Predicting match statistics in computer games using machine learning

Artem Vasenin

May 11, 2017

Proforma

Name Artem Vasenin

College Clare College

Title Predicting arbitrary events in competitive computer team games

Examination

Year 2017

Word Count [1]

Project Originator Artem Vasenin (av429)

Project Supervisor Yingzhen Li (yl494)

Original Project Aim

The aim of this project was to develop an algorithm which could predict whether certain events would happen in multiplayer computer games. In most modern multiplayer games, players are provided with a statistic after each match describing their performance in that match and summarising key choices they have made. The project should be able to predict the values of such statistics. The mean squared error of such predictions should be less than half of variance of the variable.

Summary of Work Completed

A graphical model was created which produces a probability distribution for continuous statistics. The first prototype was built using a combination of ad-hoc use of TrueSkill and machine learning functions from scikit-learn python library. The final version was created entirely using TensorFlow and Keras. The system achieved R^2 value of {} on {} variables in the dataset.

Special Difficulties

Other than the typo in success criteria, there were none.

Declaration of Originality

I, Artem Vasenin of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: May 11, 2017

[1] Calculated using `pdftotext Disseration.tex -f 7 -l 38 - | wc -w` and manually excluding appendices and bibliography

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Multiplayer computer games are becoming very popular, more than a 100 million people play League of Legends every month [1]. A game's success often rests on how enjoyable it is. Current method of improving player experience is to make sure that all players have equal chance of winning in a game. For that purpose their skill has to be tracked and teams have to be arranged such that they are of equal strength.

In many popular games several roles have to be filled on each team for optimal performance. Current algorithms, such as TrueSkill [2], only track player's overall skill and do not consider what roles the player prefers and how good they are at each one. This often leads teams being composed of players all wanting to play in the same role, which either leads to team underperforming or some players not enjoying the game as much as they could. Moreover, I believe that the events that happen in game are more important to many player's experience than the actual outcome.

To be able to match players better, a system has to be built that will take into account how the game is played and what strategies exist in it. Creating such a system using classical techniques would require a deep knowledge of workings of a game. Unfortunately, I do not have such knowledge of most of the games and obtaining such knowledge although might be fun, will take too long. Furthmore, most such games change their rules every few months to keep players interested, therefore performance of hard coded system would decrease as the time goes on. As a result I decided to use machine learning techniques which would help me create a system which can adapt to different games by itself and also update its judgement as game changes.

1.2 Related Work

Before starting the project, I have looked into papers and articles about predicting events in sports and computer games. I have found out that nothing similar has been done before. Most closely related algorithms (such as [2] and [3]) were made to only predict the outcome of the game, not any related statistic. Outside of academia, other algorithms were made which used machine learning to improve their performance, but again only focusing on the outcome of the game.

Chapter 2

Preparation

2.1 Proposal Refinement

I had two drafts of my project proposal, this being my first major project I had little experience in preparing a plan. After receiving comments from my overseers regarding the first draft of the proposal, I have incorporated several changes in the project proposal, including:

- Add a time-line, deadlines and deliverables to the project plan, to be able to track the progress of the project accurately.
- Limited and listed all of the potential variables that will be considered, to keep the project goals concise.
- Add a criteria to library selection to streamline the process.

Overall this allowed me to stay on target throughout the project and control "feature creep".

2.2 Types of variables to predict

Values included in the game statistic can be rougle separated into two types: class based and regression based.

For example in many games players have the ability to buy items. These items are usually represented as numbers in match statistic, but nearby values usually don't have much in common. Therefore, if we want to predict what items a player will buy, a classification method should be used.

On the other hand something like players score is a value which increases progressively, therefore nearby values have similar significance. In such cases regression techniques should be used.

2.3 Requirement Analysis

The projects aimed to produce a system that would be able to predict match statistics data from players past games.

2.3.1 Functional Requirements

- The system must be able to run on any properly formatted data.

- The predictions should be based on skills of all players in the game, not just the player for which predictions are being made.
- The system should be able to maintain a belief state about a player skill through time and make time-sensitive predictions, not the same prediction for all games.

2.3.2 Non-functional Requirements

- The system should be able to make predictions in under a minute after being trained, on my machine[1].
- The mean squared error of system's predictions should be less than one half of player's standard deviation for that statistic.

Regarding the last point: while this is the requirement given in project proposal, I later realised that it contains a typo which make it almost impossible to achieve. "Mean squared error", this should have been "mean absolute error". As it currently stands the systems deviation is squared, while original deviation is not, which means that achieving this requirement gets more difficult the larger the original deviation is.

2.4 Starting Point

At the beginning of the project I had:

- Basic knowledge of artificial intelligence from *Artificial Intelligence I* Part IB course.
- Programming experience in Python (acquired by working on personal projects) and Java (acquired by completing courses in first two years of my study).
- Knowledge of probability and Bayes' theorem from A-Level maths and math courses in the first two years.

During the course of the project I had to gain following qualities:

- Understanding of Bayes' inference and its use in ranking.
- Understanding of various machine learning techniques.
- Familiarity with a chosen ML library (Tensorflow).
- Knowledge of LaTeX and related packages (such as Tikz for graphs) to write-up this dissertation.

2.5 Theoretical Background

2.5.1 Probability background

There are a few probability rules, which are used in this model.

[1] Specifications of machine's hardware are given in the project proposal

Sum rule Sum/Integral rule allows us to remove unneeded variables from our probability estimates:

$$P(A) = \int_B P(A, B) \tag{2.1}$$

Product/Chain rule Product rule allows us to calculate complex probabilities only using conditional probabilities and priors:

$$P(A, B) = P(A \mid B)P(B) \tag{2.2}$$

Bayes' theorem I will be using Bayes' networks and inference in my system, which is based on Bayes' theorem:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{2.3}$$

- $\boldsymbol{P(A)}$ and $\boldsymbol{P(B)}$ are the probabilities of observing A and B, called *prior*s.
- $\boldsymbol{P(A \mid B)}$ is the probability of observing A given B, called *posterior*.
- $\boldsymbol{P(B \mid A)}$ is the probability of observing B given A, called *likelihood*.

2.5.2 Machine Learning

Machine learning allows computers to make predictions on data by learning on past examples and without being explicitly programmed. The aim of my project was to create an algorithm which could work with any game, therefore machine learning was the perfect set of techniques to apply.

Since I decided to use machine learning in my project I had to get some background in the area. Unfortunately, computer science course is in Lent term, therefore I attended a similar course in Engineering department in Michaelmas term, before beginning work on the theory.

After I have finished my research, I have decided to try the following machine learning techniques in my projeject:

- Naive bayes (NB)
- Linear Regression (LinReg)
- Logistic Regression (LogReg)
- Polynomial Regression (PolReg)
- Ridge Regression (RidReg)
- Neural Networks (NN)
- Support Vector Machines (SVM)
- Gaussian Processes (GP)
- Random Forests (RF)

2.5.3 Over and under fitting

When training an algorithm a decision has to be made when to stop the training, this is difficult decision and is usually made using some kind of heuristic. If the training is stopped too early it leads to under-fitting, if it is stopped too late, it leads to over-fitting, in both cases the algorithm performs below optimum.

Under-fitting An under-fit algorithm cannot model the training data well and is not suitable. It is often easy to detect using a good loss metric.

Over-fitting An over-fit algorithm models the data too well, the algorithm learns the noise in the training data. Noise is random, therefore it will be different in the real/test data. By modelling the noise in the training data, the real world-performance is decreased. Over-fitting is a bigger problem, since it is much more difficult to detect, therefore different methods are usually used to prevent it.

2.6 Software Engineering Techniques

2.6.1 Workflow

I have decided to use an agile approach of working on this project, since I was not sure exactly how the algorithm should work at the start of the project. The project would be done in two phases:

1. Primary research would be done during the second half of Michaelmas term. A prototype would be developed during Christmas break.

2. At the beginning of Lent term the prototype would be evaluated. In the first two weeks of February the theory would be improved using insights gained during the development of the prototype. In the second two weeks of February the prototype would be refactored to comply with changes in the theory. At the beginning of March the revised implementation would be evaluated.

This approach allowed me to incorporate the knowledge I gained during the evaluation of a prototype into the final version of the algorithm.

2.6.2 Version Control

Git was used for version control. This allowed me to roll-back to a previous version of the project in case a mistake was made. It also allowed me to compare different versions of a specific component. The repository was hosted on GitHub[2] which allowed me to work on the project from multiple machines.

2.6.3 Backup

The code was continuously synchronised to Dropbox[3]. Weekly checkpoints were saved to an external hard drive.

[2] github.com
[3] dropbox.com

2.6.4 Testing

Testing will be done using standard train-validation-test method.

2.7 Tools used

2.7.1 Programming Language and ML Libraries

Machine learning is one of the key elements of this project, therefore I needed a library that would implement key techniques used in ML for me, so that I don't have to write them myself. I have compared a number of libraries (such as Tensorflow[4] and Torch[5]), I was primarily interested in how much funcitonality they provide, their performance and how easy it is to learn them.

I have compared their functionality by completing standard machine learning task, such as creating a neaural network to classify images in the MNIST[6] dataset. Performance was compared by timing how long it would take to achieve 99% accuracy, in most cases it came down whether GPU acceleration was supported. Since I would have to learn how to use the chosen library in detail I also paid attention to the amount of support material available. I took note of quality of documentation and also compared number of questions and answers on StackOverflow.

In the end I have arrived at the conclusion that they all provided required functionality and were sufficiently easy to use. API for most of them were written in different languages. I did not want to learn a new language in addition to learning a library, therefore I decided to use TensorFlow, API for which was written in Python, a language I am most comfortable writing code in. TensorFlow is a low-level library, in which you have to construct each layer of neural network explicitly, I did not need that level of customisation, therefore I used Keras[7] to help me build the neural networks.

Additionaly Python has a higher level machine learning library called scikit-learn[8], which provides easy way to quickly implement many machine learning methods.

To a lesser extent Python was also chosen since it has a package manager[9], which makes it much easier to install additional packages.

2.7.2 IDE

Pycharm was chosen the integrated development environment (IDE) for the project. Pycharm has all of the core features of an IDE, such as syntax checking, autocompletion, running of code, etc. The use of an IDE streamlined the process of development and allowed me to focus on improving the algorithm rather than worrying about language syntax or library functions signatures.

[4] tensorflow.org
[5] torch.ch
[6] yann.lecun.com/exdb/mnist
[7] keras.io
[8] scikit-learn.org
[9] pypi.python.org/pypi/pip

Chapter 3

Implementation

3.1 Installation of required libraries

Python has a package index (PyPI) which is used to distribute most available packages. The developers of PyPI provide a tool called *pip* which can manage package installation for you, it can be installed on Ubuntu using:

```
$ sudo apt install python3-pip
```

After pip is installed all required packages can be installed using:

```
$ sudo pip install <package name>
```

Pip will manage all of the required dependencies and put the package files in the correct directories.

3.2 Data Collection and Pre-processing

3.2.1 Collecting the data

To train and evaluate my algorithm I had to find a complex dataset which would have different types of variables for me to predict. I decided to use game called "Dota 2" for several reasons:

- Summary of each professional game is publicly available and easily accessible through the provided API.
- "Dota 2" is one of the leading E-Sports with hundreds of professional matches each year, which provided me with plenty of data points.
- The game is requires a lot of skill with very little amount of luck involved, which makes it much suitable to prediction.
- The game is team-based and requires a lot of co-operation, which makes it more difficult to predict using classical rating algorithms.

3.2.2 Pre-processing

Cleaning the data The API which provides the summaries sometimes returns incomplete data with some of the values, such as the outcome of the match, missing. I had to find and remove all data points which did not contain complete set of results.

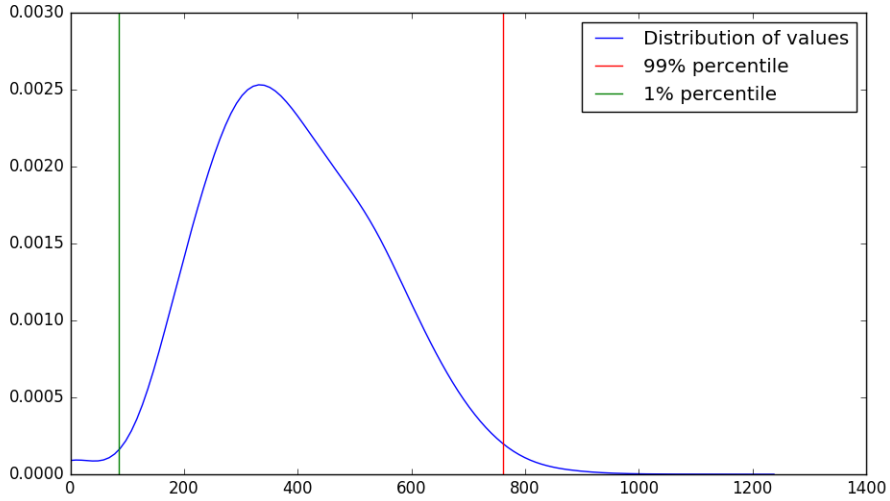


Figure 3.1: The distribution of values one of the variables.

Outliers Most variables contain some outliers which are extreme cases. In some cases the maximum value is almost twice as high as the 99% percentile, as can be seen in figure 3.1. To simplify the learning for the algorithm I decided to clip all the value for each variable between the 1st and 99th percentile.

Standardisation Some of the variables in the summary occur over much bigger scales than others. Most machine learning algorithms would not train properly and would prioritise predicting large variables more accurately than smaller ones. I would like to consider all variables with equal priority there I had to normalise the data. To normalise data we can a apply a simple transformation to all data points:

$$x' = \frac{x - \overline{x}}{\sigma} \tag{3.1}$$

Where $\overline{x}$ is the mean of x and σ is its standard deviation

Normalisation During the analysis of data I decided that the data is better modelled using a gamma distribution. Gamma distribution can only represent positive numbers, therefore I could not use standardisation for this dataset. Instead I used normalisation which also rescales the values, but keeps them all positive, it uses the following transformation:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3.2}$$

3.3 First Prototype

The main problem in machine learning is frequently generation of correct features to represent the data. In my case I had statistics of more than twenty thousand games. Most players

had a few hundred games in the dataset. Statistic for each match is represented as a dictionary of key-value pairs (in JSON[1] format), overall there are more than 200 values per statistic, an example of a statistic can be found in Appendix A. This meant that using raw data for input would be impractical, therefore some kind of aggregation had to be created.

3.3.1 Simple mean and variance

At first I tried using means and standard deviations of players results as features. While this approach made predictions which were better than just predicting player mean, they weren't very good. A variety of machine learning techniques were tried, including: Neural Networks (NN), Support Vector Machines (SVM), Gaussian Processes (GP), Ridge Regression (RR) and Naive Bayes (NB). NN, SVM and NB were used for classification variables, while GP and RR were used for regression. All of the methods had similar performance, which indicated that features generated were not sufficient. To improve the features I first tried putting a limit on how many games were used to calculate player's mean and average, while this improved performance a bit, it was still very poor. Next thing to try was rating players based on their performance in each variable.

3.3.2 Rating of players

The easiest algorithm to use for rating players was TrueSkill, since it offered rating games with multiple players and there was a library[2] for python which provided the functionality. Trueskill only uses the final ranking of the individual at the end of the game, rather than absolute value, therefore some informations is lost. This also meant that I had to do further processing on the data to convert raw statistic into ranks. This was mostly straightforward, one thing to note is that some ranks (such as death count) had to be reversed, since players try to keep those results low rather than make them high.

Such ad-hoc use of TrueSkill meant that one of the assumptions behind the algorithm was not met: the algorithm assumes that all players are competing against each other, but in reality they are separated into two teams. This meant that resulting ratings were very accurate. Player ranks were then used as features for machine learning techniques. This approach generated much better results than simple mean and variance, with best variable achieving R^2 value of 0.47. Although this is much better than previous results, it is still far from required value of 0.75.

3.3.3 Analysis of different machine learning techniques

As said above most of the techniques provided similar results. In the end I decided to use neural networks as my main technique for the following reasons:

- Neural networks can do both regression and classification, thereby I will only have to focus on learning and understanding one technique.
- One neural network can estimate values for multiple results, thereby reducing the complexity of the system and increasing efficiency.
- Chosen library (Tensorflow) has better support for neural networks than other techniques.

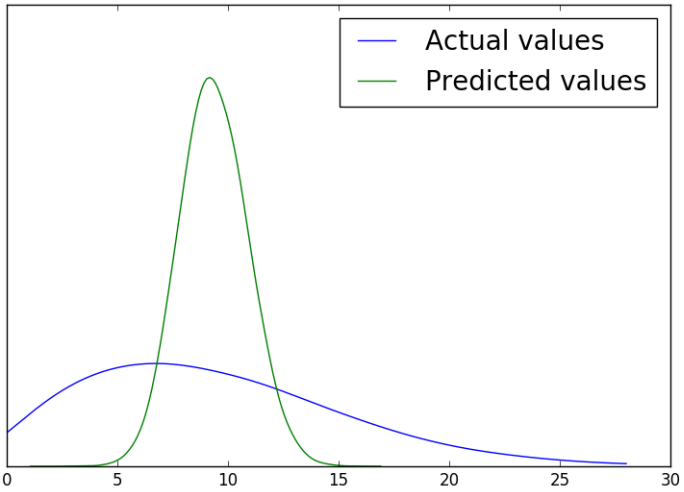


Figure 3.2: Distribution of predicted values vs. actual variable distribution

3.3.4 Analysis of the prototype

During analysis of predictions produced by the prototype, I observed that the range of predictions was much smaller than the actual range for most variables. An example of this can be seen on figure 3.2, where predictions rough range of 5 to 15, while actual variable range is roughly 0 to 30. In addition, most of the machine learning techniques had similar predictive performance. These two facts combined led me to believe that the problem is again with quality of features. Therefore, I decided to create my own graphical model to generate richer features.

3.4 Final System

3.4.1 Choosing distributions for variables

When predicting the values for each variable, rather than predicting the probability for each value independently, it is better to use a distribution to predict the probabilities for all values. To decide which distribution fits each variable best, I used *method of moments* to fit each potential distribution to all observed values of the variable. Then I used *goodness of fit* to decide which of the potential distributions is best.

Method of Moments To fit a distribution we need to estimate its parameters. Parameters for most distributions can be easily expressed in terms of mean and variance of the data. For example for Gamma distribution the equations are:

$$k = \mu^2/\sigma^2 \qquad \theta = \sigma^2/\mu \tag{3.3}$$

Goodness of fit To estimate how well the distribution fits the data we can use chi-squared test to measure sum of differences between observed and expected outcome frequencies:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3.4}$$

Where O_i is the observed frequency and E_i is the expected frequency.

[1] json.org
[2] trueskill.org

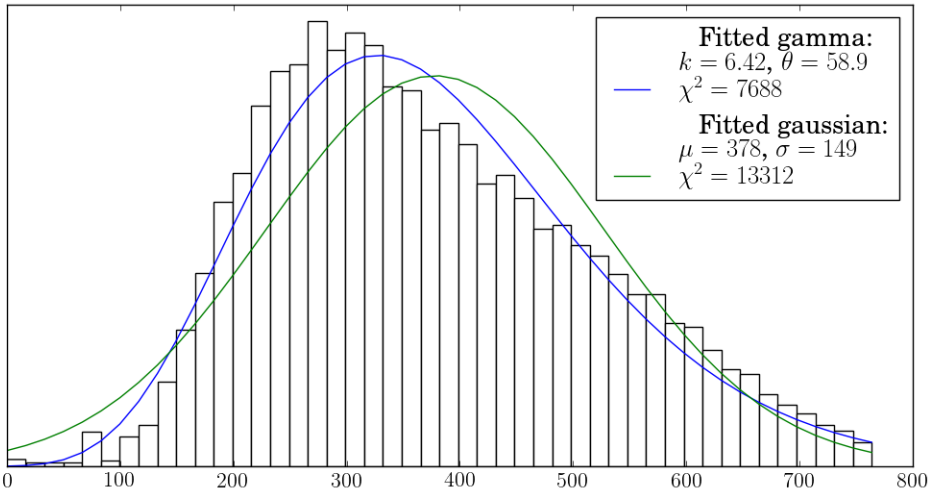


Figure 3.3: Fitted gaussian and gamma distributions on one of the variables. Gamma distribution has a lower χ^2 value, therefore it is a better fit.

3.4.2 Player skills

Every game is different and trying to pick specific set of skills for each one, would not be practical. Therefore, I decided to make an assumption that players skill in a game can be described by n real numbers. Such n will be different for each game and therefore is a hyper parameter in my model. Since the system can never be sure on the exact skill values for each player, skills of each player are represented as n Gaussian distributions.

3.4.3 Neural networks

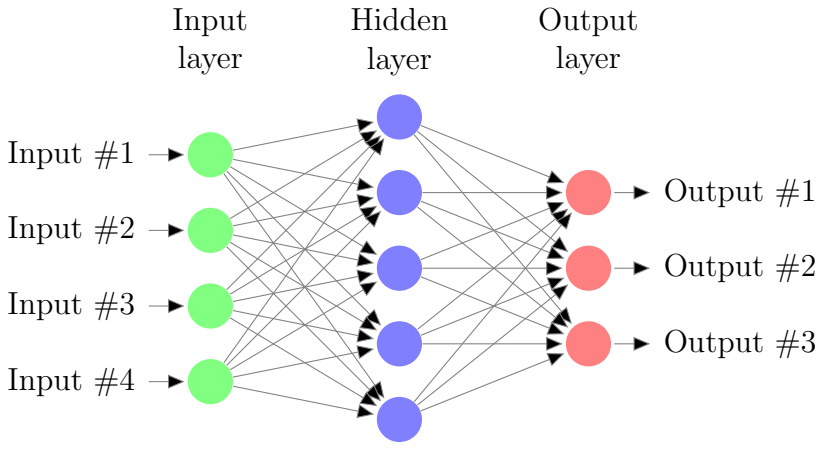


Figure 3.4: Simple fully connected feedforward neural network with one hidden layer.

Because, I decided to use neural networks in my system, I had to learn basic theory behind them and how to construct them. Neural networks made up of three types of layers: input, hidden and output. The size of the input and output layer are determined by the data being processed, deciding on the size of hidden layers is more difficult. If the hidden layer is too small under-fitting can occur, similarly if the hidden layers are too big or there too many of them over-fitting can occur.

Perceptron Each layer of a NN is composed of a number of perceptrons. Perceptron is a simple linear classifier which is can be described by:

$$y = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{3.5}$$

Where y is the output, $\sigma()$ is the activation function, $\boldsymbol{w}$ a vector of weights, $\boldsymbol{x}$ is a vector of inputs and b is the bias.

Types of activation function There are several types of activation function that can be used in neural networks, I considered the following:

Identity $\sigma(x) = x$

Rectified Linear Unit (ReLU) $\sigma(x) = \max(x, 0)$

Softplus $\sigma(x) = \ln(1 + e^x)$

TanH $\sigma(x) = \tanh(x)$

Most of them map the value of x onto a specific range, which is useful if the output must also fall in a certain range. Normally the activation function is based on which layer the perceptron is in. Activation functions such as ReLU are also useful in hidden layers to produce a non-linear transformation.

Type of NN There are many types of neural networks, I decided to use the simplest one: fully connected, feedforward neural network.

Fully connected Input for a perceptron in layer n is composed of output of each perceptron in layer $n - 1$.

Feedforward The data only flows one way in the network, there are no loops.

An example of a structure of such neural network can be seen in figure 3.4.

3.4.4 System overview

The model assumes that there are two types of results for each game:

- Player specific results, such as amount of gold earned or number of enemies killed.
- Overall game results, such as which team won or how long the game took.

The naive approach is to have only one neural network which taken player skill estimates as inputs and outputs predictions of results. There are several problems with such system:

- As the size of input grows, so does the size of the neural network required to accurately process them. This leads to massive neural network which takes a long time to process data and therefore quite inefficient.
- As the size of the neural network grows, the number of examples required to train it also increases. Although I had quite a large data-set I judged that it was not large enough to properly train such a big network.

Therefore I made a few assumptions to create a graphical model which resolves those problems. The assumptions are:

- Skills of all players playing on the same team can be summarised into a smaller set of skills.
- Player specific results only depend on the skills of relevant player and summarised skills of each team taking part in the game.
- Overall game results only depend on summarised team skills.

The summary of resulting system is seen in figure 3.5.

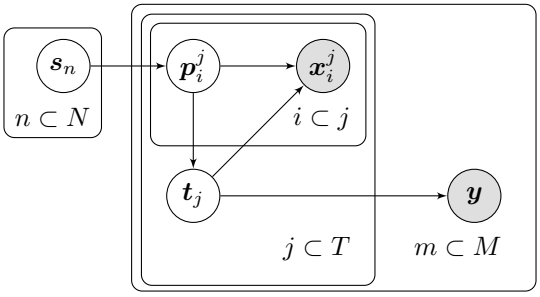


Figure 3.5: Summed up view of the prediction model.

There are N players tracked by the system, participating in M matches. In each match there are T teams. $\boldsymbol{x}_i^j$ are the results of player i in team j and $\boldsymbol{y}$ are the overall match results. Variables in bold are vectors.

There are two parts in the system: predicting results from inferred player skills and inferring player skills from their history. An example of a prediction model made for a game in which four players play in two teams can be seen in figure 3.6 and a factor graph expansion can be found in Appendix B. The skill update model is the same for all games and can be seen in figure 3.7.

Training both the variable prediction model and model for player skill updates at the same time would greatly increase the data required for processing. It will also raise some difficult problem such as *vanishing gradient*, unfortunately I did not have enough time to solve them properly. Instead I decided to use the technique called *coordinate descent*, to optimise the models alternatively.

Coordinate descent When we have multiple parameters to optimise, instead of optimising them at the same time we can optimise them alternatively. We optimise each parameter for a certain number of iterations (or until some criteria is achieved) and then move onto the next parameter. Once we optimised each parameter, we can loop back and start again from the first one. We do that until all parameters reach sufficiently stable values.

3.4.5 Prediction model

There are five variables in the model: $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{t}, \boldsymbol{p}$. To make predictions we would like to calculate $P(\boldsymbol{x} \mid \boldsymbol{p})$ and $P(\boldsymbol{y} \mid \boldsymbol{p})$. As was said above, calculating this directly is expensive, therefore we need to modify them to fit the model. We get:

$$P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}) \quad \text{and} \quad P(\boldsymbol{y} \mid \boldsymbol{t}, \boldsymbol{p}) \tag{3.6}$$

Using the Bayes' theorem we can expand the equations:

$$P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}) = \frac{P(\boldsymbol{x}_i, \boldsymbol{t}, \boldsymbol{p}) P(\boldsymbol{x}_i)}{P(\boldsymbol{t}, \boldsymbol{p} \mid \boldsymbol{x}_i)} \qquad P(\boldsymbol{y} \mid \boldsymbol{t}, \boldsymbol{p}) = \frac{P(\boldsymbol{y}, \boldsymbol{t}, \boldsymbol{p}) P(\boldsymbol{y})}{P(\boldsymbol{t}, \boldsymbol{p} \mid \boldsymbol{y})} \tag{3.7}$$

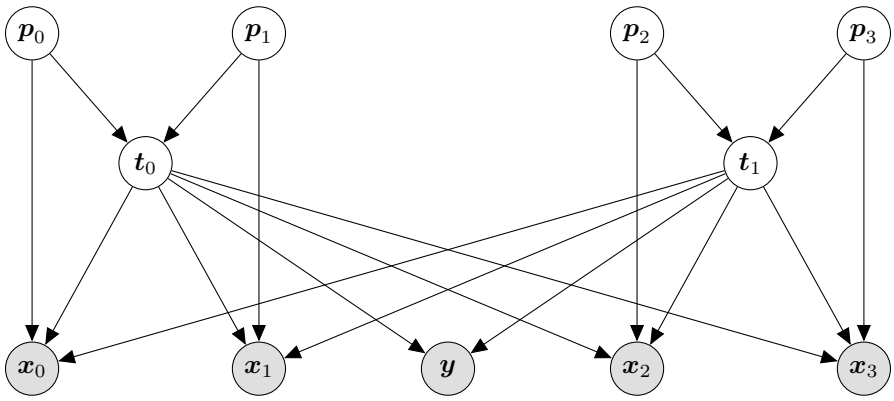


Figure 3.6: An example of an inference model for four players in two teams.
Grey circles represent observed values and white circles represent latent variables.

Using product rule we can expand the top joint probability into several conditional probabilities:

$$P(\boldsymbol{x}_i, \boldsymbol{t}, \boldsymbol{p}) = P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}) P(\boldsymbol{t} \mid \boldsymbol{p}) P(\boldsymbol{p}) \tag{3.8}$$

Using our assumption that player results only depend on the skills of that player and team skills, we can integrate most of the players out, leaving:

$$P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}) = P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}_i) \tag{3.9}$$

Doing the same to other parts we get:

$$P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}) = \frac{P(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}_i) P(\boldsymbol{t} \mid \boldsymbol{p}) P(\boldsymbol{p}) P(\boldsymbol{x}_i)}{P(\boldsymbol{p}_i \mid \boldsymbol{t}, \boldsymbol{x}_i) P(\boldsymbol{t} \mid \boldsymbol{x}_i)} \qquad P(\boldsymbol{y} \mid \boldsymbol{t}, \boldsymbol{p}) = \frac{P(\boldsymbol{y} \mid \boldsymbol{t}) P(\boldsymbol{t} \mid \boldsymbol{p}) P(\boldsymbol{p}) P(\boldsymbol{y})}{P(\boldsymbol{p} \mid \boldsymbol{t}) P(\boldsymbol{t} \mid \boldsymbol{y})} \tag{3.10}$$

Which is now expressed in terms of conditional probabilities we can calculate and priors which we know.

Neural Networks Conditional probabilities such as $P(\boldsymbol{t} \mid \boldsymbol{p})$ are specific to every game and most likely very complex. In such cases I use machine learning techniques to approximate there probabilities, since this is a probability of a vector given another vector, neural networks are the perfect candidate to use. All of the variables in the model are defined using distributions, therefore neural networks must output the parameters of required distribution. For example:

$$P(\boldsymbol{x} \mid \boldsymbol{p}, \boldsymbol{t}) = \text{gamma}(\boldsymbol{x}; \boldsymbol{k}_{\boldsymbol{\gamma}}(\boldsymbol{p}, \boldsymbol{t}), \boldsymbol{\theta}_{\boldsymbol{\gamma}}(\boldsymbol{p}, \boldsymbol{t})) \tag{3.11}$$

Where parameters $\boldsymbol{k}_{\boldsymbol{\gamma}}$ and $\boldsymbol{\theta}_{\boldsymbol{\gamma}}$ are outputs of a neural network with inputs $\boldsymbol{p}$ and $\boldsymbol{t}$ parametrised by $\boldsymbol{\gamma}$.

Variational Methods During the skill update phase, we generate expected player skills

Training neural networks To train a neural network a loss function is required. All of the outputs are probability distributions, therefore we can use the probability density function (pdf) of the chosen distribution. The higher the pdf, the better the estimate, but loss function is minimised, therefore we need to negate the pdf to define loss. For example a loss function for a Gaussian distribution would be:

$$\text{loss}(x, \mu, \sigma) = -\frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.12}$$

where x is the desired value and μ & σ are estimated parameters.

Optimising loss function The loss function is defined across the whole system, therefore we need to multiply the pdf of all the distributions. Multiplication is problematic due to two problems:

- It is computationally expensive
- Multiplying small probabilities can lead to underflow, which will mean that the final probability will be equal to zero.

To solve these problems the product of probabilities can be turned into a sum by taking a log of the joint probability:

$$\text{loss} = \log\left(\prod_{p \subset P} p(x)\right) = \sum_{p \subset P} \log(p(x)) \tag{3.13}$$

Since all probabilities are positive log of them is always defined, log is also a monotonically increasing function which means that decreasing log also decreases the function.

Computing the pdf itself also poses a problem, since potentially very small and very large values are multiplied together the pdf itself can underflow or overflow. Since the loss equation already contains log, the multiplications in pdf can turned into sums by pushing the log further in, e.g. Gaussian from equation 3.12 becomes:

$$loss(x, \mu, \sigma) = \log(2\pi)/2 + \log(\sigma) + \frac{(x-\mu)^2}{2\sigma^2} \tag{3.14}$$

The optimising function can only change the parameters produced by the neural networks, therefore constants in the loss function do not matter and can be removed, leaving:

$$\text{loss}(x, \mu, \sigma) = \log(\sigma) + \frac{(x-\mu)^2}{2\sigma^2} \tag{3.15}$$

Restricting the range of the output Some of the parameters have to fall within certain range, but the neural output produced by a node in a neural network is usually not bound to any range. This can be solved by using an appropriate activation function on the final layer of the neural network. *ReLU* is an obvious choice for enforcing positive values, but it is problematic in that the optimisation function may not properly adjust the weights of the NN, I decided to use *Softplus* instead.

Although, sometimes different outputs of the same neural network must have different bounds placed on them. Since the activation function is usually defined on the whole layer, it is frequently better to keep activation function of the last layer as *linear* and apply appropriate activation function on each element individually.

3.4.6 Skill update model

After every match the system updates its belief about the player skills using the previous belief and player performance in the match. This forms a chain of beliefs about player skill at each particular moment in time. This model treats the player performance reported by the prediction model as observations of player skill. After each game, for each player the model updates the player skill using:

$$P(\boldsymbol{s}_t) = P(\boldsymbol{s}_t \mid \boldsymbol{s}_{t-1}, \boldsymbol{p}_{t-1}) = \mathcal{N}(\boldsymbol{s}_t; \boldsymbol{\mu}_{\boldsymbol{\lambda}}(\boldsymbol{s}_{t-1}, \boldsymbol{p}_{t-1}), \boldsymbol{\sigma}_{\boldsymbol{\lambda}}(\boldsymbol{s}_{t-1}, \boldsymbol{p}_{t-1})) \tag{3.16}$$

Where $\boldsymbol{\mu}_{\boldsymbol{\lambda}}$ and $\boldsymbol{\sigma}_{\boldsymbol{\lambda}}$ are outputs of a neural network with inputs $\boldsymbol{s}_{t-1}$ and $\boldsymbol{p}_{t-1}$ parametrised by $\boldsymbol{\lambda}$. The aim of this model is to predict the performance of a player in their next match, therefore we can define the loss function for this neural network similarly to the one in equation 3.15.

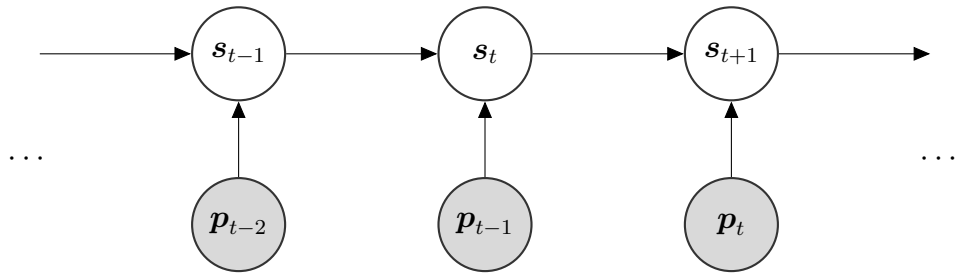


Figure 3.7: Skill update model

3.4.7 Prediction

To predict the values for a particular match-up we just need to take the set of skills corresponding to players in the match-up and feed them into the top of the model.

Restricting the time frame of training data Many popular multiplayer computer games change their rules a few times a year to keep the game fresh and players interested as well as to balance the irregularities. In addition the strategies players employ evolve through time and different skills become more or less important in the game. If the system is trained on the whole history of games, it is optimised to predict results across the whole time-frame, but most of the time we are only interested in predicting future results. Therefore, to improve the accuracy of the system we can only use recent matches in which players employ current strategies to train the system. Although, there is a trade-off: the smaller the time-frame the fewer matches are available for analysis.

Chapter 4

Evaluation

4.1 Preprocessing

When neural networks have been trained, we need process every match in chronological order to update the player skills. To make evaluation quicker we can create a dictionary which contains match id as keys and skills of the players taking part in that match. We need to store the player skills before processing the match and updating them to accurately reflect real world scenario.

4.2 Train/Validation/Test split

One of the most important rules when evaluating machine learning systems is that the algorithm must be tested on previously unseen data. Therefore right at the start we can split our data into two parts: *train/validation* and *test*. *Train/validation* set will be used to train the algorithm and adjust hyper-parameters. *Test* set will be used for evaluation of the algorithm.

4.2.1 Preventing Over-fitting

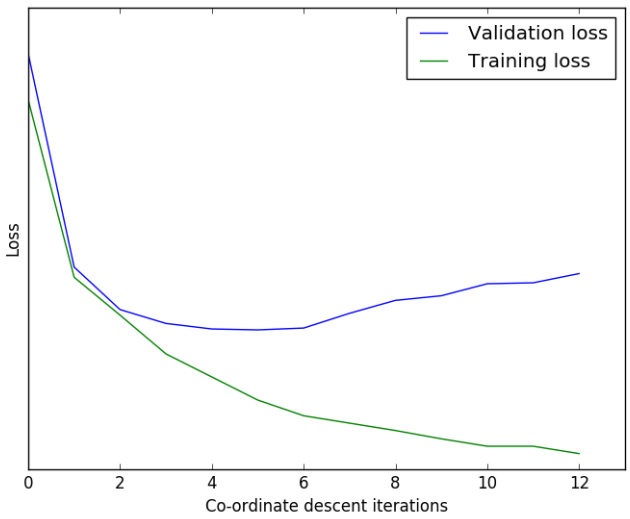


Figure 4.1: Overfitting example.

In this case the optimisation should be stopped at the fifth iteration, after which training loss keeps decreasing, but validation loss starts increasing.

Deciding on the length of training period is a difficult tasks. If the training period is not long enough the algorithm will be under-fit. Similarly, if the training period is too long, the algorithm might over-fit, as can be seen on figure 4.1, even if measures are taken to prevent over-fitting time will still be wasted. There are several methods to prevent over-fitting:

Using simple models If the model has the correct complexity, it will only be able to learn the data and not the noise. Another benefit of this approach is that training and prediction times are smaller. Using this method the model can be trained until loss stops decreasing. Unfortunately, if the model is too simple the model can suffer from under-fitting.

Model regularization Another approach is to penalise complexity in the model. For example the size of weights in neural networks can be limited to a certain value. Unfortunately, this can also lead to under-fitting.

Early stopping Over-fitting can also be prevented by emulating testing scenario during training. For that purpose the train/validation part is split into train and validation. The algorithm is trained on train part and after every iteration it is tested on validation part. When the validation error stops decreasing the training is stopped.

I decided to use validation since it does not suffer from potential of under-fitting and also has added benefit of allowing selection of hyper-parameters. The best hyper-parameter combination can be found by first training the system using each combination and validating it, and then comparing the results of each validation.

4.2.2 Cross-validation

If only one part of training data is used for validation, it might not properly represent the performance of the algorithm. To combat that, we can use cross-validation: we run train/validation multiple times, each time picking new set to use for validation. To ensure complete coverage of the train-set we can use *k-fold cross-validation*:

1. Divide the training set into k random subsets.
2. Pick a subset that has not been validated on, use it for validation and the rest for training. Record the result.
3. Repeat step two until all subsets have been used for validation.
4. Average the results.

4.2.3 Split ratio

When splitting the dataset into train/validation and test parts it is important to keep in mind the total size of the dataset. Making the size of the test part too small can lead to statistically insignificant results and also increase the chance of sampling error. On the other hand making the size of training data too small can lead to under-fitting for the model.

These problems usually only apply to small population sizes and since my population size was 14000 matches, I decided to use standard 80:20 train/test split using the Pareto principle.

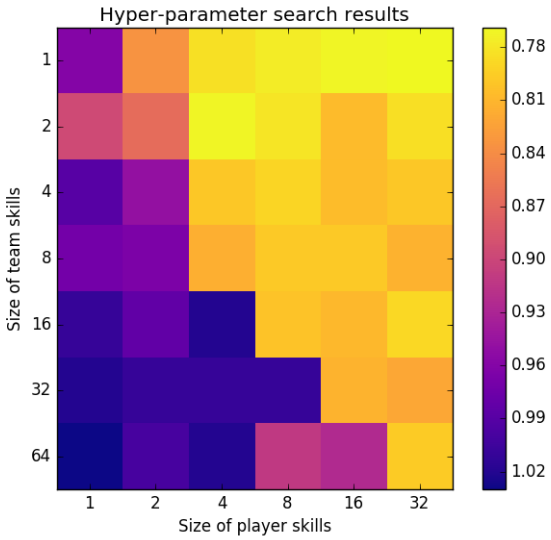


Figure 4.2: Hyper-parameter search

As can be seen, increasing the size of player skills has much bigger effect than changing size of team skills.

4.3 Hyper-parameter selection

The designed model includes several hyper-parameters which have to be adjusted for each game[1].

1. Size of the vector representing player skills.
2. Size of the vector representing team skills.
3. Size and structure of each neural network.

They are ranked in order of importance, adjusting the first one gives the most accuracy increase, but also increases required resources the most.

As can be seen in figure 4.2 the accuracy of the system is mostly dependent on the size of vectors used to represent player skills. Increasing the size of team skills does not have much of an effect and in most cases actually decreases the performance of the system. If the team skills are more than two times bigger the player skills the model overfits and accuracy is greatly reduced.

This shows the main failure in the model: team skills do not affect the accuracy of the system and therefore we can infer that they are not used by the prediction model. Therefore the present system does not take into account the skills of other players in the match and only uses the skill of the relevant player.

4.4 Performance Evaluation

The system is designed such there is as little coupling as possible between the two models. This allowed me to work on each model independently and also made evaluation simpler by allowing me to evaluate both models separately from each other.

[1]There are extra hyper-parameters such as optimisation rate, which are not specific to any game. All of the hyper-parameters which I considered I listed in appendix C.

4.4.1 Prediction model evaluation

Prediction model is responsible for two functions in the system:

- Estimating player performance in a particular match
- Predicting player results given player skills.

To evaluate how well different configurations works, pairs of skills and results are taken and split into train/validation parts. Then the different configurations are trained and validated on the respective parts. After the results for all configurations are obtained, the best configuration is chosen and used to estimate player performances in each match. The configuration consists of settings for the five different neural networks from equation 3.10, namely $p(\boldsymbol{y} \mid \boldsymbol{t})$, $p(\boldsymbol{x}_i \mid \boldsymbol{t}, \boldsymbol{p}_i)$, $p(\boldsymbol{t} \mid \boldsymbol{p})$, $q(\boldsymbol{p}_i \mid \boldsymbol{t}, \boldsymbol{x}_i)$, $q(\boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y})$.

Using hyper-parameter search I discovered that the model functions quite well even without any hidden layers, adding a single hidden layer improves performance a bit. Increasing the number of hidden layer to two or more gave negligible performance increase and therefore was not used. Therefore all five neural networks use a simple one hidden layer configuration.

4.4.2 Skill update model evaluation

The accuracy of prediction model depends directly on how accurate the skill estimations of the skill update models are, therefore it is important that skill estimation is as accurate as possible. To optimise the skill update model, sets of player skills, performances and next performances are taken and the split into train/validation parts. Skill update model uses only one neural network, $p(\boldsymbol{p}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{p}_t)$, therefore configuration consists only of the structure of this neural network. Similarly to prediction model different configurations are trained and validated and the best is chosen. The chosen neural network is then used to update the skills of each player at time t.

Similarly to neural network in prediction model, neural network without hidden layers worked quite well. Although in this model, sufficient improvements have been seen in using up to two hidden layers.

4.4.3 Overall system evaluation

To evaluate the system overall co-ordinate descent is performed where prediction model and skill update model are optimised alternatively. At the start there are no estimates of player skills, therefore the model assumes that all skills are modelled as a standard Gaussian: $\mathcal{N}(0, 1)$. In most cases the optimisation is quite fast only taking two to five rounds of co-ordinate descent.

The co-ordinate descent follows a a simple procedure:

1. Optimise prediction model and update player performances.
2. Optimise skill update model and update player skills.
3. Validate the whole system.
4. Compare validation result to previous, if it is better save the state of the system and go back to step 1. If the result is worse, load back the state of the system from last validation and proceed to next step.
5. Test the system using the loaded configuration and report the result.

The system is tested and validated by running the prediction model using the player skill estimates from appropriate time. Since the system uses cross-validation, steps one, two and three are performed k times, once for each fold of k-fold cross-validation. The validation result in step four is the mean of cross-validation results.

Chapter 5

Conclusions

5.1 Summary of Achievements

5.2 Future work and Improvement Ideas

During the work on this project I had many ideas about how different things can be done and what methods can be used. Unfortunately, I didn't have enough time to try many of them, but I think they should definitely be tested in the future, these include:

Using Kalman filter for skills updates The system represents skills and performances as gaussians, Kalman filter is a perfect candidate for the task.

Using RNN for skill update The skill update model treats player performances as observations, therefore using recurrent neural networks with previous n games as input might produce better results.

Collecting and using more results for each match The current system only used a very small amount of data available about the game, using more information about each match would definitely improve performance, it would be interesting to know how much.

5.3 Lessons Learned

The main lesson I learned, is to not overestimate my ability when working on novel projects. While the work done during improvement of an existing feature/product can be hard and challenging, usually the possible improvements can be estimated from similar work, but when working on new idea the results can be much harder to predict since what works in one area might not work in the other.

The second lesson I learned, is that machine learning is not the all-powerful tool that can solve all problems and that most of the time there are still a lot of trade-offs to be made.

Bibliography

[1] Forbes. *Riot Games Reveals 'League of Legends' Has 100 Million Monthly Players*. 2016. URL: forbes.com/sites/insertcoin/2016/09/13/riot-games-reveals-league-of-legends-has-100-million-monthly-players/#274b74155aa8 (visited on 04/16/2017).

[2] Ralf Herbrich, Tom Minka, and Thore Graepel. "TrueSkill(TM): A Bayesian Skill Rating System". In: MIT Press, Jan. 2007, pp. 569–576. URL: https://www.microsoft.com/en-us/research/publication/trueskilltm-a-bayesian-skill-rating-system/.

[3] Ruby C. Weng and Chih-Jen Lin. "A Bayesian Approximation Method for Online Ranking". In: *J. Mach. Learn. Res.* 12 (Feb. 2011), pp. 267–300. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.1953057.

Appendix A

Example of a Statistic

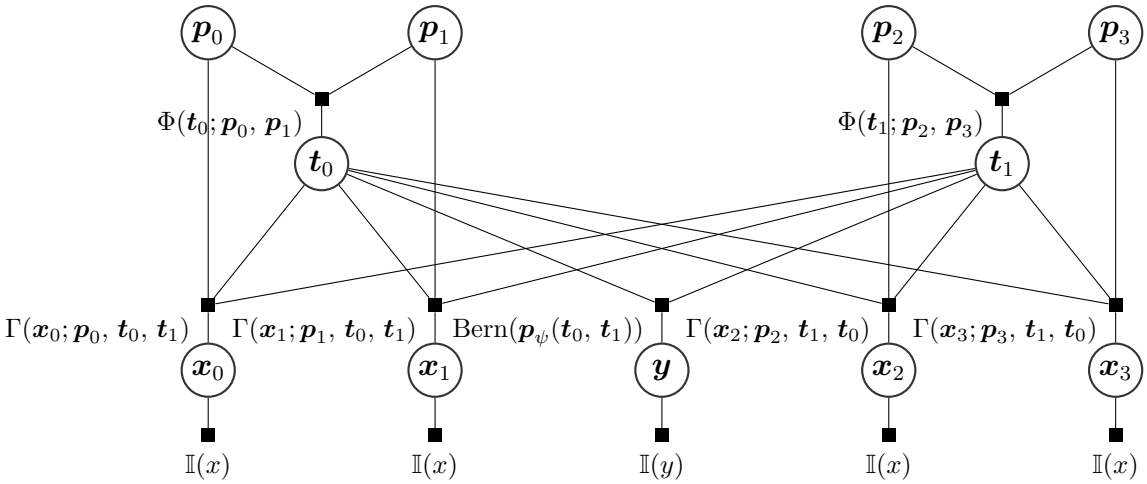
Some parts of this statistic are repetetive and take up a lot of space. In such parts, all but the first set were hidden, indicated by comments.


```
{
   "result":{
      "players":[
         {
            "account_id":87382579,
            "player_slot":0,
            "hero_id":60,
            "item_0":239,
            "item_1":92,
            "item_2":46,
            "item_3":108,
            "item_4":29,
            "item_5":6,
            "backpack_0":0,
            "backpack_1":0,
            "backpack_2":0,
            "kills":3,
            "deaths":7,
            "assists":17,
            "leaver_status":0,
            "last_hits":72,
            "denies":2,
            "gold_per_min":254,
            "xp_per_min":298,
            "level":14,
            "hero_damage":5682,
            "tower_damage":277,
            "hero_healing":170,
            "gold":73,
            "gold_spent":9220,
            "scaled_hero_damage":0,
            "scaled_tower_damage":0,
            "scaled_hero_healing":0,
            "ability_upgrades":[
               {
                  "ability":5275,
                  "time":886,
```

```
                    "level":1
                },
                "There is one set per level and players can advance up to level 25"
            ]
        },
        "There are nine more sets of player data"
    ],
    "radiant_win":false,
    "duration":2368,
    "pre_game_duration":90,
    "start_time":1471142574,
    "match_id":2569610900,
    "match_seq_num":2244488581,
    "tower_status_radiant":1540,
    "tower_status_dire":1956,
    "barracks_status_radiant":3,
    "barracks_status_dire":63,
    "cluster":113,
    "first_blood_time":89,
    "lobby_type":1,
    "human_players":10,
    "leagueid":4664,
    "positive_votes":38429,
    "negative_votes":2503,
    "game_mode":2,
    "flags":1,
    "engine":1,
    "radiant_score":27,
    "dire_score":30,
    "radiant_team_id":2512249,
    "radiant_name":"Digital Chaos",
    "radiant_logo":692780106202747975,
    "radiant_team_complete":1,
    "dire_team_id":1836806,
    "dire_name":"the wings gaming",
    "dire_logo":352770708597344369,
    "dire_team_complete":1,
    "radiant_captain":87382579,
    "dire_captain":111114687,
    "picks_bans":[
        {
            "is_pick":false,
            "hero_id":79,
            "team":1,
            "order":0
        },
        "There are about 20 sets in this list, detailing the drafting phase"
    ]
  }
}
```

Appendix B

An example of factor graph



$$\Theta(\boldsymbol{p}; \boldsymbol{s}) = \mathcal{N}(\boldsymbol{p}; \boldsymbol{\mu}_\theta(\boldsymbol{s}), \boldsymbol{\sigma}^2_\theta(\boldsymbol{s}))$$
$$\Phi(\boldsymbol{t}; \boldsymbol{p}_i, \boldsymbol{p}_j) = \mathcal{N}(\boldsymbol{t}; \boldsymbol{\mu}_\phi(\boldsymbol{p}_i, \boldsymbol{p}_j), \boldsymbol{\sigma}^2_\phi(\boldsymbol{p}_i, \boldsymbol{p}_j))$$
$$\Gamma(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{t}_i, \boldsymbol{t}_j) = \text{gamma}(\boldsymbol{x}; \boldsymbol{k}_\gamma(\boldsymbol{p}, \boldsymbol{t}_i, \boldsymbol{t}_j), \boldsymbol{\theta}_\gamma(\boldsymbol{p}, \boldsymbol{t}_i, \boldsymbol{t}_j))$$

This is the factor graph expansion for graphical model in figure 3.6.

Appendix C

Complete list of system hyper-parameters

C.1 Game-specific hyper-parameters

The following hyper-parameters have to be adjusted for every game:

1. Size of the vector representing player skills.
2. Size of the vector representing team skills.

C.2 General hyper-parameters

The following hyper-parameters are related to the system in general:

1. Optimisation algorithm and parameters of chosen algorithm.
2. Stopping condition for optimisation algorithm.

Appendix D

Project Proposal