him an advertisement call from a housekeeping company.

5) Fifth template: Now we have knowledge of the individual tag so concrete promotion and advertisement could be sent to the user. Template 5 try to figure out what is the most possible time for the user to pick up the phone call. This is achieved by finding the specific hour and day during the week that user made the most phone calls by spark streaming. The results show the time that individual is most likely to be free. For example, Bob made the most phone calls on Friday 10 am in the morning. Therefore, our marketing platform could give him a call from housekeeping company during that time period. By template 4 and 5, we can successfully cluster the person's tag and compute the most possible time for him/her to make a phone call. State sharing is implemented here to optimize the template, which will be introduced in the later section.

D. web interface

In this subsection we will introduce the interaction between front-end, the html page, and the back-end, the Django. We will also introduce what happens during a whole back-end cycle, from clicking start to changing the parameters of the data generator and start a new cycle.

1) Django: Django is a open source Python back-end framework. It gives developer less flexibility but developing quickness based on its variety developed module. A Django back-end workspace often includes several python files, such as setting, wgsi, urls and views. For a given url visited by front-end post or get request, the back-end will return a http response or render a html back to front. Since this is a stream project and all data we use is simulated by ourselves, we didn't apply a database such as MariaDB or MongoDB. We just store the processing result as json file and visualize the data in the front end.

Since the url written in the url.py file is linked to the route in views.py, I will just introduce the most frequent used functions in the views.py. Before anything happens, the queue manager for different template will start in another process first.

There are three functions related to the control of the data generator: start, stop and change. Once the system is started by clicking the button, data generator will start at a new thread in the Django process with default parameters. Start url function will be called once to go in this stage. If we want to change the current parameters and observe another distribution of the call detail records, we will receive a http request from front end first. After the receiving, we call stop function first to terminate the generating thread then restart a new thread with the new parameters input. Steps above is packaged as a url function called workload generator, which is related to the change. Other functions in views.py are mainly related to return the front end a html page. This part will be explained in detail in the front end section.

2) front end: The front end cooperates with Django and provides basic interactive UI to control data generator and templates, visualize data and make promotions to certain group of people.

- *Welcome Page*: At the very beginning, after startiing django application in server, user will get a welcome page when access to the website. The start button triggers the workload generator and redirects to dashboard.
- *Workload Generator Page*: This page is used to revise the workload generator. The user can select and fill different parameters, post to the website and update the workload generator. All those variables are the input parameters of the workload generator in the back-end. After pressing the update button, new parameters will be past to back-end and the browser will be redirected to dashboard page.
- *Dashboard*: In the dashboard page and template pages for each template, the html page visualize the data transmitted from Django. All the charts will auto refresh with certain time interval. In this project, the time interval is set to 1.5s. The auto-refresh is triggered by the javascript function in the client's browser. Beyond that, the page provide a side bar for access other components directly.

- *Policy & Plan*: This page is a simulation of real decision making platform. The table in the right shows filtered result of current time window, which represents some features of certain people. This implies that the system has already applied some rules or do some promotions to those users. The decision maker can apply some rules to those people, such as provide discount for certain people. In the next time window, new data will come in and old data will go through. The form in the right provide user defined filters to get target people. After press update button, new filters will be applied.

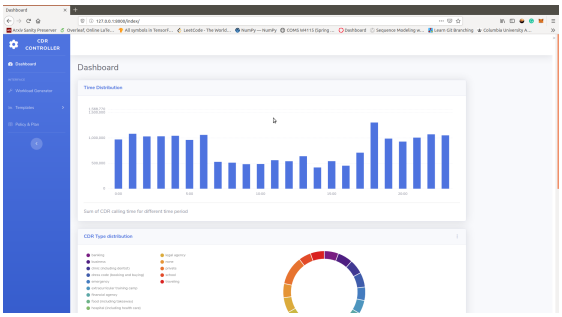

Fig. 2. Dashboard Page

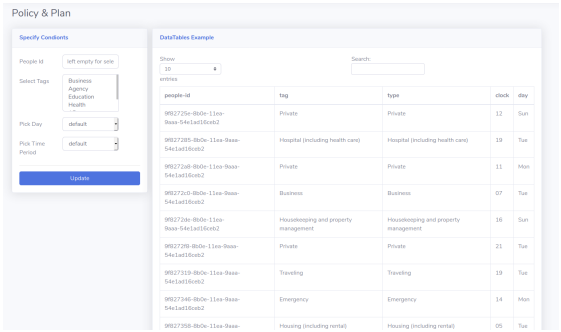

Fig. 3. Policy & Plan Page

III. Algorithm

A. optimization algorithm

As taught in class and implemented in homework before, streaming applications are programs that process continuous data streams, thus yielding various stream optimization algorithm. In our template, we have also used some optimization algorithm and made a brief analysis [2].

1) operator reordering: operator reordering is to move more selective operators upstream to filter data early. And basically we use it everywhere in our template. For example, we choose whether to put ”spark map” in front of ”spark reduce” or vice versa. It’s obvious that reduce the large data first is better, but we still do a small analysis to see if it works. We use our CPU load percentage to do the cooperation. The effect of only one used optimization algorithm is slight, about 5 percent difference. We will show the combined result when all optimization is used.

2) state sharing: State sharing is profitable for throughput if it reduces stalls due to cache misses or disk I/O, by decreasing the memory footprint. In our application, we need to continuously cluster the person with a tag and compute his/her most likely free time for a time window. If we maintain large windows for both aggregation, the memory requirement will be substantial. We let them share the same aggregation window, thereby reducing the memory for both template. Therefore, when developing and introducing the template, we split it into two independent templates. But when running the whole program, we let the templates share the same state, thus reducing memory.

3) other optimization algorithms: Redundancy elimination is also common in our program. For example, when getting number’s type from the type database and getting number’s calling time, we can reduce the duplicate operation and only connect to the database one time. Another optimization algorithm we use is operator separation. For example, we cluster the people as different tags (business tag, education tag...). The tags are based on all of their calling numbers’ type. We separated the operation, and count the most called type for each people too, thus making people’s analysis more complete. The optimization algorithms are all kind of thought, and we try to apply all of them into our program to fasten it.

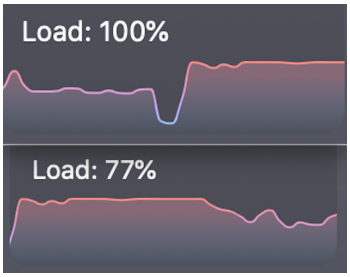

Fig. 4. cpu load

You can see in Fig 4, the cpu load comparison of two streaming procedure. In the first procedure, we write the templates and all processing program intuitively, without considering optimization. We set the cpu computing power right now as standard

load 100 percent. In the second procedure, we applied all the optimization algorithm discussed above, and consider shedding some operation, balancing the load. Then the cup load is greatly reduced, 77 percent. So our optimization algorithm is applied successfully, and the stream processing is faster now.

B. streaming algorithm

Except for streaming optimization algorithm, we can also run online clustering method to classify different kinds of call detail records. The class of each cdr data is mainly depends on the user behavior and the type of calling number is one of the most important feature. We run k-mean clustering online to classify the user behavior. The initial weight is obtained from some cdr data offline so that we can pre-define and adjust how many centroids are needed and what attribute those centroids has. We assigned numerical value to different labels and discrete numerical value to continuous range. All data used in clustering are normalize in pre-process stage and reassign different weight based on the importance of each attribute. The type of different calling numbers is most important one while the calling time has relatively low weight. We applied Euclidean distance to calculate the difference between those data. Once the system starts analysing cdr data, the center of different will be re-computed as new data come in. The class of each new coming cdr data will be given by those new centroids.

IV. Summary and Further Discussion

A. Result

To future verify the robustness of our cdr analysing system and to investigate how performance changes as data stream becomes larger, we adjust the time interval that we acquire data from our workload generator and record how our cpu performance changes. The sleep time interval is from 0s to 1s and the result is shown in Fig 5.

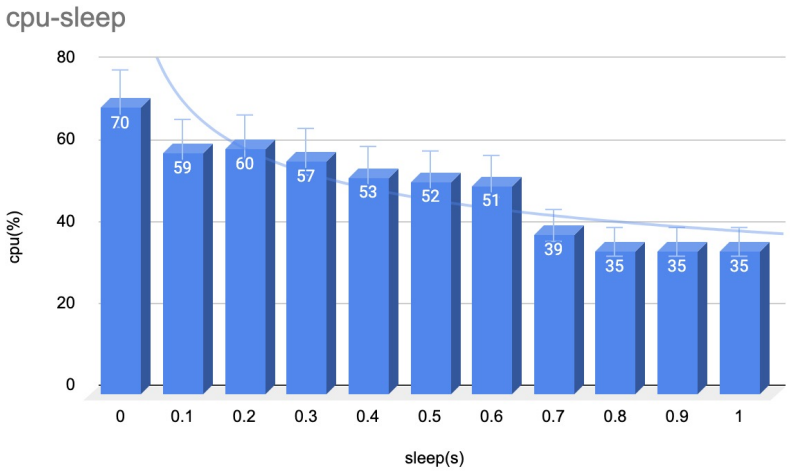

Fig. 5. CPU Performance vs. sleep time interval

In this experiment, the average cpu utilization is roughly about 70% when the sleeping time is 0s. This represents that the streaming process engine acquire data continuously without sleeping and the data streaming is largest in this experiment. As the sleep time increase, which also means the data streaming becomes smaller, the cpu utilization is decreasing. When the time interval becomes larger enough, about 0.8s, the decrease on cpu utilization becomes not evident. In this scenario, the cpu utilization is mainly spent on other back-end programs instead of data stream. Those back-end program might include the Django application, the maintaining of spark streaming or other computer programs.

B. Conclusion

In our project, we finish the rule-based marketing platform to manage the Call Detail Record, which includes the data generator to provide real-time cdr data, the message queue based on Redis and the middleware queue manager to simulate subscriber mode, different templates to analysis streaming data by using spark streaming, and the front end UI to interactive with our back-end by applying Django framework. We also apply optimization algorithm and streaming algorithm that we learned in class to improve the performance of our system and to better investigate user behavior. In the end, we discuss about how the size of data stream affect the cpu utilization in our system.

C. Future Work

Although we have completed out project successfully, we still have a lot of things to better improve our system. For the system-side, we can use kafka as message queue to deliver message for better extension ability and larger scale data stream; optimize templates and data processing flow; provide more api to control each templates; add back pressure mechanism. For the front-end, we can also add More buttons and options to control templates and login method for admin.

ACKNOWLEDGMENT

First, we would like to thank Professor Deepak S. Turaga. Without you, we can never finish our final project from scratch. Thank you for the instructions and help of a whole semester!

Second, we would like to thank all the authors and the contributors of our reference material. They have done a great work before us.

REFERENCES

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Application development – data flow programming*. Cambridge University Press, 2014, p. 106–147.

[2] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, Mar. 2014. [Online]. Available: https://doi.org/10.1145/2528412