Chess Data Analysis

20k-0388 Muhammad Ahmed
20k-0386 Syed Sufyan Imran
20k-0428 Nausherwan Baig

January 17, 2022

Contents

0.1 Introduction

Out chosen data set contains 6.25 Million chess games played on lichess.org. The data set gives insight of players statistics and its performance in each game. We have extracted the necessary data from each column and derived a reasonable conclusion on various aspects of a chess game. Opening of a chess game is the most important part of the game, and the type of opening can really help determine which direction the game will lead to. Our program analyses which players with certain ELO* prefer to use that opening, and the probability of winning while using the opening. It also considered the specific events which is used as every event duration differs from one another. While extracting our data, we have assumed that Opening referred to is always for the player playing as White.

*The ELO rating system measures the relative strength of a player in some games, such as chess, compared to other players.

0.2 Background(Research and Evaluation)

We started are research from different websites to get the suitable data set up to 1 GB for the analysis and the conclusion to be driven by our project. We asked different teachers and senior students for their opinion and then tried to incorporate all the suggestions and research in our project using the data structures we learned during the course and all the relevant materials and some out of the box things to make an effective, efficient program which can give statistical and analytical calculations through which a conclusion can be drawn

0.3 Problem Analysis

We were provided with a Large Data set and the goal of this project was to extract a valuable and reasonable meaning of the data.With small-scale data set, subtle population patterns and heterogeneity's are not possible, with large data set

0.4 Implementation

Program begins by firstly reading CSV file of 1.6GB which was reduced from 4GB using Scraped data using Python using **Panda's Library**. Once our file is created, CSV file is read using 'vincentlaucsb csv-parser'. It reads each data from the column and saves our data in our object 'rowobjects' vector of class RowData. After saving our data, We map all our User data using data structure unordered_map as shown in our variable declaration; `unordered_map unordered_map<string, UserData> userMapping`, which considers user-ID *which is represented by White and Black column* as our Key, and the value which is mapped are its statistics such as its wins, loss, draw, most frequent event he attended as well as the most frequent opening he played in all his games.
Reason for using **unordered_map** data structure is because, it is built on the implementation of hash Tables, making it very easy to map Key(user-ID) to its statistics(wins,loss..etc). It also provides Quick searching time and insertion where Average case is O(1) and Worst is O(n) where n is the number of elements.

Our program allows user to search for any player and to make searching more efficient, we have implemented the **Trie** data structure, Trie allows retrieval of information in the most effective and efficient way, if user inputs a username which doesn't match in our data. User is given suggested usernames he might have intended to search. Using Trie, search complexities can be brought to optimal limit (key length).Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. ad these branches are traversed to output all possible suggestions.We need to mark the last node of every key(word character) as end of word node. `bool isWord`.
Insertion code of Trie data structure.

```
void insertion(string word) {
        Node* temp = root;
        for (int i = 0; i < word.length(); i++) {
            if (temp->links.find(word[i]) == temp->links.end()) {
                temp->links[word[i]] = new Node(word[i]);
            }
            temp = temp->links[word[i]];

            if (i == word.length() - 1) {
                temp->isWord = true;
            }}}
```

NearSearches(which return suggestions) code of Trie data structure.

```
void nearSearches(string word) {
        Node* temp = root;
        int i = 0;

        //bool isFound;

        while (i < word.length()) {
            if (temp->links.find(word[i]) == temp->links.end()) {
                cout << "No Result Found!: " << endl;
                return;
            }
            else
                temp = temp->links[word[i]];

            i++;
        }
        unordered_map<char, Node*>::iterator it = temp->links.begin();
        printing(it->second, word + it->first);
    }
```

In one of our Program module, we are determining the number of times a type of Opening is used with a set ELO Range. As we know ELO tells us about the players Rank, hence knowing the frequency of the type of opening used in a given ELO Range can give us an insight on the type of strategy players with similar ELO prefer over others. To display the most frequent opening used in each set of ELO Range, we use the implementation of `priority_queue` which arranges each opening according to its usage, where the most frequent opening lies on the top of the queue. Overloaded comparator function is added, to make sure it works with our objects used.

```
bool operator < (const Openings& o1, const Openings& o2) {
    return o1.frequency < o2.frequency;
}
```

Our next data analysis include extracting data upsets, how that is determined is by looking at Players Rating difference* and the result of the current match for that player. Players rating difference is determined by `(it self's Rating - Opponents Rating)`. So if a players rating is in `negative` it concludes the fact that the opponent is more skill full that itself, hence weaker player winning that match is concluded as an upset. Using **Merge Sort**, we have arranged the data where players with the most Rating difference are on the top, and with the result as a *Win* strongly concluding that this specific record is our greatest upset in our data set.

*The same rating difference implies the same chance of winning. A player rated 2400 playing against a player rated 2200 has the same chance of winning as a 1400 against a 1200. The rating difference is 200 points in both cases. The most widely used rating system is known as the ELO system, through which difference is calculated

0.4.1 Logistic Regression

Logistic regression is a machine learning technique to model conditions in which the result is binary. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. In our data set, we identified that ELO's influenced the outcome of the match so we trained our model on these two parameters.

Methods and details

The hypothesis of logistic regression tends to limit the cost function between 0 and 1 so in order to map the predicted values to probabilities we used the **Sigmoid function.**

Sigmoid Function

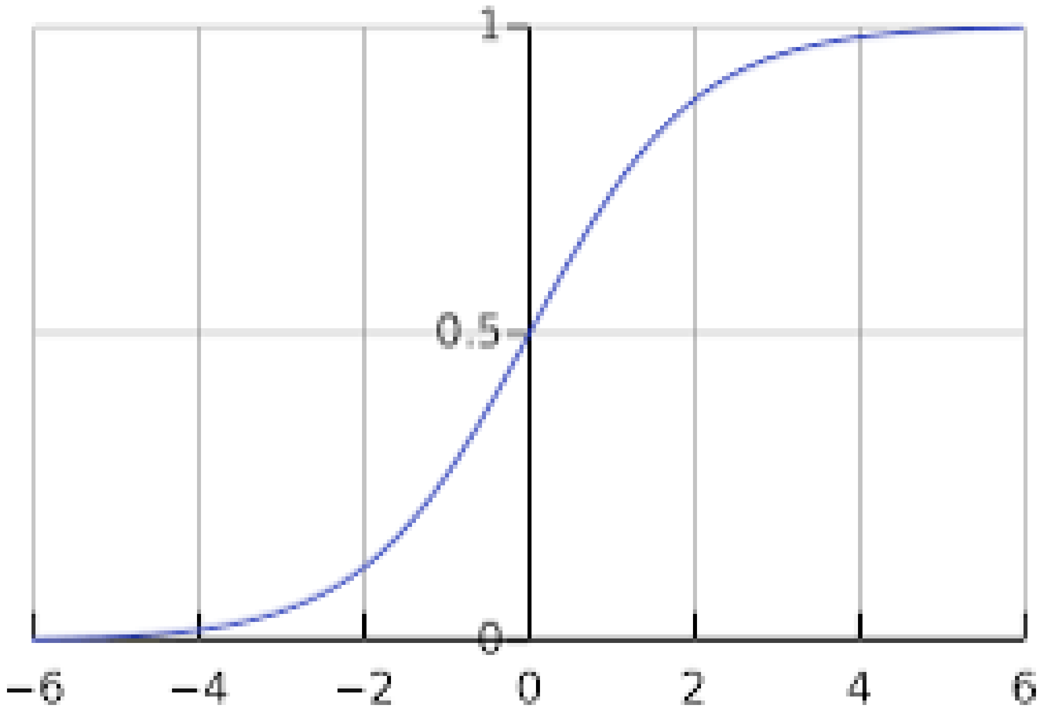
`Function:`

$$f(x) = 1/1 + e^{-}(x)$$

`Code Representation:`

```
double sigmoid(double z) {
   return (1 / (1 + exp(-1 * z)));
}
```

`Graph`



Cost Function

Cost functions in most machine learning problems measure the performance of the machine learning model. This measures how well your model is fitting the data by measuring the difference between the actual value and the predicted value. For this specific problem, we wanted to minimize the cost function so that the difference between the actual values and the predicted values is minimized.

`Function:`

$$J(\theta) = -1/m(\sum[y^(i)log(h\theta(x(i)) + (1 - y^(i))log(1 - h\theta(x(i))]$$

`Code Representation:`

```
double costFunction(vector<double>& y, vector<double>& y_hat) {
    double loss = 0;
    int m = y.size();
    for (int i = 0; i < y.size(); i++) {
        loss = loss + ((y[i] * log(y_hat[i])) + ((1 - y[i]) * log(1 - y_hat[i])));
    }
    loss = loss * -1;
    loss = loss / m;
    return loss;
}
```

Gradient Descent

To find the minimum cost function we have to update the parameters and check whether this affects the cost functions, and for this process we have used gradient descent. A good analogy that I found on the web was imagine ourselves at the top of a mountain valley and left stranded and blindfolded, our objective is to reach the bottom of the hill. Feeling the slope of the terrain around you is what everyone would do. Well, this action is analogous to calculating the gradient descent, and taking a step is analogous to one iteration of the update to the parameters.

`Code Representation:`

```
vector<double> weights = { 0.2, 0.5 };
double bias = 0.5;
vector<double> dw(2, 0);
double db;

void gradients(vector<vector<int>>& X, vector<double>& y, vector<double>& y_hat) {
    int m = X.size();
    // y_hat - y
    vector<int> diffYHat(y.size());
    for (int i = 0; i < y.size(); i++) {
        diffYHat[i] = y_hat[i] - y[i];
    }

    // X.T * diff
    for (int j = 0; j < X[0].size(); j++) {
        for (int i = 0; i < diffYHat.size(); i++) {
            dw[j] = dw[j] + X[i][j] * diffYHat[i];
        }
        dw[j] = dw[j] / m;
    }

    for (int i = 0; i < m; i++) {
        db = db + diffYHat[i];
    }
    db = db / m;
}
```

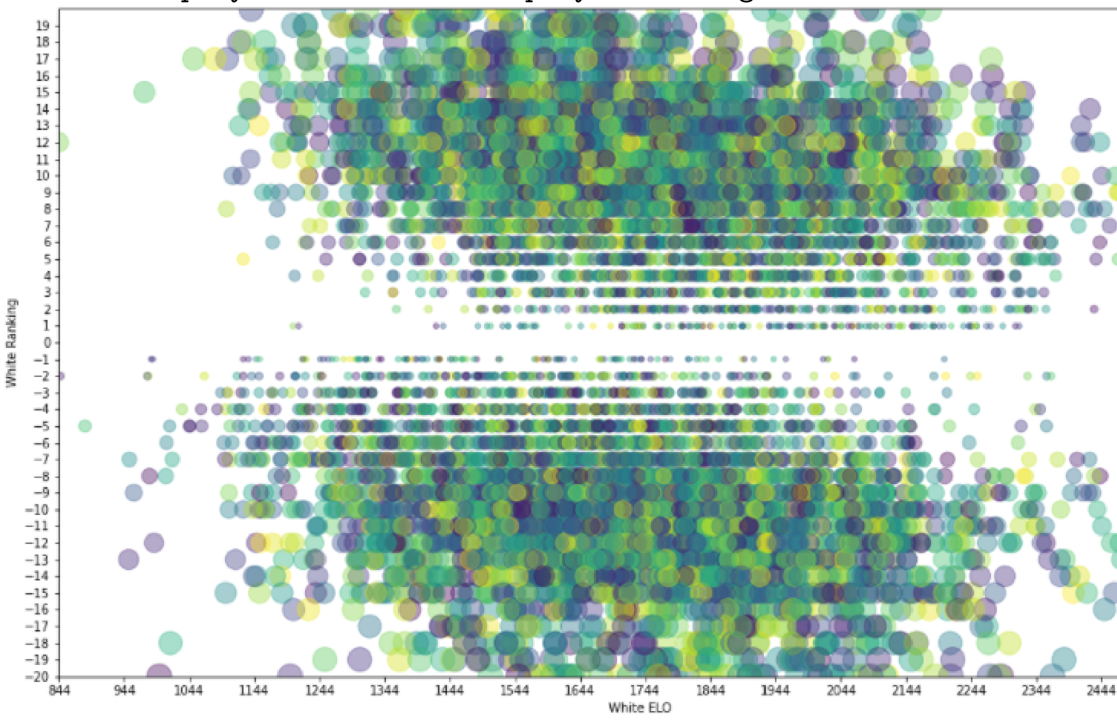
Training our program

Code Representation:

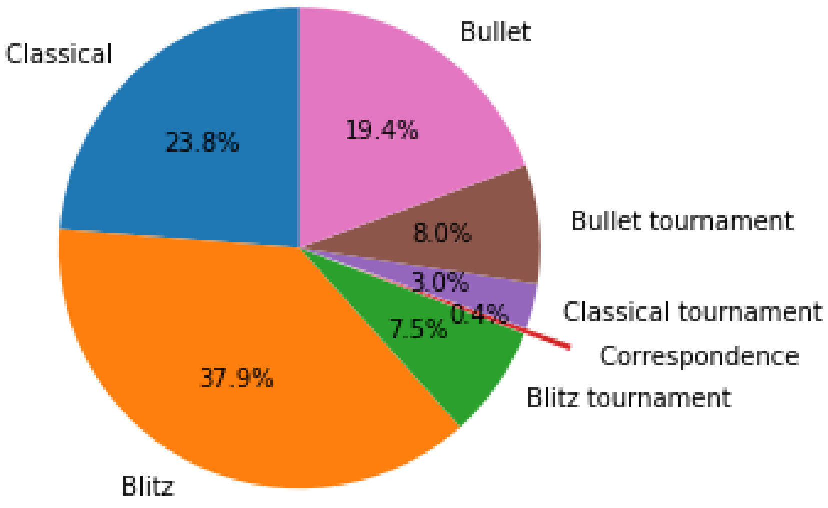
```
void train(vector<vector<int>>& X, vector<double>& y, int epochs, double lr) {
    int m = X.size();
    int n = X[0].size();

    vector<double> dummyWeights = { 0.9, 0.5 };
    double dummyB = 0;
    vector<double> losses;

    for (int i = 0; i < epochs; i++) {
        vector<double> y_hat(y.size());
        for (int j = 0; j < y.size(); j++) {
            y_hat[j] = sigmoid(dotProduct(X[j], dummyWeights) + dummyB);
        }
        gradients(X, y, y_hat);
        for (int k = 0; k < dummyWeights.size(); k++) {
            dummyWeights[k] = dummyWeights[k] - lr * dw[k];
        }
        dummyB = dummyB - lr * db;
        vector<double> dummyHat(y.size());
        for (int k = 0; k < y.size(); k++) {
            dummyHat[k] = sigmoid(dotProduct(X[k], dummyWeights) + dummyB);
        }
        double l = costFunction(y, dummyHat);
        losses.push_back(l);
        //cout << "I :  " << i << endl;
    }
    cout << "weights: ";
    for (int i = 0; i < dummyWeights.size(); i++) {
        cout << dummyWeights[i] << endl;
    }
}
}
```

0.5 Results and Conclusions

Relationship b/w White players ELO VS white players Rating



Difference.



Frequent events.