Data Mining (CS65a)
Course Project:
A Game of Cricket

Group 28

(Amitangshu Dasgupta, 161082

Akash Mondal, 171013

Shantanu Ghosh, 171137)

INTRODUCTION

India is a country obsessed with cricket. And we are no exceptions to that. So when the time came for us to decide what should we do for our data mining project, we thought instead of analysing market reports or financial historic data, why not do something more interesting and fun, for example analysing cricket. But there was one problem. That is a game of cricket is a lot more than mathematics as it involves the cricketers' current form, the pitch conditions, the mental temperament of the playing XI and luck. In other words, there's too much of noise to predict outcomes of entire matches let alone predicting tournament champions and title holders.

Therefore, we decided to build a game instead of predicting outcomes. The game can be played by two people who would have to select their respective playing XI (on paper) and then decide the batting order which the batting team would follow as well as the bowlers for every over. The algorithm would ask the player to choose three cricketers for an over (6 balls) viz. the batsman on strike, the batsman off strike and the bowler and would return the outcomes of each delivery in the over.

We considered seven of the biggest t20 tournaments across the globe for our data namely

1. Indian Premier League (India)
2. Big Bash (Australia)
3. NatWest T20 Blast (England and Wales)
4. Ram Slam T20 League (South Africa)
5. Pakistan Super League (Pakistan)
6. Bangladesh Premier League (Bangladesh)
7. Caribbean Premier League (West Indies)

We pooled the data of outcomes of every delivery ever bowled from each of these leagues in a huge dataset of size around half a million and built our game on top of it.

PROBLEM DISCUSSION

The function `overPrediction()` shall take as input the two batsmen on pitch and the bowler and will output the outcomes of each delivery in an over or until someone gets out before the over ends. In that case, the player shall have to input the new batsman according to the batting order in place of the batsman who got out and run the function again and it will output the outcomes of the deliveries remaining in that over.

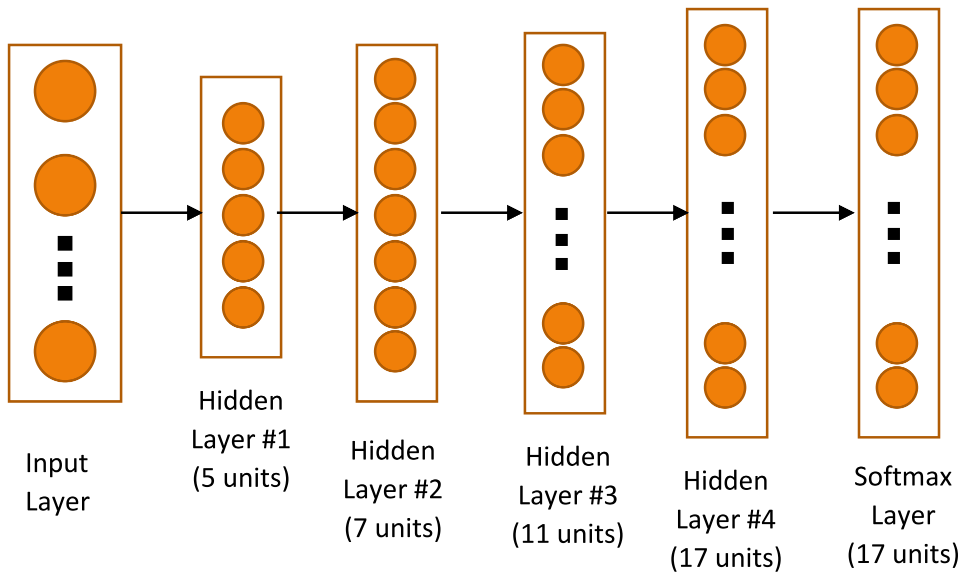
Though this sounds simple enough, the outcomes are based on the outcomes of all instances in the past when that particular batsman on strike had faced the bowler. For this kind of training we have used a neural network since we had an abundance of data (nearly half a million deliveries) and then a naïve bayes classifier for further classification.

For training the model, we used a 4 layer deep **neural network** in the first stage with softmax classifier. The first stage would classify based on the batsman and bowler whether the outcome of the delivery was a run between the wickets or a boundary or a sixer or whether it was caught or bold or stumping etc.

For the second stage, we used a **naïve bayes classifier**. This stage would further add details by classifying the running between the wickets into 1run, 2runs or more or by classifying byes or leg byes into the number of runs etc.

TECHNICAL DETAILS

We used a 4 layer neural network with respectively 5, 7, 11, 17 units in the hidden layers. The input layer contains 12 units (after encoding) while there are 17 classes for each delivery to get classified into.


Input Layer
Hidden Layer #1 (5 units)
Hidden Layer #2 (7 units)
Hidden Layer #3 (11 units)
Hidden Layer #4 (17 units)
Softmax Layer (17 units)

The choice of the architecture was an empirical one. The reason we decided to go with this architecture is that after trying out different architectures this particular architecture minimized the cost the most.

DIGGING THE PAST

DATA DESCRIPTION

- We pooled the data into a structured form using spreadsheets so as to rule out any discrepancies in names of cricketers by converting them into one name. For e.g 'Sourav Ganguly' in some datasets while 'SC Ganguly' in some others were all converted to 'SC Ganguly'.
- The pooled master data contained the following columns:
  `sl_no`: serial number of each delivery in the dataset
  `innings`: first or second innings
  `ball_id`: unique identification number of each delivery in a match (increases inspite of the delivery being a wide ball or a no ball)
  `ball_no`: which delivery of the over being bowled (doesn't increase in case of a wide or no ball
  `batting_team`: batting team name
  `batsman`: batsman on strike
  `bowler`: bowler delivering the ball
  `non_striker`: batsman standing in the non striker end
  `play_type`: outcome of the delivery (run, four, six, out, bye, wide, no ball)
  `score_value`: runs scored in the delivery
  `wicket_who`: batsman who got out
  `wicket_how`: type of out (caught, bold, lbw, stumped, run out)
  `fielder`: fielder name in case of caught out or run out
  `is_ke`eper: whether the keeper was the wicket keeper
- Initially the pooled dataset had many columns which were not relevant to our analysis. We therefore removed irrelevant columns and retained a few. We further condensed information spread across different columns into one column wherever possible. Written in details in data preprocessing below.

DATA PREPROCESSING

- A new column `score` was added to the master dataset. This was done for ranking purpose of the batsmen and the bowlers.
    - The column was filled with the score value in case the batsman scored those runs (running between the wickets, four and six) and kept zero in case the runs added were extra runs (no ball, wide ball, bye and leg bye).
    - In case of a wicket and adjustment index of -24.26 was filled in the column entries. We arrived at the adjustment index dividing the total runs scored by all batsmen (no extras) by the total number of wickets in the dataset. In a naïve manner, a wicket costs 24.26 of the runs made by that batsman.

  Using this column we can prepare a ranking table of the batsmen and bowlers. The adjustment factor was necessary to differentiate between two batsmen with same runs but one is more inconsistent than the other and often loses his wicket earlier than the other. It also helps in differentiating between two equally economical bowlers but one with more wickets than the other.

- We created two separate tables for batsmen and bowler ratings and rankings.
    - Ratings for each batsman is calculated the total of the score column entries against his name in the entire dataset while for bowlers it was the negative of the total of the score column entries against their names.
      E.g A batsman with a total of 2500 runs till date who have been dismissed 100 times till date would receive a rating of 2500-100*24.26=74 points while a bowler with 100 dismissals till date who has given away 2500 runs would receive a rating of -74 points.
    - Ranks were provided for the batsmen and bowlers according to their ratings. More the rating, higher the ranks.
- Encoding the inputs. The encoding was done in the fallowing manner:
    - Clearly, one-hot encoding isn't practical in this situation since we have around 2000 different batsmen and around 1500 different bowlers in our dataset. One-hot encoding would mean that the input vector for our neural network would be of dimension around 3500.
    - Therefore in order to reduce the number of vectors required to uniquely represent each cricketer, we used a semi one-hot encoding approach.
        - The cricketers were sorted according to their ranks in their respective fields (batting or bowling).
        - Five batsmen in the increasing order of their rank were put into one slab. The next five were put into the next slab and so on.

    - We had 5 cricketers in each slab. Therefore we needed 5 vectors in each slab to represent a batsman uniquely.

E.g. According to our data the top 10 ranked batsmen were:

Batsman	Rating	Rank
CH Gayle	6516.0692	1
DA Warner	3535.7917	2
V Kohli	3391.2271	3
SE Marsh	3244.245	4
SK Raina	3096.2144	5
AB de Villiers	2969.2617	6
M Klinger	2706.5505	7
MS Dhoni	2500.3135	8
BJ Hodge	2446.6187	9
SR Watson	2269.7265	10

Therefore the men from Gayle to Raina are put in slab 1 and those from de Villiers to Watson are put in slab 2 and so on. Now inside slab 1, we use one-hot encoding to get the following

Batsman	Slab	One-hot encoding				
CH Gayle	1	1	0	0	0	0
DA Warner	1	0	1	0	0	0
V Kohli	1	0	0	1	0	0
SE Marsh	1	0	0	0	1	0
SK Raina	1	0	0	0	0	1
AB de Villiers	2	1	0	0	0	0
M Klinger	2	0	1	0	0	0
MS Dhoni	2	0	0	1	0	0
BJ Hodge	2	0	0	0	1	0
SR Watson	2	0	0	0	0	1

This way we can represent every batsman uniquely by a combination of their slab value and 5 dimension encoding vector. I.e the vector (1,1,0,0,0,0) identifies Gayle (as a batsman) while the vector (2,1,0,0,0,0) identifies de Villiers.
We use the same approach to encode the bowlers.
We use one-hot encoding for the outcomes column in a separate table.

- We merge these tables with the actual table and get rid of the unnecessary columns to get the table of only values that would be fed to the neural netwotk. This table contains 29 columns (6 columns each to represent the batsman on strike and the bowler bowling the delivery and 17 more columns to represent the outcome, one for each outcome.) Note that columns 1:12 represent the input matrix X and columns 13:29 represent the outcome, the ground truth Y.

- The 17 classes for Y are `"leg bye", "no run", "wide", "four", "six", "run", "caught", "bye", "bowled", "caughtWK", "leg before wicket", "run out", "stumped", "caughtB", "no ball", "hit wicket", "obstructing the field"`
- The data was split in approx. 70-30 ratio randomly into train set and test set. We shall be using this training set to train the network.

TRAINING THE NETWORK

How does it work?
We stack the observations on top of each other to form the input matrix X. Each row of X is an independent observation and each column of X is a feature. Let the dimensions of X be r_1 x c_1. The input matrix X (can be considered as A_1) is multiplied by the weight matrix W_1 to get Z_2. W_1 is a matrix of dimensions c_1 x n_1 where n_1 is the number of hidden units in the first layer. We apply am activation function (sigmoid or tanh or relu) elementwize on Z_2 to get A_2. Note that the dimensions of both Z_2 and A_2 are the same i.e r_1 x n_1. This is repeated until we have reached the last layer.
Upon reaching the last layer the cost is computed and using the derivative of the cost function, we backpropagate thereby updating the weights matrices W_i after each iteration. Formulas given below:

$$A_1 = X$$

$$Z_{i+1} = A_i \cdot W_i \qquad i = 1, 2, \dots L$$

$$A_{i+1} = f(Z_{i+1}) \qquad \text{where } L \text{ is the } \#\text{layers}$$

Note. f is the activation fn and is applied elementwise on Z.

$Z_L^{(d)}$ is computed using the derivative of the cost function.

$$W_i^{(d)} := A_i^T \cdot Z_{i+1}^{(d)} \,/(\text{\# of observations})$$

$$A_i^{(d)} := Z_{i+1}^{(d)} \cdot W_i^T$$

$$W_i := W_i - \alpha \cdot W_i^{(d)} \quad , \ \alpha = \text{learning rate}$$

$$Z_i^{(d)} := A_i^{(d)} * f'(Z_i) \quad , \ \text{elementwise multiplication}$$

- We had exactly 5,29,166 observations in total. 70% of which (training set size) is 3,70,416 which is quite a large number and hence updating weights at the end of processing all the points would be computationally pretty costly as that would involve an input matrix of dimension 3,70,416x12 (12 input features, 6 for batsman and 6 for bowler). So we decided to use batch gradient descent.
- For using **batch gradient descent**, we split the entire data randomly into training set of size 3,70,000 and test set of size 1,59,166. This facilitated us to split the training set into 370 batches of size 1000 observations each.
- For each batch $r_1 = 1000$, $c_1 = 12$. The weights would be updated at the end of each batch and each batch is independent of the other which implies that the cost would not decrease monotonically as it should after each iteration of gradient descent.
- Each time all the 370 batches are processed is known as an epoch.
- For our purposes we used the **relu activation** function.
- This is a classification problem, hence we added a **softmax layer** in the end. The softmax function works for each row by using the ratio of each element of Z_L exponentiated and dividing it by the sum of all elements in that row exponentiated. E.g
  If a row of Z_L is [-1,5,2,1] then the softmax layer would output the following
  $[e^{-1}/(e^{-1}+e^5+e^2+e^1), e^5/(e^{-1}+e^5+e^2+e^1), e^2/(e^{-1}+e^5+e^2+e^1), e^1/(e^{-1}+e^5+e^2+e^1)]$
  We used the softmax function in the last layer instead of the relu function. As we can see the softmax function outputs values in the interval (0,1) which are treated as probabilities for each class.

- We used the **cross entropy cost function**, with loss $L = -\sum y_c \log(p_c)$, summation over all classes c, for each row and then the loss is summed over all rows to get the cost. Here y_c is 1 if the observation belongs to class c and p_c is the probability of class c given by the softmax function.
- We had to use 27 epochs for the cost to converge to 1.600152

TESTING

- On running the neural network using the weights we derived from the training above on the test set we got a cost of 1.598617 which is unexpectedly slightly better than the training set.

BUILDING THE GAME

- Typically a softmax classifier classifies the observation into the class for which the probability as returned by the softmax function is largest. But this wouldn't work in our game because for the same inputs it would always return the same class. Let's illustrate.
  Suppose Narine is bowling and Dhoni is on strike. Our inputs would be Dhoni and Narine and the softmax would always return the same outcome (class) every time Narine balls to Dhoni, say 1 run. But we don't want that. We need Dhoni to hit sixes and fours as well as take singles and eventually get out perhaps.
- The fundamental difference between a traditional neural network that uses the softmax classification and our application is that, the traditional network returns the class for which the probability given by the softmax function while our network returns the distribution of probability across the classes.
- This facilitates the network to return a different outcome every time a pair of batsman and bowler face each other based on their past encounters.
- We do this by generating a random observation from the probability distribution across the classes. Again an illustration.
  Let's assume for simplicity that we had 3 classes, namely DOT.BALL, RUN, WICKET. The softmax function assigns the following probability distribution:

Outcome	DOT.BALL	RUN	WICKET
P(Outcome)	0.45	0.50	0.05

  Step1: We generate a random number rn from uniform(0,1). Let's say for example rn=0.55
  Step2: We cumulate the probability of the classes and compare if rn < cumulative sum of the probabilities until that class for each class

Outcome	DOT.BALL	RUN	WICKET
Cumulated	0.45	0.95	1.00

  rn=0.55 is not less than 0.45, hence its not a dot ball
  rn=0.55 is less than 0.95 and hence the outcome of that delivery was a run

  Note that the next random number can be less than 0.45 (next delivery a dot ball) as well as it can be more than 0.95 (next delivery, a wicket).
  This way the frequency of outcomes for each class can be kept proportional to the frequency of that class in the past.

- After we can produce legitimately distributed outcomes for a delivery, we can move on to laying down basic ground rules for the game. Following are the rules that are incorporated into our game through complicated looping structures.
    - A particular input/output would involve the three parameters, the batsman on strike, batsman off strike and the bowler and would involve the outcomes of each delivery until the over is bowled or until a batsman gets out.
    - In case a batsman gets out, we will have to input the name of the new batsman and the game would continue. The outputs will be less than 6 balls in this case, as the former batsman on pitch has already taken up a few balls.
    - In case of a no ball and a wide ball, an extra delivery will be bowled.
    - In case of runs between the wickets, if the runs are odd, then the strike would be rotated.

Note that in real cricket, strike is rotated at the end of each over, but since our game inputs/outputs for one particular over, the strike rotation at the end of an over has to be taken care of the player at the same time as the player would change the name of the bowler at the end of each over.

Sample inputs/outputs:

```
> overPredictor("Gayle","Dhoni","Narine")
[1] "Batsman on strike: Gayle 1 (leg)bye run(s)"
[1] "Batsman on strike: Dhoni 1 run(s)"
[1] "Batsman on strike: Gayle 1 run(s)"
[1] "Batsman on strike: Dhoni no run"
[1] "Batsman on strike: Dhoni six"
[1] "Batsman on strike: Dhoni 1 run(s)"
> overPredictor("Dhoni","Gayle","Rashid Khan")
[1] "Batsman on strike: Dhoni 1 run(s)"
[1] "Batsman on strike: Gayle 1 run(s)"
[1] "Batsman on strike: Dhoni 1 run(s)"
[1] "Batsman on strike: Gayle 1 run(s)"
[1] "Batsman on strike: Dhoni 1 (leg)bye run(s)"
[1] "Batsman on strike: Gayle four"
> overPredictor("Dhoni","Gayle","Narine")
[1] "Batsman on strike: Dhoni 1 run(s)"
[1] "Batsman on strike: Gayle caught"
> overPredictor("McCullum","Dhoni","Narine")
[1] "Batsman on strike: McCullum caught"
> overPredictor("Sangakkara","Dhoni","Narine")
[1] "Batsman on strike: Sangakkara four"
[1] "Batsman on strike: Sangakkara 1 run(s)"
[1] "Batsman on strike: Dhoni no run"
> overPredictor("Sangakkara","Dhoni","Rashid Khan")
[1] "Batsman on strike: Sangakkara 1 run(s)"
[1] "Batsman on strike: Dhoni caught"
```

ACKNOWLEDGEMENT

I would like to thank Dr. Arnab Bhattacharya, CSE, IIT Kanpur for guiding me through the project and more importantly for giving me the knowledge necessary to plan and execute such a project.

REFERENCES

Class notes (slides) of Prof. Arnab Bhattacharya

Online course: Machine Learning (Coursera)

.. and a lot of google searches

DATA LINK

https://drive.google.com/open?id=1R4HIqNO3t25LU5uwfnaqb8rsDI0KN9tw