RideMiner

Yanning Chen (aka. LightQuantum)

December 2020

1 Introduction

As the market of online ride-hailing rising, so does the demand for the analysis of order data collected by ride-hailing companies like Didi and Uber. They need to know the demand-supply pattern of their existing orders, like the temporal and spatio distribution of orders, and in which way they are generating revenue, like the total revenue and temporal distribution of it. They also want to know about the user experience of their service, which is indicated by travel time and fee. These information is critical for ride-hailing companies to have insight on their business, optimize their service and improve the user experience.

Here we propose RideMiner – an interactive data analysis and visualization application designed for ride-hailing order data. Just load your dataset (for example, the order data in Chengdu provided by KDD Cup 2020) into RideMiner, and you may easily acquire interactive and nice-printed graphs (including line charts and pie charts) on spatio-temporal demand pattern, distribution insights on user travel time, order fee and revenue on a per-hour basis. Besides, by utilizing intelliPredict function provided by RideMiner, you will get a data-backed eta for a specific travel plan, which can be integrated into mobile applications or web pages.

2 Implementation

2.1 Data Storage and Query

Dataset is first stored in SQLite before further query and analysis. By integrating with SQLite, query logic is seperated into SQL layer, which makes it easier to employ additional analysis in the future. Additionally, SQLite provides with a better interface for data exchange. To accelerate later use, users may choose to persist data into filesystem, so RideMiner only needs to load the dataset in the first run. Database table definitions are listed as below.

```
create table grids (
    grid_id     INT primary key,
    vertex0_lng REAL,
    vertex0_lat REAL,
    vertex1_lng REAL,
    vertex1_lat REAL,
    vertex2_lng REAL,
    vertex2_lat REAL,
    vertex3_lng REAL,
    vertex3_lat REAL
);

create table kv (
    key   TEXT primary key,
    value TEXT
) without rowid;

create table orders (
    order_id       TEXT primary key,
    departure_time INT,
    end_time       INT,
    orig_lng       REAL,
    orig_lat       REAL,
    dest_lng       REAL,
    dest_lat       REAL,
    fee            REAL
) without rowid;
```

2.2 Interactive Graphing

RideMiner offers various types of graphs to represent data distribution, including line charts, pie charts and histograms. Data analysis has never been easier thanks to interactive graphs. Interactive graphs are implemented by integrating QtChart components with rubberbands. Now users may dig deep into the graphs by scrolling mouse wheel or dragging on the graph directly. When users lose themself in the data, just double click the mouse, and they will be brought back home. Smooth and intuitive animations are added, giving users helpful visual guides when scaling graphs. Zooming effects are added to pie charts additionally, and labels will show only when slices are hovered, providing a better ux for small screen devices.

2.3 intelliPredict

Besides data visualization and graphing functions, RideMiner has a unique feature called intelliPredict. Given a travel plan, more specifically the departure time, origin coordinate and target coordinate, RideMiner is capable of giving an ETA.

To calculate the ETA, it's assumed that travel time is only related to departure time in a day, and the start and end coords of a travel plan. First, a metric function is defined as in Eq 1, where T stands for departure time, ed stands for euclidean distance, D stands for departure coords, and A stands for arrival coords.

$$dist\left(X,Y\right) = \left|T_X - T_Y\right| + 5000 \times \Big(ed\left(D_X, D_Y\right) + ed\left(A_X, A_Y\right)\Big) \tag{1}$$

Next, distances between travel plan and existing records are calculated using the metric function we just defined. Distances and travel time of records form a 2d euclidean space, which is the search space. To narrow the search space, orders with coords that has a manhattan distance greater than 0.1, or has a distance higher than 3000 are ruled out.

Then, mean shift algorithm is applied to points to cluster the points. The ETA is estimated by finding the largest cluster that has a central point with a distance lower than 600. Outliners are excluded during this process. There's a possibility that all points will be elided, which is expected because of the insufficience of order data. The accuracy of this model is questionable for the same reason. Still, it yields reasonable predictions.

2.4 Scheduling System

A task scheduling system inspired by the one used by Meteor[1] is implemented. All CPU-bound operations should be wrapped into tasks, which are threads running parallel with the main thread to avoid freezing UI. Tasks may emit UI update signals to notify users of current progress or possible failures. For example, *LoadDataset* and *ETAPrediction* are two typical tasks.

Like the one in Meteor, the task scheduler runs in a standalone thread, and maintains a task queue. The UI thread creates tasks according to user requests, and the scheduler will push the task into task queue. The UI thread now becomes idle, and logic continues in the task thread. Later, the task returns the result or an exception by emitting a signal or calling a callback function. Now, the logic continues in the UI thread. In other words, control is passed in the form of continuation. This is called CPS (continuation-passing style).

In traditional Qt, this logic flow switch is accomplished by emitting and catching signals. However, CPS is much harder to understand than direct style by programmers. To solve this problem, the scheduler of RideMiner has the ability to wrap tasks into Promise objects. It handles all the heavy lifting of message passing and proxy setting, and now you can write something like this:

```
void TaskCalculateSomething::run() {
    auto inputParam1 = args.at(0).toString();
    auto inputParam2 = args.at(1).toInt();
    if (!isValid()) {
        emit task_complete(TaskError("Oops"));
        return;
    }
    // ...
    emit task_complete(result);
}

void btnCalculate_clicked() {
    auto *t = TaskCalculateSomething({"blabla", 0};
    scheduler.promise(t).then([=](const TaskResult &r) {
        if (auto error = std::get_if<TaskError>(&r) {
            // clean up
```

[1]Meteor, an insightful, efficient and responsive app for analyzing Hangzhou metro, written by Alex Chi

```
        } else {
            // display the result
        };
    };
}
```

3 Results

Please refer to source code and demo video for details.

3.1 Tempro-spatio Distribution of Demands

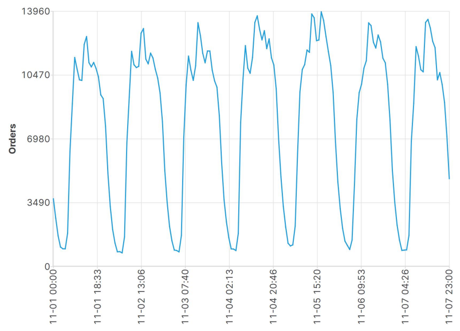


Figure 1: Spatio Distribution

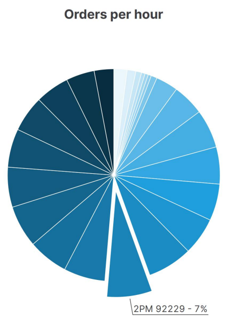


Figure 2: Tempro Distribution

3.2 Distribution of Travel Time, Order Fees and Revenue

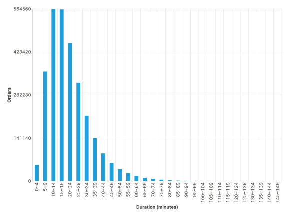


Figure 3: Travel Time

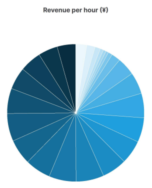


Figure 4: Revenue

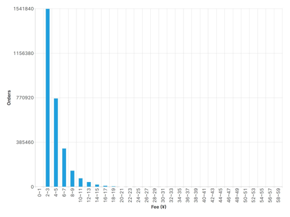
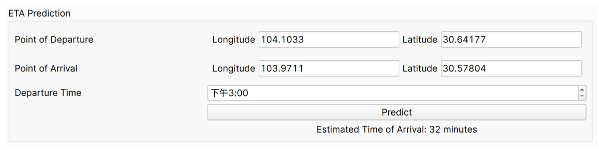


Figure 5: Fee

3.3 intelliPredict

RideMiner generally gives similar predictions as Google Map ETAs.


ETA Prediction

Point of Departure Longitude 104.1033 Latitude 30.64177

Point of Arrival Longitude 103.9711 Latitude 30.57804

Departure Time 下午3:00

Estimated Time of Arrival: 32 minutes

Figure 6: intelliPredict

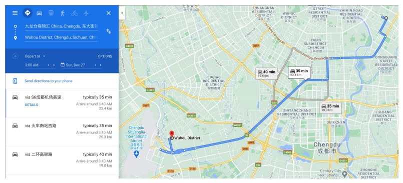


Figure 7: Google Maps ETA

4 Discussion on Possible Improvements

In current version of RideMiner, there's no map display function. Users must choose analysis grids manually, and input coordinates without any visual guide. This is largely due to the lack of map components in QtWidgets.

QtWidgets is announced to be 'finished' in 2012. No new components or features are implemented since then, and the Qt Project has shifted their attention to QtQuicks. QtQuicks is a brand new framework. It doesn't implement logic in CPP. Instead, there's a new DSL called QML. It's a hybrid of HTML, CSS and JS. User interface and logic is implemented with QML. Therefore, using QtQuicks doesn't meet the requirements of this assignment.

There's a component called Qt Location in QtQuicks, and the openstreetmap library introduced in the guideline is also based on it. Unfortunately, it doesn't have a QtWidgets counterpart. However, it's still possible to embed into QtWidgets project by using QQuickView. Qt provides convenient FFI between QML and CPP. Actually, CPP code may invoke QML(JS) methods, and data structures can be shared. Due to limited work time of this project, the techniques described above are not implemented.

5 Acknowledgement and License

This project is built with Qt Framework. Thanks for the work done by Alex Chi on Meteor. RideMiner's task scheduling design is heavily inspired by it. And also thank the authors of various third-party libraries. A complete list of third-party libraries can be found in README.md.