Large-scale Customer Location Inference

Student: Duc Trung NGUYEN
Supervisor: Prof. Pietro MICHIARDI

Contents

Acknowledgment

I would like to express my great appreciation to my supervisor, Professor Pietro Michiardi, who not only gave me the opportunity to work on this wonderful project on the topic **Large-scale Customer Location Inference** but also taught and guided me many things: from research, describing ideas,... to other aspects in life.

Secondly, I would also like to thanks Marcin, Michal and other SWISSCOM colleges who helped me a lot in this project generally, and in my trip to Switzerland.

Advices given by Xiaolan and Yufei has been a great help in developing the background knowledge, refining the algorithms and the report.

I wish to acknowledge the helps provided by Mateo, Mario and Hung about technical stuffs from the starting time till now.

I would like to offer my thanks to my parent, my dear friends who gave me the advices and the best condition to do this project.

Thanks again to all.

Abstract

The next locations of mobile users attracts the consideration of many researchers and telecommunications providers, including Swisscom. Having knowledge about customers' locations can help Swisscom in providing better services such as increasing media streaming experience, network congestion detecting... In order to forecast users' locations in the near future, we design algorithms in two approaches to build models from overlarge datasets : tree-based approach and trajectory-based approach. Both solutions provide the acceptable results with the accuracy around 80% and 65% respectively. Furthermore, the second approach also supports adjusting models incrementally time to time.

Introduction

The context of this project lies in mobile network service provisioning, and it is related to the quest for better customer experience when offering mobile network services to customers.

In the mobile network we consider in this work, some critical components – labeled enforcement points (EP) – execute traffic engineering tasks to decide, for example, given the current state of the mobile network for a particular user, which media encoding to use to deliver data to a mobile terminal. Today, EPs may behave similarly to an admission control mechanism and they accept all traffic or none.

The endeavor of this work is to inject additional information in EPs, such that better decisions (and not only all-or-nothing) could be taken. Specifically, we consider radio access network (RAN) conditions, which include: congestion in the RAN, when EP should implement their decisions, knowledge about the serving cell (SC) for each customer, ...

Now, the knowledge about the serving cell for each user is hard to obtain in real time, which may limit the usefulness of such information to EP. In particular, due to the current state of the technology used to operate the mobile network, it is possible to measure the serving cell for a user only with a non-negligible delay (in the order of tens of minutes, e.g. 15 minutes). Given such limitation, the customer location inference problem can be tackled with simple heuristics, such as bounding the region surrounding critical EPs, where the bounding box should be sufficiently loose to allow recovering the delay in obtaining the SC information. These heuristic may work in practice, but their merit still need to be assessed. Moreover, the heuristic approach is non-scalable, and can only be applied to a selected subset of "hot" EPs.

As such, in this work we tackle the problem of providing estimates of customer location, in the sense of trying to estimate (or predict), at a given point in time, which users will be served by which SC. Based on delayed (and possibly stale) measurement data, combined with historical data of customer location, we want to produce accurate estimates of customers' SCs.

We believe addressing this problem to be feasible based on prior work such as [16], in which Barabasi et. al. state that "human trajectories show a high degree of temporal and spatial regularity, each individual being characterized by a time-independent characteristic travel distance and a significant probability to return to a few highly frequented locations" and "93% potential predictability in user mobility across the whole user base". Therefore, it's possible to build a model to predict the next location of user.

There are two system architectures that we can consider to build model: Batch processing and real time system [15]. In our work, currently, we only focus on batch processing.

In the batch processing architecture, by using machine learning algorithms, historical data is used to train a model, which will predict the next location for users. Note that, in general, a model has a lifetime, or "age". When the model is stale, historical data combined with new data is used to train a new, updated model.

Another problem we should consider is, what is the scope of models. In literature, there are two kinds of models: Individual models and Global models. Individual models are the trained models for each user, and be used to predict the next location of this user only. In the contrast, global models are built and used to make predictions for all users. The former is more widely used. However, in this project, we will build and evaluate both model scopes.

In this project, we use two approaches to train models : tree-based and trajectory-based approaches. As the former's name, we use Decision Tree and Random Forest to predict the target feature: Next location. In trajectory-based approach, we try to identify and group the similar daily mobility paths of users, generate models and use them to make predictions. Decision Tree and Random Forest will be used as the baseline to compare the performance with our new algorithm in trajectory-based approach. The implementations of both approaches is written in SPARK - a very powerful scalable framework - which be introduced in section 1.2.

The rest of this document is organized as follows: Chapter 2 is related works and some background knowledge will be used in this project. In chapter 3, we introduce and describe in detail the data we use to build our statistical models, and to validate our algorithms. Chapter 4 and 5 talk about tree-based approach and trajectory-based approach. Finally, we come up with the conclusion in chapter 5.

Chapter 1

Background and related works

1.1 Related works

Our goal in this project is designing scalable algorithms to build models from very huge data from SWISSCOM, and use them to predict the next location of users. Therefore, we focus on two families of related works: the studies of location inference problems and scalable machine learning algorithms.

1.1.1 State of the art in location inherence

We reviewed a number of related works that address similar (or sometimes, the same) problem we focus on in our project. Note that almost all papers are the result of research efforts in addressing machine learning challenges in 2012 from NOKIA and ORANGE. Unfortunately, the original data used by the research papers below are no longer available.

Tran et. al. [21] predicted the next place which user will go to in the future using an approach based on user-specific decision trees learned from *each user's history.*They discrete time series data into 10-minute windows and then label some special places: home , office (by using some properties), and use them to detect when the user changes his habit (in order to update our model). Additions, they used the Holidays Detection mechanism to aggregate the data. After that, Decision Tree algorithms, i.e. $J48$ - an implementation of the C4.5 algorithms to build the tree model for each user and use these models for prediction later on. The used features are: "PlaceID", "IsHoliday", "IsWeekend", "Weekday", "LeavingTime", "Duration". Feature "IsHoliday" will be determined based on the holiday calender type of each user, which will be chosen by cross-validation method. They also used the information of call logs, sms logs, user's calendar and some other parameters to adjust the decision. This study used many mechanisms, such as Holiday Detector, Parameter Optimizer, Location Changing detection... to increase the performance, The accuracy of this method is about 61.11%. If the users have simple movement patterns, the accuracy can reach more than 80%. However, they lack the mechanism to auto update models when having the new data, and cannot make prediction for some some who didn't participate in the training before.

In 2013, João Bártolo Gomes et. al. [6] focus on the predicting the next location problem of a

mobile given data on his current location. They provide a framework which using spatial, temporal data as well as other contextual data such as bluetooth, call/sms logs, accelerometer. Besides, this framework is executed on the mobile device itself in order to respect the privacy of users. They use Massive Online Analysis + Hoeffding Tree + Probabilistic model. They tried to build the model for each user to predict the next location based on the current location only (without data about previous locations). The raw data of each user's location will be transformed infor semantic places by using Four-square, Facebook and Google Latitude... to build *Anytime Model*, which has ability to adapt to the new data. *Accuracy Estimator* - the component takes the role of comparing the anytime model prediction and the actual location is set up to keep an estimation of the next place prediction accuracy. The features were used in their study are temporal features (duration of the visit, the day of week, isWeekend, period of the day, hour of the day), Phone status (general, silence, min/max of battery level...), Phone usage (call logs, sms logs...), Environment features(accelerometer, bluetooth, wlan, gms).The accuracy when applying their method on data from Nokia challenges is 59.6%. The models of this approach are adaptive time to time. However But the users have to install the software which contains the proposed framework on their phone.

Still in the Nokia Challenge "The next place prediction", Etter et. al. addressed the problem based on graphical models, neural networks , decision trees and some blending strategies. In there, they listed three characteristics of the data what they think are critical to the prediction task:

- User Specificity : It is not possible to build joint models over the user population to learn from some one to make prediction for another.
- Non-Stationarity : User can change his/her habit overtime (when moving house or office)
- Data Gaps : For some user, we have no information about them in long periods. And these gaps are sometimes followed by change of mobility habit.

To solve the problem, they used Dynamical Bayesian Network, Artificial Neural Networks and Gradient Boosted Decision Trees to build three different models for each user, compare them and combine several predictors to increase the accuracy. The features were used in building model phase are: Location, Start time of visiting (Hour, Day, Weekday), End time of visiting (Hour, Day, Weekday), IsTrustedTransition. The average of accuracy is more than 60%.

Using another approach, J. Wang et. al. want to predict the next location of a user based only on his trajectory [22]. They assume that user behavior exhibits strong periodic patterns. The model for each user will be built based on Periodicity Based Model and Multi-class Classification algorithms. The features which were used are: starting time of a visit, end time of a visit, current location. Where the time included: day of week, hour of day, hour of week, weekend, weekday, morning, noon, afternoon, evening, midnight. The accuracy of this approach when applying on Nokia challenge's data is around 55%. This approach doesn't consider the relationship between the next location and the previous locations and isn't suitable with small training dataset.

1.1.2 Scalable machine learning algorithm

In 2009, Google introduced **PLANET**, a scalable framework for learning tree models from over large datasets [18]. PLANET defines the tree building procedure as a series of distributed computation,

and implements each one using the MapReduce model of distributed computation.Besides, because tree learning is an iterative algorithm, PLANET needs to use a special program to chedules and controls the entire tree induction process on distributed machines. The machine running that program called "Controller".In order to control and coordinate tree construction, the Controller maintains the followings:

- *ModelFile*: the entire tree contructed
- *MapReduceQueue (MRQ)* : contains the nodes which need to be expanded but having too large training data to fit in memory
- *InMemoryQueue (IMQ)* : contains the nodes which need to be expanded, and having the training data that can fit in memory

Whenever a node is expanded to two children, each child will be appended into the MRQ or IMQ, depends on its training data. Once PLANET checks that MRQ or IMQ is not empty and there are enough resources, it launches a MapReduce job in a separate thread to expand the current nodes in this queue. The memory limitations of a machine and the number of available machines on the cluster often prevent the Controller from scheduling MapReduce jobs for all nodes on a queue at once.

The induction procedure will stop if there is no running job and all queues are empty.

In the design of PLANET, when scheduling a set of nodes, the Controller does not determine the set of input records require by the nodes. It simply them the whole training dataset **D** to every job.If the input to the set of nodes being expanded is much smaller than **D**, then the Controller will send much unnecessary input for processing.

Although Google shows that PLANET work well in practice, but it's still closed-source and not directly usable by the broader community. Inspired from this study, Wei Yin et. al. introduce an open-source version of PLANET, called "OpenPlanet", which helps building scalable regression tree on Hadoop. (We will discuss about Hadoop in detail in section 1.2.2). The Controller runs one job at a time to build all nodes at the same level of the regression tree.They also tuned and analyzed the impact of parameters such as HDFS block sizes and threshold for in memory handoff on the performance of OpenPlanet to improve the default performance.

1.2 Background

1.2.1 Map-Reduce

MapReduce is a programming model and an associated implementation for processing scalable problems accross huge datasets using a large number of computers (cluster). MapReduce can take advantage of locality if data, processing it on or near the storage assets to reduce the distance over which must be transmitted (communication cost). By words, MapReduce helps us move the computation instead of moving data. MapReduce was introduced by Google in 2004 [12]. It's inspired

by Map/Reduce in functional programming languages, such as LISP from 1960's, but its purpose is no the same as in their original form.

In perspective of programming model, a MapReduce program (also called job) is a compose of a *map function* and a *reduce function*. The map function takes the input, transform it into intermediate results. The reduce function performs a summary operation on these intermediate result to return the final output.

1.2.1.1 Locical view

In logical view, a job has two main phases:

- **Map phase:** The master node takes the input, divides it into many smaller parts and distributes them into worker nodes. The worker nodes scan over the its input, compute the pair of key/value pair for each piece of input in parallel and pass the answers back to its master node.
- **Reduce phase:** The master node collects the intermediate result from workers, groups them by key and dispatches each group to a worker. These workers apply a reduce function on each group to calculate the final result in parallel.

For example, we want to calculate the frequency of each word in a huge document. In the map function, every time we meet a word w, we emit a pair $(w, 1)$. The input of the reduce function is a pair of word, and the associated list of frequencies of this word.

Algorithm 1 Example of map-reduce job

```
function MAP(document)
    for each word w in document do
        Emit(w, 1)                                   ▷ key = w, value = 1
    end for
end function
                          ▷ The pairs are group by keys, and be the input for reduce function
function REDUCE(w, List_of_frequencies)
    sum = 0
    for each value in List_of_frequencies do
        sum = sum + value
    end for
    Emit(w, sum)
end function
```

1.2.1.2 Execution View

The map invocations are distributed across multiple machines by partitioning the input data into a set of M pieces. This partitioning task can be done in parallel on many machines. Similarly, the

reduce invocations are distributed by partitioning the intermediate key space into R pieces using a hash function. The number of partitions R is decided by the user. The below figure show the execution of a MapReduce job when running with the implementation of Google:

1. The MapReduce library firstly splits the input data into M parts and then starts many copies of user program on a cluster of machines.
2. Among the copies, there is a special program : master. The others are workers that are assigned work by the master. The master picks up the ide workers and assigns each one map or reduce task in total M map tasks and R reduce tasks.
3. The workers which were assigned map tasks read the corresponding data, apply the map function and then produce the intermediate key/value pair and store them in memory buffers.
4. The buffers are written into local disk and partitioned into R regions periodically. The information of these regions are sent back to master who is responsible for forwarding these location information to reducers.
5. When a reducer gets the notice from master about these locations, it reads the data from the local disk of map workers by using remote procedure calls. After reading all intermediate data, the reducer group them by their keys.
6. For each unique intermediate key, the reducer pass the key and the associated set of values into the Reduce function. The ouput of Reduce function are appended to a final output file for this reduce partition.
7. After MapReduce job completed, we have R output files, each file is produced from one reducer.

Figure 1.1 shows the workflow of a MapReduce job.

1.2.2 Hadoop

Hadoop is an open-source implementation of Google MapReduce, an Apache top-level project being built and used by a global community. The Appache Hadoop framework contains some main components [1]:

- **Hadoop Common:** contains the libraries and ultilities for other modules
- **Hadoop Distributed File System (HDFS):** a distributed file-system that stores data on commondity machines, provides high aggregate bandwidth across the cluster
- **Hadoop MapReduce:** a programming model as described above

The architecture of Hadoop is a little bit different from Google MapReduce, but the key idea is the same.

Hadoop is used widely by many big companies, such as Yahoo, Facebook... and many organizations.

Advantages:

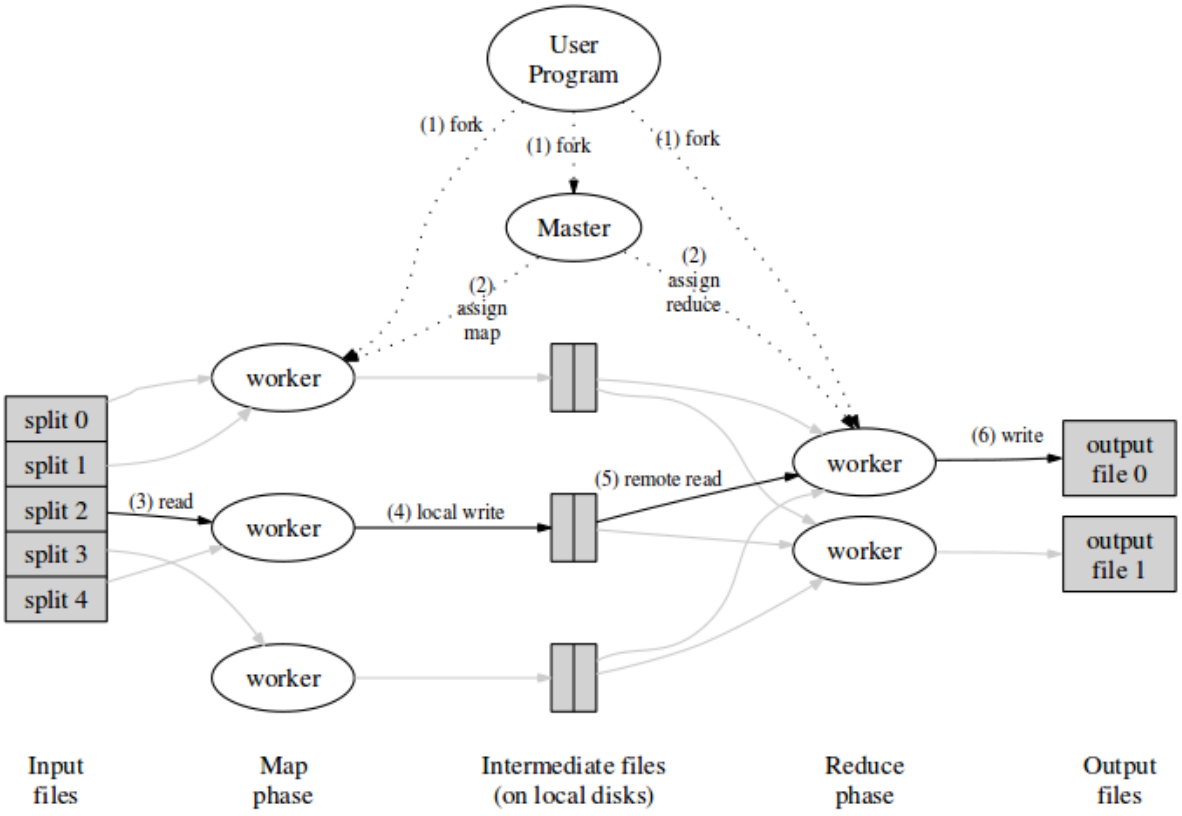


Figure 1.1: MapReduce's execution flow

- Simple and elegant concept
- Powerful
- Building block for other parallel programming tools: such as Pig, Hive...
- Extensible for different applications
- Good failure tolerant

Disadvantages:

- Poor support iterative algorithms
- Doesn't support real time processing itself

1.2.3 SPARK

Like Hadoop, SPARK is an open-source framework for fast and flexible large-scale data analysis, originally developed in the AMPLab at UC Berkeley. SPARK also was built on top of HDFS.

However, SPARK is not tied to the two-stage MapReduce model. It allows job to load data into cluster's memory and query it repeatedly, making it suited to machine learning algorithms. It also promises increasing the performance up to 100 times faster than Hadoop for certain applications.

SPARK's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from HDFS files, (or other Hadoop InputFormat), or transformed from other RDDs. Each RDD has transformation functions that transform itself to other RDD, and action functions that return values, like map and reduce functions in Hadoop. RDDs support many persistence levels of storage: Memory only, Memory and Disk and Disk only... Since the RDD is mainly stored in memory and reduces I/O (by caching mechanism) and the complexity of driver (it's driver itself), it supports very well to iterative algorithms. Besides, SPARK can handle well both batch processing and real-time processing.

Besides, SPARK support both streaming and batch processing.

Because of these advantages, we choose SPARK as a framework for scalable environment.n,

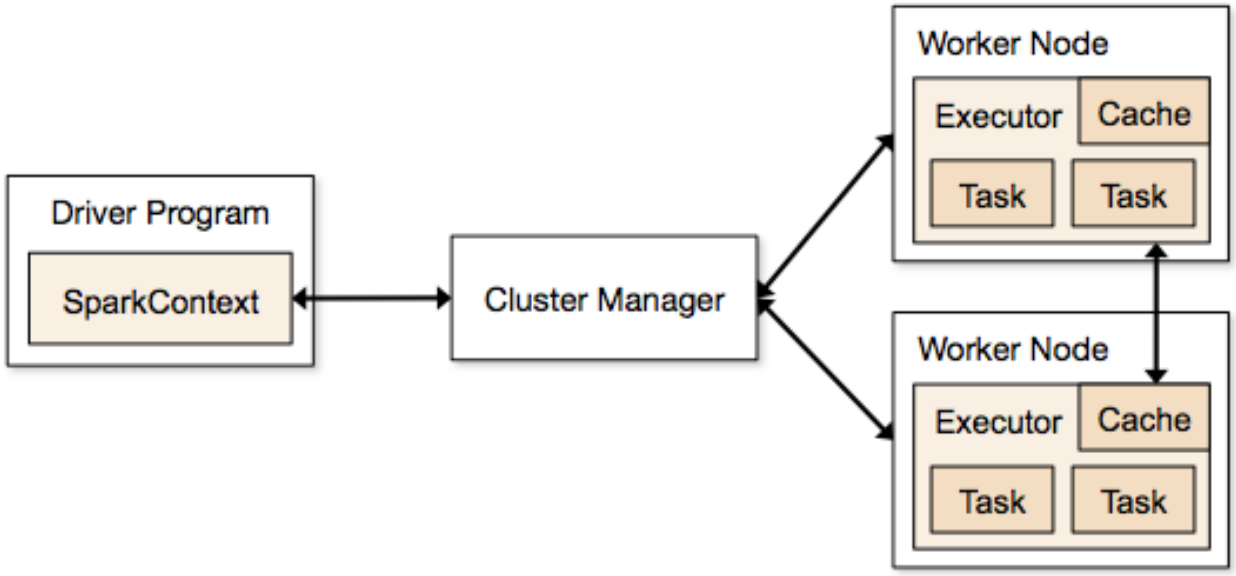


Figure 1.2: SPARK's architecture

Chapter 2

Data

2.1 Introduction

Beside the data from SWISSCOM, which is private, we used an additional similar dataset from a project of MIT. In this chapter, section 2.2 introduces about these two datasets as well as describe some tasks to analyze them.The first task is pre-processing data to make it adapt to our approaches (section 2.3). After that, in section 2.4, we do some data analysis to retrieve the properties of data. Finally, we discuss about the method to sample the data for our algorithms in section 2.5.

2.2 Datasets

2.2.1 Data from SWISSCOM

Each day, SWISSCOM collects terabytes data of customers behavior data contains the following fields:

cid	customer ID, anonymized version of a customer unique identifier
cell	serving cell unique identifier, where actually this field conveys information about the sector inside a serving cell and the bound
dt	timestamp
event type	events to related to signaling

An example line of such a log file is given below:

cid, cell, dt, event type

7fb514974789b80b91692f849f3c0c5,FLAH2F,2014-01-12 16:11:32.000265,12

It should be noted that precise location information about SC is available through a separate database (or dataset) which contains information about the topology of the mobile network and its SCs. Precisely, this database provides information about the GPS location of SCs plus a lookup service to discover neighboring SCs of that provided for a particular customer at a particular point in time.

This means, in practice, that some relational operators akin to a JOIN will be required when working on the full-fledged prototype, whereas for the initial study, it is possible to simplify the problem by denormalizing the information contained into multiple tables and the log file into a larger log file, enriched with precise location information of SCs. We shall call the latter log file, the offline log.

In the context of this project, we are interested in predicting the next Service cell ID (and eventually its coordinates) to which a user will connect to, as opposed to a precise tracking of the GPS coordinates of a user. We will address the "next cell problem" considering a range of forecasting scopes, from few tens of minutes to few hours in the future. An important requirement that we will take into consideration is the time it takes to make a prediction, given a model is available.

2.2.2 Additional dataset

Due to some data availability problems, we first focused on finding publicly available data with a "schema" similar to what we expect to have from Swisscom. Currently, we use publicly available data collected in the Reality Mining project [13] at MIT. In the Reality Mining project, information about device logs, Bluetooth devices in proximity, cell tower IDs, application usage and phone status were collected from more than one hundred Nokia 6600 smart-phones via several pre-installed pieces of developed software and a version of the Context application from University of Helsinki. This study generated data covering over nine months, with about 500,000 hours of data on users' locations, communication and usage behavior. In the middle of 9-months study, the research group at MIT conducted an online survey for users, which had to answer questions related to social relations so as to infer also a friendship network. The subjects of the study are seventy-five students or faculty in the MIT Media Laboratory, twenty-five incoming students at MIT Sloan Business School adjacent to the Laboratory.

The dataset we used comes in the proprietary MATLAB's format, and contains many features. Below, we list a subset of these features that we deem relevant for our study:

	StartDay	the starting day of the study duration on users
Communication	Date EventId ContactId Type of event Direction Duration	The timestamp Unique event ID The contact ID in phone's address book (-1 = not in address book) Type of communication Direction of the event ("Outgoing" or "Incomming") Duration of the event
Tower transition	Time Area.CellID	The time when user swiched cell tower Complex value XXX.YYY where XXX os tje area ID and YYY is the cell ID

In addition, we are interested in considering additional features, such as information about the days SMS were sent or received, number of voice calls, home id, and many more. However, this information may not be directly available in the Swisscom dataset: for this reason, in this report we neglect such additional features. It is important to notice that there is no available mapping between Serving Cell IDs and their physical location. Instead, this information will be available in the Swisscom dataset.

2.3 Data pre-processing

In order to facilitate pouring data to different processing frameworks,different algorithms..., we transformed, filtered and saved the dataset from MATLAB format to a CSV format. The size of the final file is 124.5 MB with more than 3 millions records (or observations) with the schema: *userID, year, month, day of month, day of week, hour of day, minute, areaCellID, AreaCellIndex.*

The data is ordered by $UserID$ and then *timestamp.* For instance:

UserID,Year,Month,DayOfMonth,DayOfWeek,Hour,Minute,AreaCell,AreaCellIndex

2,2005,1,26,Wed,16,42,24127.0011,35

2,2005,1,27,Thu,1,45,24127.0011,35

3,2005,1,23,Sun,2,42,24123.0011,38

The column $areaCellIndex$ is the mapping from $areaCellID$ value to an unique value.

2.4 Data pre-analysis

Following a similar approach to that in [14], we now proceed with an exploratory data analysis and plot the "movement habit" of each user. As an illustrative example, Figure 2.2 and figure 2.1 depict the location (in terms of Serving Cell ID) in different times of different days for user 93 and 65 respectively. Specifically: the x-axis is the relative day of the study; the y-axis is the hour of day.

Figure 2.1: The movement pattern of user 93

Location information uses a color code, with a different color for each unique Serving Cell ID, and with the convention of an empty line indicating that there is no information or no signal at all. For example, Figure 2.1 indicates that user 93 has frequent movements from 9AM to 9PM in most of the days. So, this user may be "classified" as a "regular user". In the contrast, as Figure 2.2 indicates, user 65 is an "irregular user".

2.5 Data sampling methods

Next, we discuss another important step for the preparation of data in light of training a model for the prediction task. Sampling methods amount to devise a methodology to split the dataset in a number of parts, which will be used for the model training, and for the model testing, that is, for assessing the prediction quality of the model.

There are many strategies to perform sampling:

Strategy 1: Given a dataset with N observations, randomly pick up samples to obtain two disjoint

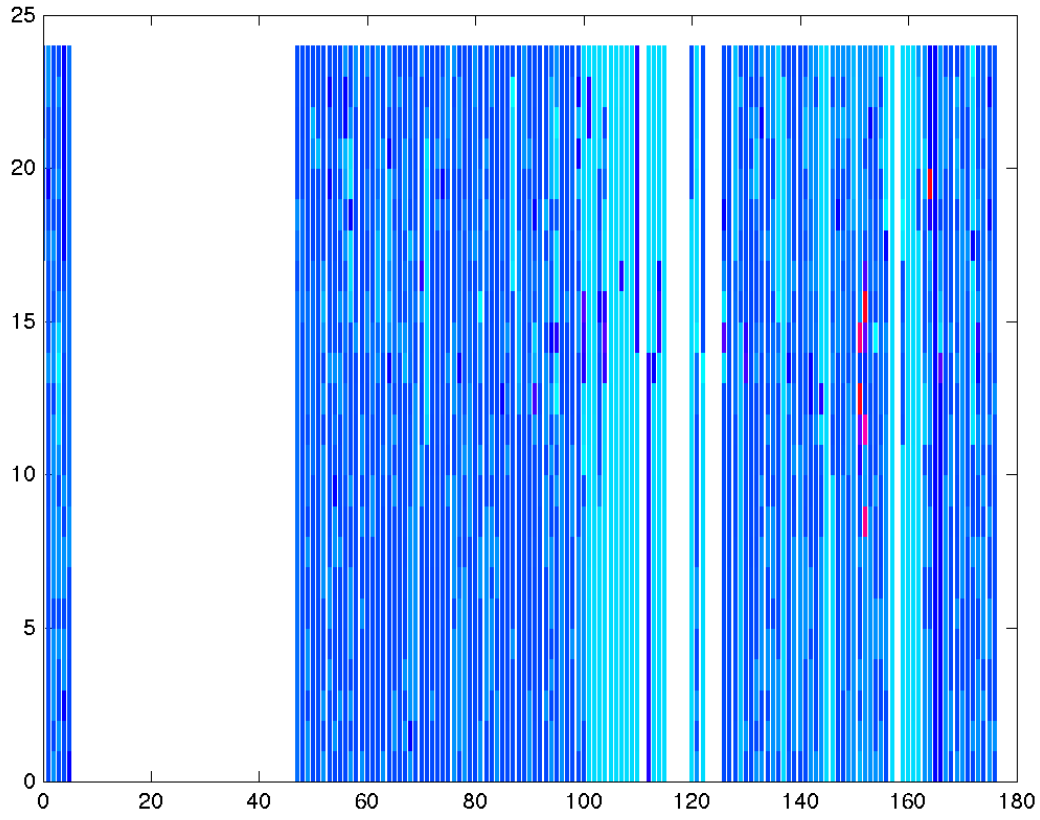


Figure 2.2: The movement pattern of user 65

sets: training set with pN observations, testing set with $(1-p)N$ observations, where p is the the proportion of training set over total data, determined by user ($0 < p < 1$). This strategy is very simple, but it has bias toward users that have more observations than others.

Strategy 2: For every user, consider every observations (ordered by timestamp already), take a subset of such observations to use as training, and the rest for testing. E.g. $userID = 60$ has 100 records, consider the first 80 records as training, and the last 20 records for testing.

The advantages of this strategy are : simple, and can adapt to algorithms that use historical information to predict the future data. But this strategy requires more data scanning to sort data by time and user, and still have some bias for users that do not have many observations or have many missing values.

Strategy 3: Given a dataset with N observations, take the first pN observations for training instead of random selection as Strategy 1, and the rest for testing ($0 < p < 1$). This strategy is simple and very fast but still has the same problem like other strategies: bias problem (some users may can have any observation, or very little).

In this project, we want to use the historical information for building models, and every user should have their observations in the training set - the factor we think that can have positive effects to the accuracy of algorithm. Therefore, in our evaluations, we will use strategy 2 to sample the data.

Chapter 3

Tree-based approach : Decision Trees

3.1 Introduction

Decision trees are one of the oldest and most popular data mining models. With many advantages such as inexpensive to construct models, extremely fast at classifying unseen records, the accuracy is comparable to other classification techniques... decision tree is often used to address the location inference problem.

In this chapter, we introduce about different kinds of trees and the procedures to construct them and some techniques which help increasing the performance accuracy of models.

The main challenges when using this approach in the project is how to parallelize the tree inducting process in the scalable environment with SPARK. We address these challenges in section 3.3.2 by introducing ***Labelling Tree Building*** algorithm to construct decision trees, and the parallel implementation of other techniques.

The idea of our approach is very simple: we consider the next location of user as the target feature (what we want to predict), the other information is the predictors (what we use for predict), and using tree learning algorithms to build models. In order to use these algorithms effectively, the domain value of the target feature must be discrete or be a number in $\mathcal{R}$. It means, if locations are represented by a service cell IDs (categorical feature), we use the classification (CART or ID3) for building models. In contrast, if locations are constituted by GPS coordinates, we can not use tree model for predicting these complex values (because there are no way to sum GPS coordinates or calculate the centroid point of coordinate).

Therefore, with this approach, we only use classification tree, which we will introduce in next sections, to forecast the next service cell id which users will connect to.

3.2 Background

3.2.1 Preliminaries

Before going into details of decision trees, we would like to introduce some notations which will be used in the following sections.

The training data of this approach is often in text file format. In which, each line (or called records, observations... analogy) contains a set values of attributes.

The below are 2 lines of the dummy dataset with 3 attributes: Temperature, Humidity and Play golf.

12, High, No
27 , High, Yes

We also used word "feature" with the same meaning as "attribute". An attribute is called "Categorical" feature if it is unordered. For example, Gender, Car Models,... In contrast, an ordered attribute is called "Numerical" feature. For instance, Age, Salary,...

With a given data, we want to predict the value of a specific feature based on some of other features. That feature can be called: *the target feature*. Its dependency features are *predictors*. In the above example, the attribute "Play golf" can be the target feature. The others are predictors. Formally, let $\mathcal{X} = X_1,\ X_2,\ ..,\ X_n$ is the set of attributes with domain values $\mathcal{D}_{X_1}, \mathcal{D}_{X_2}, .., \mathcal{D}_{X_n}$ respectively. Let Y is an output with domain $\mathcal{D}_Y$. Denote $D* = \{(x_i, y_i) | x_i \in \mathcal{D}_{X_1} \times \mathcal{D}_{X_2} \times ..\mathcal{D}_{X_n}\}, y_i \in \mathcal{D}_Y$.

If $\mathcal{D}_Y$ is continuous, the learning problem is regression problem. If $\mathcal{D}_Y$ is categorical, it is classification problem.

Our goal is defining a function $\mathcal{F} : \mathcal{D}_{X_1} \times \mathcal{D}_{X_2} \times ..\mathcal{D}_{X_n} \rightarrow \mathcal{D}_Y$ that best approximates the true distribution of $D*$. The tree model represents $\mathcal{F}$ by partitioning the data space $\mathcal{D}_{X_1} \times \mathcal{D}_{X_2} \times ..\mathcal{D}_{X_n}$ recursively into non-overlapping regions. The boundary of each region is represented by a predicate. If X is categorical feature, the predicate is of the form: $X < v$, $v \in \mathcal{D}_X$. For example, "Age < 10". Unordered attributes have predicates of the form $X \in C = \{c_1, c_2, .., c_k\}$, $v_i \in \mathcal{D}_X$. For example, "Car Models $\in$ {MH1, MH5, CH4}". v and C are called ***split points***.

3.2.2 Impurity functions

The key idea of decision tree learning is : the data space will be divided into many rectangles by determining their boundaries,recursively, until they can not be split anymore.The decision will be made from these final rectangles. We will need a function to qualify the boundaries to find out the best one for dividing data. These function also called "impurity" functions.

Entropy, Information Gain, Gini-index... are popular functions which used in trees constructing.

3.2.2.1 Entropy

Entropy $H(\mathcal{S})$ is a measure of the amount of uncertainty or the impurity of the (data) set $\mathcal{S}$.

$H(\mathcal{S}) = -\sum_{x \in X} p(x) \log_2 p(x)$, x belongs to $\mathcal{X}$

Where,

$\mathcal{S}$: The current (data) set for which entropy is being calculated

$\mathcal{X}$: Set of classes in S

$p(x)$: The proportion of the number of elements in class x to the number of elements in set $\mathcal{S}$

So, the smaller impurity, the better quality of split.

3.2.2.2 Information Gain

Information gain $IG(A)$ is the measure of the reducing in entropy from before to after the set S is split on an feature A. In other words, how much uncertainty in S was reduced after splitting set S on feature A.

$IG(A, \mathcal{S}) = H(S) - \sum_{t \in T} p(t) H(t)$

Where,

$H(\mathcal{S})$ - Entropy of set $\mathcal{S}$

T - The subsets created from splitting set $\mathcal{S}$ by attribute A such that $S = \bigcup_{t \in T} t$

$p(t)$ - The proportion of the number of elements in t to the number of elements in set $\mathcal{S}$

$H(t)$ - Entropy of subset t

3.2.2.3 Gini-Index

Gini-Index (or Gini impurity) is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. Gini impurity can be computed by summing the probability of each item being chosen times the probability of a mistake in categorizing that item. It reaches its minimum (zero) when all cases in the node fall into a single target category.

$I_G(f) = \sum_{i=1}^{m} f_i(1 - f_i) = \sum_{i=1}^{m} (f_i - {f_i}^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} {f_i}^2 = 1 - \sum_{i=1}^{m} {f_i}^2$

Where, i takes on values in $1, 2, ..., m$

m is number of class of the target feature

f_i is the fraction of items labeled with value i in the set

3.2.2.4 Least Square Error

Another way to qualify a split is calculating its Least Square Error. This criterion is used for building regression tree by CART algorithm, which we will describe in section 3.2.3.

$D = \frac{1}{n}\sum_i^n (y_i - r(\beta, \mathbf{x}_i))^2$

Where,

n is the sample size

($\mathbf{x}_i, y_i$ is the data point)

$r(\beta, \mathbf{x}_i)$ is the prediction of regression model $r(\beta, \mathbf{x})$ for case ($\mathbf{x}_i$, y_i)

3.2.3 Decision Tree

A decision tree is a hierarchical structure of nodes and directed edges. There are three kind of nodes in a decision tree:

- **Root node**: The only one which has no incoming edges and zero or more outgoing edges. It is the entrance of the tree
- **Internal nodes (decision nodes)**: each of them has one incoming edge and two or more outgoing edges
- **Leaf nodes**: each of them has one incoming edge and no outgoing edges

The path from the root to a leaf node in the tree defines a region.

Assume that in the tree, we have a directed edge: Node1 $\rightarrow$ Node2. We say, Node1 is the parent of Node2; or Node2 is a child of Node1.

There are two ways to categorize decision trees: (i) using number children of non-leaf nodes and (ii) using the type of the target feature.

If in a tree, the non-leaf nodes contain exactly 2 outgoing edges, we call it ***"binary tree"***, otherwise, ***"multiway tree"***.

In a tree, if the target feature is *categorical*, we call ***"classification tree"***; if the target feature is *numerical* we call***"regression tree"***.

There are many well-known algorithms to build a tree.For build a "binary tree", the most popular algorithm is "CART" (Classification and Regression Tree) which was invented by Breiman from 1984 and has been using until now [9].

For building "multiway classification tree", "ID3" (Iterative Dichotomiser) is the essential algorithm, which was introduced by Quinlan (1986) [19] . In 1993, Quinlan published an improvement version of ID3, called "C4.5" which overcomes some disadvantages of ID3. C5.0 is a commercially successor of C4.5.

"CART" and "ID3" will be detail described in the next sections.

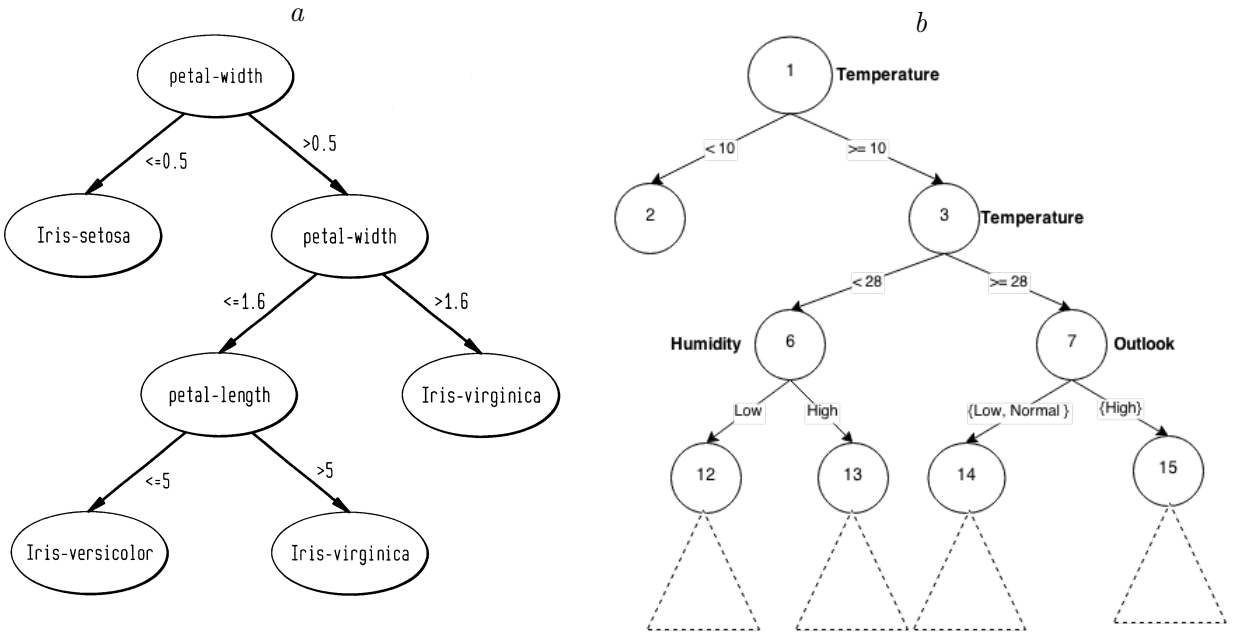


Figure 3.1: (a) Multi-ways tree
(b) Binary tree

3.2.3.1 CART

The CART methodology is known as binary recursive partitioning because each data space alway be split into **two** disjoin subspaces. At the initial state, the tree is null, which is corresponding to the full data space. If the current data space meets the stop criteria, we create a leaf node. Otherwise, we need to divide the data into two sub-disjoin spaces by finding a split point on one of features of data by considering the quality of it. For example, we want to predict a person will go playing golf or not by training model from data which has schema: *Temperature, Humidity, IsWindy, PlayGolf.* Assume that, the predicate is "$Temperatue \leq 25''$, we create a node associated with it. The records which satisfy this condition will be go into the left branch, otherwise, go to the right branch to form up two sub-spaces, and so on. If a categorical feature is chosen for splitting, the value of split point can be a subset of values of this feature in the current data space. For example: "$Humidity = \{High, Medium\}''$.

A leaf node can be created by taking a representative value of the target feature in the current data : the value have the most frequency (in case of Classification Tree), the average value (in case of Regression Tree).

The split point of a feature is chosen by a exhausted scanning through all possible split point candidates, calculate the quality of each split point and choose the one has maximum quality. The algorithm 2 expresses the whole tree learning process in the point of global view.

The output tree model from CART, as its name, can be a Regression Tree or Classification Tree based on the type of the target feature.

Because CART scans through all possible split points to find the optimal partitioning, it can be time

Algorithm 2 CART algorithm

```
function ExpandNode(data)
    if canExpand then
        (bestFeature, bestSpitpoint) ← findSplitPoint(data)
        D1 ← datapoints in data that satisfy the condition of bestSpitpoint
        D2 ← datapoints in data that don't satisfy the condition of bestSpitpoint
        root ← createNode(bestFeature, bestSplitpoint)
        root.left ← ExpandNode(D1)
        root.right ← ExpandNode(D2)
        Return root
    else
        Return LeafNode(data)
    end if
end function
function findSplitPoint(data)
    for each feature f_i in F do                    ▷ F : the set of features
        for each split point sp_j of feature f_i do
            qualityOfSplit(spj) ← qualityFunc(sp)
        end for
        bestSplitPoint(fi) ← argmax(qualityOfSplit)
    end for
    splitFeature ← argmax(bestSplitPoint)
    splitpoint ← bestSplitPoint(splitFeature)
end function
tree ← ExpandNode(fullData)
```

consuming. If a numerical feature has N distinct values, it has $N-1$ possible spit point. It's not too big. But, if a categorical feature has N distinct values, the number of split points is 2^N-1. The bigger N, the bigger problem. Fortunately, Breiman et. al. have proven that if the target feature has zero class (in case Regression Tree) or has only two class (in case Classification Tree), the number of split point of a predictor, which are need to be scanned is only $N-1$ [9].

Finding the best split points The function *findSplitPoint* in Algorithm 2 described the overview progress of finding the best split points of features in the training data. However, there is some differences between treating Categorical feature and Numerical feature. In this section, we will go into details of how to find the best split for each type of feature.

Finding the best split point of numerical feature In CART, the best split of Categorical feature is determined by evaluating all candidates via the *Least Square Error* function. However, with an continuous attribute, the number of split points is infinity, we can not evaluate all of them. These splits are known to be the major bottleneck in terms of computational efficiency of tree learning algorithms.

Fortunately, in [20], Torgo presented a fast incremental updating method to evaluate all possible splits of continuous attributes with significant computational gains in binary regression tree building. He defined that the best split **s*** is the split belonging to S that maximize the expression:

$Q = \frac{S_L^2}{n_L} + \frac{S_R^2}{n_R}$

where,

Q is the quality of the split

S is the set of possible split points

t_L, t_R are the left and right branch of the current node

$S_L = \sum_{D_{t_L}} y_i$, $S_R = \sum_{D_{t_R}} y_i$ are the sum of target feature's values in the left and right branch

n_L, n_R are the number of samples in t_L and t_R respectively.

Algorithm 3 is the pseudo-code of the implementation of the above formula.

With any pair of adjacent values (x_i, x_{i+1}) of continuous feature X in dataset D, all split point x_j , such that $x_i < x_j < x_{i+1}$, use the same way to divide D into two disjoin part with the same quality Q. Therefore,the possible split points set of numerical feature X is the ordered sequence of midpoints of every adjacent X's values. We try to calculate how many data points go into the left and right branch, and the sum of Y's values in each branch. The best split point is the one which maximize Q.

For example, suppose that we have the training data:

X, Y 10, 900
10, 1000
10, 1200
11, 300

Algorithm 3 Find best split point of a numerical feature

function FINDBESTSPLITPOINT_NUMERICALFEATURE($Data, TargetFeatureY, PredictorX$)
▷ Data : the training data of the current node
▷ Y : The feature which we want to predict its values
▷ X: the numerical feature, which we want to find the split point for
Sort the cases in $Data$ according to their value in X
$n_t \leftarrow$ number of lines in $Data$
$S_t \leftarrow$ sum of the target feature' values in $Data$
$S_R = S_t; S_L = 0$ ▷ S_R, S_T are the sum of the target feature' values of cases that are in the "left" and in the "right" the split point relatively
$n_R = n_t; n_L = 0$ ▷ n_R, n_T are the number of cases that are in the "left" and in the "right" the split point relatively
$BestTillNow = 0$ ▷ $BestTillNow$ is the best quality of split point until now
for all instance i in $Data$ **do**
$S_L = S_L + y_i; S_R = S_R - y_i$ ▷ y_i is the value of the target feature in instance i
$n_L = n_L + 1; n_R = n_R - 1$
if $x_{i+1} > x_i$ **then**
$NewSplitQuality = \frac{S_L^2}{n_L} + \frac{S_R^2}{n_R}$
if NewSplitQuality $>$ BestTillNow **then**
$BestTillNow = NewSplitQuality$
$BestCutPoint = \frac{x_{i+1}+x_i}{2}$
end if
end if
end for
Return $(BestCutPoint, BestTillNow)$
end function

12, 500
12, 800

For each X's value, we try to calculate its frequencies in the training data and the sum associated Y values after sorting data by X's values. X, SumY, Frequency
10, 3100, 3
11, 300, 1
12, 1300, 2

In this case, we have 2 possible split points: 10.5 and 11.5.

$Quality(10.5) = 3100^2/3 + 1600^2/3 = 40566666.66$, because if we split the data by value 10.5 on this predictor, there are 3 lines will go into the left node, with the sum of target feature is 3100; and 3 lines will go to the right node with the sum of Y is 1600.

and similarly, $Quality(11.5) = 3400^2/4 + 1300^2/2 = 37350000$

So, 10.5 is the best split point of this feature.

Finding the best split point of categorical feature Given a categorical attribute X with K distinct values. There are 2^{K-1} subsets which can be considered as split points. The bigger K, the more computation time to evaluate all split points. Fortunately, Breiman et. al. (1984) [9] proved an interesting result that changes the complexity of obtaining this type of split from $O(2^{K-1})$ into $O(K-1)$. Assuming that B is the set of values of X that occur in the current node. Defining $\widehat{y}(b_i)$ as the average Y value of the instances having value b_i in feature X, we sort the value such that:

$\widehat{y}(b_1) \leq \widehat{y}(b_2) \leq ... \leq \widehat{y}(b_K)$

Having the feature values sorted this way, Breiman and his colleagues have proven that the best split on categorical attribute X in node t is one of the $K-1$ splits:

$X_V \in \{b_1, b_2, .., b_{K-1}\}$

Algorithm 4 described the pseudo-code of this method.

The complexity of this algorithm is lower compared to the case of continuous variables. In effect, it is dominated by the number of values of the attribute. The exception is the part of sorting the values according to their average Y value. The sorting has complexity is $O(KlogK)$ but to obtain the average Y value associated to each value b we need to scan through all given instances with the complexity $O(N)$ which is most probability more complex than the sorting operation (N is the number of training samples).

For example, suppose that in a node t, we have the following instances:

Color, Y
green, 24
red, 56
green, 29
green, 13
blue, 120

Algorithm 4 Find best split point of a categorical feature

function FINDBESTSPLITPOINT_CATERICALFEATURE($Data, TargetFeatureY, PredictorX$)

▷ Data : the training data of the current node

▷ Y : The feature which we want to predict its values

▷ X: the categorical feature, which we want to find the split point for

Obtain the average Y value associated to each value of X

$D \leftarrow$ Sort the values of X according to the average Y associated to each value

$n_t \leftarrow$ number of lines in $Data$

$S_t \leftarrow$ sum of the target feature' values in $Data$

$S_R = S_t; S_L = 0$ ▷ S_R, S_T are the sum of the target feature' values of cases that are in the "left" and in the "right" the split point relatively

$n_R = n_t; n_L = 0$ ▷ n_R, n_T are the number of cases that are in the "left" and in the "right" the split point relatively

$BestTillNow = 0$

for each value b of obtained ordered set of values D **do**

$YB \leftarrow$ Sum of the Y values of the cases with $x = b$

$N_B \leftarrow$ Number of the cases with $x = b$

$S_L = S_L + YB; S_R = S_R - YB$

$n_L = n_L + NB; n_R = n_R - NB$

$NewSplitQuality = \frac{S_L^2}{N_L} + \frac{S_R^2}{N_R}$

if $NewSplitQuality > BestTillNow$ **then**

$BestTillNow = NewSplitQuality$

$BestPosition =$ Position of b in the set of ordered values D

end if

end for

$BestSplitPoint \leftarrow$ the set of X value in the set of ordered values D that have position $\leq BestPosition$

Return $(BestSplitPoint, BestTillNow)$

end function

red, 45
blue, 100

To calculate the average Y of each value in attribute "Color", we calculate the frequency of each Color's value and the sum associated Y values:

Color, Frequency, SumY
green, 3 , 66
red , 2, 101
blue, 2, 220

The average Y of each value in attribute "Color" is:

$\widehat{y}(green) = (24 + 29 + 13)/3 = 22$
$\widehat{y}(red) = (56 + 45)/2 = 50.5$
$\widehat{y}(blue) = (120 + 100)/2 = 110$

By sorting values according to their average Y values respectively, we obtain the sequence: <green, red, blue >

According to Breiman's theorem the best split would be one of the $K - 1$ (in this case $K - 1 = 2$) splits, namely the best split point $X_v \in$ { {green} , {green, red} }.

If X_v = {green}, 3 observations will go into the left branch, the rest belongs to the right branch. $Quality(\{green\}) = (24 + 29 + 13)^2/3 + (56 + 120 + 45 + 10)^2/4 = 14792.25$.

Similarly, $Quality(\{green, red\}) = (24 + 56 + 29 + 13 + 45)^2/5 + (120 + 100)^2/2 = 29777.8$

Because of having the biggest quality of splitting, {green, red} is the best split point.

This above procedures look like the word-count problem which we already introduced in section 1.2.1.1 : how many times a word appears in a document. The difference is, here, beside calculating the frequency of an predictor's value, we compute the sum of target feature also. The detail implementation of these algorithms will be discussed in the next sections.

From the best split points of all predictors in the current expanding node, we can find the best one, which has the maximum splitting quality. It will become the split point of this node, and helps dividing the data into two set. These two data sets are the training data for two child nodes.

3.2.3.2 ID3

To avoid the problem of the huge possible split points when having more than 2 classes in CART, Quilan introduced ID3 (Iterative Dichotomiser 3) which partitions data by a different way. Instead of finding best split point on a feature like CART, ID3 only find the predictor which can make a best "splitting" on the data. He assumes if a categorical feature X is chosen for splitting, it will divides data into N parts, where N is number of distinct values of X. That means we don't need to scan any split point of a categorical feature. The original ID3 doesn't support to deal with a numerical

feature. C4.5 overcomes this problem by treating numerical features in the similar way to CART.

The general pseudo code of ID3 algorithm with a customization to support numerical features is shown below.

The default function to calculate the quality of split in ID3 is Information Gain (section 3.2.2.2).

3.2.4 Pruning

The output model from tree learning procedure can face to *over-fitting* issue: The tree is too fit to the training data, but isn't accuracy enough in the other testing sets. Or formally: Given a hypothesis space H, a hypothesis $h \in H$ is said to over-fit the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

Pruning is a technique to reduce the size of decision tree, aim to reduce the complexity of the final classifier as well as increase predictive accuracy by solving problem of over-fitting and noisy data. There are two kinds of pruning techniques:

- Pre-pruning (or Early Stopping Rules) : stops the algorithm before it becomes a full-grown tree. For example: stop algorithm if the number of instances is less than a threshold or if the class of the target feature are independent of the available features....
- Post-pruning: Grow the tree to its entirety and then trim the nodes in a bottom-up fashion. If the generalization errors are improve after trimming, replace the sub-tree by a leaf node.

Because of pruning from the full tree, Post-pruning technique is more powerful than Pre-pruning, but the running time from the beginning to get the final tree is more longer. Sometimes, we have to make a weak split to be able to follow up with a good one. As a consequence, post pruning is used more widely. In the next section, we only introduce about Post-pruning, particularly, Weakest Link Cutting, a very popular approach.

3.2.4.1 Post Pruning with Weakest Link Cutting

Weakest Link Cutting or Minimal Cost-Complexity Pruning, has been proposed by Breiman et al. in 1984, helps us find a optimal sub-tree from the full tree.

Denote, t is a node. T_t is a branch with the root node is t.

$R(t)$ is the risk when predicting if we prune branch T_t (t becomes a leaf node)

$R(T_t)$ is the risk when predicting before pruning if we don't prune branch T_t (use its leave to make predictions)

In case of classification tree, the risk can be the misclassification rate.

$\tilde{\mathrm{T}}$ is a set of leaves of T_t

$R(T_t) = \sum_{t \in \tilde{\mathrm{T}}} R(t)$

Algorithm 5 ID3 algorithm

function BUILDTREEID3($Data, Target_Feature, Predictors$)
  Create a root node for the tree
  if all data belong to the same class or all predictor's values is the same **then**
    Return the single node with the most common target value
  end if
  if number of predictors is zero **then**
    Return the single node with the most common target value
  else
    $(bestFeature, bestSplitpoint) \leftarrow getBestSplitPoint(Data)$
    $Root \leftarrow$ create node which contains the predictor is bestFeature
    if $bestFeature$ is categorical **then**
      for each possible value v_i of $bestFeature$ **do**
        Add a new tree branch below Root, corresponding to the test $bestFeature = v_i$.
        $Data(v_i) \leftarrow$ the subset of data that have the value $v_i$for A
        if $Data(v_i)$ is empty **then**
          add a leaf node below Root with label = most common target value in the data
        else
          below this new branch add the subtree BuildTreeID3 ($Data(v_i)$, Target_Feature, Predictors - {bestFeature})
        end if
      end for
    else if $bestFeature$ is numerical **then**
      $D_1 \leftarrow$ the subset of data that satisfy the condition of $bestSplitpoint$
      $D_2 \leftarrow$ the subset of data that don't satisfy the condition of $thebestSplitpoint$
    end if
    Return $Root$
  end if
end function

function GETBESTSPLITPOINT($Data$)
  for each predictor f_i in $Data$ **do**
    if f_i is categorical **then**
      $Q(f_i) \leftarrow$ Quality of the full split on $Data$ by choosing f_i
      $SP(f_i) \leftarrow null$ ▷ sign of a full split
    else if f_i is numerical **then**
      for each split point sp_j of feature f_i **do**
        $qualityOfSplit(sp_j) \leftarrow$ Quality of binary split f_i on $Data$
      end for
      $Q(f_i) \leftarrow max(qualityOfSplit)$
      $SP(f_i) \leftarrow argmax(qualityOfSplit)$
    end if
  end for
  $bestFeature \leftarrow argmax(Q)$
  $bestSplitpoint \leftarrow SP(bestFeature)$
  Return $(bestFeature, bestSplitpoint)$
end function

$R(T_t)$ is biased downward. Specifically, the weighted misclassification rate for the parent node is guaranteed to be greater or equal to the sum of the weighted misclassification rates of its children:

$R(t) \geq \sum_{t_i} R(t_i)$, t_i is children of t

This means that if we simply minimize R, we always prefer a bigger tree, that can not solve over-fitting problem. We need to add a complexity penalty to this misclassification error rate. The penalty term favors smaller trees. Therefore, Breiman introduced *"complexity parameter"* $\alpha > 0$ and defined the cost-complexity measure $C_\alpha(T_t) = R(T_t) + \alpha \left\|\tilde{\mathrm{T}}\right\|$

It means that the more leaf nodes, the higher complexity. The cost complexity measure is considered as a penalized version of misclassification error rate. This is the function need to be minimized when pruning the tree. With $\alpha = 0$, we prefer the biggest tree. With α approaches infinitive, the tree of a single node will be selected. Generally, given a pre-selected α, we can find a tree $T(\alpha)$ that minimize $C_\alpha(T)$. This minimizing subtree for any value of α always exists because of the finiteness of number sub-trees.

As long as $C_\alpha(T_t) < C_\alpha(t)$, the branch T_t contribute the less complexity cost to tree than node t. The inequation is satisfied with small α. When α increase to a certain value α^*, the equality of two cost-complexity is achieved. At this point, the branch T_t can be replaced by t because it's no longer help improve the classification :

$C_{\alpha^*}(T_t) = C_{\alpha^*}(t)$

$\Leftrightarrow R(T_t) + \alpha^* \left\|\tilde{\mathrm{T}}\right\| = R(t) + \alpha^*$

$\Leftrightarrow \alpha^* = \frac{R(t) - R(T_t)}{\left\|\tilde{\mathrm{T}}\right\| - 1}$

Generate subtrees

Let $\mathcal{T}_1$ is the full tree. Let $g1(t) = \frac{R(t) - R(T_t)}{\left\|\tilde{\mathrm{T}}\right\| - 1}$ if t is non-leaf of $\mathcal{T}_1$, infinitive if otherwise

$g_1(t)$ is call the strength of the link from node t. In Minimal Cost-Complexity Pruning, the nodes that have the weakest link will be trimmed first. Let $\alpha_1 = 0$, $\alpha_2 = g(t_1^*) = min(g_1(t))$, t belongs to leaf nodes of $\mathcal{T}_1$. To get the optimal subtree corresponding to α_2, simply remove then branch of t_1^*. When α increases, t_1^* is the first node that becomes more preferable than branch $T_{t_1}^*$.

Let $\mathcal{T}_2 = \mathcal{T}_1 - T_{t_1}^*$ Do the same thing with $\mathcal{T}_2$ to obtain $\mathcal{T}_3$, and so on. Till the end, we have a list : $[(\alpha_1, \mathcal{T}_1), (\alpha_2, \mathcal{T}_2), ..., (\alpha_K, \mathcal{T}_k)]$

The theorem states that the α_k are an increasing sequence, that is, $\alpha_k < \alpha_{k+1}$, $k \geq 1$,where $\alpha_1 = 0$. The pruning procedure gives sequence of nested subtrees: $\mathcal{T}_1 > \mathcal{T}_2 > \mathcal{T}_3 > ... > \mathcal{T}_k = rootnode$.

For any $k \geq 1$, $\alpha_k \leq \alpha < \alpha_{k+1}$, the smallest optimal subtree $\mathcal{T}(\alpha) = \mathcal{T}(\alpha_k) = \mathcal{T}_k$, i.e., is the same as the smallest optimal subtree for α_k. Basically, this means that smallest optimal subtree $\mathcal{T}_k$ stays optimal for all the α's starting from α_k until it reaches α_{k+1} or we can say $[\alpha_k, \alpha_{k+1}) \rightarrow \mathcal{T}_k$. Although we have a sequence of finite subtrees, they are optimal for a continuum of α .

We have a sequence of subtree but we don't know what tree is the best, or by other word, we don't know what value of α will give us the best tree. To choose value of α, we do Cross-Validation.

Cross- Validation

The idea is that for each interval of α: $I_k = [\alpha_k, \alpha_{k+1})$, we try to estimate the error if we use $\alpha_1 \leq \alpha \leq \alpha_2$ and then select value of α minimize the error.

Split full training data into N-parts: $D = D_1 \cup D_2 \cup D_3 \cup ... \cup D_N$, where N is number of fold times.

In fold i, using data set $L = D$
D_i as a training set to build the full tree Tree^{i}. Do the same thing in Generate subtrees to get the list of $(\alpha, Tree^{\text{i}}{}_alpha)$ For each interval $I_k = [\alpha_k, \alpha_{k+1})$, we select a representative value $\beta_k = \sqrt{\alpha_k \alpha_{k+1}}$, and find $Tree^{\text{i}}(\beta_k)$ and then calculate the misclassification error rate of it with the testing set is D_i.

In the next fold $i + 1$, we continue to find out list of $Tree^{\text{i}+1}(\beta_k)$ and its error rate.

After N-folds, we calculate the average error of each $Tree^{\text{i}}(\beta_k)$ and select the tree has minimum error. The value β^* which associated to this tree will be the best value for α. The best prunned tree is $T(\beta^*)$.

But this tree is still a local optimal tree. The solution is using 1-SE rule to select the best $Tree^{\text{i}}(\beta_k)$ instead of select tree has minimum value of average error. Another improvement can be done is, we use the smaller version of the full tree, which have the equivalent error rate. Let's call the full tree from the beginning is $\mathcal{T}_0$. By bottom-up fashion, we prune all branch T_t which $R(T_t) = T(t)$ to get $\mathcal{T}_1$.

Cross-Validation is a very powerful solution to select value of α.

3.2.5 Random Forest

Random Forest[10] is a technique helps increasing predictive accuracy significantly by combine many different trees(without pruning) and select the prediction which were voted by the most of trees. The idea of Random Forest is simple, but it's really powerful. We have two ways to generate a "random tree" : randomize the training data, and randomize the expanding nodes.

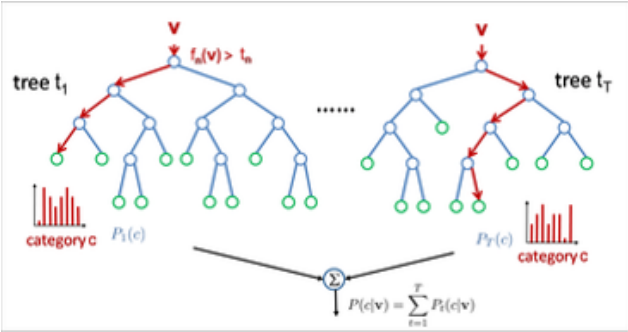


Figure 3.2: Random Forest

Let's K is the number of tree in the forest which we want to build, N is the number of lines in training set D. To build each tree, **select randomly N lines of D with replacement** to form up a new training set $D1$. Literature has proven that 2/3 instances of D will be selected with some duplications. $D2 = D \backslash D1$ is called *"out-of-bag"* data, which will be used for testing and other purposes. On the other hand, to expand a node during building tree, instead of using all of predictors

in $D1$, we **select randomly p predictors** to consider for splitting ($p << M$, M is the number of predictors in the data of each expanding node time). p is often equal $\sqrt{M}$ or $\frac{\sqrt{M}}{2}$.

After building the forest, when we need to predict a target value of unseen records, just pass it into all trees and select the predicted values which the most of tree agree on.

Random Forest also be used for calculating the variable important factor and many other functions. But for this project, we only implement a basic level of random forest, which provide enough functionality for make decision.

3.3 Scalable Algorithm Design

3.3.1 Challenges

The main challenge of this approach is how to build tree models from over large datasets. Although we can OpenPlanet can build Regression Tree in Hadoop, it still has some disadvantages. PLANET and OpenPlanet makes a trade-off between finding the perfect split for an ordered attribute and simple data partitioning [12, 23]. Before each iteration, OpenPlanet reads the whole training data for MapReduce jobs with MapReduce. The result of jobs will be written into HDFS after each iteration. That can lead to I/O overhead. Besides, OpenPlanet doesn't support pruning and Random Forest. Therefore, we address this challenges by designing new algorithms for building tree model as well as for pruning and Random Forest.

The second challenge is determining the predictors: what features should be predictors ? We have two kind of predictors: directly and indirectly. The directly features are the attributes that exist already in the training data. In contrast, we have to aggregate features, combine with other data to infer the indirectly features. In this project, due to data availability and feature lacking, we only use directly features. Besides, among many features, to select the best predictors, function "Feature Selection" of Random Forest is often used in literature. But within the limited time of this project, we didn't develop this function on scalable Random Forest.

In the next section, we focus to address the challenge of algorithm scalability. In spite of only using classification tree, we also design the scalable algorithm for building regression tree for future purposes.

3.3.2 Solutions

3.3.2.1 Labelling CART : The parallel tree building algorithm for CART

Inspired from Google Planet, we developed Regression Tree algorithm in SPARK with a scalable version of CART, called Labelling CART. SPARK is a Hadoop-like, but it supports iterative algorithm very well. The data will be stored in Resilient Distributed Dataset (RDD) in both memory and disk. That helps to reduce I/O operations.With this algorithm, we use label to manage all expanding node jobs at the same tree level in once, instead of using queue like Planet.

In this section, we will describe how to construct the tree. As the meaning of the algorithm's name,

"label" is an important factor to build the tree.

Assume that the input data D has N lines (or records/observations). We mark each record with a "label", which determines the node that this record will be used for expanding. When a node is constructed, the "label"s are updated for the next building phase of it's child nodes (if this node can be expanded more).

3.3.2.1.1 Walk through Let's take a walk-through to understand the idea of this algorithm. At the initial state, the tree model is empty and we want to construct the root node, which associated to ID **1** (or label = 1, analogy). Every records in the dataset will be marked label **1** for meaning : these records are used for building node **1**. Assume that , in the first iteration, after evaluating the quality of possible split points, the best split is "Temperature < 10". We create and add the root node with this split information into the tree model. The records in data which satisfy the predicate "Temperature < 10" are marked label **2**, then be used for constructing node 2 in the next iteration. The rest are marked label **3** and be used for expanding node 3. In this example, the best split has 10 records ($D1$) in the left branch and 140 records ($D2$) in the right branch. The controller then start the new iteration to expand node 2 and 3 once. Node 2 stop expanding because it matches the stop criterion (assume that 25 records is the smallest data for splitting).Therefore, it will be added to the tree as *a leaf*. In opposition, Node 3 is expandable, then finds out the best split "Temperature < 28" and be added to the tree as *a internal node*. The data in $D1$ which satisfy this predicate are marked label **6** (=3*2). The remaining in $D1$ are marked as label **7** (=3*2 + 1). Node 6 and 7 are constructed in the third iteration with the same procedure, and so on. Our algorithm expands trees breadth first. The algorithm will stopped if there is no expandable nodes.

The more details of each iteration are described below.

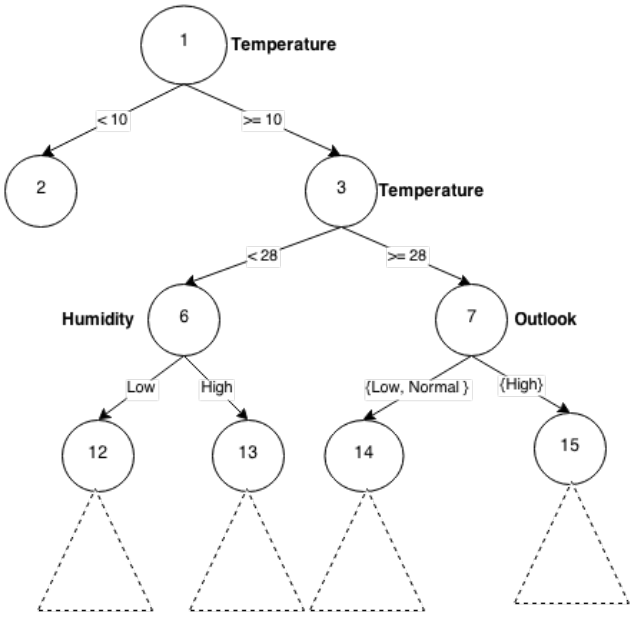


Figure 3.3: Labelling CART's Tree model

3.3.2.1.2 Technical Details As mentioned 3.2.3, each iteration of CART has 4 steps/tasks:

- For each value of each feature, calculate the aggregate information of the associated values of target feature
- Check the expandability of each node depend on its training data. Create leaf nodes for which can not be expanded anymore
- For each feature in each node, find the best split point and calculate the quality of this split
- For each node, among the best split points of features, select one which has the maximum value of quality and construct a internal node. Then update the label for the next iteration

In the scalable environment, each tasks will be done as a map-reduce jobs. We use a controller to manage the iterations, update model and check the convergent point of the whole algorithm (Algorithm 6). The controller maintains the followings:

- **ExpandableNodes**: The list of nodes which will be expanded
- **Model** (M) : The current tree model

After each iteration, the Model M will be written into disk once. The algorithm will be stop if the list ExpandableNodes is empty.

Denote the feature i^{th} is F_i (the target feature included)

X_i : the input feature or predictor i

Y : the target feature.

$\mathcal{X}$: the set of predictors

k : the number of features

label : the label current node that we want to expand, is an integer number and equal to 1 if the current expanding node is the root.

D : the original training data

Each iteration of this algorithm has 5 steps, which is represented as map-reduce jobs: *Counting Frequencies* , *Computing the statistical information* of the target feature in each node, *Finding features's best split point*, *Finding nodes's best split point* and *Updating labels*. Exceptionally, the additional step "Preparation" is only executed before the first iteration.

Figure 3.4 describes the main steps of each iteration:

- (1) **Preparation**: initializing the first label for every sample of the training data
- (2) **MR_CalculateFrequencies**: calculating the frequency of each value of each feature in each node

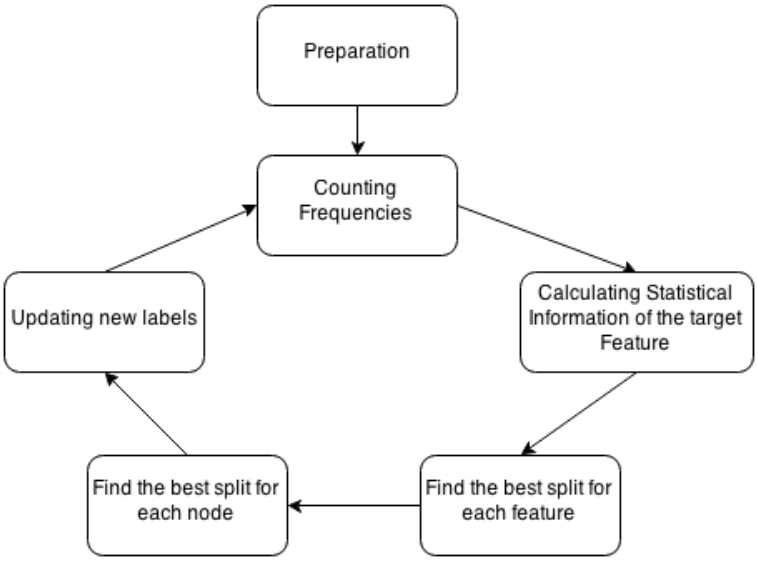


Figure 3.4: Labelling Tree Building

- (3) **MR_TargetFeatureInfoAgg**: calculating the aggregated information of the target feature in each node (for checking the expandability)
- (4) **MR_FindBestSplit_EachFeature**: Finding the best split point of each feature in each node
- (5) **MR_FindBestSplit_EachNode**: Finding the best split point of each node
- (6) **Updating labels**: Updating the new labels for the training data for preparing expanding nodes in the next level.

Steps 2,3,4,5,6 will be looped until there is no node to expand anymore (Algorithm 6)

The detail implementation of each phase is described below.

In the *Preparation*, after reading data D from disk, we mark the first label for records in this data. The first label is the root node's ID (default is 1). We call the new data is D_1.

The map-reduce job *MR_ CalculateFrequencies* will take input which has schema: $(label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))$, where v_{F_i} is the value of feature F_i in this line, y is the target feature's value.

In the map phase of *MR_ CalculateFrequencies*, for each valid line, we separate values of features to k values, and then emit k tuples, each tuple has form: $((label, F_i, v_{F_i}), (y, y^2, 1))$. A line of data is valid if the associated label are positive, otherwise, that line is not used in tree learning anymore. The information from $label, F_i$ helps us remember this value is belong to what feature and be used to build what node. These information also aim to expand nodes in parallel.

In the reduce phase of *MR_ CalculateFrequencies*, we aggregate information by calculating the partial sum of the values.

Algorithm 6 Controller - the main loop

Require: D : the training data
  function BUILDTREE
    D1 ← Preparation(D)
    while true do
      Frequencies ← MR_CalculateFrequencies(D1)
      StatisticalInfoY ← MR_TargetFeatureInfoAgg(Frequencies)
      checkExpansion(StatisticalInfoY)
      Update model: Create leaf nodes for which are inexpandable
      BestSplitOfEachFeature ← MR_FindBestSplit_EachFeature(Frequencies)
      BestSplitOfEachNode ← MR_FindBestSplit_EachNode(BestSplitOfEachFeature)
      CreateInternalNodes(BestSplitOfEachNode)
      D1 ← UpdateLabels(D1)
      if *ExpandableNodes* is empty **then**
        exit
      end if
    end while
  end function

Algorithm 7 Preparation

Require: Feature set $\mathcal{F} = \{F_1, F_2, ..., F_k\}$
Require: $D = RDD[(v_{F_1}, v_{F_2}, ..., v_{F_k-1}, y)]$
  $FirstLabel = 1$ ▷ 1 is the ID of the root node
  for each element e in D **do**
    EMIT $(FirstLabel, e)$
  end for

Algorithm 8 MR_CalculateFrequencies

Require: $D1 = RDD[(label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))]$
function MAP(D_1)
  for each element $e = (label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))$ in $D1$ **do**
    if $label > 0$ **then**
      for $i = \{1..k\}$ **do**
        EMIT $((label, F_i, v_{F_i}), (y, y^2, 1))$
      end for
    end if
  end for
end function

function REDUCE(<Key=(label, F, v), Value Set V = $\{(y_i, y_i^2, 1)\}$>)
  EMIT <Key, $(\sum y_i, \sum y_i^2, \sum 1)$ >
end function

The map-reduce job *MR_ TargetFeatureInfoAgg* filters out the statistical information of the target feature only in each node. The statistical information is form of ($label$, $\sum y$, $sumy^2$, $frequency$), where $frequency$ is the number records in the training data of node which indicated by $label$, $\sum y$ and $\sum y^2$ are the sum and the sum square of target feature's values. Based on these information, the controller calculates the deviance of the target feature's values in each node to decide which nodes should be stop to create a leaf node, which nodes can be expand to two children (called expandable). A node will be stopped if it's data is less than $threshold_1$ lines or the deviance of Y in the data is smaller than $threshold_2$. These parameter were configured before starting the algorithm. In this step, the map phase will filter D_2 - the output of *MR_ CalculateFrequencies* - to get only the tuples which contains information of the target feature ($F_i == Y$). The reduce phase will calculate the deviance of Y in each node.

Algorithm 9 MR_TargetFeatureInfoAgg

function MAP($D_2 = RDD[((label, F, v), (\sum Y, \sum Y^2, frequency))]$)
  for each element e in D_2 **do**
    $((label, F, v), (sumY, sumY2, fre)) = e$
    if F == Target Feature **then**
      EMIT (label, (sumY, sumY2, fre)
    end if
  end for
end function

function REDUCE(<Key K=label, Value Set $V = \{(sumY_i, sumY2_i, fre_i)\}$>)
  EMIT <Key, $(\sum sumY_i, \sum sumY2_i, \sum fre_i)$ >
end function

Algorithm 10 Controller - Check the expandability of nodes

function CHECKEXPANSION($arr = Array[(label, (\sum Y, \sum Y^2, frequency))]$)
  StopNodes $\leftarrow \emptyset$
  ExpandableNodes $\leftarrow \emptyset$
  for each element $e = (label, sumY, sumY2, frequency)$ in arr **do**
    Deviance(Y) = DevCalculate(sumY, sumY2, frequency)
    if $frequency < threshold_1$ or $Deviance(Y) < threshold_2$ **then**
      Create a leaf node with id=label and value = Average(Y)
    else
      ExpandableNodes = ExpandableNodes + label
    end if
  end for
  Return ExpandableNodes
end function

After that, these information will be send back the controller for checking the expandability of each node (Algorithm 10). The nodes which are expandable will be appended to the list $ExpandableNodes$. The others are represented as new leaf nodes in the tree model.

In step 3, we try to find the best split point of each feature in each node by executing *MR_ FindBestSplit_ EachFeatur*

Algorithm 11 MR_FindBestSplit_EachFeature

function MAP($D_2 = RDD[((label, F, v), (\sum Y, \sum Y^2, frequency))], ExpandableNodes$)
  for each element e in D_2 **do**
    $((label, F, v), (sumY, sumY2, fre)) = e$
    if F $\neq$ Target Feature and label $\in$ ExpandableNodes **then**
      EMIT ((label, F), (v, sumY, sumY2, fre)
    end if
  end for
end function

function REDUCE(<Key K=(label,F), Value Set $V = \{(sumY_i, sumY2_i, fre_i)\}$>)
  if F is Categorical Feature **then**
    EMIT $(label, (F, FindBestSplitpoint\_CatericalFeature(V)))$
  else
    EMIT $(label, (F, FindBestSplitpoint\_NumericalFeature(V)))$
  end if
    ▷ In both cases, the emitting tuple has schema: ((label, (F,bestSplitPoint,qualityOfSplit))
end function

MapReduce job *MR_ FindBestSplit_ EachNode* will find the split point which maximizes the quality of split of each node and then send back these information to driver to update the model by creating the internal nodes.

Besides, one of the most important tasks in this step is updating the label for the next iteration (for construction the child nodes). The part of data which belongs to leaf nodes will be marked a negative label. The remaining are update to the new label. Let consider an expandable node which has ID=$label_k$, the best split point is sp on the feature f. For each element e of the data D_1 (which is the input of *MR_ CalculateFrequencies*), if v_f is less than sp (in case f is numerical) or if $v_f \in sp$ (in case f is categorical), we mark the new label for e is $label * 2$, otherwise, $label * 2 + 1$, where $label$ is the current label of e (Algorithm 14).

The next iteration will take the new data, and do the same process on it.

By this algorithm, all nodes in the same level will be expanded in one iteration. Note that "Preparation" and "Updating labels" are the map only jobs because they only transform the data but not aggregate them. In SPARK, a series of map-only job or map functions will be aggregated and done as a simple map phase. It means that step 1 (*Preparation*) and step 6 (Updating labels) are parts of *MR_ CalculateFrequencies*'s map phase. Therefore, the algorithm has only 4 Map-Reduce jobs indeed.

3.3.2.2 Parallel algorithm for ID3

Different from CART, each node in ID3 tree model can have more than 2 nodes. If we expand a node by choosing a categorical feature X, it will have n children, where n is the number of distinct values of X. For example, if we use "DayOfWeek" as the splitting feature and it has 7 different

Algorithm 12 MR_FindBestSplit_EachNode

function MAP($D_3 = RDD[(label, (F, bestSplitPoint, qualityOfSplit))]$)
  for each element e in D_3 **do**
    EMIT e
  end for
end function

function REDUCE(<Key K=label, Value Set $V = \{(F_i, splitpoint_i, quality_i)\}$ >)
  MaxQuality $\leftarrow -\infty$
  Best $\leftarrow$ UNKNOWN
  for each $v_i = (F_i, splitpoint_i, quality_i) \in V$ **do**
    if (**then**MaxQuality $< quality_i$)
      MaxQuality = $quality_i$
      Best = v_i
    end if
  end for
  EMIT (label, Best)
end function

Algorithm 13 Controller - Create internal nodes

function CREATEINTERNALNODES(arr = Array [(label, (F, bestSplipoint, bestQuality))])
  for each element $e = (label, (F, bestSplipoint, bestQuality))$ in arr **do**
    Create an internal node with (label, F, bestSplitpoint)
  end for
end function

Algorithm 14 Updating labels

function UPDATELABELS(arr = Array [(label, (F, bestSplipoint, bestQuality)), D_1])
▷ D_1 is the input of $MR\_CalculateFrequencies$ – the labeled data
for each element $e = (currentlabel, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))$ in D_1 **do**
$(label, (F, bestSplipoint, bestQuality) \leftarrow$ element in arr which has label = $currentLabel$
if $currentLabel \in ExpandableNodes$ **then** ▷ If this line of data is still used in the next iteration
if F is CategoricalFeature **then**
if $v_F \in bestSplitpoint$ **then** ▷ $bestSplitpoint$ is a set of values
Update label of e to $currentLabel * 2$
else
Update label of e to $currentLabel * 2 + 1$
end if
else
if $v_F < bestSplitpoint$ **then** ▷ $bestSplitpoint$ is a number
Update label of e to $currentLabel * 2$
else
Update label of e to $currentLabel * 2 + 1$
end if
end if
else
Update label of e to -9
end if
end for
Return D_1
end function

values: {Sun, Mon, Tue, Wed, Thu, Fri, Sat} then we will expand the current node into 7 child nodes. That means, for unordered feature, it's not necessary to evaluate all possible split points like CART. We only need to qualify each splitting feature candidate. In this section, we introduce the parallel algorithm for building ID3 tree model in scalable environment, called Lablelling ID3.

Besides, our algorithm overcomes a shortage of the original ID3: numerical feature support.

We use *Labelling technique*, which is described in section 3.3.2.1 for constructing all nodes in the same level in parallel like Labelling CART. However, because of some differences in tree model, the controller also has some changes.

In CART Tree model, because every internal node has two and only two children, we can manage the ID of nodes, and use these ID for labelling the data easily. Nodes ID management in ID3 is more complex : the ID of a child node is given by the controller via *NextNode*, instead of referring from its parent.

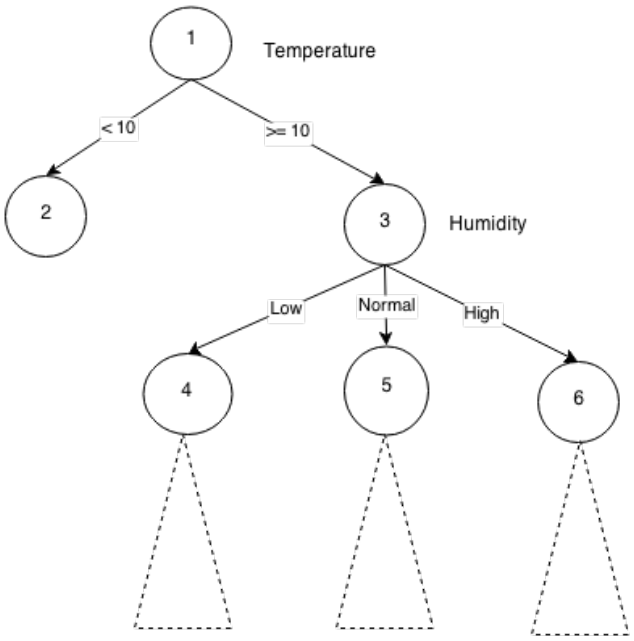


Figure 3.5: Labelling ID3's Tree model

3.3.2.2.1 Walkthrough When tree induction begins, the tree model has only root node (ID=1) without any information. In addition, every record in the training data D is marked label **1** for expressing that they are used for constructing node 1. In the first iteration, assume that after evaluating all possible split points, the best split is "Temperature < 10". We create the root node associated to these splitting information. This split point leads to two empty children of the root which are assigned ID 2 and 3 respectively. In dataset D, we suppose that there are 10 records ($D1$) that satisfy the predicate "Temperature < 10". These records are marked label 2 and belong to the constructing of node 2. Similarly, the rest, 190 records ($D2$), are marked label 3. In the second iteration, node 2 and 3 are expanded in parallel. With the given minimum record threshold for expanding (minRecords ≥ 25), Node 2 matches the stop criteria. Thus, the controller updates

the tree by treating Node 2 as a leaf with the decision value is the most frequent value of the target feature in $D1$. when expanding node 3, we see that the splitting on the categorical feature "Humidity" brings the best quality. Therefore, the controller update these information for node 3 in the tree model and then it create 3 empty children for him, corresponding to 3 distinct values of attribute "Humidity" in $D2$ (figure 3.5). The records in $D2$ which match the conditions "Humidity=Low", "Humidity=Normal" , "Humidity=High" are marked the new labels 4, 5 and 6 respectively to prepare for next iteration....

The algorithm is terminated when there is no empty node in tree model.

The IDs of nodes are an increment sequence, ordered by their creation times.

Thanks to Lablelling technique, we can expand all nodes in the the same level in a single step without maintaining any queue. We use empty nodes in the tree models to manage to expanding nodes instead.

3.3.2.2.2 Technical Details Similar to Labelling CART, Labelling ID3 has a controller to manage iterations and the tree models by maintaining and updating the current tree model through the loop of iterations.

In this algorithm, we use the same terms as before in section 3.3.2.1.

Each iteration of Labelling ID3 has some main steps:

- (1) **Preparation**: initializing the first label for every sample of the training data
- (2) **MR_CalculateFrequencies**: calculating the frequency of each value of each feature in each node
- (3) **MR_GroupStatYByXValue**: group the statistical information of the target feature by each associated predictor's value
- (4) **MR_FindBestSplit_EachFeature**: Finding the best split point of each feature in each node
- (5) **MR_FindBestSplit_EachNode**: Finding the best split point of each node
- (6) **Updating labels**: Updating the new labels for the training data for preparing expanding nodes in the next level.

Steps 2,3,4,5,6 will be looped until there is no empty node in the tree model (no node to expand anymore) (Algorithm 15)

In the *Preparation*, after reading data D from disk, we mark the first label for records as same as in the Labelling algorithm. The first label is the root node's ID (default is 1). We call the new data is D_1. Besides, we create a empty root node in the tree model.

The map-reduce job *MR_CalculateFrequencies* (algorithm 16) takes input which has schema: $(label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))$, where v_{F_i} is the value of feature F_i in this line, y is the target feature's value.

Algorithm 15 ID3 - Controller - the main loop

Require: D : the training data
  function BUILDTREE
    $D_1 \leftarrow$ Preparation(D)
    Create an empty root node in tree model M
    while true do
      frequencies $\leftarrow$ MR_CalculateFrequencies(D_1)
      groupedStatY $\leftarrow$ MR_GroupStatYByXValue(Frequencies)
      bestSplitOfEachFeature $\leftarrow$ MR_FindBestSplit_EachFeature(groupedStatY)
      bestSplitOfEachNode $\leftarrow$ MR_FindBestSplit_EachNode(groupedStatY)
      UpdateModel(bestSplitOfEachNode)
      $D_1 \leftarrow$ UpdateLabels(D_1, M)
      numEmtpy $\leftarrow$ the number of empty node in M.
      if numEmpty = 0 **then**
        exit
      end if
    end while
  end function
  function UPDATEMODEL(statisticalInfo = $\{label_i$, $splitFeature_i$, $splitpoint_i$, $gain_i$, $predictedValue_i\}$)
    for each element e in $statisticalInfo$ **do**
      ($label$, $splitFeature$, $splitpoint$, $gain$, $predictedValue$) = e
      if $splitpoint$ = NULL **then**
        update node which has ID=$label$ to be a leaf node
      else
        update node which has ID=$label$ to be an internal node
        if $splitFeature$ is categorical **then**
          create N empty children for node which has ID=$label$ ▷ N is the number of value of $splitFeature$ in this node's training data
        else
          create 2 empty children (left,right) for node has ID=$label$
        end if
      end if
    end for
  end function

In the map phase of *MR_ CalculateFrequencies*, for each valid line, we separate values of features to k values, and then emit k tuples, each tuple has form: $((label, F_i, v_{F_i}, y), 1)$. A line of data is valid if the associated label are positive, otherwise, that line is not used in tree learning anymore.

In the reduce phase of *MR_ CalculateFrequencies*, we simply compute the partial sum of the values to get the frequency of each feature's value.

Algorithm 16 MR_CalculateFrequencies

Require: $D_1 = <(label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))>$
1: **function** MAP(D_1)
2: **for** each element $e = (label, (v_{F_1}, v_{F_2}, ..., v_{F_{k-1}}, y))$ in $D1$ **do**
3: **if** $label > 0$ **then**
4: **for** $i = \{1..k\}$ **do**
5: EMIT <(label, F_i, v_{F_i}, y), 1) >
6: **end for**
7: **end if**
8: **end for**
9: **end function**
10:
11: **function** REDUCE(<Key=(label, F_i, v_{F_i}, y_i), Value Set V = {1}>)
12: EMIT <Key, $\sum 1$ >
13: **end function**

In step 2, for each value of each feature, we calculate how many time each target feature's value appears (Algorithm 17). For example, assume that the input of this step is like below:

//(Label, Feature, XValue, YValue) , Frequency
(1,F1,20, yes), 1
(1,F1,22, yes), 2
(1,F1,20, no), 5

The output of this step will be:
(1, F1, 20), (yes, 1), (no , 5)
(1, F1, 22), (yes, 2)

The output of step 2 is also the input of step 3 (*MR_ FindBestSplit_ EachFeature*). The map phase of this job sent the statistical information of each feature in step 2 to a reducer. Each reducer receives something like (we use the json-like format for easier reading):

// (label, feature), statistical information
(1, F1), 20 : (yes, 1), (no , 5) , 22: (yes, 2)

Depends on the type of the feature, the reducer determines the way to find the best split point. A categorical feature, which expands on all its values, will have the split point is these values. We only need to calculate the quality of split (called "gain"). In contrast, with a numerical feature, we have to scan for all possible splipoints by using the method of Breiman in CART, and calculate the

Algorithm 17 MR_GroupStatYByXValue

function MAP(D_2= {<($label_i$, F_i, v_i, y_i), $frequency_i$ >})
  for each element e in D_2 **do**
    ((label, F, v, y), fre) = e
    EMIT <(label, F, v), (y, fre)>
  end for
end function

function REDUCE(S = <Key=(label, F, v), Value{(y_i, fre_i)} >)
  EMIT <Key, Value >
end function

"gain" of each split. The best split point is the one maximize its gain (algorithm 20). Besides, the statistical information of each feature also can be used to check the stop criteria of the associated node. A node is inexpandable if (i) there is no predictor or (ii) the target feature has only one value or (iii) the number of samples is smaller than a threshold.

Algorithm 18 MR_FindBestSplit_EachFeature

function MAP(D_3 = { <($label_i$, F_i, v_i) , {($y_{i,j}$, $fre_{i,j}$)} >})
  for each element e in D_3 **do**
    <(label, F, v) , {(y_j, fre_j)}>= e
    EMIT <(label, F), (v, {(y_j, fre_j)}) >
  end for
end function

function REDUCE(<Key = (label, F), Value={v_i, { $y_{i,j}$, $fre_{i,j}$}} >)
  if F is categorical **then**
    (bestsplit, gain, predictedValue) = FindBestSplit_CategoricalFeature(Value)
  else
    (bestsplit, gain, predictedValue) = FindBestSplit_NumericalFeature(Value)
  end if
  EMIT <Key, (bestsplit, gain, predictedValue) >
end function

The information of the best split point of each node is sent back to the controller to update the current tree model. If a node has the best split is NULL, we update this node from empty to a leaf node with the predicted value including. Otherwise, it is updated to a internal node with the value for predicting. In addition, we create empty children for this node. The number of children is exactly equal to the number of values in the split point's feature.

Before running the next iteration, the labels of each record in the training data are updated by using Algorithm 21. For each record, we traverse the tree model by checking it with the predicate of each node, from the root. If we reach a empty node at the end, that means "this record will be used to expand this node in the next iteration". Therefore, the ID of this node will be the new label for this record (Algorithm 21).

Algorithm 19 MR_FindBestSplit_EachNode

function MAP(D_4 = { <($label_i$, F_i), ($bestsplit_i$, $gain_i$, $predictedValue_i$) >})
  for each element e in D_4 **do**
    <(label, F), (bestsplit, gain, predictedValue) >= e
    EMIT <label, (F, bestsplit, gain, predictedValue) >
  end for
end function

function REDUCE(<Key=label, Value={F_i, $split_i$, $gain_i$, $predictedValue_i$})
  $maxgain$ = -∞
  $bestSplit$ = NULL
  for each element e in Value **do**
    (F, $split$, $gain$, $predictedValue$) = e
    if $gain$ >$maxgain$ and $F \neq$ target feature **then**
      $maxgain = gain$
      $bestsplit = e$
    end if
  end for
  EMIT <label, bestsplit >
end function

Remember that step *Preparing* and step UpdatingLabels are the map-only jobs. So, these job will be parts of the map phase in Map-Reduce job *CalculatingFrequencies* by default in SPARK.

The next iteration will take the updated data, and do the same process on it.

3.3.2.3 Pruning

In this section, we introduce a parallel version of Pruning procedures, which helps reducing the size of tree models to avoid over-fitting problem. Although the size of each tree model is quite small, we still focus on the scalable implementation because the whole procedure has to run cross-validation to re-learn tree models from different large datasets and evaluates all of them.

3.3.2.4 Random Forest

For building Random Forest in parallel, we need a Controller, which manage the learning processes of trees in forest. The Controller maintains:

- **Model Collection (MC)**: The set of tree model in the forest
- **ExpandingTreeQueue (EQ**: The queue of tree need to be constructed

Given **K** is the number of trees in the forest, **MaxParallel** is the maximum number of trees which can be constructed in parallel. The Controller schedules the expanding tree off of EQ until the

Algorithm 20 Function to find the best split point

function FindBestSplit_CategoricalFeature($S = \{v_i, \{ y_{i,j}, fre_{i,j}\}\}$)
  predictedValue ← most common target value in S
  if S match stop criteria **then**
    return (NULL, -∞, predictedValue)
  end if
  XValueSet = $\{v_i\}$ ▷ all values are split points
  gain = InformationGain(S)
  return (XValueSet, gain, predictedValue)
end function
function FindBestSplit_NumericalFeature($S = \{v_i, \{ y_{i,j}, fre_{i,j}\}\}$)
  predictedValue ← most common target value in S
  if S match stop criteria **then**
    return (NULL, -∞, predictedValue)
  end if
  $S\prime$ ← sort S by v_i ascending
  maxgain = -∞
  bestsplit = NULL
  for each adjacent (x_i, x_{i+1}) **do**
    $sp = (x_i + x_{i+1})/2$
    gain ← InformationGain when splitting by sp
    if gain >maxgain **then**
      maxgain = gain
      bestsplit = sp
    end if
  end for
  return (bestsplit, maxgain, predictedValue)
end function

Algorithm 21 UpdatingLabels

function U(p)datingLabelD_1 = {(label, (v_{F_1}, v_{F_2}, ..., $v_{F_{k-1}}$, y)) }, M ▷ M is the current tree model
  for each element e of D_1 **do**
    node ← the deepest node in M when traversing e
    if *node* is leaf **then**
      update label of e to -9 ▷ this record won't be used in the next iterations
    else
      update label of e to *newLabel*
    end if
  end for
  return D_1
end function

queue is empty and none of the tree it schedules remain running construction. Each tree building is launched in a separate thread, so that Controller can send out multiple expanding tree in parallel.

Two important tasks which make a forest learning to be Random Forest learning is randomizing the training set and randomizing the splitting features. The first task can be done easily with a built-in function in SPARK. To do the second task, we use the same procedure of Labelling Tree Building, which was presented in section 3.3.2.1, but modify the reduce function of Algorithm 12 a little bit (Algorithm 22): for a given node, we only select the best split point on features that belong to a random feature set (lines 12 and 14).

Algorithm 22 The modified version of MR_FindBestSplit_EachNode

```
1: function MAP(D_3 = RDD[(label, (F, bestSplitPoint, qualityOfSplit))])
2:     for each element e in D_3 do
3:         EMIT e
4:     end for
5: end function
6:
7: function REDUCE(<Key K=label, Value Set V = {(F_i, splitpoint_i, quality_i)} >)
8:     MaxQuality ← −∞
9:     Best ← UNKNOWN
10:    FullFeatureSet ← {F_i}
11:    N ← number features in FullFeatureSet
12:    RandomFeatureSet ← select randomly N/2 features in FullFeatureSet
13:    for each v_i = (F_i, splitpoint_i, quality_i) ∈ V do
14:        if ( thenMaxQuality < quality_i and F_i ∈ RandomFeatureSet)
15:            MaxQuality = quality_i
16:            Best = v_i
17:        end if
18:    end for
19:    EMIT (label, Best)
20: end function
```

3.4 Experiment Evaluation

In this section, we used decision tree with pruning and random forest to predict the next location of mobile user from data from MIT.

3.4.1 Data

In the experiment, we used MIT data which be introduced in chapter 2. This data is in schema: $Userid$, $Year$, $Month$, Day, $DayOfWeek$, $LocationAtTimeInterval_1$, $LocationAtTimeInterval_2$,..., $LocationAtTimeInterval_L$.Because time is a continuous variable, we tried to discrete it into L equally time interval with length M (minutes).$LocationAtTimeInterval_i$ means the location which user was be there in the longest time at time interval i.

Algorithm 23 Random Forest Algorithm

```
  function BUILDFOREST(K, MaxParallel)
Require: Training set D
    CurrentExpanding ← 0
    while true do
      if NumTrees < K and number expanding trees < MaxParallel then
        D1 = select N records from D with replacement      ▷ N is the number of records in D
        NewThread(LabellingID3())
      end if
      if NumTrees ≥ K and no running job then
        break
      end if
    end while
  end function
```

For using with Tree-based approach, we transformed the current data into schema:

$Userid$, $Month$, $DayOfMonth$, $DayOfWeek$, $currentTimeInterval$, $currentLocation$, $nextTimeInterval$, $nextLocation$

Where ($currentTimeInterval$, $currentLocation$) and ($nextTimeInterval$, $nextLocation$) are two pairs of time/location in two adjacent time intervals.

To reduce the impact of the old information (of a long time before) and use this tree for unseen user, we don't use Userid, Month in building tree. The full data is sorted by userid and time, then we select the first 80% days of the study of each user to form up the training data. The rest is used for making testing data : in each day, generate randomly k test cases (k is random too). After transforming, we have 163024 lines of training instances and 9371 lines of testing instances.

As mentioned before, we run evaluation with 2 model scopes: i) Global model ii) Individual model. With each approach, we have many ways to choose the predictors. Using classification tree, we try to predict the next location based on information of the day of week, the current location and the next time interval:

$NextLocation \leftarrow DayOfWeek + CurrentLocation + NextTimeInterval$

Where,

$DayOfWeek$ and $CurrentLocation$ are used as categorical features

$NextTimeInterval$ is used as numerical feature

In the experiments below, we only use ID3 and Random Forest of ID3 to test the performances, with different approaches and parameters:

- Global model: Use the whole training data to build a single full tree (without pruning) and use this tree to make predictions for everyone
- Global model with pruning: The global model will be pruned to avoid over-fitting problem

- Individual models: Use the data of each user to build a single full tree for theirself, and use this tree to make predictions only for him/her
- Individual models with pruning: The individual model will be pruned before using
- Random Forest: Use the whole data to build a global forest, and then use this forest to make predictions for everyone. Parameter N - number of trees will be used with different values: 5, 10, 20 to customize the forest

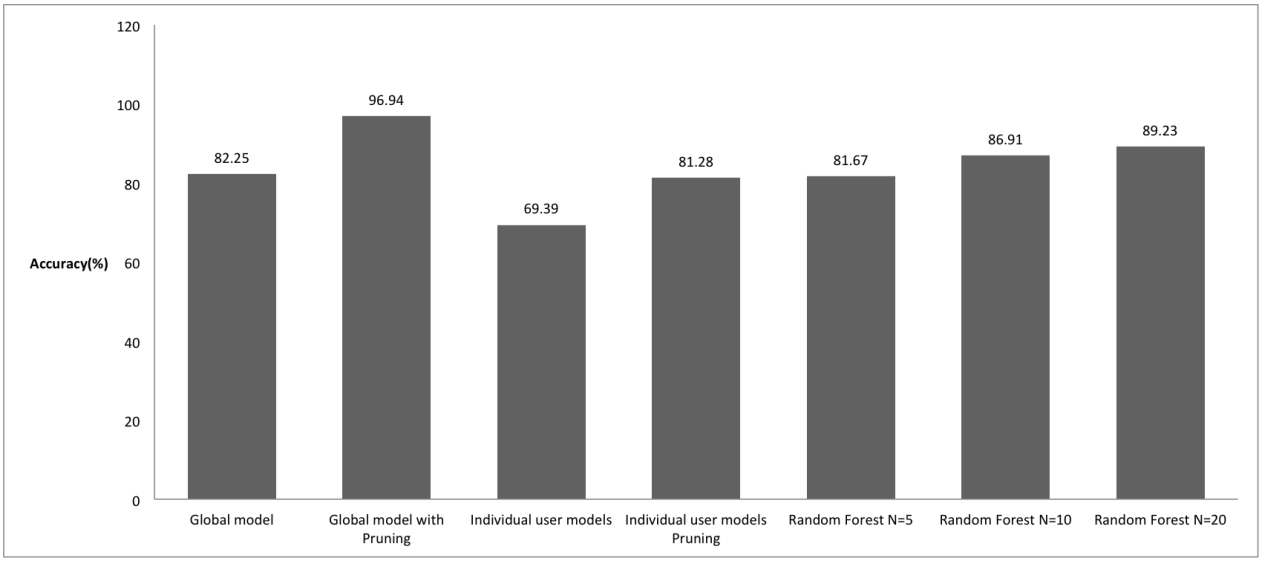
3.4.2 Set up

Because the data after pre-processing is quite small (around 4.4MB), we run it on the local environment, although the program can support to run in the scalable environment. The specifications of computer: Macbook Pro, 2.2 GHz Intel Core i7, 8GB RAM, 128 GB SSD.

3.4.3 Results

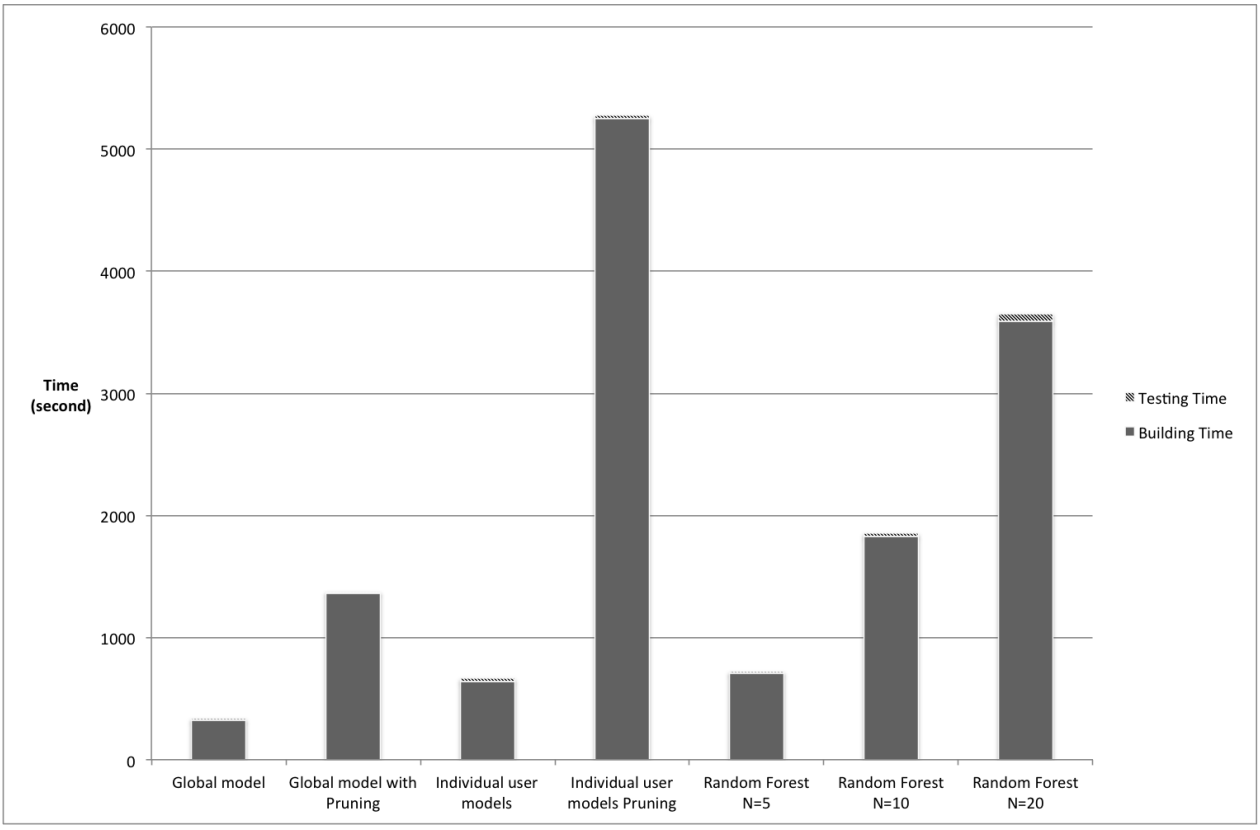
According to the result of experiments, the Global model with pruning gives the best result with the accuracy is 96.94%. The Random Forest with N=20 and N=10 also brings us the good performance with the accuracies are 89% and 87% respectively.

Figure 3.6



The individual approach takes the most time to build the tree, but the performance is the worst.

Figure 3.7


6000
5000
4000
3000
2000
1000
0
Time (second)
Global model
Global model with Pruning
Individual user models
Individual user models Pruning
Random Forest N=5
Random Forest N=10
Random Forest N=20
Testing Time
Building Time

Chapter 4

Trajectory-based approach : Movement patterns identifying

4.1 Introduction

Although having quite high accuracy performance, tree-based algorithm still have some disadvantages. Algorithms like CART, ID3, C4.5... don't support incremental building. It means when we have the new data input, we have to build the models from scratch again. Of course, we can use Streaming Parallel Decision Tree (SPDT) [7] to overcome this shortage, but it can not be used with Random Forest, which helps increase the accuracy significantly. Besides, because the input is historical data, we have to take care the "out-of-date" data, which can take impact directly on the tree structure.

In other hand, algorithmic scalability of traditional approaches is problematic. Based on our experience with, for example, regression trees, it is clear that complex traditional techniques to address regression and classification problems are difficult to parallelize. Instead, simpler – yet effective – approaches designed with scalability in mind are amenable to a parallel implementation with substantially smaller effort.

To address these lackings, we try another approach: trajectory-based approach. This approach can helps us build incremental models, reduce the suffer from "old" data, can implement on both streaming or batch processing architecture.

In this project, we use both temporal and spatial information, uses time-series knowledge and clustering technique to prediction the next locations of a user at a given time in near future. The new approach, named "Movement Pattern Identifying" (MPI), helps us predict the next location of a given user at any time in the near future by using any sequence of locations in the near past. It means the new approach is more flexible than the other in the literature, according to our knowledge.

4.2 Background

The underlying idea of our algorithm is to treat user mobility as patterns, defined in a period of time. By representing users as mobility patterns, our goal is to extract patterns from the data, and to group similar patterns together. We treat the data as time-series information, and then use clustering algorithms to do pattern extracting.

4.2.1 Time-Series

A time series is is a sequence S of historical measurements x_t of an observable variable x at equal time intervals.Time series are studied for several purposes such as the forecasting of the future based on historical knowledge, the understanding of the phenomenon underlying the measures, or simply a succinct description of the salient features of the series....[8].

In order to applied time-series in our approach, we need to transform the data into the suitable schema.

As mentioned in section 2.3, the transformed dataset has the following "schema":

$userid, year, month, dayOfMonth, dayOfWeek, hour, minute, area.cellID, uniqueCellID$

where $uniqueCellID$ is an arbitrary mapping from $area.cellID$ to a unique identifier. This is done as follows: each tuple is sorted by area.cellID to extract distinct values; unique $area.cellID$ are assigned to a unique index to create the mapping.

In words, a single tuple in the original schema reads as "for a given user, he/she was connected to the service cell $uniqueCellID$ since time(year, month,day, hour, minute)". For example:

20,2004,10,24,Fri,12,09,2200.453,18
20,2004,10,24,Fri,12,15,2200.493,32

which in words, indicates that at 12:09 Friday, 24/10/2004, user 20 changed the service cell which his phone connect to to cellID 18. (Before this time, his phone connected to another cell ID, let's say 17, for instance). His phone kept connecting to cell 18 until 12:15 of the same day then changed to cell 32.

Our goal is to develop a time series of cell IDs for each user in a given discrete time granularity. We select such granularity to be a day-worth of data. The new data is in schema:[64]

$userid, date, (time0, cell0), (time1, cell1), (time2, cell2), ..., (timeN, cellN)$

which reads as: in the day indicated in the "$date''$ attribute, the user indicated in the "$userid''$ attribute changed the cell service N times. The sequence above, thus, indicates service cell transitions, that happened at time indicated in the "$time_i''$ attribute of the inner tuple $(time_i, cell_i)$. Our pre-processing is not over. To train/detect patterns, we need to transform data from "transition information" to "location information", which look like: for a given user, at a given time, the service cell is X. Because time is a continuous variable, we have to discretize it into smaller ***time intervals***.

Why do we have to do it ? The fact that there are 2 approaches:

The first is keeping information in continuous of time. For example, we have a 2 movement paths like bellow:

$User_0$, $time_0 \rightarrow time_1 : location_1$, $time_1 \rightarrow time_2 : location_2$, ..., $time_{N-1} \rightarrow time_N : location_N$
$User_1$, $time_0\prime \rightarrow time_1\prime : location_1\prime$, $time_1\prime \rightarrow time_2\prime : location_2\prime$, ..., $time_{N-1}\prime \rightarrow time_N\prime : location_N\prime$

In this approach, the length from time0 to time1 may be different from the length from $time_1$ to $time_2$.

The second approach is we standardize each movement path into a sequence of locations in many ***equally time intervals with length M***. For example: ($\mathbf{M} = 60$ minutes)

$User0$, $00:00 \rightarrow 01:00 : locationA$, $01:00 \rightarrow 02:00 : locationB$,...., $23:00 \rightarrow 24:00 : locationZ$
$User1$, $00:00 \rightarrow 01:00 : locationA'$, $01:00 \rightarrow 02:00 : locationB'$,...., $23:00 \rightarrow 24:00 : locationZ'$

or simplify it:

User0, locationA, locationB,....,locationZ
User1, locationA', locationB',...., locationZ'

Figure 4.1 shows the visualization of a movement path. The x-axis is hours in day. The y-axis indicates the locations of a given user.

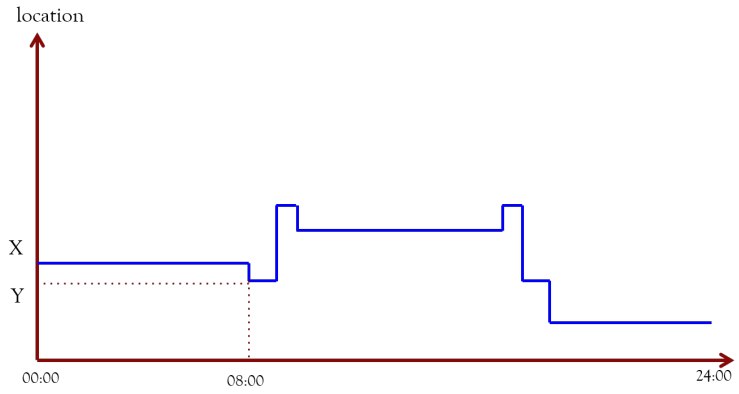
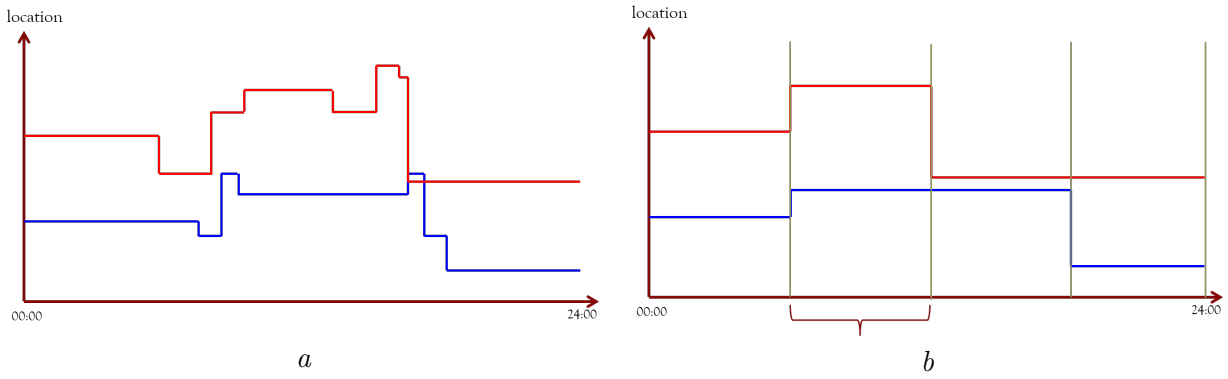


Figure 4.1: Example of a movement path

In our algorithm, we try to identify the similar movement paths and group them together. Using a essentially sampling the original time series at regular time intervals to "align" their movement paths (the second approach) will help this task done easier.

The challenge is: in each time interval, user can change and connect to many cells. For example, from 12:00 to 13:00, the user connect to cell X in 15 minutes, and cell Y in 35 minutes, and then reconnect to cell X in the rest of time. So what is the only one cell we put into data ? Our solution is choosing the service cell which he spent the most of time in this time interval to connect to. It means, we choose the service cell has the largest probability that the user will connect to in this time

interval. Because we select only one cell to make data, we will ignore some other information which can affect the accuracy of the whole algorithm. If M is small enough, this problem will disappear. But how can we choose value for M ? It depends on the speed of the user's movements. In our project, we use different values of M. With MIT's data, users don't move a lot, M equals to 1 hour. With the practical data from SWISSCOM, we set M to 15 minutes.



a *b*

Figure 4.2: (a) The unaligned movement paths are hard to calculate the similarity
(b) The movement paths are aligned with 4 time intervals (M = 360 minutes)

4.2.2 Clustering algorithm

Clustering algorithm helps grouping a set of objects in such a way that objects in the same group (called a *cluster*) are more similar (in some sense or another) to each other than to those in other groups (clusters). Clustering is a well studied problem, for which a number of popular algorithms exist, belongs to 3 families: Centroid-based clustering, Distribution-based clustering, Density-based clustering and Hierarchical clustering, such as: K-Mean (MacQueen, 1967) [17], DBSCAN (Martin Ester et. al., 1996) [3], ...

K-Mean firstly initializes K center points, and assign data points to the nearest center to form up groups, then adjust the center by calculating the centroid point of data points in each group. We will continue this procedure until the center points can not adjust anymore (or terminate the algorithm after n iterations.) DBSCAN firstly initialize the set of centers with 1 point and then consider the distance of each data point to the nearest center. If this distance is smaller a threshold, the data point will be assigned to that center, otherwise, use this data point is a new center. The algorithm will be stopped when there is no data point to be considered.

Each algorithm has some properties and drawbacks. For example, K-Mean and the other Centroid-based algorithm are very simple, powerful and easy to implement in parallel but can not deal with noises and requires the input value of the number clusters, what we don't know definitely. Or with DBSCAN and other Density-based algorithm, we often specify a threshold of maximum distance of two points, and it often runs sequentially.

Because of these advantages of K-Mean, in our work, we chose it and a variation thereof (which is customized to deal to its drawback) to help building models and compare their behavior.

Next, we introduce 2 simple clustering algorithms: one is K-Means, and the other is the variant of

K-Means. Two algorithms are only different in the way to adjust center points after each iteration.

4.2.2.1 K-Means

As mention before, the algorithm of K-Means is quite simple: From the initial state, we select random K points in the data to be the centers.

This algorithm will be executed in N iterations (or until when there is no change of the centers).

In each iteration:

* **Assigning**: for every point in the data, we calculate the distance to each center, and assign it to the nearest center.

* **Adjusting**: adjust the center of each cluster by calculating the centroids point (by average) and use it as a new center in the next iteration.

In 2009, W. Zhao introduced the parallel version of K-Means in MapReduce model [24]. At the initial phase, they select randomly K centers as before. Each iteration of this algorithm is a MapReduce job called *MR_ Clustering* (Algorithm 24). The map phase does "Assigning" task. It assigns the center point to the nearest center. The output of the map phase is pairs of form $<c, p>$, where c is the nearest center of data point p. In the reduce phase, the data points which are assigned to the same cluster will be dispatched to the same reducer. At each reducer, these data point are used to calculate the new new position for the associated center. The new center points become the input for the map phase in the next iteration.

4.2.2.2 K-Modes

The algorithm of K-Modes [11] is similar to K-Means, but instead of considering the average of each cluster to be the new centroid, we consider the "mode" of this cluster.

Let X, Y be two categorical objects formed up by *mcategorical* attributes.The distance between X and Y is the total mismatches of corresponding attributes of two objects. The smaller number of mismatching, the more similarity. Formally,

$$distance(X, Y) = \sum_{j=1}^{m} \delta(x_j, y_j) \tag{4.1}$$

where,

$$\delta(x_j, y_j) = \begin{cases} 1 & x_j \neq y_j \\ 0 & x_j = y_j \end{cases}$$

For example, the distance from (1,2,3) to (4,2,5) is 2 because of the differences of values in the first and third dimensions.

Let $S = X_1, X_2, .., X_n$ is a set of categorical objects which are described by m categorical attributes.

Algorithm 24 Parallel K-Means

Require: Data points P = $\{p_i\}$
  function MR_Clustering_map(Data points P = $\{p_i\}$, Centers set C = $\{c_i\}$)
    for each data point p in P **do**
      $nearestCenter \leftarrow$ NULL
      $nearestDistance \leftarrow 0$
      for each center c in C **do**
        $d \leftarrow$ distance(p, c)
        if d $<$ $nearestDistance$ **then**
          $nearestCenter \leftarrow c$
          $nearestDistance \leftarrow d$
        end if
      end for
      EMIT($nearestCenter$, p)
    end for
  end function

  function MR_Clustering_Reduce(Center c, set of data point $S = \{p_i\}$)
    $newCenter \leftarrow$ average(S)
    EMIT(c, $newCenter$) ▷ this new center will be used in the next iteration
  end function

  function Controller
    $Centers \leftarrow$ select randomly K point in P
    $i \leftarrow 1$
    while *true* **do**
      (OldCenters, NewCenters) $\leftarrow$ MR_Clustering(P, Centers)
      if NewCenters $\neq$ OldCenters or $i \geq$ MaxInterations **then**
        break
      end if
      i $\leftarrow$ i + 1
    end while
  end function

A mode Q of S is a vector $[q_1,q_2, .., q_m]$ that minimize the total distance to all objects in S. Q is not necessary an object of S. S can have more than one mode.

For instance, the mode of 3 data points: (1,2,3); (1,6,5); (2;6; 3) is (1,6,3).

Like K-Means, at the initial stage, we select random K points in the data to be the centers.

The below process is executed until the centers have no change or the number of iterations is more than a threshold:

* **Assigning**: for every point in the data, we calculate the distance to each center (by using the equation 4.1), and assign it to the nearest center.

* **Adjusting**: adjust the center of each cluster by calculating the **mode** of the associated data points and use it as a new center in the next iteration.

4.2.2.3 Distance Functions Overview

In any clustering algorithm, one of the most important point is: what is the distance function. The distance of two movement paths is their similarity.

There are many ways to calculate the distance (now is the distance of two cell index) of two vectors in which each dimension values is a metric space's element, but the three below are simple and often be used:

- Euclidean distance [4]
- Manhattan distance [5]
- Cosine similarity [2]

In the Evaluation section, we will use all of three to run algorithm and compare the accuracy in predictions.

4.3 Algorithm

In this section, we introduce our algorithm, which combine time-series and clustering technique, helps building incremental models and avoid the impact of the old data.

Firstly, let's consider one observation: "The locations of a user is determined by the action which he/she does". Therefore, a sequence of locations has a strong relationship to the sequence of actions of this user. Besides, people often change their locations only when they awake, or by words, they start moving and create such sequence from the time they wake up and end it when sleeping (within a day). For each new day, they start the new sequence of actions, or the new sequence of locations. From that observation, we denote a ***movement path*** is a sequence of locations which a user visited, order by time **within a day** (from 00:00AM to midnight).

Obviously, the position of a user for a time in the future is given by the aggregate behaviors of the user in the near past.

With a "regular" user, there are many similar movement paths. For a given user, is the "current" movement path till now similar to any path which we already known ? Can we identify them, and use them to make prediction ? That is the key idea of this approach.

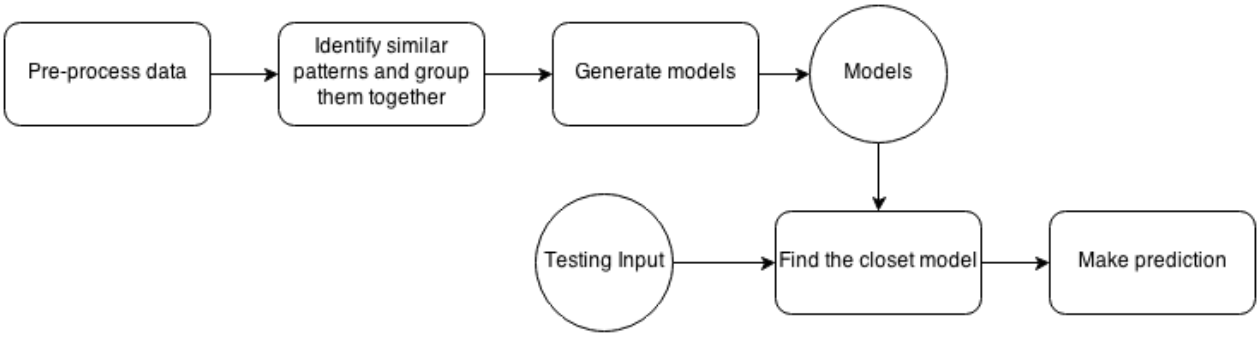


Figure 4.3: Workflow of trajectory-based approach

From the training data, we do pre-processing to filter out the invalid data, and transform it to the collection of daily movement paths of users. Then, using clustering algorithm to identify the similar patterns and group them together. For each pattern, a representative model is generated. When finishing the training phase, we have some global models. In the testing phase, for a given input, we try to find the closet model for it, and use this model to make prediction for the next location. An input is a sequence of pair time/location in the near past.

Next, we present the two main phases of our approach: i) the training phase, or model building phase, which uses clustering to find groups of similar users and build a model of their mobility patterns, and ii) the testing phase, or prediction phase, in which, for a given user, we use the available historical information to select the closest model and use the latter to predict future possible locations of the user.

4.3.0.3.1 Training phase Before this phase, we assume that the training data is preprocessed as described in section 4.2.1 and has schema:

$User$, $Location_1$, $Location_2$,...., $Location_L$.

Where, $Location_i$ is the location of user which indicated by $User$, in time interval i; L is the number of time interval in each day.

The training phase has 2 main steps:

- Identify and group the similar paths (or also called *clustering*)
- Generate approximate models that are representative for each group

Identifying and grouping the similar paths With a regular user, his movement paths can be very similar over days. In this phase, we try to identify them, and group them together. We can

use any clustering algorithm, such as K-Mean, K-Mode, DBScan... For quick proving the potential of this approach, and for simplicity in develop scalable algorithm, we use K-Mean and it's variants.

The output of this step is groups (clusters) and their associated movement paths.

Generating models After clustering the data points, we get many clusters, each cluster has information of the center points and the associated data points. Remember that each point is sequence of locations and be viewed as a vector. Although these centers are the representatives of the clusters, but it's not good enough for using to make prediction. Because with the output of K-Mean, the centers are the average of data points within the same cluster. If we use these center points to make prediction, the results are not good (because the prediction can be a cell ID which not exist in data). Besides, each dimension of centers has only one value. It's mean, we can only provide one value in each prediction.

Could we build models which can provide us more than one prediction ? Instead of store only one value in each dimension of center point, we can store more, and make it become a ***Probabilistic model***. This is the purpose of this step.

Denote $\mathbf{L = 60*24/M}$ is the number of time intervals in each day, where M is the length of each time interval (in minute).

After "Clustering" step, we have numbers of cluster of movement path. In this step, we will generate the **Probabilistic model** for each cluster.

Given cluster with N paths:

$Loc_{1,1}, Loc_{1,2}, .., Loc_{1,L}$
$Loc_{2,1}, Loc_{2,2}, .., Loc_{2,L}$
...
$Loc_{N,1}, Loc_{N,2}, .., Loc_{N,L}$

We will generate a matrix Locs has size $T \times L$. Each column contains value of cell ID and the probability if connect to that cell, order by probabilities. T is the maximum number of location in each time we should keep track. For each time interval i, select top T locations (and its probability) which have the most frequency to be the column i of the matrix.

For example: $M = 360$ minutes, $L = 4$ time interval, we want to keep track $T = 3$ locations in each time interval. Assume that after clustering, a cluster contains 100 movement paths, which is summary like below:

Time interval 1		Time interval 2		Time interval 3		Time interval 4	
Location	Frequency	Location	Frequency	Location	Frequency	Location	Frequency
10	37	33	50	27	67	89	100
12	30	49	50	22	20		
23	20			25	12		
44	6			44	1		
94	7						

After selecting top 3 locations which have the most frequency in each time interval, we get the

matrix:

$(10, 0.37)$ $(33, 0.5)$ $(27, 0.67)$ $(89, 1.0)$
$(12, 0.30)$ $(49, 0.5)$ $(22, 0.20)$ $(\_, \_)$
$(23, 0.20)$ $(\_, \_)$ $(25, 0.12)$ $(\_, \_)$

It means that from 00:00 AM until 06:00AM, user connect to cell 10 in 37% of time; connect to cell 12 in 30% of time; and connect to cell 23 in 20% of time. From 06:00AM until 12:00 AM, user can connect to : cell 33 with 50% and cell 49 with 50% of time. From 12:00 AM to 6:00 PM, user can connect to cell 27, 22 or 25 with probabilities is 67%, 20%, 12% respectively. From 06:00 PM to midnight, he only connect to cell 89 in all cases.

If using this model to make a prediction, we can give a distribution of up to 3 possible locations which user can be visited in the time to come. By selecting top K locations as above, we make sure the prediction has the biggest probability to be true.

The figure 4.4 expresses the key idea of this step;

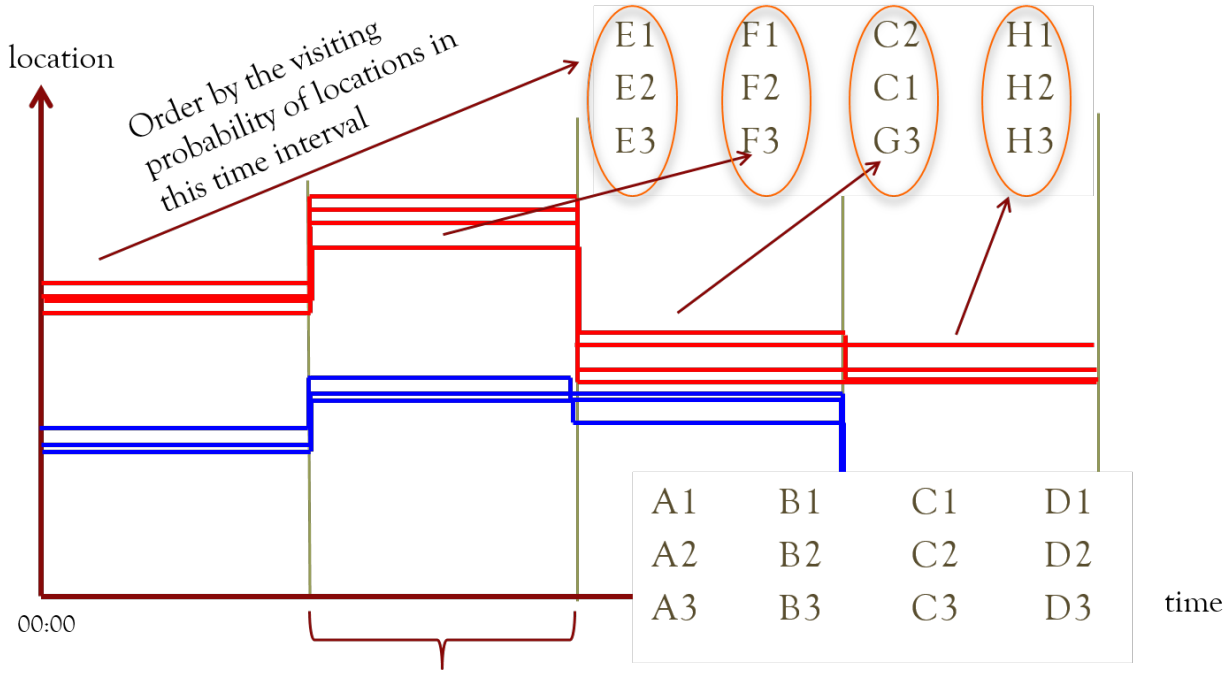


Figure 4.4: Identifying and grouping similar movement paths

4.3.0.3.2 Testing phase : Make prediction In this phase, when the new input comes, we have to select the best model, and use this model to make prediction.

Given model with data is matrix $T \times L$. Each column contains value of cell ID and the probability if connect to that cell, order by probabilities. (T is the maximum number of location in each time we should keep track, L is the number of time interval in each day)

Given a series of time-location: $S = [(time\_interval_1, cell_1), (time\_interval_2, cell_2), ..., (time\_interval_N, cell_N)]$

The number of elements of S may be smaller than $\mathbf{L}$ and $time\_interval_1, time\_interval_2, time\_interval_N$ is not necessary be in the continuous sequence.

We try to find out *which model is the best fit* by calculate the matching rate of the input values on each model:

$rate = \sum_{i=1}^{m} f(loc_i, time_i)$

Where,

(loc_i, $time_i$) is pair of (location, time_interval) – the ith element of the input

m is the number previous locations of the input

$$f(loc, time\_interval) = \begin{cases} 2^{-j/2} & \text{if location } loc \text{ is found at row } j \text{ of column } time\_interval \\ 0 & \text{if location } loc \text{ is not found in column } time\_interval \end{cases}$$

The best matching model is the model holding the biggest matching rate.

Assume that we want to predict the location of a given user at time interval t. The whole column t^{th} of the best matching model, which contains a sequence of locations ordered by its probability, can be used as the prediction.

The matching formulation will keep the properties: matching 2 points at row $i + 1$ *is better* than matching 1 point at row i.

Algorithm 25 described the implementation of this phase.

For more understanding, let's take an example: assume that we have the input: $(0, 12), (2, 27)$ means in time interval **0**, user connect to cell 12; in time interval **2**, user connect to cell 27. We want to predict the cell ID in time interval **3**.

The model is :

(10, 0.37) (33, 0.5) (27, 0.67) (89,1.0)
(12, 0.33) (49, 0.5) (22, 0.28) (_,_)
(23, 0.30) (_, _) (25, 0.15) (_,_)

With value of input $(0, 12)$, in column 0, we found cell 12 at row index 1.

With value of input $(2, 27)$, in column 2, we found cell 27 at row index 0.

$Weight = 2^{-1/2} + 2^{-0/2} = 1.707$

The figure below shows the similar example.

Assume that we have 2 daily mobility patterns in the training data. They are clustered into two groups (red and blue), and be generated two approximate models corresponding. The time interval length M is 360 minutes (each day has 4 time intervals) . The testing input is a set of three pairs time/location of the first three time intervals (indeed, the input is not necessary be list of locations in the continuous time intervals). We want to predict the location of user in the last time interval. In this example, the matching rate of the blue's model is bigger than the red's one. So, the prediction is made base on the blue's model.

Algorithm 25 Make prediction from the probabilistic model

Require: $models = array[model_i]$, is a collection of models
  function FindBestMachingModel(input= array[($location_i$, $time\_interval_i$)])
    $n_{input} \leftarrow$ number elements of input
    $bestRate \leftarrow -\infty$
    $bestModel \leftarrow$ NULL
    for each model m in $models$ **do**
      $rate_m \leftarrow 0$
      for j from 0 to $n_{input} - 1$ **do**
        $h \leftarrow$ number rows of m
        for r from 0 to $h - 1$ **do**
          if $location_i$ is found at row r **then**
            $rate_m = rate_m + 2^{-r/2}$
            break
          end if
        end for
      end for
      if $rate_m >$*bestRate* **then**
        $bestRate \leftarrow rate_m$
        $bestModel \leftarrow m$
      end if
    end for
    return $bestModel$
  end function
  function MakePrediction($input$= array[($location_i$, $time\_interval_i$)], t)
                                                ▷ t is the time interval in future
    $bestModel \leftarrow$ FindBestMachingModel($input$)
    return column t^{th} of $bestModel$
  end function

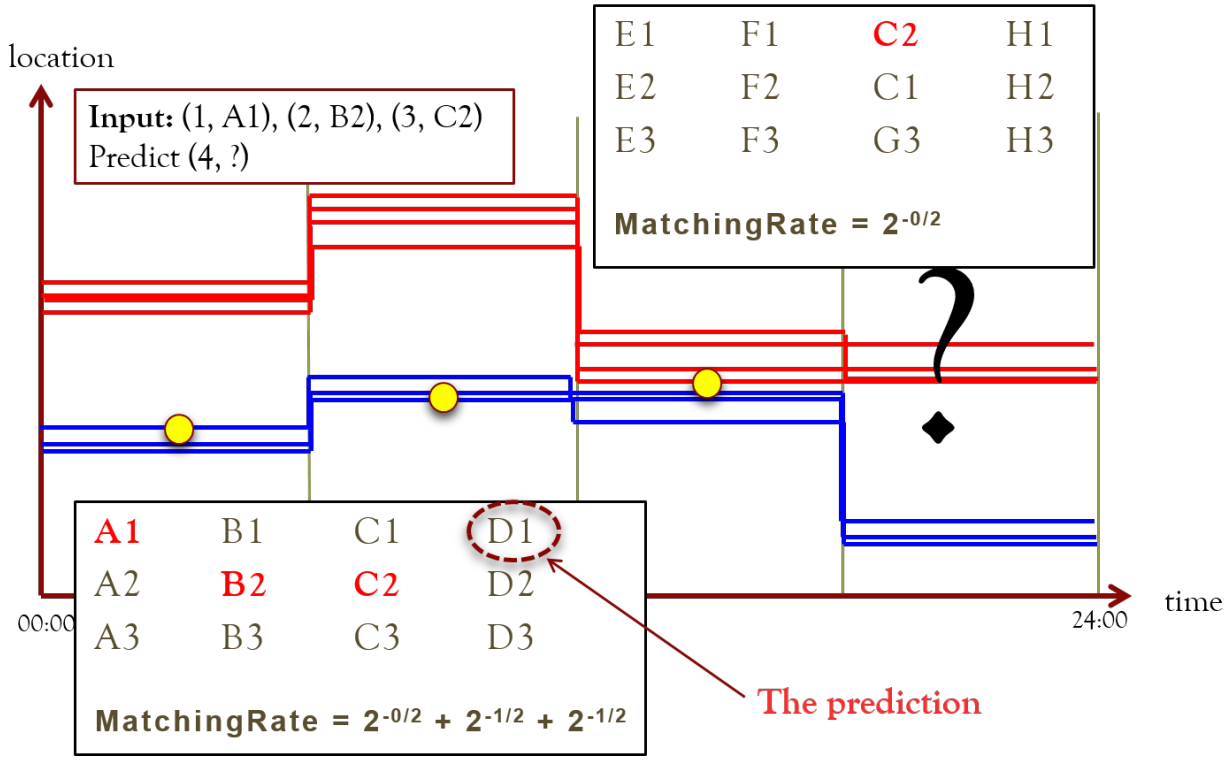


Figure 4.5: Example of making prediction

4.4 Scalable Algorithm Design

4.4.1 Challenges

Still be a main challenge of this project, over large dataset processing, the algorithms must have ability to run in scalable environment, particularly SPARK. Consequently, our machine learning algorithm need to have a controller to manage the whole distributed processes.

Besides, to prove the potential ability of our approach, we use K-Means and its variant to extracting mobility patterns. As subsequence, there are two points we should address:

First, what is the value of K , or how many patterns in the data ? We have no knowledge about K before doing clustering.

Seconds, how can we calculate the distance between movement paths ? Remember that each movement path is a vector of locations. Each location can be the service cell ID, or the coordinate, depends on the data. In the special case, if a location is represented by a service cell ID, and be treat as a categorical attribute, we can use the distance function in equation 4.1. Otherwise, if a location is constituted from coordinate, K-Modes' distance is not a good choice. In this project, we want to solve the general case : each location is a geographical point. With this point of view, the algorithm can be changed and adapt to any kind of location in any data. Therefore, the movement paths' distance problem becomes how to calculate the distance between two locations.

4.4.2 Solutions

In order to address the challenge of no knowledge about K when using K-Means, we can use some solutions:

- (1) Using *The most frequency clustering* , our customized algorithm from K-Modes and K-Modes, which is introduced in section 4.4.2.1, instead of K-Means
- (2) Do cross-validation for finding the best value for K: The computation of this solution is expensive
- (3) Using other existing scalable clustering algorithms: unfortunately, at the beginning time of this project, there is only K-Means is developed in scalable environment

Among them, the first choice is the most reasonable because "The most frequency" not only keeps the simplicity of K-Means, but also has ability to adjust the number of cluster automatically. In section 4.4.2.1, we introduce about this algorithm in both local and scalable environment.

Besides, to solve the challenge of the scalability from huge dataset, we design a scalable version of the algorithm which is introduced before in section 4.4.2.2.We assume that the output models from the training phase are small enough to stored on a local machine. That means, the testing phase is done in a local environment, instead of scalable environment. Therefore, in section 4.4.2.2, we only focus on how to parallelize steps in the training phase.

4.4.2.1 The most frequency clustering

The idea of this algorithm is similar to K-Means: instead of updating the center's position of a cluster by taking the average, we try to move the centers to the regions that contains as many data points as possible, or we want to chose a "mode" that minimizes the distance to all other points in the cluster. However, there are some differences with K-Modes or K-Means:

- Each point of input data in this approach is not necessary be a vector of categorical value
- The number of clusters after each iteration is not necessary equal to K

The implementation is describe in algorithm 26.

From the initial state, from N data points, select random K points as the centers. ($\sqrt{(N)} <= K < N$)

The algorithm will run until there is no change in the centers or until the number of iteration reaches a given threshold. In each iteration, there are two main tasks: (i)form up clusters and (ii) adjust the centers.

To group data points into clusters, we assign each data point to its nearest center. A center point will be removed if it doesn't have any assigned data point.

Algorithm 26 The most frequency algorithm

Require: *maxIter* is the maximum number of iterations, which is given by user

```
1:
2: function THE_MOST_FREQUENCY(D = Array [datapoints])
3:     N ← the number of data points
4:     k ← √N
                                                    ▷ K is the initial number of cluster
5:     Centers ← sample K points from D
6:     iteration ← 1
7:     while Centers are not changed or iteration ≤ maxIter do
8:         iteration ← iteration + 1
9:         for each point p in D do
10:            c ← the nearest center of p
11:            cluter_c ← cluter_c + p
                                    ▷ cluter_c is the set of data point which are assigned to center c
12:        end for
13:        Centers ← Centers removes clusters which don't have data points
14:        for each center_i in Centers do
15:            center_i ← adjustCenter(cluster_{center_i})
16:        end for
17:    end while
18:    return Centers
19: end function
20:
21: function ADJUSTCENTER(D = set of data points)
22:    L ← number of dimension of data points in D
                                                    ▷ each point is form of (x_1, x_2, ..., x_L)
23:    NewCenter ← any point in L-dimension space
24:    for i from 1 to L do
25:        if dimension i is categorical then
26:            x ← the value occurs most frequently in all x_{i,j}
27:        else
28:            x ← the average value of all x_{i,j}
29:        end if
                                        ▷ x_{i,j} is the value in dimension i of point j^{th}
30:        Update the value of dimension i of NewCenter to x
31:    end for
32:    return NewCenter
33: end function
```

To adjust a center, we will change its value of each dimension. Because there is no constrain about the value type of each dimension in data points, we calculate the value of dimension of the new center in two different ways, depend on its type.

If the type of a dimension is categorical, the new value is the one that occurs in the data point most frequently. If the type of a dimension is numerical, the new value is the average of values in the same dimension of data points.

For example, suppose that there is a cluster that has 4 data points, each data point is an element of 3-Dimension space, two first dimensions are categorical, the last one is numerical.

("1", "5", 3)
("1", "4", 8)
("3", "4", 1)
("2", "3", 4)

In the first dimension, value **1** occurs most frequent (2 times). Similarly, in the second dimension, value **4** has the most frequency with 2 time. Because the last dimension is numerical, we select *4 = (3 + 8 + 1 + 4)/2* as representative value. Therefore, new center of this cluster will be ("1", "4", 4).

If types of all dimensions in data points are numerical, the "Updating" phase of this algorithm is the same as of K-Means. If types of all dimensions in data points are categorical, it is the same as of K-Modes.

Because we remove the center which has no associated data points, the number of cluster is changed after iteration. At the end of the whole algorithm, we get K′ clusters (K′ < K).

Parallel version Like Parallel K-Means, each iteration of *The most frequency* is a MapReduce job, called MR_Clustering (Algorithm 27).

The input of map phase is the set of data points, and the current centers. After finding and assigning the nearest center for each data point, we submit a tuple them for reducer. Each reducer receives a tuple: a center is the key, the list of associated points it the value. Depend on the type of each dimension of data points, we update the value in this dimension of the center by different ways.

4.4.2.2 Algorithm

In this section, we introduce the scalable version of the algorithm which is describe in section 4.4.2.2. We only focus on the parallelism of steps in the training phase: (i) Identifying and grouping the similar paths and (ii) generating models.

4.4.2.2.1 Identifying and grouping the similar paths To identify and group the similar mobility patterns, we use K-Mean and our algorithm "The Most Frequency" because of their simplicities. Indeed, we can use any scalable clustering algorithm.

Algorithm 27 Parallel The Most Frequency

Require: Data points P = $\{p_i\}$
function MR_CLUSTERING_MAP(Data points P = $\{p_i\}$, Centers set C = $\{c_i\}$)
for each data point p in P **do**
$nearestCenter \leftarrow$ NULL
$nearestDistance \leftarrow 0$
for each center c in C **do**
$d \leftarrow$ distance(p, c)
if d $<$ $nearestDistance$ **then**
$nearestCenter \leftarrow c$
$nearestDistance \leftarrow d$
end if
end for
EMIT($nearestCenter$, p)
end for
end function

function MR_CLUSTERING_REDUCE(Center c, set of data point $S = \{p_i\}$)
$L \leftarrow$ number of dimensions in each p_i
$newCenter \leftarrow$ NULL
for i = 0 to L **do**
if dimension i^{th} is categorical **then**
$newCenter_i \leftarrow$ the value occurs most frequently in all $x_{i,j}$
else
$newCenter_i \leftarrow$ the average of $x_{i,j}$
end if
▷ $newCenter_i$ is the value in dimension i of $newCenter$ ▷ $x_{i,j}$ is the value in dimension i of point j^{th}
end for
$newCenter \leftarrow$ average(S)
EMIT(c, $newCenter$) ▷ this new center will be used in the next iteration
end function

function CONTROLLER
$Centers \leftarrow$ select randomly K point in P
$i \leftarrow 1$
while *true* **do**
(OldCenters, NewCenters) $\leftarrow$ MR_Clustering(P, Centers)
if NewCenters $\neq$ OldCenters or $i \geq$ MaxInterations **then**
break
end if
i $\leftarrow$ i + 1
end while
end function

4.4.2.2.2 Generating models In this step, we want to generate the probabilistic model for each group in the output of clustering step. Each group contains movement paths which are similar together. In this algorithm, probabilistic model is a matrix of $T \times L$, where T is the number locations which we want to keep track, L is the number of time interval in each day. Column i^{th} in the model is the set of locations which users visit the most frequent in time interval i^{th}.

In order to construct the model, we simply order locations in each time interval of movement paths in the current cluster by their frequencies, and then select top T of them. This task is done by a series MapReduce jobs: (1) Calculating frequencies of locations (*MR_ CalculatingFrequency*), (2) Calculating Top-K Locations (*MR_ SelectingTopK*), (3) Constructing models (*MR_ ConstructingModels*) as Figure . Every single step is a MapReduce job. That makes sure the algorithm can run even if $K * L$ is very big. (K is the number of clusters).

Figure 4.6: Steps in the training phase of Trajectory-based approach

The map phase's input of *MR_ CalculatingFrequency* is a set of tuples (p, c), in which, each element expresses that data point p belongs to cluster c (Algorithm 28). This job is very similar to the word count example, which is introduce in section 1.2.1.1.

Algorithm 28 MR_CalculatingFrequency

```
function MAP(Clusters = {(p, clusterid)})
▷ L is the number of time interval in each movement path
    for timeInterval = 0 to L − 1 do
        loc ← the location at time interval i of p
        EMIT(<(clusterid , timeInterval, loc) , 1>)
    end for
end function

function REDUCE(<Key=(clusterid, timeInterval, location), Value = {1} >)
    frequency ← ∑1
    EMIT(<(clusterid, timeInterval, location), frequency >)
end function
```

The output of this job represents the frequency of locations in a given time interval in a given cluster by tuples in the form of <(clusterid, timeInterval, location), frequency) >. This is also the input of the next MapReduce job - *MR_ SelectingTopK*. In the second job, each mapper scans through the input, and then submits a tuple <K,V>for each records, where K = ($clusterid_i$, $timeInterval_i$) and V=($location_i$, $frequency_i$). The frequency information of locations at the same time interval of the same cluster will be dispatched to the same reducer. In the reduce phase, we select the top K locations based on their frequency (Algorithm 29).

In MapReduce job *MR_ ConstructingModels*, we sent all time intervals and the associated top K

Algorithm 29 MR_SelectinngTopK

function MAP(F = {<($clusterid_i$, $timeInterval_i$, $location_i$), $frequency_i$ >)})
  for each element f = <($clusterid_i$, $timeInterval_i$), ($location_i$, $frequency_i$) >) in F **do**
    EMIT(<($clusterid_i$, $timeInterval_i$), ($location_i$, $frequency_i$) >)
  end for
end function

function REDUCE(<Key=($clusterid$, $timeInterval$), Value = {($location_i$, $frequency_i$)} >)
  $V\prime \leftarrow$ sort V by frequency descending
  $TOPK \leftarrow$ Select top K of $V\prime$
  EMIT(<($clusterid$, $timeInterval$), $TOPK$)
end function

locations of each cluster to a reducer (Algorithm 30). At each reducer, the probabilistic model will be constructed by aggregation information of all top K locations. Column i^{th} is the Top K locations at time interval i^{th}.

Algorithm 30 MR_ConstructingModels

function MAP(F = {<($clusterid_i$, $timeInterval_i$), $TOPK_i$ >)})
  for each element f = <($clusterid_i$, $timeInterval_i$), $TOPK_i$ >) in F **do**
    EMIT(<$clusterid_i$, ($timeInterval_i$, $TOPK_i$) >)
  end for
end function

function REDUCE(<Key=$clusterid$, Value = {($timeInterval_i$, $TOPK_i$)} >)
  $Matrix \leftarrow$ construct matrix from $Value$
  EMIT($clusterid$, $Matrix$)
end function

4.5 Evaluation

In this section, we will describe our evaluation the algorithm with the two clustering approaches (KMean, The most frequency) and different parameters for each algorithm. The models is evaluated under two scope: Global and Individual.

4.5.1 Testing on MIT's data

4.5.1.1 Methodology

We know that one of draw-backs of K-Mean is the input of number clusters in data , which we don't know at all. The number of cluster effects to the accuracy of the whole algorithm. So, we have to find out some way to estimate this number. In these evaluation, we applied a simple technique to

reach that goal. Because K-Mean requires the number of cluster, which is a mystery at that time, we tried the algorithm with different values of K : $300, 350, 400, 450, 500$. With The most frequency algorithm, it isn't necessary to specify value of K.

4.5.1.2 Data

We use the data which has been presented in chapter 2, with 8904 records, corresponding to 8904 day of the study. Our data has schema: $userid$, $locationAtTimeInterval_0$, $locationAtTimeInterval_1$,..., $locationAtTimeInterval_L$

The first value of each line is the unique id of users. The next L values are the locations of that user at L time intervals in each day. Because this data is not sorted by any feature, we selected randomly 80% as the training set. For the testing set, we select randomly 20% lines of data, and then, for each line, select randomly a sequence of 4 locations: the first three locations and the time interval corresponding will be use as the input for the testing phase; the last location is the actual outcome which we want to predict. Specially, with the SINGLE approach, we have to input the userid also.

4.5.1.3 Set up

Because of the small size of training data, we run the algorithms on local mode(although it was written for scalable environment) with the specs: Macbook Pro, 2.2GHz, Core i7, 8GB RAM.

4.5.1.4 Accuracy metric

With the GLOBAL approach, we input the sequences of three $(location, time\_interval)$ and predict the fourth location as mentioned in step a. With the SINGLE approach, we add one more information in the input: userid, because the models are built for each individual user. Each prediction of the algorithm will contains $TOP_K = 3$ pairs of $(location, probability)$, where "$location''$ is the predicted location and "$probability''$ is the percentage of time which the current user has visited that "$location''$. A prediction is true if one of three values contains value of "$location''$ is equal to value of the actual location.

4.5.1.5 Testing result on MIT's data

The location of MIT's data is the service cell ID.

We tried to make prediction by the models built by 2 above algorithms : K-Mean and The most frequency with different distance functions, and then compared them in the figures 4.7 and 4.8.

The x-axis is algorithms, y-axis is the accuracy percentage of each algorithm. Because of the space availability in the figure, we use some notations of the algorithm's name, such as:

"Global-Kmean50-E" means "Building global models by KMean with $K = 50$, the distance function is Euclidean"; "Global-Kmean50-M" can be read as "Buiding global models by KMean with $K = 50$

and Manhattan distance"; "Single-TheMostFreq-E" means "Building models for each individual user by The Most Frequency algorithm and Euclidean distance" and use the similar way for the others.

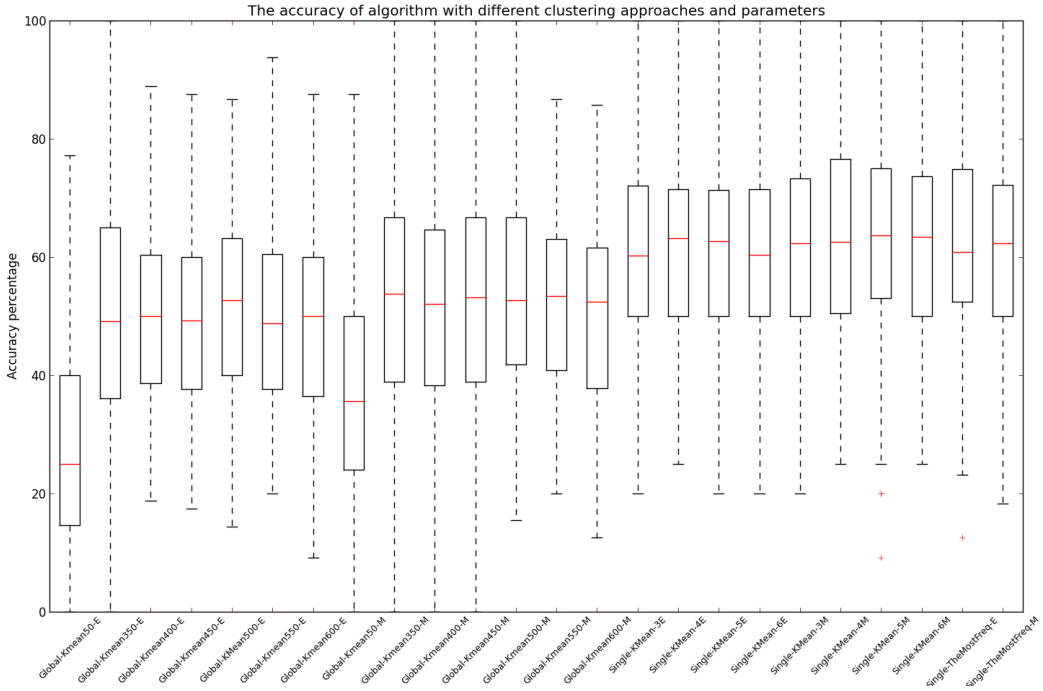


Figure 4.7: The accuracy of the algorithm with different clustering approaches and parameters

In the figure 4.7, we can see the accuracies of building models for each individual user approach are higher than building global models for all user. But, with approach SINGLE by using KMean, because of the mystery of how many mobility pattern of each user, we tried many values of K. The bigger K, the more over-fitting problem. This weakness is solved by using "The most frequency clustering" algorithm.

Figure 4.8 describes the time using for building models and testing with different approaches.

The time using for building models phase is proportional with the number of K - the number pattern which we input for KMean. The testing time is almost a constant value. Because of building model for each user sequentially in approach SINGLE, the training time is more higher than building GLOBAL models.

Missing the geographical information lead to the less accuracy in calculating distance of two service cells, that affect directly to the algorithm performance.

4.5.2 Testing result on SWISSCOM Data

4.5.2.1 Data

The data is collected in 9 days, split into 9 day based on the continuous of time:

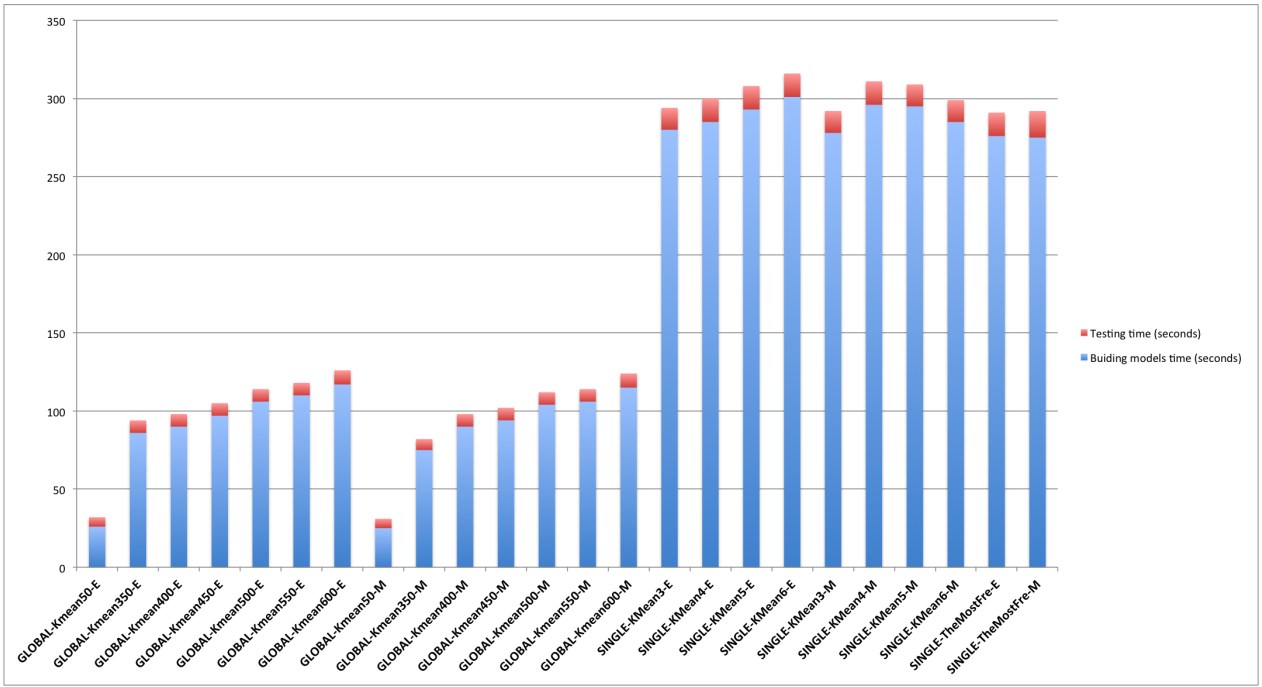


Figure 4.8: Time using for building model and testing

- **Part 1:** [13/05/2014 - 17/05/2014] : 16 GB
- **Part 2:** [23/06/2014 - 26/06/2014] : 2 TB

In this evaluation, we only build the *global models* on Part 1.

Part 1 contains information of 282768 users. We select 80% data from each user to form up the training set. The rest is testing set. It means, each user has only 3 movement paths for training. In case there is no user share the same pattern with another, the ratio of number clusters on the number of data points is around 1/3 (not good for clustering).

Besides, the data from SWISSCOM not only contain the information of cell like ID and coordinate, but also contains coordinates of users.

4.5.2.2 Methodology

Because each value of a daily movement path is the compound of location and time, two users share the same movement path means that these users have the close houses, go to the close place in the same time intervals. The bigger time interval length M, the bigger probability of the case that two users can share the same pattern. It seems that, there are not many pattern which are shared by multiple users. With 282768 users in the training data, each user has only 3 movement paths, the number of clusters can be up to 200000. This is a very big number, and can not be run in the current implementation and the current infrastructure. Besides, ***The most frequency*** is only suitable to applied on the data which K is more smaller than N. (K is the number of clusters, N is the number

of samples). Therefore, we try to evaluate the trajectory-based approach by build global models on Part 1 data with K-Means, K= 4000.

In this testing, we use **coordinate of each user** as his/her location. The coordinate is compound of latitude and longitude, for example (43755553.0, 343929.0)...

4.5.2.3 Set up

The cluster is used for evaluation has 1 master (34 cores, 40GB RAM) and 17 workers (2 cores, 2GB RAM).

4.5.2.4 Result

By considering each location coordinate, which is compound of latitude and longitude, as a tuple of two numbers, we calculate the error of the prediction on each component separately instead of calculating the geographical error.

The average error of prediction is (2110.622525430246, 1001.1482063166551)

Besides, in 11030/19970 $\approx$ 50% cases, we predict exactly the coordinates of a user.

The value K=4000 is too small with its true value. So, the result is not good as expected.

Chapter 5

Conclusion and future works

5.1 Conclusion

We have addressed the problem of location inference using two different algorithmic approaches: tree-based and trajectory-based approach for batch processing architecture.

As studied in literature, the tree-based approach, using Decision tree (ID3, CART) with the support of Random Forest, give us a very good performance. However, it's inflexible and hard to port to streaming architecture.

The second approach: Movement pattern identifying is more flexible, near with natural thinking, also give us the very promising results. Especially, the algorithm for this approach is incremental and easy to implement on streaming architecture.

5.2 Future works

Because of the time constrain of this project, the acceptable results from both approaches is not described all their abilities. In the future, respect to each algorithm, we can do some improvement to increase the performance overall.

5.2.1 Tree-based approach

In this project, we only used some directly features as predictors. These predictors didn't assure getting the best result.

The first improvement we can do is applying the "feature selection" to choose the good predictors. A feature selection algorithm is a the combination of a search technique for proposing new feature subsets, along with an evaluation measure which scores the different feature subsets.

In addition, the existing feature can combine with data from other source to refer the new feature, which can useful in our algorithm, such as : The semantic places near by the current location, the

actual road map...

5.2.2 Trajectory-based approach

Section 4.3 described the steps to identify and group the similar patterns: From the whole data after pre-processing, consider each line as a data points, and run the clustering algorithm to divide data points into groups and generate model for each group.

The problem is, the data from SWISSCOM contains information of millions users, each user can have a lot of days of study. There is not many users have the similar movement paths (because they live at different houses, go to office at different hour...). Besides, each user can have more than 1 mobility pattern. It means that there may be more than millions patterns, and we can not run clustering algorithms with big value of K like that to build global models!

The disadvantage of building "Individual models" is inflexible. With a new user, we don't have his model for making prediction. That why we prefer global models to the individual models. So, how can we build global models in the better way ?

5.2.2.1 Improvement 1

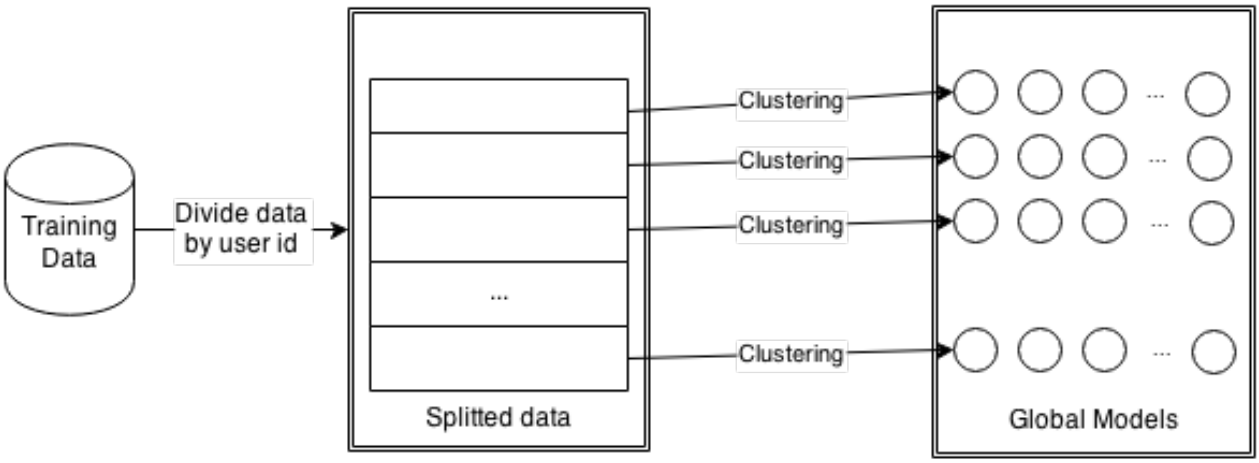
Let's consider the requirements of clustering algorithm's output. We want to minimize the internal-distance (distance between datapoints in the same cluster) and maximize the intra-distance (distance between clusters). One observation is, the movement paths in the same cluster often only belongs to very little users. If the number clusters of the output is smaller than its true values (case 1), the internal distance is big. If the number of clusters of the output is bigger than its true value (case 2), the intra-distance is big, and we get the over-fitting problem. But, in this project, using the habit of a user to predict the next behavior of this user (case 2) is often better than using the habit of a groups similar users for this task (case 1).

On the other hand, in case already having a good clustering result, we can make a strong assumption that: for a given cluster, the movement paths in this cluster is often belong to very little users. Because each value of a daily movement path is the compound of location and time. Two users share the same movement path means that these users have the close houses, close office, go to the close locations in the same time intervals. The bigger time interval length M, the bigger probability of the case that two users can share the same pattern. It seems that, the mobility patterns can be firstly clustered by user id before running clustering algorithm for each user. The disadvantage of this approach is wasting resources. Because, if there is some users share the same patterns (such as pattern of cases: 3 people in a family go to the park at every weekend), we have to store one pattern three times. Although this problem doesn't affect to the accuracy of the algorithm, we will waste the disk space. But it's not a big deal.

The improved algorithm is summarized in the figure 5.1:

The training data firstly split into many smaller parts by user id, and then the clustering algorithm is ran on each part to divide movement paths of each user into group. For each group, generate a model. This model will be a part of the global models. This procedure can be done in parallel:

Figure 5.1: Improved version of identifying and grouping similar movement paths



Algorithm 31 Parallel version of Improvement 1

function MAP_IDENTIFYING_AND_GROUPING_PATHS(*data*)
  for each movement paths m in *data* **do**
    $uid \leftarrow$ user id of m
    EMIT (uid, m)
  end for
end function

function REDUCE_IDENTIFYING_AND_GROUPING_PATHS(Key K=uid, Value Set $V = \{m_i\}$)
  EMIT (uid, K_Means_Parallel(V))
end function

The testing phase is the same as before: With a testing input, among the global models, we will find the closet one and use it to make prediction. If we can not find any acceptable closet model, we can consider the testing input belongs to a completely new cluster.

5.2.2.2 Improvement 2

This improvement's purpose is overcome the disadvantage of Improvement 1. According to the algorithm which is described in figure 5.1, among the global models, there are some duplications. These duplications lead to wasting resources. To solve this problem, we can do a merging phase after finish running clustering algorithms on all data parts. This phase can be done in parallel:

Instead of mering all models in a single machine, we try to identify the **possible** similar models by hashing each model into a value. The key/value pair output of the map phase is $< hashedValue, (model, uid) >$. The models have the same hash value will be sent to the same reducer. In reduce phase, the possible similar models will be merged in the local worker.

The most important factor of this improvement is the hash function. It has to minimize the number

Algorithm 32 Merging duplicated models

```
function MAP_MERGING(models)
    for each model m in models do
        uid ← user id of m
        EMIT (hashFunction(m), (m, uid)))
    end for
end function

function REDUCE_MERGING(Key K=hashedModel, Value Set V = {((m_i, uid_i)})
    EMIT (Merging_local(V))
end function
```

of cases: two models which are similar but have different hash value.

The simple hash function we used is: $H(model) = X$, where X is the location has the most frequent in this model.

5.2.2.3 Improvement 3

Until now, the closet model is picked by calculate the matching rate r of the testing input. Because the testing input is only a few pairs of location/time, it can lead to the "local" closet model, which will affect the accuracy result.

We can reduce this problem by calculate the **fitting_value** F instead of matching rate of each model by the following formula:

$F = \alpha\delta + \beta r$

where,

δ equal to 1 if the model is belong to the testing user

r is the matching rate (as before)

α is the weight of matching user

β is the weight of matching model

If $\beta = 0$ the *fitting value* is equal to the matching rate.

Bibliography

[1] Apache Hadoop. `http://en.wikipedia.org/wiki/Apache_Hadoop`.

[2] Cosine Similarity. `http://en.wikipedia.org/wiki/Cosine_similarity`.

[3] DBScan. `http://en.wikipedia.org/wiki/DBSCAN`.

[4] Euclidean distance. `http://en.wikipedia.org/wiki/Euclidean_distance`.

[5] Manhattan distance. `http://en.wikipedia.org/wiki/Taxicab_geometry`.

[6] Where Will You Go? Mobile Data Mining for Next Place Prediction. In Ladjel Bellatreche and Mukesh K. Mohania, editors, *Data Warehousing and Knowledge Discovery*, volume 8057 of *Lecture Notes in Computer Science*, chapter Where Will, pages 146–158. 15th International Conference, DaWaK 2013, Prague, Czech Republic, August 26-29, 2013. Proceedings, Berlin, Heidelberg, 2013.

[7] Y Ben-Haim and E Tom-Tov. A streaming parallel decision tree algorithm. *The Journal of Machine Learning Research*, 11:849–872, 2010.

[8] Gianluca Bontempi, Souhaib Ben Taieb, Yann-Aël Borgne, Souhaib Ben Taieb, and Le Borgne. Machine learning strategies for time series forecasting. In *Business Intelligence*, volume 138, pages 62–77. 2013.

[9] L Breiman, J H Friedman, R A Olshen, and C J Stone. *Classification and Regression Trees*, volume 19. 1984.

[10] Leo (University of California) Breiman. *Random forest*, volume 45. 1999.

[11] Anil Chaturvedi, Paul E. Green, and J. Douglas Caroll. K-modes clustering. *Journal of Classification*, 18:35–55, 2001.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. *Communications of the ACM*, 51:1–13, 2008.

[13] Nathan Eagle and Alex (Sandy) Pentland. Reality mining: sensing complex social systems. *Personal and Ubiquitous Computing*, 10(4):255–268, 11 2005.

[14] Vincent Etter, Mohamed Kafsi, and Ehsan Kazemi. Been there, done that: What your mobility traces reveal about your behavior. *the Procedings of Mobile Data Challenge by Nokia Workshop at the Tenth International Conference on Pervasive Computing*, 2012.

[15] Andras Garzo, Andras A. Benczur, Csaba Istvan Sidlo, Daniel Tahara, and Erik Francis Wyatt. Real-time streaming mobility analytics. In *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pages 697–702, 2013.

[16] Marta C González, César A Hidalgo, and Albert-László Barabási. Understanding individual human mobility patterns. *Nature*, 453(7196):779–82, 6 2008.

[17] J B MacQueen. Kmeans some methods for classification and analysis of multivariate observations. *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*, 1:281–297, 1967.

[18] Biswanath Panda, JS Herbach, Sugato Basu, and RJ Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *Proceedings of the VLDB . . .*, 2009.

[19] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[20] Luis Fernando Rainho Alves Torgo. *Inductive Learning of Tree-Based Regression Models*. PhD thesis, 1999.

[21] Le Hung Tran, Michele Catasta, Lucas Kelsey McDowell, and Karl Aberer. Next place prediction using mobile data. *Proceedings of the Mobile Data Challenge Workshop (MDC 2012)*, 2012.

[22] Jingjing Wang. Periodicity based next place prediction, 2012.

[23] Wei Yin, Yogesh Simmhan, and Viktor K. Prasanna. Scalable regression tree learning on hadoop using openplanet. *Proceedings of third international workshop on MapReduce and its Applications Date - MapReduce '12*, page 57, 2012.

[24] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5931 LNCS, pages 674–679, 2009.