CIS 550 Project Final Report

Group 15

Hu Chenchen 11021747
Xiang Chen 66770481
Zhong Wen 54397279
Dhwani Kapoor 84152362

Abstract

The project is based on developing a prediction model for predicting the medals that a country will get in the next Olympic Games. Also, we aim to provide analysis on the previous Olympic results that might interest our users.

A nation's performance in previous Olympic games, past performance of athletes, economic factors and population are some of the important prediction variables that can decide the preparation of the nation for upcoming Olympic Games.

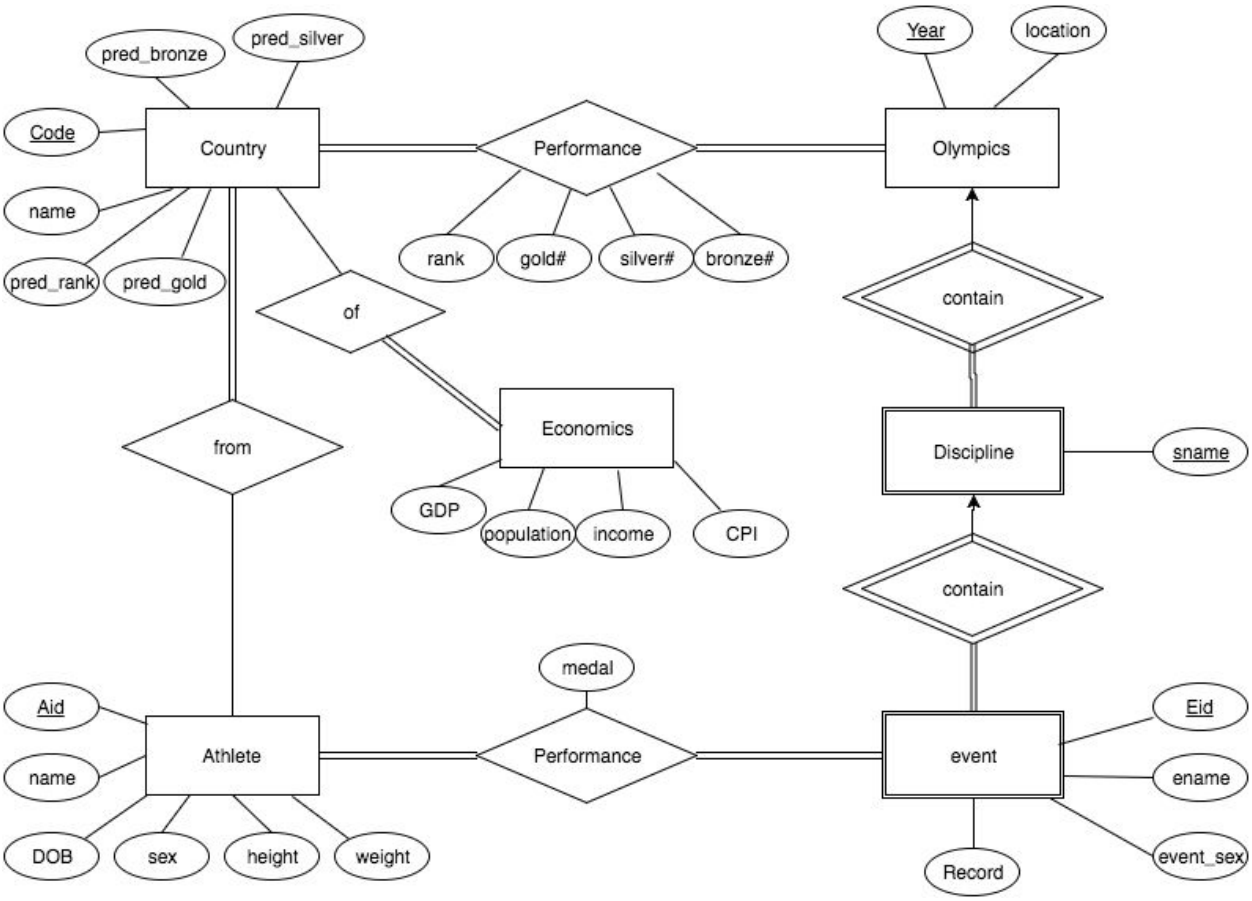
- Economic factors – People of affluent nations have affinity towards spending more time on recreation and sports than poorer nations
- Population – Separates larger nations of the world from many small nations, both in terms of number and genetic diversity
- historical data (2008 - 2016 data needed)
- Athletes' performance – An athlete's' performance is related with a variety of other entities (such as the country, the year, other competitions) which supplements the development of a diverse schema
- Host factor

Modules and Architecture

From the raw data sets (Olympics results 1896-2008), we have designed many possible schemas, after took these principles into consideration:

1. Efficient to implement simple queries
2. Easy to read for development and test
3. Clear enough to find the structure of our prediction model
4. Persistent to avoid update, delete and insert anomalies

We get this final ER diagram to use:



Also, because the profiles can be very different among athletes, if we import all of them into relational database, it will definitely use up space unneces unnecessarily. Therefore, we established AWS DynamoDB to store profiles of athletes, here is a basic description:

```
Table: Athlete
{
"Athlete":[
   {
"_id":1,
"name":{
      "first": "Yannick",
      "last": "AGNEL"
},
"DOB":{
      "year": 1992,
      "month": 6,
      "day": 9
},
"Sex": "Male",
"Height": 202,
"Weigth": 90,
"Nationality": "FRA",
"Rank":[
      {
            "Event":"200 Metres Freestyle Male LC",
            "Year": 2011,
            "Rank": 5
      },
      {
            "Event": "400 Metres Freestyle Male LC",
            "Year": 2011,
            "Rank": 3
      }
],
"Social" :{
      "Facebook":"https://www.facebook.com/Yannick-AGNEL-215498776943/",
      "Twitter": "https://twitter.com/YannickAgnel",
      "Fans on Facebook": 132163,
      "Followers on Twitter": 149628
},
"Record":
[
   {
      "Event": "50m Freestyle",
      "Course": "50m",
      "Time": 22.87,
      "Points": 764,
      "Date": "3 Nov 2015",
      "City": "Doha (QAT)",
      "Meet": "FINA: World Cup No 7 - 2015 Series"
   },
   {
      "Event": "50m Freestyle",
      "Course": "25m",
      "Time": 22.06,
      "Points": 774,
      "Date": "11 Aug 2013",
      "City": "Berlin (GER)",
      "Meet": "FINA: World Cup No 2 - 2013 Series"
   }
]
}
}
```


When come to schema refinement, firstly we found some possible dependencies to avoid, such as :

- aid (athletes) -> code (country to represent)
- year -> location (host country)
- (year, code) -> medal

  ...

So, in the end, we broke large tables into several smaller ones and created new keys like:
aid (athlete id), eid (event id), dname (discipline name) to make sure every table is BCNF and more convenient for us to retrieve information.
We used Node.js for the backend of application, and set up the architecture of the application mainly as follows, below are details of the main files:

- Server application
  - Sets up server environment and server, initializes Express and view engine EJS, sets up paths for routing
  - app.js
- Database connection
  - Provides methods to access both OracleDB and DynamoDB. There are multiple queries to use, in order to display statistical results as well as satisfy user's various requests.
  - db.js
- Routing
  - The paths of each route are set up in the server application, while the functionality for each route is implemented in the routing component of application. The methods handle request from the frontend and perform the relevant queries to the database and then send a response.
  - index.js , athletes.js
- Views
  - The HTML pages that will be displayed to the user are compiled by the EJS engine, and the .ejs files are templates that the engine will compile the page with, given the relevant information from the routing.
  - views/*.ejs
- Public
  - Stores style definition files such as .css and a d3 geomap component implemented in our views.
  - public/*

Data instance use

Since one of the motivation of our project is to predict the performance of each country in 2020, we have used the economic data (population, GDP, CPI and revenue) from World bank (in csv) and the past performance of each country (parsed from web) to make a prediction model in Python. We present the result of the prediction in the format of a world map. In addition, for the NoSQL part, we have provided the information of the top 10 swimmer to our user. For example, user can enter the swimmers' facebook and twitter from our website. User can also

get the physical indice of those swimmer from our website. What's more, user can get some interesting information about countries and Olympics including the total medals a country has got in past Olympics, the total medals a country's male/female athletes have got, the best record of each discipline, etc.

Complementary data sources:
1. Prediction data (in HTML): Olympics records after 2008
2. Country page (in CSV): Macroeconomic features
3. Athletes page: Top ten swimmers profile (in PDF)
    Swim rankings: swimrankings
    Bibliography: http://www.sports-reference.com/olympics/athletes/
    Social networks and fan pages:
    http://fanpagelist.com/category/athletes/olympics/view/list/sort/fans/page1

Data cleaning and import mechanism

We adopt ipython notebook to do data cleaning. Since the data come from multiple sources, we firstly check the consistency of country code as well as athletes' names across all databases. Then we delete the entries with missing data. All data used in SQL part are written into csv tables. As for athlete's profile data, since the attributes of different athletes' profiles could be quite different, we decide to use NoSQL database to implement this module and convert these data into json. One minor problem is that after we test our query result in HTML, some french character couldn't be displayed properly, even though we add $< meta\ charset = "utf-8" >$ in the header. In the end we replace those characters with English in the database and solve the problem.

The NoSQL import:
Since we only have one table 'Athlete' to import in the Dynamodb, there's no foreign constraints to worry about. Dynamodb supports table creation, primary key setting and data importing (in json) from the console. Since our NoSQL queries are all based on athlete_id, we set the athlete_id as the primary key and import the json file.

The SQL import:
We used sqldeveloper to create tables, import data and check those restrictions including primary key, foreign key, or other integrity constraints. Some of these constraints were not perfect, we either changed the schema of our table or made tiny modification on the constraints that we designed.

Algorithms and communication protocols

The primary goal of our project is to dig into the Olympics datasets and predict the medal distribution and rank for major countries in 2020 Olympics. As described in the Abstract, we

focus on four dimensions to do feature selection, namely macroeconomic data, athlete's performance on other international games, historical data and host factor. For example, the feature pool of the gold medal number prediction model includes: 2012_CPI, 2012_GDP, 2012_Population, 2012_revenue, 2012_revenue/population, 2008_CPI, 2008_GDP, 2008_Population, 2008_revenue ,2008_revenue/population, 2008_gold/TOTAL, 2012_gold/TOTAL, HOST_indicator, 2008_gold, 2012_gold. Since the economics data provided by the World Bank do not include revenue information after 2015, we have to drop the 2012_revenue factor during the training process.

In preprocessing phase, to avoid the side effects of the differences between feature scales and variances, we normalize all economics data with 0 mean and unit variance. Then, the dataset (around 60 countries) are split into training and test sets with the ratio 4:1. Considering that many countries got 0 medals in the previous Olympics and the metal matrix may be sparse, we adopt Lasso regression with alpha = 0.3. As comparison, we also adopt least square error regression (pandas.stats.api) model to cross check the correctness of prediction model.

In feature engineering phase, we evaluate the importance of features by the p-value of their coefficients. Based on the backward selection mechanism, we delete those features whose coefficients are close to zero step by step. The final model is as follows:

$$y\,(2016\ gold) = 2008\_CPI + 2008\_GDP + 2008\_POP + 2008\_revenue + 2008\ gold + 2012\ gold$$

The adjusted R square is 0.9656, and the mean squared error on test set is 6.49, which means the mean deviation from true value is around 2 medals. The result could be refined by adding more countries in the training set.

We adopt similar methods to predict the number of silver and bronze medals. As for the prediction on the rank, we come up with two methods. The first one is to follow the idea in medal prediction; while the second one is to sort all countries by their predicted numbers of gold, silver and bronze medals in 2020 and generate the rank. The second method comes naturally with the definition of ranks, but strongly depends on the performance of previous prediction models, therefore we finally adopt the first one. For ranking data prediction, people will value most the results of top ranks. Therefore we assign each records (country) with different weights of error

$$weight = (max(X['2012\_rank']) - X['2012\_rank']) ** 2 / max(X['2012\_rank'])$$

And adopt weighted least squares model. After similar process of feature selection, the adjusted R square of final model is 0.833.

Use cases

This app is geared towards people interested in analyzing past Olympic results and predictions of upcoming Olympic games.

- A user can see highlights of the previous Olympic games (2016) on the homepage.

- When the user clicks on ‘Country’ tab, a world map is displayed which shows the prediction of ‘Tokyo 2020 Olympics’ medal count for each country when the user hovers the cursor over the map. Further, the user can compare performance of different countries and male v/s female medal counts.

- The user can analyze performance of athletes by clicking on ‘Athlete’ tab. Top 10 athletes in the swimming category are shown with links detailed information. When the user clicks on ‘details’ link of a particular athlete, athlete’s world ranking, past records and profile are displayed with links to Twitter and Facebook profiles. Further, the user can click on athletes list and compare performance of athletes.

- The user can also analyze the performance of countries with respect to different disciplines by clicking on the ‘Disciplines’ tab. Also, top 3 all time world records for different events can be viewed from the left navigation menu.

Optimization techniques employed (indexing, optimization)

We noticed that since our db has been decomposed into several smaller tables in order to preserve BCNF, sometimes a single query requires many joins between different tables, thus we implemented optimization techniques in three ways: creating views, building index, refining time consuming queries.

Creating views

In order to integrate both the athletes’ performances and details of the country they represent, we have created such view to accelerate the query time:

```
CREATE VIEW TPERFORMANCE
AS
SELECT c.code, A.aid, hE.eid, hD.dname, O.year
FROM Country C INNER JOIN Represents R ON C.code = R.code
               INNER JOIN Athletes A ON R.aid = A.aid
               INNER JOIN PerformanceOfAthletes POA ON A.aid = POA.aid
               INNER JOIN hasEvents hE ON POA.eid = hE.eid
               INNER JOIN hasDiscipline hD ON hE.dname = hD.dname AND hE.year = hD.year
               INNER JOIN Olympics O ON hD.year = O.year;
```

It is helpful, for example, one of our query is to estimate the performance of women athletes and men athletes on 2008 Beijing, without the temporary view table, the elapsed time would be 0.114 s, after using the view that we created, the elapsed time was reduced to 0.025 s.

Building index

At first, we did come up with index suggestions on our database like we can build index on (code, medal), so that we can retrieve the performance of countries more efficiently. However, the difference was not so obvious, in the test query we used 0.064 s without index and used 0.018 s on index (code, medal) to fetch all the rows. On the other hand, we have broken large

tables into several smaller ones, so if we want to retrieve partial information of the whole data, we can directly commit selection on that specific table.

Refining queries

There is one query that aims at getting the country which has accumulative medals less than Michael Phelps's medals, before refinement we designed the query as:

```
SELECT *
FROM
(SELECT P.CODE, C.NAME, SUM(P.NUM_OF_GOLD) AS ALLYEARGOLD
FROM PERFORMANCEOFCOUNTRIES P, COUNTRY C
WHERE P.CODE = C.CODE
GROUP BY P.CODE, C.NAME)
WHERE ALLYEARGOLD < (
SELECT COUNT(*)
FROM PERFORMANCEOFATHLETES P INNER JOIN ATHLETES A ON P.AID = A.AID
WHERE A.NAME = 'PHELPS, Michael' AND A.AID = P.AID AND P.MEDAL = 'Gold'
);
```

There is a nest in the query, however, it is easy to find that Phelps's medals number is a constant and there is no need to compare each row of countries' performance in the nest.
So after refinement, we can get the query as:

```
WITH PHELPS AS(
SELECT COUNT(*) AS PCOUNT
FROM PERFORMANCEOFATHLETES P, ATHLETES A
WHERE A.NAME = 'PHELPS, Michael' AND A.AID = P.AID AND P.MEDAL = 'Gold'),
COUNTRYCOUNT AS(
SELECT CODE, SUM(NUM_OF_GOLD) AS ALLYEARGOLD
FROM PERFORMANCEOFCOUNTRIES
GROUP BY CODE)
SELECT C.NAME AS Country, CC.ALLYEARGOLD AS totalgold
FROM COUNTRYCOUNT CC, PHELPS Ph, COUNTRY C
WHERE CC.ALLYEARGOLD < Ph.PCOUNT AND C.CODE = CC.CODE
ORDER BY CC.ALLYEARGOLD DESC
```

The time consumption is that the new query is slightly 0.01 s faster than the old one.

Technical specifications

1. Backend: we use Node.js as the server to connect databases, and express.js as the application framework.
   In particular, we add the following modules within node.js:
   Async, Awssdk, Bodyparser, ejs-locals, nib, stylus, Oracledb, consolidate, google, open
2. Frontend: we tried to adopt Angular.js as a handy tool to implement dynamic websites; however we got stuck in the functions to pass angular variables to express. Finally we adopt jQuery, ejs and boostrap for frontend development.

3. Database: AWS
   Relational Database Service (SQL)
   DynamoDB (NoSQL)
4. Data cleaning and prediction: we use ipython notebook as the analysis tool and the following modules are imported: pandas, numpy, random, pandas.stats.api, sklearn.
5. Data visualization: we use d3.js (geomap) template to display the prediction results of countries.
6. Other APIs; Facebook login and sharing, Google search, Youtube

Special Features to Highlight

First of all, we have used many APIs:

1. Google API. User can search in our website and the first result of the google search which is related to Olympics will pop up.
2. Facebook API. There's a share button in our website to let user log in their facebook and share our website with others.
3. Youtube API. User can watch the official video of 2016 Rio Olympic Games in our website.

In addition, we have used d3.js to make a world map. User can hover over the certain part of the map to get the prediction data of that country.
Also a minor but tricky technical feature is that in one web page, user can get the results of several queries.

Division of work between group members

1. Data extraction and preprocessing
   2008 - 2016 Olympics data: Xiang Chen
   Economics data: Zhong Wen
   Athletes profile: Hu Chenchen
   Prediction model: Hu Chenchen

2. Schema design, normalization and queries: All
   Oracle: Xiang Chen, Zhong Wen, Dhwani
   Dynamodb: Hu Chenchen, Xiang Chen

3. Populating the database (data cleaning, formatting and entity resolution)

4. Web design
   Backend (DB connection, queries): Xiang Chen, Hu Chenchen
   Frontend (Boostrap, Web frame): Zhong Wen
   Google Search and Facebook login in: Xiang Chen

Modules: home page: Xiang Chen
Country: Zhong Wen
Athletes: Hu Chenchen
Discipline: Dhwani

Technical challenges and how they were overcome

Initially we wanted to query the athlete's profile by their first name and last name as follows:

```
var params = {
    TableName : "Athlete",
    FilterExpression: "#fn = :first AND #ln = :last",
    ExpressionAttributeNames: {
        "#fn":"name.first",
        "#ln":"name.last"
    },
    ExpressionAttributeValues: {
        ":first":key.first,
        ":last": key.last
    }
};
```

However it always returned null. We followed the instruction on official documents to query on sub attributes while could not solve it. In the end we decide to do all queries by the primary key athlete_id.

Potential future extensions

We implement SQL and NoSQL parts in separate modules, which could not provide our users a comprehensive perspective of Olympics dataset. future research on how to communicate between mysql and mongodb could be conducted.

Initially We hoped to bind the searching box and result within one page by using angular.js and spent several days on that. However we ran out of time and in the end rendered the callback data in a new page. In future work we'd like to explore more about angular.