Minna Wahlroos

ESports match analytics

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and communications technology

Bachelor's Thesis

20 November 2018

Metropolia

Author Title Number of Pages Date	Minna Wahlroos Esports match analytics 64 pages + 1 appendices 20 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Media Technology
Instructors	Aarne Klemetti, Researching Lecturer

The purpose of this thesis, was to research the predictability of competitive sports, and to develop a software system to gather and analyse said predictability. The final goal is to find correlating features in available data.

A video game (League of Legends) was picked as a sample subject, due to its well documented API and the author's previous experience with it. During this thesis project, a system was built to continuously gather match data that was later on transferred to another environment for data analysis. Both phases were documented in the thesis.

The system was built using Django framework (for rapid prototyping), and data analysis was performed using Jupyter Notebooks (for portability). Data was transferred via compressed PostgreSQL table dumps.

After an extensive exploratory data analysis as well as binary classification using neural networks, the conclusion was that with the available data is it impossible to predict match results accurately (beyond 55% accuracy on a hold-out test dataset).

This work could be used as a foundation for future research on the topic of match analytics, for example on predicting an individual participant's behaviour which was not performed in this work.

Keywords	e-sports ETL predictive analytics

Tekijä Otsikko Sivumäärä Aika	Minna Wahlroos E-urheilun otteluanalytiikka 64 sivua + 1 liite 20.11.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	mediatekniikka
Ohjaaja	tutkijaopettaja Aarne Klemetti

Insinöörityössä selvitettiin joukkuepohjaisten e-urheiluottelujen tulosten ennustamista käytännön esimerkin avulla. Työn kohde oli videopeli League of Legends. Tavoitteena oli ennustaa ottelun lopputulos mahdollisimman tarkasti käyttäen aiempaa (ottelun alkua edeltävää) tietoa. Vaihtoehtoisesti työ pyrki selvittämään, onko tämä ylipäätänsä mahdollista työssä toteutetulla tavalla.

Työn alussa aihealue rajattiin tiettyyn osalohkoon kyseistä videopeliä ("Solo-/Duo Ranked") ja tästä osalohkosta listattiin saatavissa olevia tietovarantoja. Näiden tietojen ominaisuudet kartoitettiin, niille haettiin keskenäissuhteita, ja lopuksi ne mallinnettiin muodollisesti ohjelmallisesti käsiteltävään muotoon (Entity-Relationship- eli ER-malleina).

Toteutus jatkui rakentamalla ensin tiedonkeruuta toteuttava palvelinohjelmisto Python-ohjelmointikieltä käyttäen. Ohjelmistokehityksessä ensin valmistetun prototyypin pohjalta tehtiin lopullinen toteutus. Tätä varten tehtiin taustaselvitystä prosessien tausta-ajosta ja rakennettiin monitorointia varten tiedonkeruun seurantapaneeli. Lähdekoodi pidettiin johdonmukaisena ja päätelaiteriippumattomana Git-versionhallintaohjelmistoa käyttäen.

Kun tiedonkeruu saavutti muutaman sadan tuhannen tietueen lukumäärän, ryhdyttiin suunnittelemaan ja toteuttamaan analytiikkaa. Työn lopuksi tietueet visualisoitiin ja toteutettiin syvä neuroverkko, joka tuotti 55 %:n ennustustarkkuuden. Tarkkuutta pyrittiin parantamaan hakemalla lisää dataa ja jalostamalla olemassa olevaa dataa heuristisin menetelmin.

Lopputuloksena ei saavutettu riittävää ennustustarkkuutta, jotta se oikeuttaisi toiminnan (oli toiminta sitten vedonlyönti tai ottelun kesken jättäminen). Vaikka tämän voi laskea epäonnistumiseksi, työstä saa käsityksen ihmispelaajien ennustettavuuden vaikeudesta.

Jatkotyö voisi keskittyä yksityiskohtaisemmin yksittäisen pelaajan tulosten ennustettavuuteen.

Avainsanat	e-urheilu tiedonkeruu analytiikka

Table of Contents

Abbreviations

big data	Large data sets, of size where traditional methods in storing and analysing (e.g. Excel spreadsheets) are ineffective.
esports	Electronic sports. Competition in video games on a professional level, with fiat money prizes at stake (often provided by event sponsors).
API	Application Programming Interface. An (often HTTP based) interface, which provides programmatical access to third-party resources.
MOBA	Multiplayer Online Battle Arena. A category of video games, where the players work in teams of equal numbers, to battle other teams in an area with fixed borders.
Elo-rating	A measurement of the relative skill level of a player (relative against the skill levels of other players), created by Arpad Elo.
(rank) ladder	A figurative "ladder" (ordering) of players or teams, set in an order based on a quantity which measures their rank.
(object) class	A data structure, to contain data and (in programmatic context) a set of subroutines to communicate with the data. A fundamental building block of object-oriented programming.
foreign key	An address to a dataset within a relational database, avoiding duplication of data. A key concept in relational databases.
ORM	Object-Relational Mapping. A utility to provide programmatic interface for interacting with a relational database using objects, abstracting away the need for explicit SQL queries.
UML	Universal Markup Language. A standard for several diagrams and their related (often repeatedly used) symbols, altogether describing a system from certain point of view (depending on diagram).

1 Introduction

1.1 Preface

The topic of this thesis is researching applied predictive analytics on big data, and methodologies to implement such a stand-alone system. This is simultaneously a software and a research project, with the focus on research.

The aim of this research is to find out whether it is possible to apply processes from statistics, on big data from competitive sports (namely esports), in order to create predictive models that outshine existing naive models. In order to reach this aim, we design and build a system for gathering data, analysing it, and providing insights on it. The subject for the practical part of this research is a popular (Volk, 2016) video game "League of Legends", and its historical match records.

This thesis does not involve an essential client company, but should the research succeed, it is possible to be practically utilised (e.g. gambling, sports training). In this case it has at least theoretical monetary value.

People constantly make presumptions and predictions of competitive sports, both in the context of everyday conversations, as well as in the context of financial interest (i.e. betting). For example amongst ice hockey fans, the topic of the day could be *"Will team Ilves or team Tappara win today's ice hockey match"*. In the aforementioned situation, some fans may base their prediction on individual(s) of a team (i.e. *"Player X recently joined Tappara and that's why they are stronger than Ilves"*). On the other hand some other fans could guess based on naive interpretation of statistical tables (i.e. *"Ilves is on a winning streak, and that's why Ilves will surely win"*). Sometimes people even go as far as to try earning money illegally with match-fixing. There are numerous examples of this, for example the year 2011 scandal in Finnish football (BBC News, 2011).

The same phenomenon is nowadays also apparent in online computer and console games; Both in the casual and the professional gaming, latter also known as "esports" (short for "electronic sports"). These matches include the same presumptions and pre-

dictions as physical sports, mentioned in the previous paragraph. They may become topics of everyday conversations (especially amongst young generations) and people do betting on the match results. As betting gets involved, it induces match-fixing. Examples of this include year 2013 Starladder tournament in video game Dota2 (PCGamesN, 2013) and year 2015 CEVO tournament in video game CounterStrike:GO (Valve, 2015).

In addition to the potential monetary value, this thesis has theoretical benefits to the people playing the subject video game, League of Legends, be it a casual or a professional player. The research may also assist those who like making intuition-based predictions on competitive sports (in more generic manner - not explicitly esports), giving them means to evaluate their own predictions' statistical validity.

This specific topic was chosen, because predictive analytics is a major part of machine learning, which on the other hand has become an increasingly trendy topic and according to Gartner's estimations is in its peak of inflated expectations (Panetta, 2017). The topic is also relevant from a financial perspective, because should the hypothesises turn out true, their effect on gambling industry could be revolutionary.

In addition to former, a more practical reason behind this specific topic (and the subject), is an existing API access to the subject (League of Legends) video game's data, provided by the game publisher (RiotGames), who does not endorse this research in any way. This API access enables fetching match related data and utilizing the data models described in the game publisher's API documentation (Developer.riotgames.com, 2018a). Aforementioned fact makes data gathering a very streamlined process, leaving more time and resources for the other parts of research (further data modelling, data analytics, conclusions).

The research problem in form of a question is the following, *"How accurately can we predict the binary outcome (win or loss) of match results, in a situation where statistical data from past matches is known?"*. We approach this problem by determining the variables affecting a given match's outcome, and then find the relationship between a given variable and outcome (e.g. "correlates", "does not correlate"), and finally making predictions based on these findings. However, we must keep in mind at all times that correlation does not imply causation.

I divide the aforementioned research problem into the following sub-problems, which we attempt to answer as the research progresses:

- "How do we model the data to a computationally processable form?"
- "How do we implement data gathering with respect to data analytics?"
- "How can we make predictions based on the gathered data?"
- "What are the most statistically significant variables affecting the outcome?"
- "How does the amount of gathered data affect accuracy of predictions?"
- "Are the predictive models, created during research, practically effective?"

Records pertaining to played matches are generated on a daily basis, and an essential feature of this data is, that there is "more data generated than is humanly possible to read". This creates the setting where using information technology to analyse the data is necessary in order to obtain statistical facts.

There are several software projects based on the same API, such as "League of Graphs" (Leagueofgraphs.com, 2018), but they solely aggregate the existing data to graph it as trends and tables, without attempting to make conclusions based on these. On the other hand, the primary objective of this thesis is to find causal connections from this data based on continuously accumulating (past and current) data, and use those to create predictive model(s).

Additionally, these existing projects are based upon the assumption, that "every player on a given (Elo-rating) tier are highly similar to each other" (i.e. "do not form subcategories within said tier -category"); A secondary objective of this thesis is to recognize each players' features that have a statistically significant relationship with either winning or losing a given match.

Should this research result in generic methodologies to create analytics, it might be possible to be applied even to other target domains than merely esports.

1.2 About subject

As mentioned in the preface, the subject of this research is the match data of a video game called "League of Legends". This is one of the few popular games in the category of MOBA games (Signup.leagueoflegends.com, 2018) that are played internationally on a professional level. The other professionally played MOBAs worth mentioning are Dota2 and SMITE. Many solely casual MOBA games also exist, as well as some with annual tournaments, but none of them are in the same order of magnitude as these three.

For a ballpark figure of said magnitude, League of Legends' annual "2017 Season World Championship" prize pool was $ 4,946,970 USD (Leaguepedia, 2017), and Dota2's annual "The International 2017" tournament prize pool was $ 24,787,916 USD (Liquipedia Dota2 Wiki, 2017).

However, the focus of this research is on the subject's match data. Because our subject game contains three distinct game modes that are competitively playable (each with its own Elo-rating based ranked ladder), we have to narrow down our target data. Our choice of these is the most renowned (and also the most recognized - more on that follows) game mode, "Ranked Solo/Duo". This mode is identified in the API by the identifier "420".

The reasoning behind choosing "Ranked Solo/Duo" as our focus, is that it is the most valued game mode amongst game's professional players (Blitz Esports, 2017) and according to numerous observations it is also the most popular game mode amongst game's streamers (Twitch, 2017).

2 Data modelling

2.1 Variables affecting the outcome of a match

The focus of our research ("Ranked Solo/Duo" game mode, ID 420) contained a lot of variables that affected the outcome of a given match, but we split all said variables into the following eleven rough parts:

- the 5x2 players (aka participants) in both teams (total of 10)
- the match environment

Additionally, we classified any individual variables, by the following division.

- varying, uncertain variables (e.g. individual participant's choices during the game, or randomly available resources in match environment)
- static variables (e.g. strengths and weaknesses of participant's character)

All known variables (based on former game knowledge, and skimming through the game's static data) from game version 8.1.1 were compiled to figure 1. Should these variables had changed during the research (due to major game version differences), the image would have been updated accordingly.

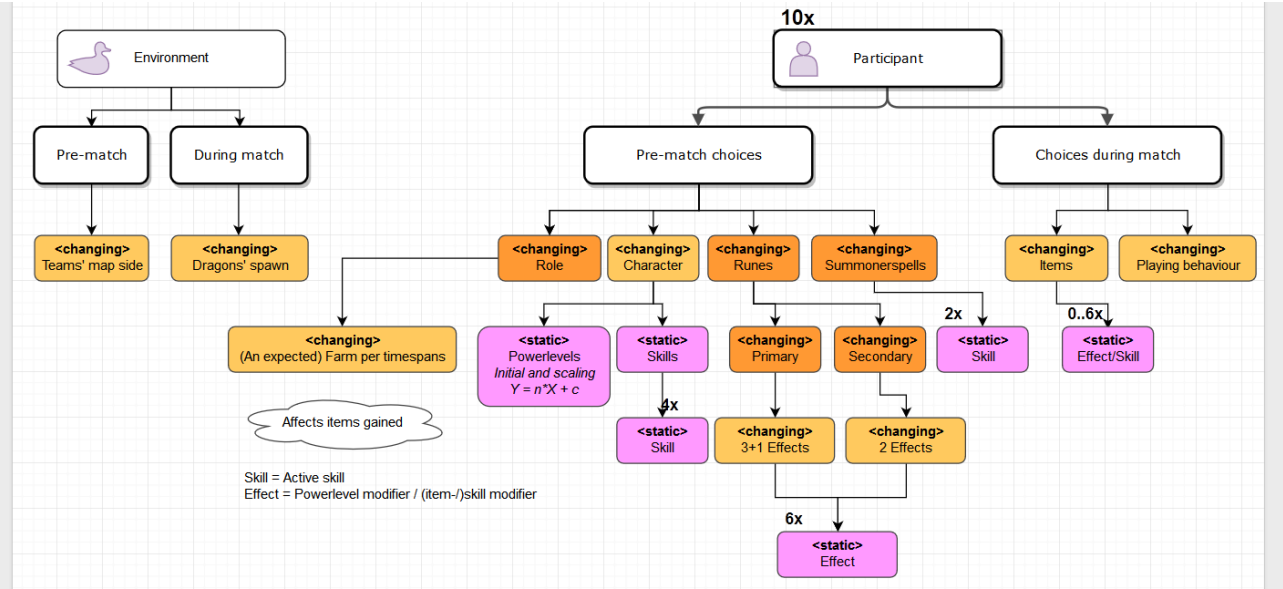


Figure 1. Known game variables affecting match outcome. Static variables in a purple hue.

Pre-match environment variables

The only environmental variable set prior to the match, visible in so-called "Champion Selection" screen, is the side of the map each team starts from. While the game map is very much mirrored on both sides, there are some notable differences.

Imagining the map as a square cartesian plane, where teams start from opposite corners (e.g. $[x_{min}, y_{min}]$ vs. $[x_{max}, y_{max}]$), two of map's important secondary objectives (Baron Nashor, and Dragon) spawn halfway to the other two corners (e.g. $[x_{min}, y_{max}]$, $[x_{max}, y_{min}]$). These spawns are static, and so is the terrain surrounding them (more easily accessible for each team respectively). They were shown in figure 2 as bright red filled circles.

Figure 2. Teams' starting points, and two major secondary object spawns shown as red circles. © Riot Games (http://ddragon.leagueoflegends.com/cdn/6.8.1/img/map/map1.png)

Pre-match participant variables

As players accept an invitation to a match, they are assigned to one of five available roles. The game server attempts to prioritize those roles, which each player has preferred (having selected two preferred roles prior to queueing for the match), but often times there are more people preferring one over the other, so people are thrown off

from their preferred roles. Most often a specific role is mapped to a specific lane, but exceptions to this rule do exist. The role together with the lane determines the location of the map where the player is initially supposed to go, as well as the resources that are dedicated to them. The role also implies a kind of setup (combination of selectable character, runes, and summoner spells) the player is assumed to choose, however in practice this is often not the case. For the aforementioned reasons, role of a participant mostly affects the number of choices available to that participant during the game (namely earlier purchasable items and upgradeable skills).

Each player must choose a character they play during the match. The game contains over 130-character choices (Na.leagueoflegends.com, 2018a), but these are limited by "bans" as well as the uniqueness of a character (disallowing duplicates of a single character in a match). Initially before choosing, each participant has an optional choice to ban (as in remove) one of the characters, making it unavailable for everyone in that match. After the bans are set, each participant has a randomly chosen turn (alternating between teams) to select his or her character, out of non-previously chosen characters (conforming to aforementioned rule of uniqueness). It is relatively common for players to have their favourite character either banned or chosen first by someone else, in which case they have to select another one (or forcibly cancel the match creation, by leaving the game lobby before everyone has their pre-match choices set, resulting in penalties to the leaving player). Each character has a distinct set of four skills, one passive effect, initial power levels and scaling power levels.

In addition to the passive effect and the four skills provided by the character of choice, each player has to choose two from a total of nine summoner spells. Pictures of each of nine summoner spells were displayed in figure 3.

Figure 3. Available summoner spells as of game version 8.1.1. © Riot Games (from game client)

Summoner spells may be chosen irrespectively of other participants' choices and are always available. The order of choices does not matter, therefore using combinatorics we calculated the number of distinct summoner spell choices as a number of 2-combin-

ations of the set with nine elements (one for each distinct choice), which resulted in a total of 72 combinations as additionally formalised in eq. (1).

$$\binom{9}{2} = \frac{9!}{(9-2)!} = \frac{1*2*3*4*5*6*7*8*9}{1*2*3*4*5*6*7} = 8 \times 9 = 72 \quad (1)$$

The last participant specific pre-match variable is a set of runes. Runes are passive benefits that manifest as (conditional or unconditional) power level increments, effects triggered by specific conditions (e.g. "every 30 seconds your next attack is empowered by X factor"), or extra items.

All available runes belong to one of five categories, of which player is required to choose two. The order of aforementioned two ("primary and secondary") categories matters, granting a different passive effect to the participant. Pictures of these five categories were placed side-by-side in figure 4.

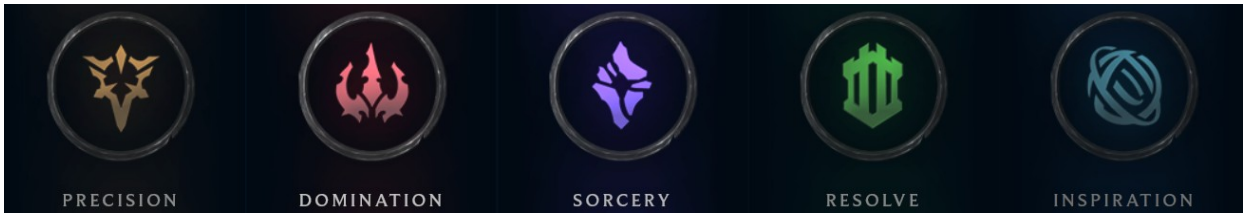


Figure 4. Available rune categories as of game version 8.1.1. © Riot Games (from game client)

Having chosen the two order-specific categories, the player is required to choose individual benefits from both categories. From the primary category, player has to choose four benefits, one from each group of three (displayed visually in the left side of figure 5). From the secondary category, player has to choose two effects, one from each, of two out of three groups (displayed visually in the right side of figure 5).

Figure 5. Available individual runes from primary precision and secondary inspiration categories as of game version 8.1.1. Each empty circle denotes an available choice to be made. © Riot Games (from game client)

Using combinatorics, we calculated the number of distinct primary+secondary category combinations as 2-variations of a set with five elements, resulting in total 20 variations as per the formula in eq. (2).

$$\frac{5!}{(5-2)!}=\frac{1\times2\times3\times4\times5}{1\times2\times3}=4\times5=20 \qquad (2)$$

On the other hand, we calculated the number of choices in a primary category to $3\times3\times3\times3=81$ alternatives, since those choices are fully independent of each other (i.e. choosing one does not limit the remaining choices).

In secondary category, there are three groups of three choices, of which two groups have to be chosen (irrespective of order), and from each of those two groups, one of three benefits. We calculated the number of ways to choose two groups as 2-combinations of a set with 3 elements, amounting to a total of 3 combinations as formalised in

eq. (3). These two groups both require selecting one of three options, totalling to 9 alternatives as formalised in eq. (4). Since the choice of groups is independent of choosing an option from the group, the total number of alternatives in the secondary category is 27 as formalised in eq. (5).

$$\binom{3}{2}=\frac{3!}{2!(3-2)!}=\frac{1\times2\times3}{1\times2\times1}=3 \qquad (3)$$

$$3\times3=9 \qquad (4)$$

$$3\times9=27 \qquad (5)$$

Choices in the secondary category are independent of the primary category, and therefore each primary+secondary combination includes 2187 alternatives as calculated in eq. (6).

$$81\times27=2187 \qquad (6)$$

Adding this to the number of distinct primary+secondary categories, we have a total of 43740 alternative sets of runes, as summarised in eq. (7).

$$20\times2187=43740 \qquad (7)$$

To recap all of this; We concluded that pre-match participant variables include one of five unique roles (1 of 5), one of over 130 unique characters limited by possible 10 bans in addition to the uniqueness constraint (1 of 130-[0..10]), two of nine summoner spells resulting in a total of 72 alternatives (1 of 72), and a set of runes from 43740 alternative combinations to choose (1 of 43740).

Mid-match environment variables

During an ongoing match, the only environmental variable we can keep track of (based on all available data) is the type of dragon which spawned. Dragon is a secondary objective, granting a permanent buff to the entire team obtaining it. The specifics of this buff vary per the type of dragon, which is randomly one of four possible types. One team obtaining the dragon means that another dragon of random type will re-spawn (until 35 minutes of the game, after which the spawn will be a special type of dragon, which is always the same from match to match).

Mid-match participant variables

Mid-match participant variables consist of each participant's purchasing choices and other playing behaviours.

During the game, each participant gains a varying amount of a resource ("gold") that can be used to obtain items. Items' effects vary, but generally, they can be divided in three: Effects that increase current power levels, effects that grant a passive effect (e.g. "every attack slows the target enemy"), and active effects (e.g. "grants a momentary shield on activation"). An item may include one or more effects of varying kind.

Different characters benefit more from different items, and often a good indication of a given item's effectiveness is, whether the character's skills benefit from the type of power levels increased by the item. Therefore all players playing a specific character tend to buy more or less the same set of items over the span of each game. Despite this, the impact of both, minor variations and the order of purchases, can be huge from match to match (e.g. a low-cost one-time use item that makes the participant temporarily invulnerable, but unable to act, may be useless if never utilised, but on the other hand may prevent an unlimited number of offensive acts done towards the participant during that temporary period of time). The optimal purchases also depend on other variables, such as the type of enemy damage that must be defended from.

In addition to item purchasing choices, there is each participant's playing behaviour. The playing behaviour refers to patterns in movement and skill activations from match

to match. This is recorded minutely in match timeline, and therefore we cannot obtain very accurate biometrics (movement and skill activations happening every second), unlike the game's publisher (Boards.na.leagueoflegends.com, 2017). However this behaviour (combined with all other variables) results in events such as "player kill", "player death", and "reaching an objective". These events are recorded as they happen in match timeline, and therefore we can (at least try to) deduce participants' playing behaviour, based on events they are part of.

Aggregate metrics

All aforementioned variables may affect the outcome of a given match, but in addition to the binary classifiable outcome, we obtained other participant specific metrics from the API such as each participant's "total damage dealt to enemy participants", "total healing applied to friendly participants", "total damage dealt to primary objectives", etcetera. We may interpret these metrics as aggregates of pre-match and mid-match variables, as they have a causal relationship altogether. Alternatively, we may ignore these metrics if our initial results deem them useless.

As for environment-related variables, some specifics of these are included in available data, namely the number of total secondary objects obtained by each team, and the team which was first to obtain each distinct secondary objective.

The way these (namely participants' aggregate) metrics map to individual variables, and to victory, were summarised to figure 6. While the sum of participant's pre-match and mid-match variables remain one and the same, they have multiple interpretations depending on the context. Figure 6 shows them in three contexts: As a part of team's capability to obtain an objective, as representations of an individual participant's choices, and as a part of team's overall strength (as in each team member's individual contribution towards victory).

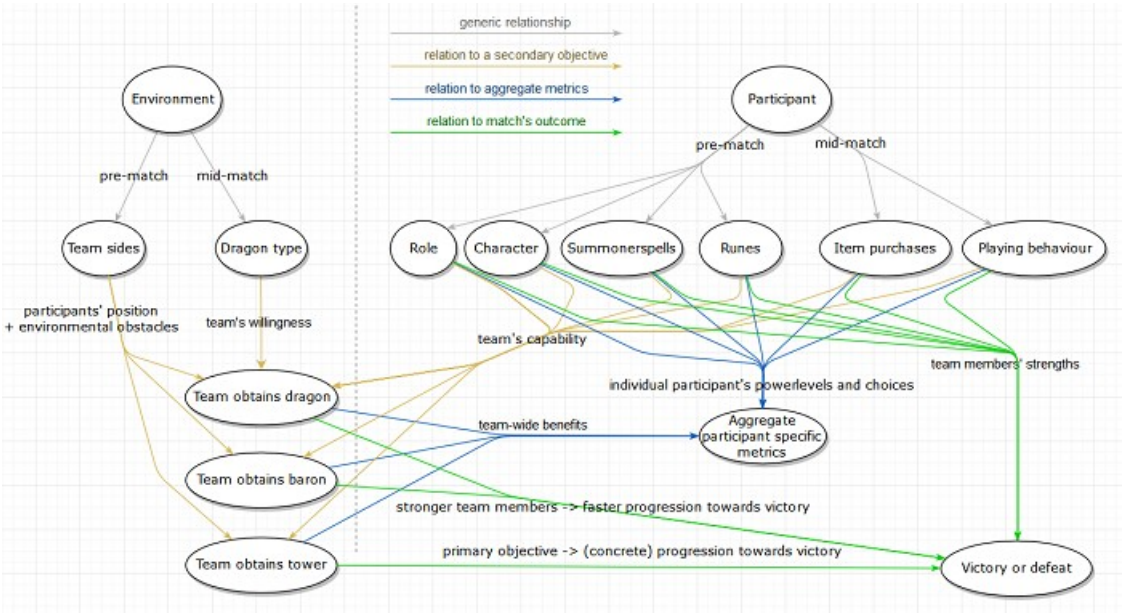


Figure 6. All variables' relationships to objectives, aggregate metrics, and match's outcome.

Relevant (API) response classes

Most importantly, we obtained the binary outcome of each match from the match result data. Match result can be found in the object class MatchDto under ParticipantStatsDto class (Developer.riotgames.com, 2018b).

Additionally, we found the aggregate metrics described in chapter 2.2.6, from ParticipantStatsDto and TeamStatsDto (included in MatchDto) for participants and teams respectively.

Third and final set of variables we found in MatchDto (under class ParticipantDto) is the pre-game participant variables (role, character, summoner spells, and runes). These were represented as numerical identifiers (generally integers) in the data, and have to be separately mapped to human readable representations using Static Data API endpoints during data analysis (Developer.riotgames.com, 2018c).

The rest of variables were all found in class MatchTimelineDto (Developer.riotgames.com, 2018d).

Coordinates of each participant are recorded to MatchTimelineDto on a minute basis. We find teams' side on the map by deducing it from each participants' coordinates at the first minute of the match.

The type of dragon that spawned is not recorded at all, but each event where one of the teams obtains it is recorded. Therefore we can keep track of all but the last dragon's type. However, this last dragon should not affect much on our analysis, as the unobtained dragon means neither of the teams had its effect (and so it did not affect the match's outcome).

Same applies to all other secondary and primary objectives. We find them as individual events in MatchTimelineDto, along with timestamps of when they happened.

Regarding participants' mid-match variables, we found a detailed series of every item purchase in MatchTimelineDto's events (including the timestamp of when the purchase took place). On the other hand, participants' movement is recorded only once every minute, and skill activations are not recorded at all. Instead of relying on this non-existing movement and skill activation data, we can deduce participants' playing behaviour based on "player kill" events they took part in, correlating those events with the timestamp of when they happened (e.g. "many kills early in the match, many deaths later in the match").

2.2 Inconsistencies of mapped properties

In an ideal world (from the perspective of data analytics), every participant in every competitively played match would be in a constant fit to play. In reality this is not the case. We have to live with that, and consider ways to handle the resulting exceptions.

Cases that may misguide our analysis (to a varying extent) fall primarily within one of two categories; Either the participant is "more effective than he or she should be", or the participant is "much worse than he or she should be" (to the point of being detrimental to his or her team).

Cases where the participant is so to speak "too good", happen when the participant is either cheating (which may skyrocket his or her positive metrics), or the participant is not the real owner of that account but someone "helping him or her to climb ranks". Although cheating in video games is a saddening matter, we cannot help it and so we concluded these the same way as we concluded "simply good players". The reasoning for this is that statistically speaking these are difficult to distinguish from very successful players who are climbing ranks on their new account (in a lower initial ranking). Any spare time analytics on these matters are carried out outside this research, and are not included in the scope of this research.

On the other hand, cases where the participant is much worse than usual are more easily detected due to their nature. Leaving out the cases where a participant is simply sleepy or distracted (resulting in a below average performance), possible reasons for greatly underperforming include, the participant being utterly drunk (or otherwise suffering from a major cognitive impairment), and another, relatively common, case of intentional will to lose a match (to cause a loss to specific teammate(s) in the process). Latter case may be a result of a prior disagreement with the teammate, or a general will to cause grief to others. These are supposedly outliers (we didn't have the time resources during this work to delve deeper into this subject) in the grand scheme of things, and thus we naively included them in our data as well as the previously mentioned "too good" cases.

2.3 Choosing database

To persist our initial data, as mapped in chapter 2.2, we chose to use a relational database system. An alternative would have been a distributed processing system such as Hadoop. The reason we used a relational database is that many parts of the data are of relational nature, even though the number of individual datasets (per each historical match) was expected to be relatively big (in tens of thousands, based on rough estimations using permitted API quota and available processing power). For instance, every match has it's own game version (that increments as the game updates, but between the updates it stays the same for a number of games), which consequently was put in it's own GameVersion table, and only referenced as a single SERIAL type FOREIGN KEY (Postgresql.org, 2018a) in each row of HistoricalMatch table. The entity-relationship diagram in figure 7 illustrates this.

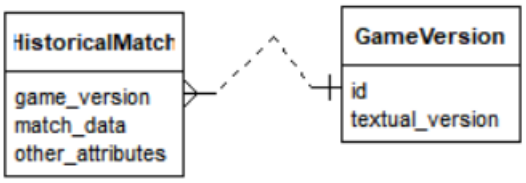


Figure 7. ER-diagram of each match's relationship to the game version.

Most common non-proprietary relational database systems include MySQL, MariaDB, and PostgreSQL. We chose to use PostgreSQL since it processes data in a more robust manner, constrained by types and checks set by the user (Cybertec, 2017). The trade-off is between data integrity and speed, and here we err on the safe side.

Looking into future, PostgreSQL supports table sizes up to 32TB (Postgresql.org, 2018b). If we had (somehow) been able to gather tens of terabytes of data, then we would have had to consider moving on to a distributed solution such as Hadoop, but in this work PostgreSQL was sufficient.

2.4 Data model

As previously declared in chapter 2.1, the focus of this research was the continuously increasing match data. In addition to the datasets enumerated in chapter 2.2.7, we utilised Summoner data (representing each player). This was necessary for automated data gathering to recognise an already observed player from a new player (for recursive "tracking" of players). More on this later in the implementation of data gathering. Figure 8 shows how relations in the database were implemented.

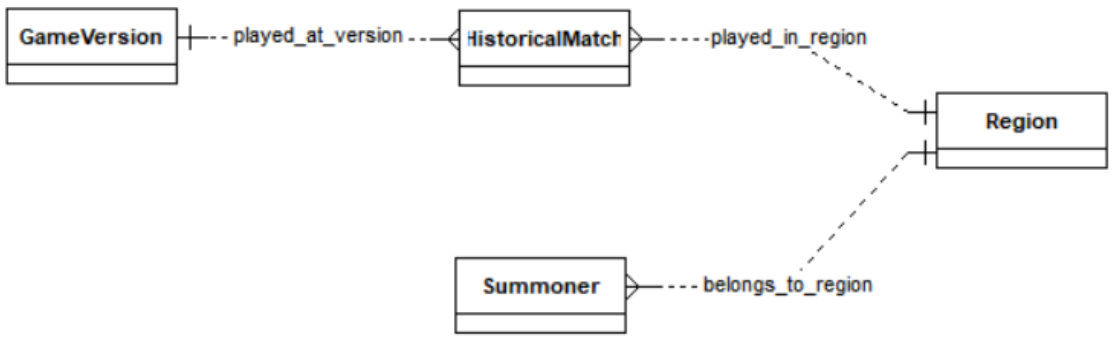


Figure 8. ER-diagram of datasets stored in the database.

3 Data gathering

3.1 Prototype

Rationale for prototype-first

An ideal workflow when creating a new kind of system (with no existing described way to implement one) is to create a proof of concept (to prove feasibility of the system), then to create a prototype (to benchmark and troubleshoot potential issues), and finally to construct the final implementation based on the prototype. In reality however, this is often not possible, due to time and resource constraints.

In this research, we already had an interactive user interface provided by the game publisher RiotGames (Developer.riotgames.com, 2018a) that replaced the need for an explicit proof of concept. Using this web interface we can retrieve samples of the data (to inspect obtainable datasets and their properties), to construct the data models prior to using them (relying on that they match the models described in the game publisher's API documentation).

The reason we still made a prototype (instead of rushing to the final implementation) was that using a prototype we did the following:

- Validated our data models. (Persisting new data and retrieving persisted data.)
- Found out errors, and a good enough (initial) abstraction for error handling.
- Experimented on various abstractions (HTTP, program states) by trial and error, in preparation for the final implementation, that must have at least some automation (and therefore be independent of user interaction, should errors occur).

The primary objective of our prototype was to ensure consistency of the APIs (as per documentation that is referenced in chapter 2.2.7). For each error or inconsistency encountered, we adjusted the software appropriately so the final implementation could run automatically without the need for constant human supervision. A successful (final) execution of the prototype was an errorless program flow, resulting in a logically sensible outcome.

For example, inputting a non-existing username should prompt for a new username, while inputting a valid username should provide the winrate of that player (even if 0 games played, logically resulting in a 0% winrate and **not** a DivisionByZeroError or so).

Our data gathering prototype does the following operations:

- Two inputs are passed to the prototype: An existing player's username and the region of said player.
- The prototype retrieves the respective Summoner ("player") dataset and persists it if it doesn't already exist in the database.
- Using the accountId -property of the Summoner dataset, prototype then retrieves that specific player's 100 most recent matches in game mode 420 ("Solo/Duo Ranked", as described in chapter 1.2).
- Finally, the prototype finds the boolean outcome of each match respective to the player (i.e. victory or defeat) and calculates the player's win percentage based on these matches.

(Initial) Database schema implementation

The game publisher's API returns all data in a valid JSON format, that is natively supported by most programming languages (Developer.riotgames.com, 2018e). We used Python language for this work because it is a familiar language and a lot of data science these days is performed using Python.

The datasets fetched from the game publisher's API are saved for the purpose of re-using them at will, without doing unnecessary repeated queries (straining the API).

Since the data is highly relational (as described in chapter 2), we save it to a relational database after retrieval. We were going to use a database connector in any case, but additionally we used an Object-Relational Mapping library. ORM was useful because the data structures did not change radically between prototype and final implementation, but it abstracted that away, removing the threat of duplicating database related code. The ORM for this project was Django Framework's model library (Docs.djangoproject.com, 2018). Reason for choosing specifically this ORM was the familiarity of it to the author of this research.

The produced data models, found in project source code (GitHub, 2018a), were normalized with respect to distinct datasets retrieved from game publisher's API (e.g. "Player's account" versus "A historical match"), but the data within each dataset was not normalized per sé (e.g. "A historical match" contains names of players involved, and these are not normalized). We also saved the duration of the match in addition to the game version and the region, so that we can filter away invalid (under 6 minute) matches which occur when one or more participants fail to join the match (usually due to connectivity issues).

Normalization of these (datasets' essential attributes) was implemented up to the Third Normal Form. An ER-diagram representing the subset of datasets used in the prototype is described in figure 9.

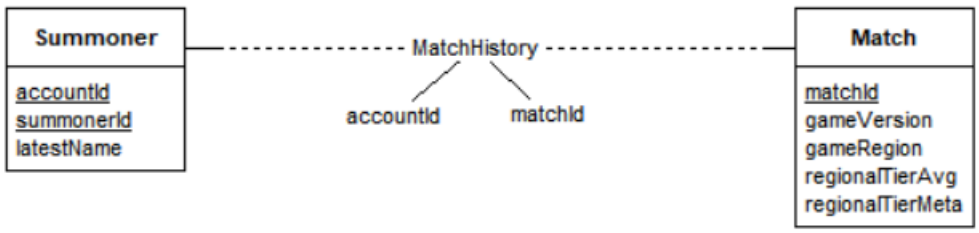


Figure 9. ER-diagram of prototype's datasets.

Program exceptions

To minimize the need for unnecessary manual work, we also prepared for possible errors that software may encounter. While we cannot account for every possible error scenario, we enumerated the most likely errors and created handling routines for those events.

As is a good software practice, we implemented these in software code using try-catch control structures. This way we keep the business logic concise (only throwing an exception in case of an error) yet modular (with independent components that each have their own possible exception cases - respecting the single responsibility principle).

Most likely handleable error scenarios (some of which were found during initial testing of the half-made prototype) included:

- Exceeding rate limit(s), a maximum number of requests permitted in a time frame
- Interface unavailability (non-rate-limit caused HTTP errors in application layer)
- Duplicate database inserts (if two or more parallel processes find, and try to save the same dataset in database simultaneously, also known as "concurrency")

In addition to aforementioned handleable error scenarios, we also had scenarios that could not be handled with simple program flow changes, and would take too much of effort to implement as full blown features versus handling them manually (i.e. very rare occurrences). However, we could (to some extent) prepare for these with technical methods, namely benchmarking. These scenarios and their respective solutions included:

- Out of memory -scenario, which we could prevent to an extent by benchmarking the final software's memory consumption, and calculating the maximum number of parallel processes.
- Internet connection outage, which we cannot prepare for (the computing platform is a non-dedicated cloud virtual machine), but we could monitor it using log files.
- Power outage, which we also cannot prepare for at all (the computing platform being a non-dedicated cloud virtual machine).

Finally, if the game publisher was to deny access to the APIs, that would effectively had made this research impossible to finish. Also, the data models could change drastically in the next season update (that is per every year), so we most likely had a few months' time to finish this research with the current revision of data models.

Program flow

As earlier described, the handleable program exceptions are separated from business logic using try-catch clauses and throw-able exceptions. Altogether these formed a stateful program flow, that is best described using an industry standard UML (activity) diagram. This is essentially identical for every API request and displayed in figure 10.

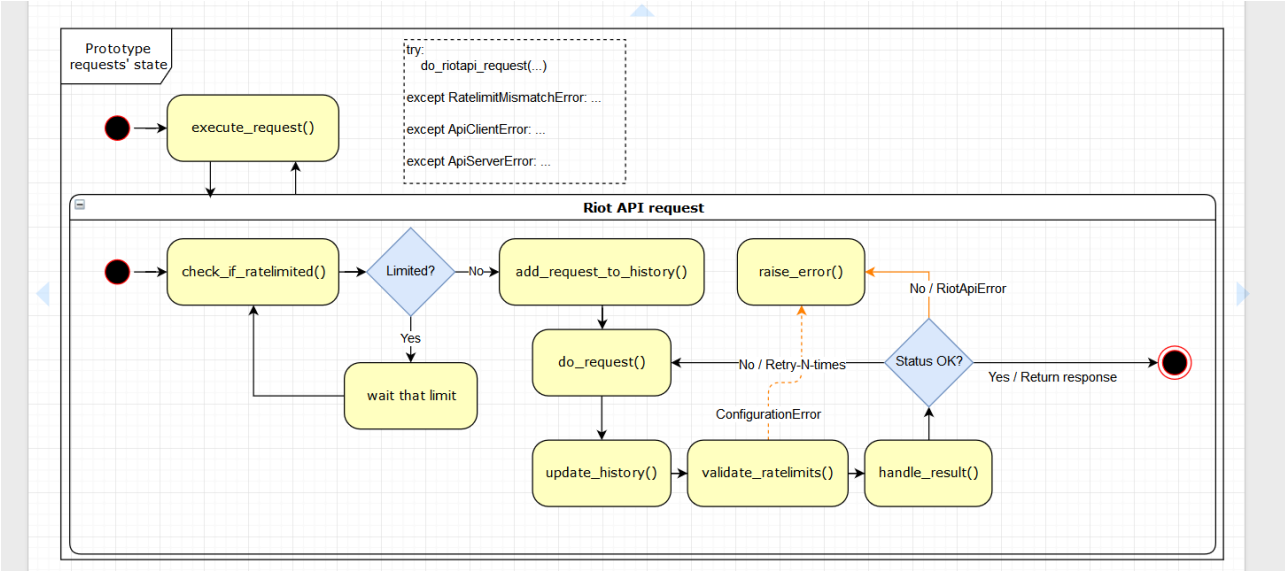


Figure 10. Prototype's program flow when requesting data from API, emphasising error handling

Running the prototype & conclusions

Having run the prototype a few times, we concluded that the data models are consistent throughout hundreds of consecutive API calls, and that errors do occur but get caught successfully by error handling. This meant we could continue working on the final implementation of data gathering. Prototype source code was saved to a GitHub repository (GitHub, 2018b), along with the final implementation.

3.2 Final implementation

(Final) Database schema implementation

Since the data gathering was meant to operate over a long period of time, there were inevitably game updates while the data gathering is active. This research was planned to finish before the next major update (so our models are essentially possible to remain intact), but the minor game updates that inevitably occurred did change the balance between the variables shown in figure 6 (Aggregate metrics). Therefore, we had to account for game version when doing analysis on historical matches, and we saved this in the relational database as shown in figure 11. Region is treated in same manner as game version, since all tiers are regional and as of result the tiers of two regions may not match in their respective levels of play.

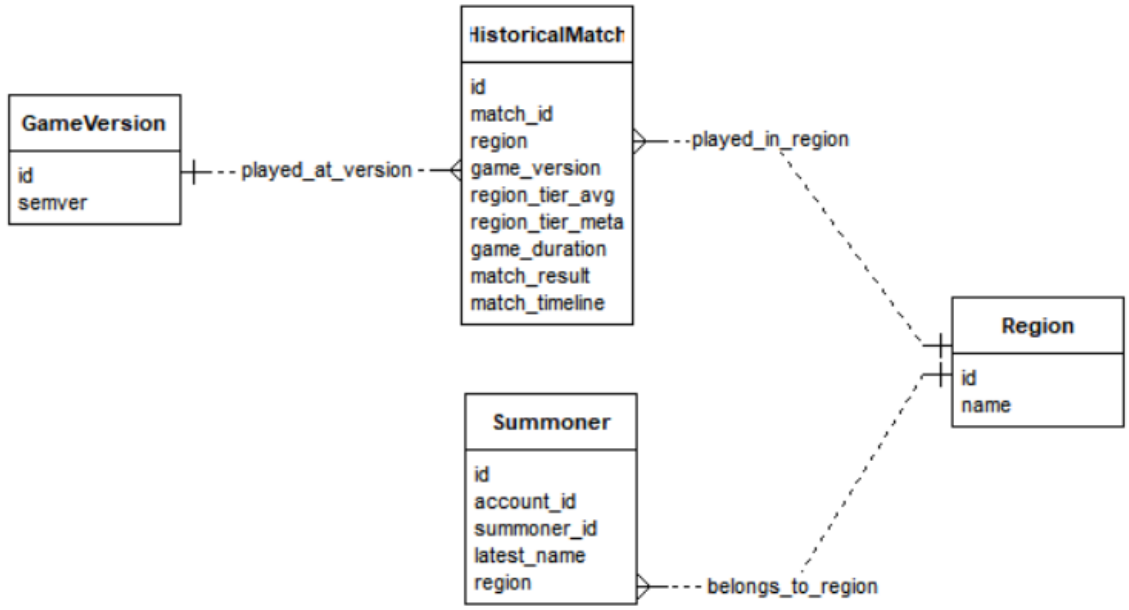


Figure 11. ER-diagram of the final data gathering database schema using Crow's Foot Notation.

Program flow

The final data gathering has three distinct program states, which are the following:

- **Input state**, during which the software will wait for user to manually input a player in active match (with the requirements of the subject - namely it must be a "ranked" match of type "420").

- **Mid-match state**, during which the software will fetch and save each participants current ELO-rating based tiers, in order to calculate the average tier where that specific match belongs to. After this the software will wait, polling periodically whether the match has ended.

- **Post-match state**, where the software will fetch and save the match results, and combine them to previously saved information (all of it stored in the database). Then the software will iteratively search if any of the participants have entered a new match (with aforementioned requirements). If a new match is found, then the software will automatically move to **mid-match state**, gathering more data. If a new match is not found within a certain time threshold (30 minutes), then the software will move back to **input state** requiring manual input to continue.

Overall, these states enabled a high level of automation, letting us have many parallel processes running while requiring a minimal interaction to control them. This program flow is more accurately depicted as an activity diagram in figure 12.

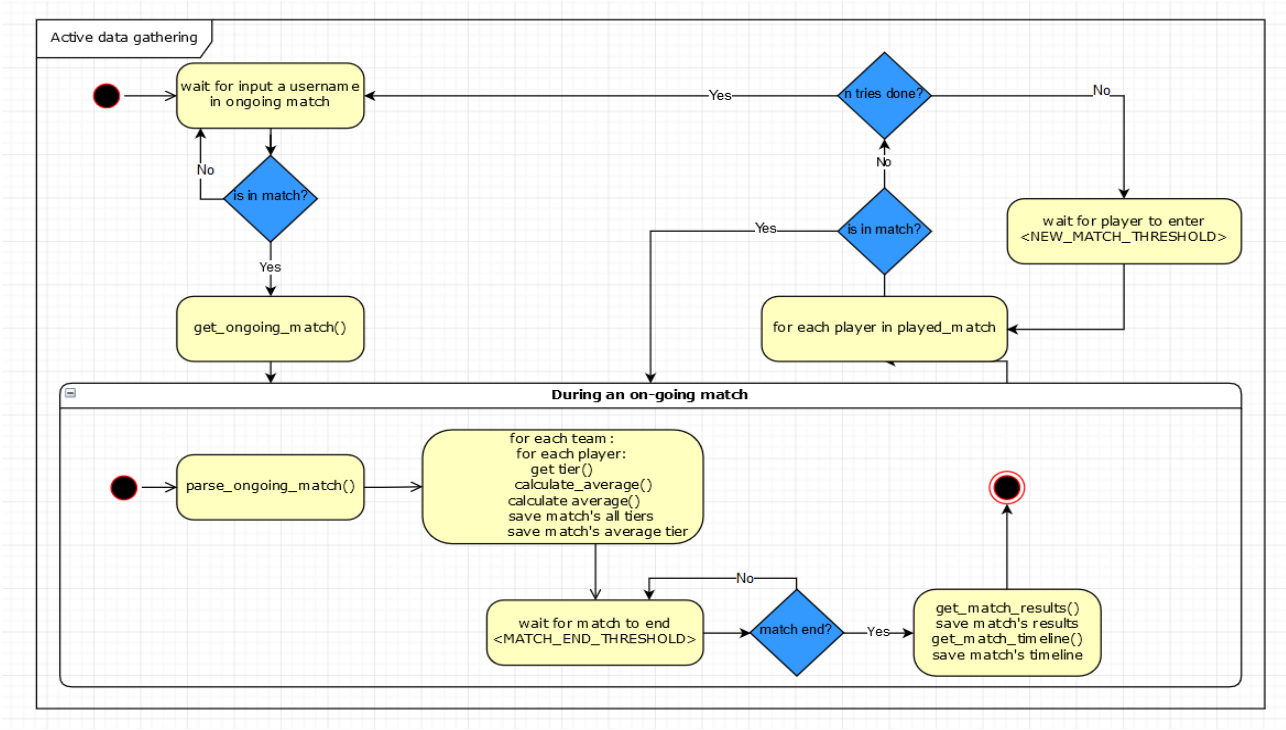


Figure 12. Final program flow, capable of gathering data for periods of time independent of user.

Since there are multiple instances of the software, all gathering essentially same sets of matches using same API, we needed a mechanic to ensure that no duplicate data is persisted. As we initially chose to use Django ORM as our data persistence library, this let us easily check "whether an ongoing match is already in database" (indicating it is observed by another process).

We also had to ensure that API restrictions are honoured across all simultaneously running processes. The API restrictions are based on generic "maximum N requests of type X within time-frame M seconds" rules, and must be checked prior to every API request that every process does. For this we maintained a separate database table, counting each and every API request that was to be initiated. This way we could simply query the database prior every next request *"whether we have reached the periodical quota, or if we may proceed with a new request (subsequently logged in database)"*. The only problem here is so-called thread safety, where multiple processes may access the same resource. We addressed this using table-level locking, where a process obtains an explicit write/read access to the table, denying other processes from accessing it (and instead, having them polling in a queue for the next access).

Unfortunately PostgreSQL does not support an absolute table-level locking (where other processes would be forbidden to use it under all imaginable circumstances). This was proven after an exhaustive testing (trying to make it work) on PostgreSQL. Therefore we deployed a separate (MySQL) relational database in order to achieve this. Since the feature was simple enough, we did not use an ORM library for this. Connection to the database was achieved by using MySQLdb (Mysqlclient.readthedocs.io, 2018) python library.

Process persistence, background processing

Because the properties of the subject of this research change in time (due to game updates), we had to collect data cumulatively throughout the entire research in order to have enough relevant (up-to-date) data at the time of analysis. This need translated to having multiple continuously running data gathering processes that persist as much data as our resources (API quota and available computing power) permits.

While the most naive solution to run parallel python scripts would be "having N interactive terminals open for N processes", we want to be able to communicate with each process individually, at will, from a single terminal, and that terminal has to have the ability to go offline. In other words, we wish to do background processing that has some sort of interface to respond to user input. Specific user controllable features we need are process termination (without need to utilise `ps` and `kill` commands) and text input related to business logic (when in input state as described earlier).

We can implement background processing on a Linux server using any of the following methods: cron, nohup, screen, and daemon. We separately evaluated the suitability of each of these methods, and then concluded what was most suitable solution for our use case.

Cron is a software based on a user specific list (also known as "crontab") containing scheduled commands. These commands are executed using user's shell (command-line interpreter) on times specified in the list, repeatedly (e.g. "8 o'clock in the morning" means the command will be ran every single day thereon at 8 o'clock in the morning). In our use case, using cron would have the following consequences:

- Processes could be scheduled minutely, but the schedule would be permanent (until manually changed). This is not suitable to our process that is meant to be running constantly (with highest uptime possible)
- Process could not obtain input from the user. This alone makes the method unviable. While we could pipe initial input as part of the command, it is a requirement to be able to pass input to the process during its runtime.
- Process would either shut down as per program logic, or it would need to be shut down using `kill` command. This is an additional inconvenience, as we wish to shut down as conveniently as possible (without need of searching the process ID for `kill` command).

nohup -command is an abbreviation of "no hang up". This modifies a process so the process disregards any "HUP" (short for "hangup") signal that is coming from the user's terminal. It is implemented by executing the process as usual with an addition of "nohup" prior the command and an ampersand after the command. E.g. for our "active_data_gathering.py" python script the syntax of executing said script with nohup is *"nohup python data_gathering.py &"*. The consequences of running our data gathering using nohup would be the following:

- Process keeps running regardless whether the terminal (used to start it) is closed or not. This is ideal for our use case, as we wish to be able to close the user terminal at will.
- Process could not obtain input from the user. Again this alone makes this method (like cron) unviable. We would be limited to only passing an initial input to the process.
- Process would either shut down as per program logic, or it would need to be shut down using `kill` command. This is not ideal to our use case, as we wish to shut down as conveniently as possible.

A user specific "screen terminal" is created using the syntax "screen -S name-of-the-terminal". It is virtually identical to a normal user terminal (created by user login through either an SSH connection or local physical connection), but has the ability to "detach" and "re-attach" at will from a normal terminal. Figuratively speaking one can think of it as a virtual overlay on top of a normal terminal, that has all essential properties of a normal terminal. One detaches from a screen terminal (to return to a normal terminal) either using the key combination CTRL+A -> D or by closing the normal terminal (e.g. by suddenly disconnecting an ongoing SSH session). One can return to the screen terminal at any time afterwards using the syntax "screen -rd name-of-the-terminal". Using multiple screen terminals to run our data gathering processes would have the following consequences:

- Process keeps running regardless of whether the normal terminal (used to start the screen terminal) is closed or not. This is ideal for our use case, as we wish to be able to close the user terminal at will.
- Process can obtain input from the user, through the screen terminal that can be re-attached to user's normal terminal of any sort. This is good enough for our use case, as we are able to obtain control of the process at will.
- Process can be shut down as any interactive process, sending an interrupt by the key combination CTRL+C (SIGINT signal) through the command line (as long as we have first re-attached to its respective screen terminal). This is also good enough for our use case, as we have the same procedure to execute whenever we wish to terminate the process (as opposed to searching the process ID with `ps` command to use together with `kill` command).

We could also utilise the C-language unistd.h library based "daemon() -function" to create background processes programmatically. This is also known as daemonising processes. These processes are irrespective of the user terminal that was used to start them (whether that terminal is closed or utilised in other ways) similar to nohup -command, but in addition a daemon process can be programmed to obtain input from the user e.g. utilising UNIX sockets (such as network interfaces).

The downside to daemonising processes is the added program code in order to daemonise the processes and to communicate with them (by whatever means chosen). Aside from this downside they would fulfil the necessary requirements.

In conclusion, the only methods that fulfilled our requirements were the screen terminal and process daemonisation. In principle, a correctly implemented daemonisation would have more room for customisation (e.g. having a local network-based command to terminate a given process, as opposed to re-attaching its respective screen terminal and sending an interrupt to it). Despite the potential benefits of daemonisation, we chose to use screen terminals because they are faster to implement and modify. Daeamonisation would be an ideal target of improvement (if the system was to be reproduced), but for this initial implementation we sticked with screen terminals.

Implementing monitoring

While our data gathering processes' program logic is highly automated, there may be unhandled errors as previously stated, as well as the need to pass manual input to individual processes. In order to keep our data gathering processes steadily running despite of these, we need periodical metrics and information on state of our processes (if a process is waiting for user input and therefore is not actively gathering more data).

First and foremost metric we needed was the utilised API quota per points in time. This is most convenient to be viewed as a time series graph. We can calculate each point in time using information from each process' log files, and we visualise it as an image using a python plotting library known as "matplotlib". Matplotlib is the de-facto generic use plotting library for python, and since our software is written in python this choice is a no-brainer. An alternative would have been to output the results of those calculations (points in time) as numeric data to an interactive graphical user interface (such as web-based user interface utilising javascript), but that would have required much more effort than a (good enough) image of a graph. This is again (as with process daemonisation) something that would be a viable target of improvement (if the system was to be reproduced).

The primary limiting factors being the permitted API quota and the available computing power, the second metric we needed was CPU and RAM usage per process in contrast with total available CPU and RAM. However, this monitoring is already implemented in Linux systems as an executable `top` (Man7.org, 2018) command. While we could have unified these metrics together with API quota graphs (e.g. producing graphs of them in the same destination), it would have been an unnecessary overhead since the CPU and RAM usage is to be expected to remain steady after an initial benchmarking (and less relevant after having reached the optimal number of parallel processes).

We could have had the metrics calculated as background processes (e.g. scheduled or daemonised) as we did with the data gathering processes, but an on-demand calculation was more appealing since it does not use processing power except when needed to. Having metrics calculated continuously would also had enabled us to create alerts (e.g. sending an email or an instant message) based on thresholds, but this was left as yet another possible improvement after the initial implementation (since the data gathering is not very sensitive to downtimes of even multiple days - as the matches are played constantly and a typical period of time between game updates is measured in weeks).

Since we already use Django framework (for its ORM modules), we combined it with an Apache web-server to generate a dynamic website with aforementioned graphs and numeric metrics. This way we were able to stick with Python, and generate everything necessary on a webpage load.

For fast prototyping we used a wireframe in order to create a minimal sketch of the web page contents. As opposed to a higher-fidelity mock-up, this wireframe was used to provide insight on feasibility of the page (i.e. "do all these metrics and graphs fit one page, or do we need tabbed panes, etc.") as well as granting us a blueprint to memorize and follow while building the web page. This wireframe is shown in figure 13.

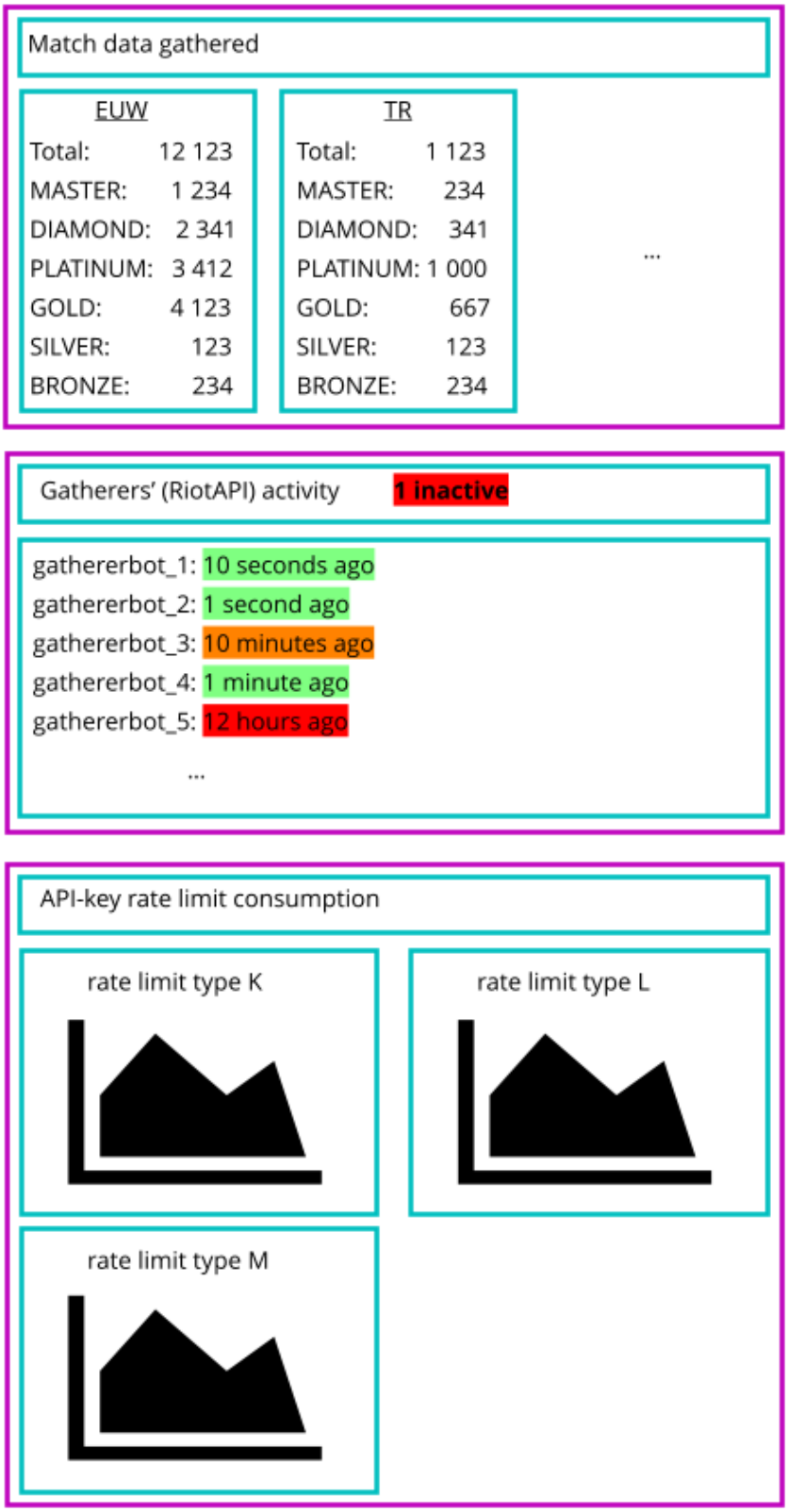


Figure 13. Wireframe of the monitoring user interface.

The user interface was built using the "React" javascript library, to support the plan of building the user interface using independent components as the building blocks. This enabled us to later modify each component individually when necessary, without affecting the rest of the user interface. This is less of an error-prone approach than creating the entire UI as a single monolithic entity. Any possibly duplicate resources shared by the components, were unified using SASS preprocessor language and React library, for cascading style sheets and javascript program logic, respectively. The only other javascript library that was used, is axios library, to provide a promise-based AJAX alternative since the Fetch-standard is not yet finalized (Fetch.spec.whatwg.org, 2018).

Since the modern user interface development has its own development infrastructure (decoupled from server-side programming environments), we stored all the source code related to this monitoring user interface in its own GitHub repository (GitHub, 2018c).

3.3 Retrospective

Restarting stopped daemons

As previously mentioned, there are reasons why individual processes may halt, and we have prepared to account for these throughout the research. Having had the data gathering active for over 2 months (since 19th January 2018 - to 9th April 2018) there were namely three distinct types of interruptions of daemons.

The first and the most prevalent type of an interruption was the expected scenario of an individual process not finding a consecutive match to follow within the defined threshold (30 minutes) and as of result returning to the manual input state. This is not too common, and easy to manage with an occasional log on to the system to input new matches for those processes to continue with. The monitoring user interface proved to be an invaluable asset in this respect, giving up-to-date information whether there are halted data gathering processes and which specific ones they are.

The second most prevalent interruption was similar to the first, but for another reason. As the game acquires minor version updates, there is always a related period of downtime on the game servers. There are no matches being played during this period of

time, and as of result all processes eventually result in the manual input state whenever a game update occurs. While it is momentarily inconvenient, the current process' behaviour seems ideal as the game updates could (unlikely but) potentially change the data structures and therefore it is good that processes halt until manually continued.

The third kind of an interruption occurred when the cloud server provider had to restart their servers to update in response to Meltdown and Spectre vulnerabilities (Meltdown-attack.com, 2018). This resulted in stopping of all processes and their respective screen terminals. While this was the largest scale halt imaginable, it did not take long to restart absolutely everything, due to a high level of automation. On an estimation, it took half an hour to start all processes and to double-check that they are running as intended.

The automation proved to be working effectively over multiple months, with a minimal administrative overhead.

Incomplete records

As a new minor game update is applied, new matches occur but the API receives knowledge of the new version retroactively. This results in some of the data (retrieved immediately after a game update) missing version information. However, this was fixed retroactively by searching that specific match (identified by the match id together with the region) later on (once version data is updated). The delayed API behaviour is undocumented, but based on history it takes between 1-to-7 days to be updated after a game update.

Another potential issue with the data gathering, may occur when the internet connection cuts while downloading remaining part of the match data (with an independent part already being saved), e.g. match results having been saved but match timeline being unavailable. Some times the API may be simply overloaded with requests and therefore be temporarily unavailable. This behaviour is documented, and does not obstruct data gathering since the saved data is intact in either case. It simply meant we had to download those missing records at a later time when the respective API was again available.

For these two cases we created a supplementary software to repair both missing version data and any missing records, since both are mapped to the same combination of match id and region. Since there is no pre-released information on upcoming minor game updates, this periodical data repair (GitHub, 2018d) was instantiated manually after each game update (indicated by the monitoring user interface). This kept all datasets consistent retroactively.

Data backups

As the gathered data is essential to our data analysis, we needed backup copies of it to ensure continuity even in the case of a disaster (such as the cloud provider's disk system breaking, or discontinuing their services). If we were to own the physical storage devices ourself, we could have used methods such as RAID1 disk mirroring (to counter the possibility of a hard disk drive failure) and UPS power system (to counter the possibility of a power outage, providing time to save everything gracefully regardless of said power outage). Since we did not own the physical media where data is stored, we had to acquire a copy of it to a location in our possession.

Prior to relocating the entire database, it had to be saved to a transferable file format. PostgreSQL provides a binary executable named pg_dump (Postgresql.org, 2018c) for this purpose. This executable outputs the database (specified as a parameter) to the standard output stream, optionally in a compressed format using the "-Fc" switch. The compressed file format may be restored in any other environment using the counterpart binary executable pg_restore (Postgresql.org, 2018c).

In order to create a scheduled cron task of the backup process, we needed a method to pass the database password to the scheduled script. For this we used PostgreSQL specific environment variables, namely PGPASSWORD. We set this directly to the crontask call as opposed to the recommended method (that is using a password file) (Postgresql.org, 2018c). Reason for going against the recommended method was to have one less file at the expense of multi-user safety. As described in the documentation, directly using the environment variable allows any other users to see it via `ps` command, however, in our system were (throughout the research) the only user, so that was not an issue. We prepended the task with a command "rm *.file-extension-of-our-choice" to remove (if existing) previous version of the backup, and then redirected the output of the cron task using `>` operator to a new file, with the current date in the

file name. Reason why we needed the current date in the filename, was to have an easy way of affirming how old the backup is, when periodically copied from a remote machine (since the modified time on the remote machine will be the timestamp of the file transfer). An alternative method would had been listing the directory (along with its files' modified time), but the former method was simpler in terms of execution (i.e. "less chained commands").

On the computer that stores the periodical backups, we scheduled another crontask to instantiate `scp` file copy on the same interval that the backup is produced (while at the same considering the time it takes to produce each new backup, so with a minimum of at least one hour of time in between). To perform the safe copy without a password, we utilised ssh keys (Linux.die.net, 2018).

4 Data analytics

4.1 Approach

We essentially have two ways to approach the problem of making predictions utilizing the gathered data. The first way is to begin with the presumption of having any prior knowledge of variables affecting each response (i.e. "win" or "loss"), and an estimation of their relationship to the response (i.e. whether it is a negative relationship or a positive relationship).

The second way is to objectively begin with no presumptions of the data, other than the knowledge of how each variable computationally maps to other variables (as described and visualised earlier). This research focuses explicitly on the workflow that is necessary to create predictive model(s) and finding new correlations between variables (as opposed to limiting the scope of correlations to the "known" relationships). This second way encourages such focus more than the first way, but we will make some use of both to seek optimal results. Namely, we utilise the first way (with prior domain knowledge) in order to perform feature engineering.

Variables that exist in every match, are straightforward to interpret (regardless whether they are continuous or categorical variables), since they either have or don't have a statistically significant relationship to the outcome of a match.

In addition to variables from match data, we can augment the dataset using historical aggregated data of each participant.

4.2 Tools

Work environment

The two most popular choices for exploratory data analysis tend to be the R language (R-project.org, 2018) and a mixture of Python (Docs.python.org, 2018) libraries, as advised by the Kaggle tutorials FAQ (Kaggle.com, 2018). While both of them are programming languages that ultimately permit the use of extensions written in C language, Python is the more familiar of these and that is why this research was conducted using Python programming language.

As we already established the tools necessary to move our database across computing environments, we could conduct the exploratory data analysis on a separate computer than where data gathering is done. This permits utilising more memory (supporting larger data structures) and specialized hardware (e.g. CUDA enabled GPUs) to assist in computationally intensive operations. To emphasise portability of the research, we used IPython environment and Jupyter Notebook file format to move any work in progress to whichever workstation we were working on next.

The data analysis libraries used were *Pandas* for its additional (expressive) abstract data structures "Series" and "DataFrame" (Pandas.pydata.org, 2018a) along with its prerequisite library *Numpy* (Docs.scipy.org, 2018), and *Matplotlib* for 2D data visualisations (Matplotlib.org, 2018) along with its extension *Seaborn* for both easier access to altering colour palette as well as the additional types of plots it provides (Seaborn.pydata.org, 2018).

Preparing data

Ultimately, we wished to have the data in a pandas DataFrame, but since the original data is stored as JSON formatted strings, we first had to deserialise it.

While we could have immediately mapped respective attributes we wish to use (from the raw datasets) to create a DataFrame, we did it after first processing the datasets into a singular three-tiered hierarchical structure. Reason for this was, that the original data format was fragmentary and contained a lot redundancy (e.g. unique ID numbers in multiple places), and it is much easier to add/remove variables from a hierarchical tiered data structure. This hierarchical structure of ours is depicted in figure 14 as a flowchart. The original data was fragmentary most probably, because it is an aggregate of the game publisher's own, relational database format where relations are optimized as per game functionality (and not for analytics).

In addition to simply reorganizing existing data into tiers, we transformed some sets of individual values to ordered lists. An example of this is an individual player's "final items" which can be re-ordered in-game indefinitely, while their order does not affect their effects. These seven items are enumerated in the raw dataset as "item0", "item1", "item2", etc. up until "item6". We put them in an ordered list under key "final_items", preserving their order while making them easier to process programmatically (e.g. "does item X exist in final items" as opposed to checking each item key individually).

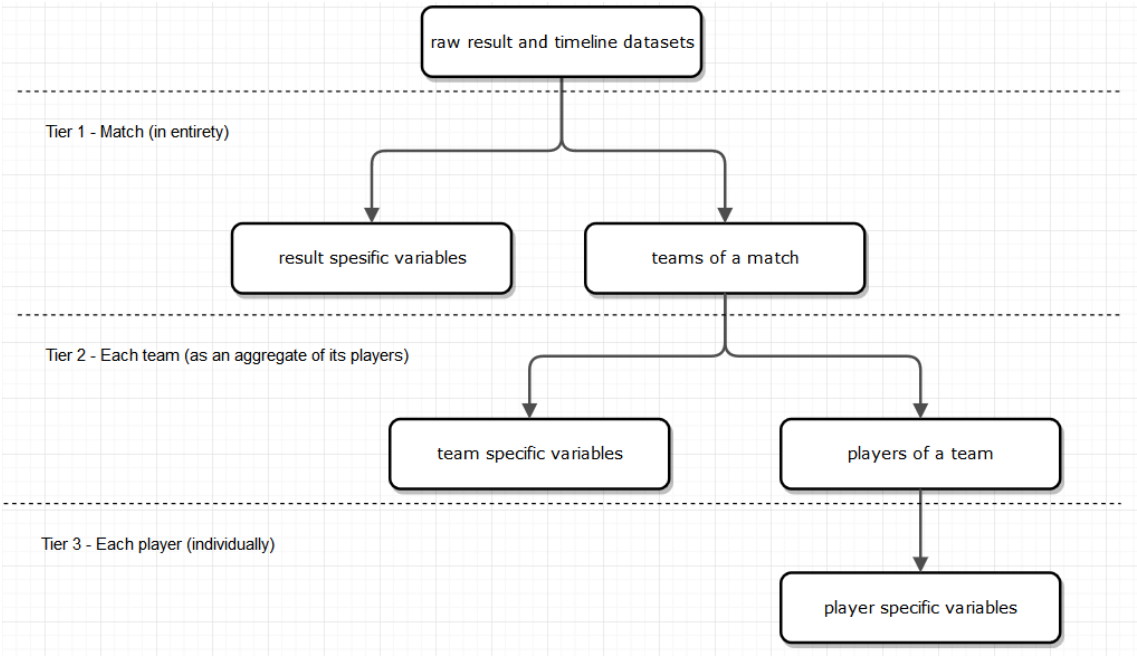


Figure 14. Raw match specific datasets re-mapped into a hierarchical, three-tiered data structure. Tier 1 represents the initial depth (e.g. {'result_var1': 'val1', 'teams': [...]}).

The downside of this monolithic data structure was the inherent need to have two sets of mapping rules. First the rules to map the original datasets to this monolith, then the rules to flatten and transform individual variables from this monolith to a format that is fittable to machine learning algorithms. An alternative to this would have been flattening our monolithic structure in entirety utilising JSON serialization together with pandas library's json_normalize function that flattens a given JSON data structure (Pandas.pydata.org, 2018), but a reason to map each existing variable individually was to additionally assess whether each variable was worth including in the first place (e.g. redundant variables such as unique identifier numbers).

In addition to straightforward data transformations (e.g. integer based categorical identifiers to strings – that are later to be one-hot encoded), we had some missing values to fix in the source data. An example of these missing values is an occasionally missing "perkSubStyle" key-value pair, as shown in figure 15. Technically speaking the value exists in every single match (since the game client enforces having both "primary" and "sub" perk styles), but for whatever reasons, the information on sub style is missing in some cases as indicated by a KeyError exception. We can deduce whatever it was based on a patch specific perk lookup, mapping all of the chosen perks (3 from primary style and 2 from sub style) then reducing the primary style off from the 2 found styles.

```
normalized_data_collection = []
for tier in ['SILVER', 'GOLD', 'PLATINUM', 'DIAMOND', 'MASTER', 'CHALLENGER']:
    normalized_data = list(matchdata_generator(tier, 'EUW'))
    print('{} matches from tier {}'.format(len(normalized_data), tier))
    normalized_data_collection = normalized_data_collection + normalized_data

print('Total {} matches loaded and normalized'.format(len(normalized_data_collection)))
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-10-66d00ea43235> in <module>()
      1 normalized_data_collection = []
      2 for tier in ['SILVER', 'GOLD', 'PLATINUM', 'DIAMOND', 'MASTER', 'CHALLENGER']:
----> 3     normalized_data = list(matchdata_generator(tier, 'EUW'))
      4     print('{} matches from tier {}'.format(len(normalized_data), tier))
      5     normalized_data_collection = normalized_data_collection + normalized_data

                    redacted mid-section of the traceback

<ipython-input-6-a25b23a85374> in <lambda>(participant)
     41     ],
     42     'primary_rune_cat': lambda participant: str(participant['stats']['perkPrimaryStyle']),
---> 43     'secondary_rune_cat': lambda participant: str(participant['stats']['perkSubStyle']),
     44     'runes': lambda participant: [
     45         str(participant['stats']['perk0']),

KeyError: 'perkSubStyle'
```

Figure 15. KeyError resulting from a missing field in the original dataset.

Finally we decided the context we correlated to the result of predictions (whether a match will be a win or a loss). We could have correlated the result against 3 distinct entities.

The first and most simple of aforementioned contexts is an individual player ("a participant in a match"). This is the common point of view people assume when predicting their own or someone else's match's outcome. Additionally this is the point of view of all the websites that aggregate data from the subject game. An individual participant does contain many measurable variables even prior to the start of a match (e.g. chosen character, chosen role, and chosen summoner spells), but this point of views is based on trying to find ”a silver bullet”. Most of these aggregations on said websites are correlated with a win rate between 45% and 55% (Leagueofgraphs.com, 2018), therefore themselves disproving the existence of any ”silver bullets”. However, this information can give useful indications of "what generally works and what does not work". Inspection of the dataset from the previous patch confirms this, as shown in figure 16.

```
players_df.groupby('champion')['is_winner'].apply(lambda x: '{:.0f}% winrate (over {} games)'.format(x.sum() / x.count()
```

```
champion
429     61% winrate (over 38 games)
68      60% winrate (over 77 games)
85      59% winrate (over 66 games)
134    59% winrate (over 120 games)
9      57% winrate (over 124 games)
44     57% winrate (over 121 games)
38     56% winrate (over 245 games)
163    56% winrate (over 197 games)
91     56% winrate (over 172 games)
30     56% winrate (over 122 games)
22      55% winrate (over 88 games)
133     55% winrate (over 76 games)
164    55% winrate (over 666 games)
121    55% winrate (over 489 games)
32      55% winrate (over 31 games)
96     55% winrate (over 112 games)
497    54% winrate (over 826 games)
267    54% winrate (over 615 games)
102     54% winrate (over 50 games)
83      54% winrate (over 41 games)
92     54% winrate (over 277 games)
35     54% winrate (over 242 games)
55     54% winrate (over 196 games)
105    54% winrate (over 167 games)
161    54% winrate (over 123 games)
3      54% winrate (over 101 games)
60      53% winrate (over 91 games)
157    53% winrate (over 605 games)
80      53% winrate (over 57 games)
8      53% winrate (over 475 games)
                ...
```

Figure 16. Character specific win rates from near-professional-tier games of the previous patch, confirming they are relatively close to the coin toss probability of 50%.

The second, and practically speaking the most useful to a player, is the context of an individual team. A team includes five players with their related variables. It is the most useful context for a player, since during the character selection all of the teammates are already known, but there is an option to leave from that team (with much lower penalties that come from losing a match).

In addition to the players there are very few "team specific" variables (i.e. champion bans and the side of map for the team). Most importantly in this second context, each player is a set of related (non-independent) variables, and must be transformed to categorical values that altogether form the measurable values of "a team". An example of this is the existence of a given champion in a team. Each champion ("character") can exist only once per team, but it makes a notable difference what role (and what resources as of result) is allocated to that champion. In practise this means that every champion variable must be combined with every role and every lane that may exist in the data.

The third context is the entire match, including both teams. This would give us the most insight of all three, since it includes all of the variables from two smaller contexts, but in addition the information of team vs team. Problem with it is, that the number of existing variations is increased so high that processing it statistically requires very high sample sizes.

Since we had access to hundreds of thousands records, we chose to focus on the third context that is the entire match. However, we did evaluate the first and second context in terms of exploratory data analysis.

The entire analysis data pipeline involves aforementioned three forms of data (raw original data, hierarchical "fixed" data, and finally the context-split two-dimensional flat dataframes). This pipeline is depicted in the figure 17. There are two essential points of transformation (one between each form of data), and both of them require the additional knowledge of static game data (i.e. enumeration of all possible categories and/or values for each data point).

The rules used to transform data between their forms, were made as configurable as possible, in order to support both any future changes in the variables (between game versions) and any engineered features (implemented later during the research). Any transformations applied to the dataframes (e.g. standardizing data to around 0 for specific machine learning algorithms) were excluded from this initial description of the pipeline, and are included in their specific use cases.

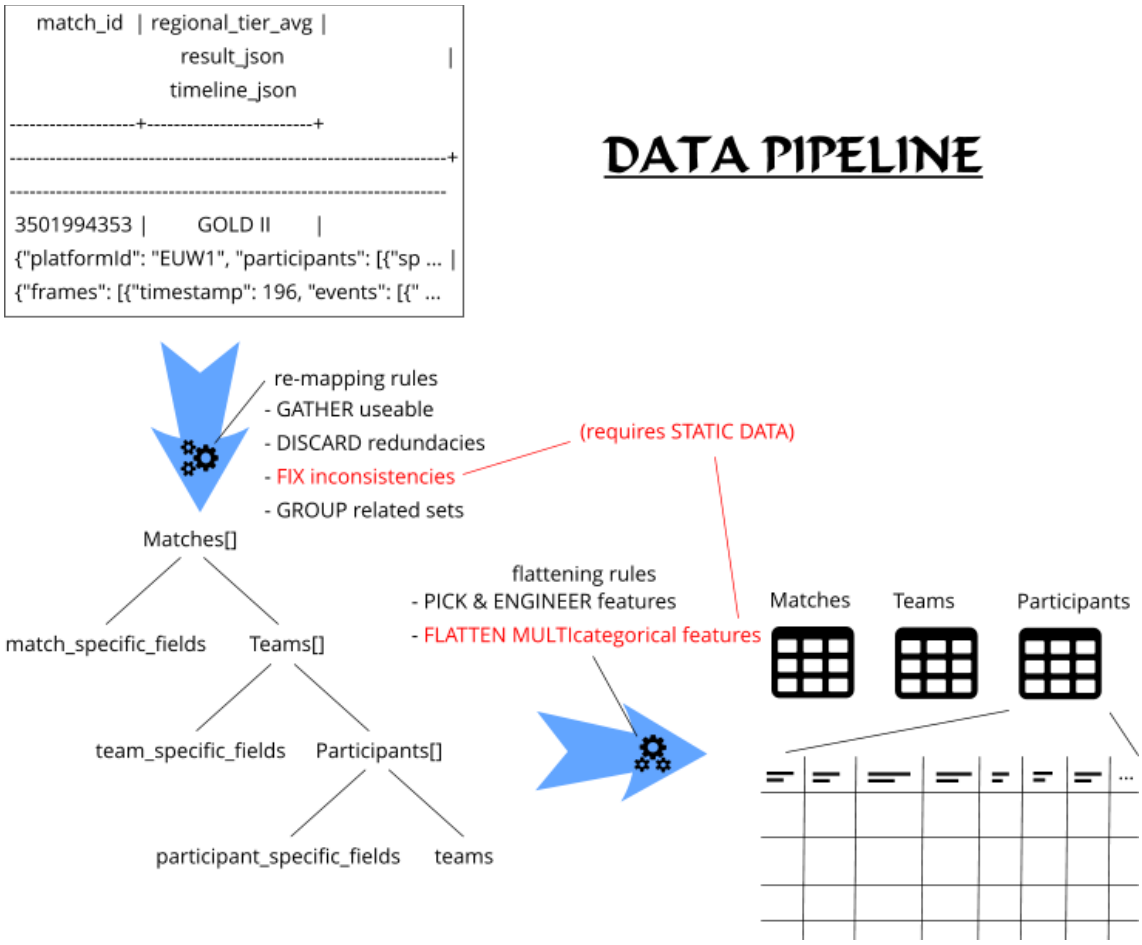


Figure 17. A chart describing the data pipeline used to transform the gathered raw datasets to statistically processable form.

4.3 Exploratory Data Analysis

Summaries

We used the pandas library's DataFrame method "describe" to obtain a set of summarising values of each data column. A part of this output in the context of the participants (summaries from a total of 400 columns) is shown in figure 18. We have the median, mean, and standard deviation (abbreviated to "std") for each column as well as the quartile values. An important note here is that these columns also include some of the mid-match statistics (that we do not really know at the time of prediction). We can assess their usefulness in initial data analysis, then later augment the dataset with information on past matches' respective values (i.e. historical averages of each participant).

```
participants_df.describe(exclude=np.number)
```

	previous_season_max_tier	lane	role	scored_first_blood_kill	scored_first_blood_assist	scored_first_tower_kill	scored_first_tower_assist	scored_firs
count	36960	36960	36960	36960	36960	36960	36960	
unique	8	5	5	2	2	2	2	
top	DIAMOND	BOTTOM	SOLO	False	False	False	False	
freq	19422	12739	11503	33264	33264	34032	33538	
first	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
last	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

```
participants_df.describe().reindex(['min', '25%', '50%', 'mean', 'std', '75%', 'max', 'count'])
```

	gold_earned	gold_spent	gold_per_min_0_to_10	gold_per_min_10_to_20	gold_per_min_20_to_30	gold_per_min_30_to_40	damage_to_champions_total	damag
min	1338	500	0	0	0	0	0	
25%	8483	7650	235	317	0	0	8931	
50%	10807	9775	275	390	0	0	14083	
mean	11049	10073	274	375	161	0	16310	
std	3616	3533	66	146	222	0	10165	
75%	13338	12250	314	462	377	0	21237	
max	31150	48225	720	943	1089	0	119752	
count	36960	36960	36960	36960	36960	36960	36960	

Figure 18. The output of DataFrame.describe() -method on the participants dataframe, showing common statistical information representing distribution of values for each column of the dataset.

Glancing through each of the columns tells us how the data is varied. It doesn't inherently tell us the usefulness of each of the extracted columns. We cannot use it alone to predict any future results. At the best we can notice which variables include outliers.

Plotting pre-game choices in search of a ”silver bullet”

A naive solution would have been to correlate all of the records' distinct pre-game choices against the boolean is_winner value, calculating averages of each relationship, then predicting based on merely those averages.

This is the approach used by most of the fansites such as op.gg (OP.GG Europe West, 2018a), leagueofgraphs.com (Leagueofgraphs.com, 2018), and champion.gg (Champion.gg, 2018). The approach works to some extent for 1-dimensional data. For example with 140 distinct characters (Na.leagueoflegends.com, 2018), 282 distinct items (Ddragon.leagueoflegends.com, 2018), and 9 summoner spells (as depicted in chapter 2.2), we have a total of 431 categorical values to plot against the win rate. However, with only 2-dimensional data mapped only along the characters, we already have 140x(282+9) = 40740 combinations which is not a realistic amount of graphs to manually review. The aforementioned fansites offer this kind of (2-dimensional) statistics and generally speaking they tend to be in the same range of 40-60% win rate (close to the "coin toss propability" of 50%).

While it is unlikely for us to find "a silver bullet" amongst an individual participants pre-game choices (roles x characters x summonerspells x runes) while considering neither the teammates nor the enemy team's composition, it is worth calculating these values any way, since we can programmatically find any relevant outliers (i.e. "silver bullets") if they were to exist. This was performed to not neglect the possibility of such occurrence, since the required computations are relatively inexpensive. The way we were able to further reduce the total number of computations, was by only considering the existing categories (e.g. only the combinations that exist in rows of our dataset). This way, theoretically speaking the maximum possible number of variations was either the number of samples in the dataset or the number of combinations amongst the choices whichever of the two was smaller. However, practically speaking it is highly unlikely for us to encounter every imaginable combination in the samples and therefore we saved computations doing this.

The number of combinations per category is increased "times two" every time a new category is introduced, due to that category being possibly either included or excluded. For example we may initially have the category "characters", to which we add another category "summonerspells". We can map them 1-dimensionally [[characters], [sum-

monerspells]] or 2-dimensionally [[characters, summonerspells]], and if we introduce a new category "runes" we can either keep the existing mappings ("times one") or add the new category to each of the combinations ("times two" with former), as [[characters], [summonerspells], [runes]], [[characters, summonerspells], [characters, runes], [summonerspells, runes]], [[characters, summonerspells, runes]]. Since the "0 categories" is not a viable combination, we reduce one from total, resulting in $\mathbf{2^n-1}$ combinations per n categories. Together with the tens of thousands combinations within each category, we have a lot of potential results (but only the number of rows of maximum existing results).

It is very likely that some combinations are more popular than other. To not miss any possibly relevant combinations, we fetched all existing combinations per each row of the dataset, and counted their individual winrates. This new dataset (depicted in figure 19) represented a momentary aggregate of combinations contained in the current dataframe along with their winrates, number of occurrences, and number of individual player ids that used them (to distinguish a possible scenario where one person may be the only one using a certain combination).

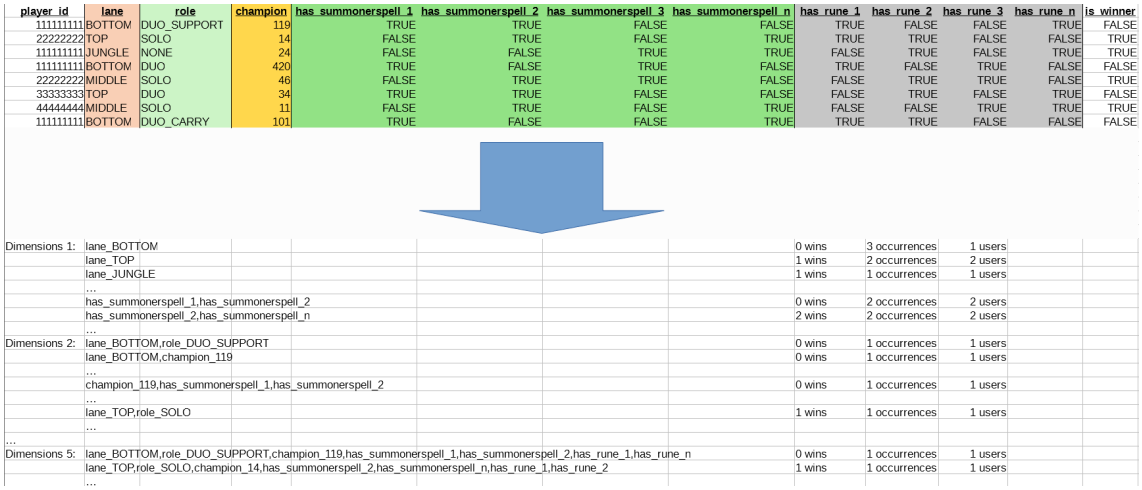


Figure 19. Aggregation from (cumulative) participant dataframe to an aggregate of combinations along with their winrates and usage metadata. Contains example pseudodata.

The problem in this method was, that from a mere two weeks period of data gathering in the top <2% tier playerbase of Diamond and higher (OP.GG Europe West, 2018b) we had 36'900 participant dataframes, altogether containing 186'453 different combinations (across the five dimensionalities containing categories "lane", "role", "champion", "summonerspells", and "runes"). That is a lot of data to go through, and as of result we

had to either aggregate the data (that is already a bunch of aggregates) or filter it by parametrized limits. To have an idea of its distribution we used box plotting together with a histogram. A box plot shown in figure 20 reveals how the distribution of winrates is very evenly split around the 50 per cent, and that a large portion of values (1st and 4th quantiles) exist at the minimum and maximum of 0 and 100 respectively. A histogram shown in figure 21 confirms that most values are either 0 or 100.

```
import matplotlib.pyplot as plt
import seaborn as sb
sb.set()

def boxplot_dimensions(dimensions):
    x = []
    for dim in dimensions:
        for combination in dimensions[dim]:
            statistics = dimensions[dim][combination]
            x.append(statistics[1]/statistics[2]*100)  # Calculate and add each winrate to x
    plt.boxplot(x)
    plt.title('{} combinations boxplotted by their winrate'.format(len(x)))
    plt.show()
    plt.close()

boxplot_dimensions(dimensions)
```

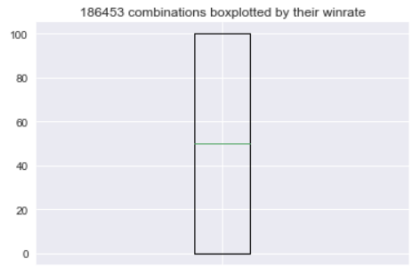


Figure 20. All combinations boxplotted by their winrates, disregarding the number of games for each combination. (E.g. combination being used 1 time vs. another combination being used 1000 times with different winrate).

Figure 21. Distribution of all existing combinations plotted as a histogram over full percentages.

The difference in winrates is expected to vary more with a smaller number of samples, and so we used a scatter plot to validate this assumption. This scatter plot is shown in figure 22 and confirms the aforementioned assumption. As the number of samples increases, the winrate nears towards 50 per cent.

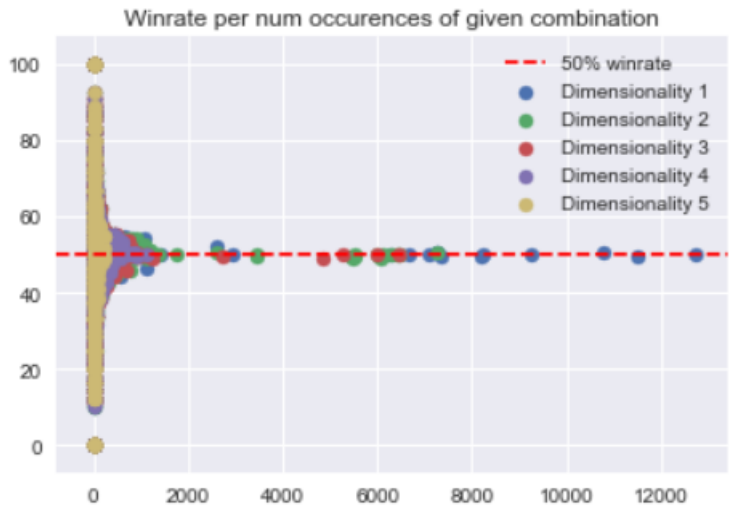


Figure 22. Scatter plot of winrate against number of times each combination was used. Color-coded dimension of each combination.

There is a cluster near a few hundreds' occurrences' mark, between 60 and 40 percentages. We could inspect the result closer at either >10 occurrences (which is a minimal number of matches to acquire a starting Elo rating in matchmade games) or near >200 occurrences to inspect that cluster more closely. Figure 23 shows both of these cases. Several hundred repetitions of a single combination is already a considerable amount, but them all existing between 40 and 60 per cent confirms the fact that while there are distinguishable "good" and "bad" choices, there are none "silver bullets" (worth using in accurately predicting an outcome of a match).

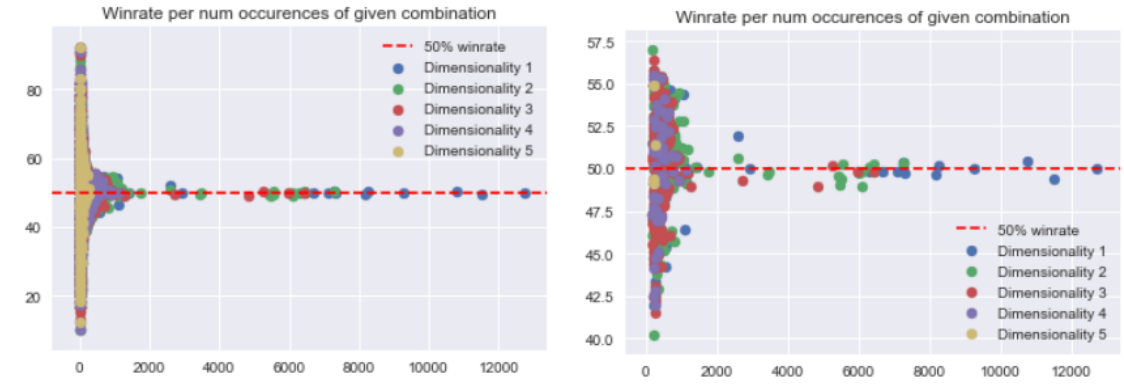


Figure 23. Scatterplots of winrate against number of times each combination was used, filtered by a minimum number of times a combination was used. Left graph excluding any winrates with less than 10 occurrences, right graph excluding any winrates with less than 200 occurrences.

We can perform this same procedure to the team dataframe, using the number of teammates as dimensionality (e.g. "a pair of participant A's choices with participant B's choices" is one two-dimensional combination). All first dimension's values are essentially the fifth dimension of participant dataframe (containing all the individual columns). Performing the exact same steps as before, we end up with similar looking graphs that show distribution near edges (0 and 100) at low numbers (i.e. below 10) of occurrence, that converge towards the area between 40 and 60 per cent. These are shown in figure 24.

Figure 24. Boxplot and histogram showing the distribution of combinations contained by the team dataframe. Additionally a filtered (>200 occurrences) scatterplot displaying how all combinations seem to converge to the range between 40 and 60 per cent as number of occurrences increases.

We skipped this procedure for the match dataframe (containing essentially two team dataframe rows in each of its rows) due to time constraints, since the numbers of combinations is increased a lot (by each of the teams' 5 participants' combinations being able to be mapped to the enemy teams 5 participants' all possible combinations). It would very likely result in the same as the former two, that is a scatter plot converging

towards the range between 40 and 60 per cent. This may be explored in future research on the subject.

4.4 Machine learning

Premise

We can utilise machine learning to automatically discover correlations and mappings between known independent variables (i.e. input) and dependent variables (i.e. output) in the hypothesis space of possible variations. The combination of a chosen algorithm with a specific set of pre-train parameters (also known as ”hyperparameters”) is referred to as a "model". (Google Developers, 2018) We utilised solely supervised machine learning, where a model is trained by passing it samples of input and output, and this training is known as ”fitting”. Once a model is fitted to a satisfactory extent, it can then be used to predict unforeseen outputs by sending it respective input datasets. Predicting new outcomes based on a known input is an essential part of our research problem, and so these algorithms are useful tools for us to process the vast amount of data we have.

Machine learning consists of many distinct algorithms to produce aforementioned models. We began by utilising deep neural networks on pre-game data, namely on the character selection choices, to see how sufficient character choices alone are in order to predict the outcome. A similar work has been previously implemented by Ong et al. where they reportedly obtained 54.4% and 70.4% prediction accuracies on a total of 130’000 samples using a 10% hold-out test set with a support vector machine algorithm (Hao Yi Ong, Sunil Deolalikar & Mark Peng, 2015). The former prediction accuracy was reportedly obtained by using the game publisher’s then-standard classifications for each selectable character, and the latter by using player behaviour based KMeans derived clusters. Unfortunately the paper does not go into the specifics of how these clusters were mapped to the predictive model *(as presumably not via mid-game values – that would not be realistically obtainable in future predictions)*. Although the game has changed since (in three years) we obtained our initial model in a similar manner utilising only champion selection choices, though directly one-hot encoding them as the input for an artificial neural network (as opposed to reducing them to clusters of similar characters).

Predicting the match outcome by character choices

A common (and a very popular) approach to utilising MOBA games' predictive systems is to ask for an optimal character choice in the beginning of a match. This implies that some team combinations of characters would be better than the others, but in practise these suggestions are often based on anecdotal evidence (regardless if they come from another participant or a software system). Character choices are also expected to influence any other features greatly, so we started with them and progressively increment upon that (expecting better results with better additional features).

All the work was implemented in IPython Jupyter Notebook environment using GPU backed TensorFlow abstraction via Keras library. We additionally used an explicit early stopping with 0 delta sensitivity and 30 epoch patience, as well as the (TensorFlow framework specific) TensorBoard visualisation tools to keep track of the training process, to make sure it works in the first place. (TensorFlow, 2018)

The baseline is based on presuming ”all samples belong to category that is most prevalent in the dataset”. In this case the dataset was equally split to 50% winning teams 50% losing teams, thus the baseline is 50 per cent. Training the vector of 280 values using 166’283 sample games from three adjacent game versions (8.7, 8.8, 8.9), resulted in validation accuracy of 55 per cent at the best as displayed in figure 25. We do not expect the significances of the individual characters to vary enough between these game versions for those significances to matter, but we had the capabilities and did double-check this as follows.

The networks were of varying complexity starting from a single node sigmoid function to a three-layered network of 120, 80, and 40 nodes with rectified linear unit activation. The simplest single node ”network” (practically an implementation of logistic regression) proved to be the most efficient (reaching the optimal weights in 40 epochs as zoomed in figure 26) as well as the most accurate. In addition to the shown models, there were at least two alternative versions for each of all three multi-layer models, containing various choices for generalizing the architecture including the use of 34% dropout on each layer and the exponential L2 regularizer with 0.001 value. However, none of these generalization methods improved the validation accuracy to the same height as the single node ”network”. The two most complex networks (depicted in the

figure 25 as green and pink lines) included some of these (34% dropout and L2 regularizer) in the models that were plotted.

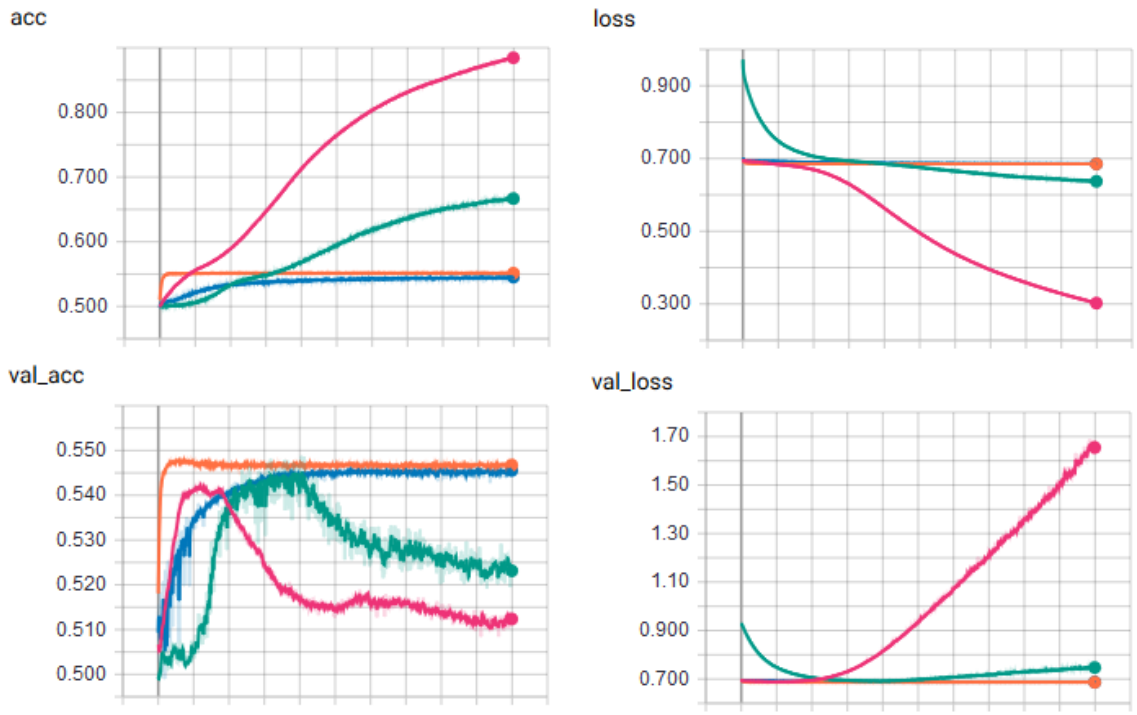


Figure 25. Neural networks' training- and validation accuracy graphs, next to respective training- and validation loss graphs, trained for 1000 epochs. As the complexity increased, the validation accuracy eventually fell and never reached higher than the least complex model (orange). Notice the varying Y-scales per each graph.

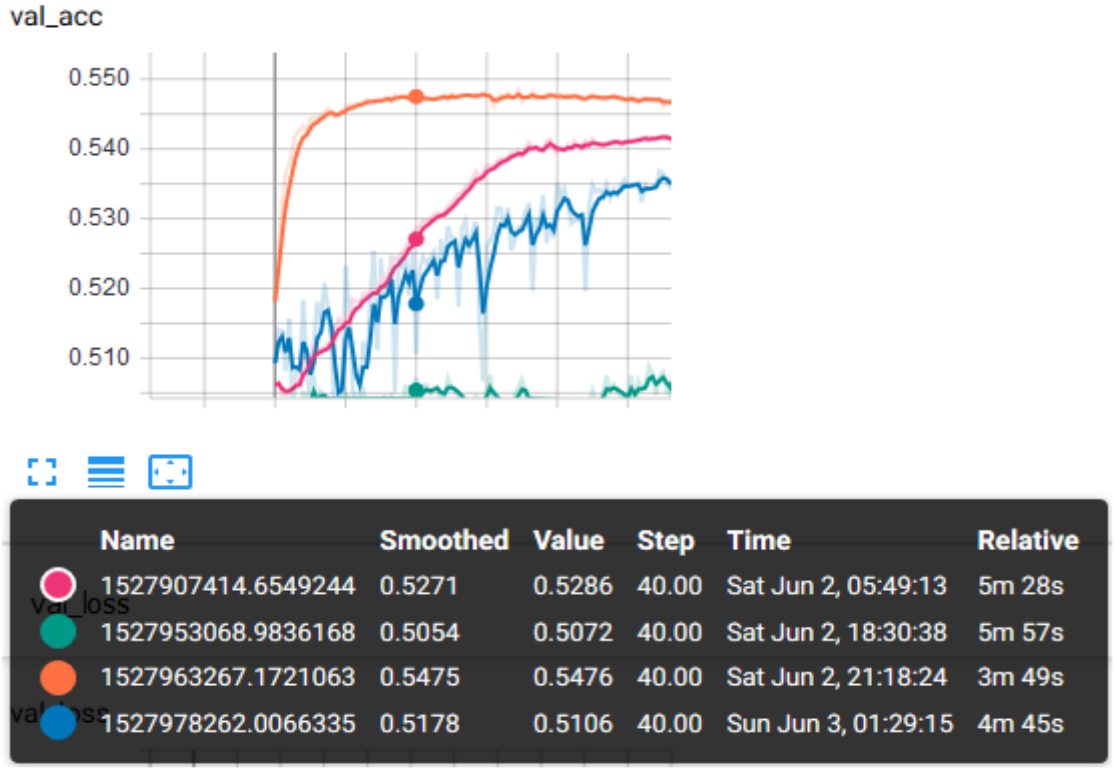


Figure 26. The point of training (depicted with filled circles) with the most optimal weights for the most optimal model.

An accuracy of 55 per cent for mere champion choices is a good starting point, higher than baseline, but is not sufficient for practical use.

Since we used samples spanning over the versions, we tried again with more new data, having tens of thousands of samples from each individual version to confirm whether it returns the same result of 55 per cent. Specifically we have 88606, 72323, 41393, 141673, and 64683 samples from game versions 8.7, 8.8, 8.9, 8.10, and 8.11 respectively. Most of the data is from a tier named ”Diamond” that includes the most of the top second percentile (2%) of the ranked player base, so we can assume it consists of games that are serious-mindedly played.

We used a 80/20 hold-out test set, in addition to k-fold cross-validating (with 5 folds) the 80 per cent of training data, for each game version. These were applied to 6 distinct architectures of varying complexity between a single node ”network” and networks with two hidden layers. Every network’s output node uses the sigmoid activation function, and any hidden layer nodes use the rectified linear unit ("ReLU") activation func-

tion. All utilised network architectures were summarised in appendix 1 along with the results.

Because there were a total of 150 networks (from 5 game versions, 6 network architectures, and 5 folds) trained in the process, we utilised early stopping instead of manually observing the loss and accuracy graphs (in order to greatly reduce the training time and limit redundant training). The parameters used for early stopping were a delta (difference against the previous minimum) of 0 and a patience (as in the number epochs to wait before concluding the network isn't improving) of 30.

We obtain average validation accuracies ranging from 0.49 to 0.55, and the test accuracies for each model peak at 0.55 (mostly staying in the range between 0.53 and 0.55).

Mid-game role-based feature engineering

While we could have a look at all the individual pre-match choices, which participants have to make (summonerspells and runes) in addition to the character, there is no ”guaranteed winning combination” (aka silver bullet) as we explored in the chapter 4.3 (Exploratory Data Analysis), and realistically speaking the possibilities of any highly effective non-linear relationships amongst them (e.g. character A with summonerspell X combined with character B with rune Y) are non-existent. This is additionally supported by the author's first-hand experience of playing the game for years in the top second percentile of ranked player base. Any further research on these (sets consisting of exclusively pre-match) variables and their potential interaction is beyond this thesis.

We could have extracted a lot of historical post-game data (containing statistical summaries of what happened in the games) and aggregated all of it. Additionally we could have feature engineered variables that more better represent the actual flow of a game. An example of the latter is a kill event that occurs during a team fight (where the amount of contribution is unknown in contrast to the other 4 team members), as opposed to a vastly different kill event in a 1-on-1 situation (which represents a positive combination of decision making and capability to play on the character). These two events are indistinguishable without feature engineering. We chose to do the latter (feature engineering) together with the former (simple aggregations) to encompass as many variables as potentially useful.

In order to perform feature engineering we had to utilise the mappings amongst the gathered data (more closely described in chapter 2) along with domain knowledge. In this research the domain knowledge was limited to author's own knowledge of the game. Therefore this part of the study may be opinionated, biased, and subjective, but that is sadly a necessity at this point to continue, due to the massive number of possible combinations of acquired metrics. It could be a target of later critique on the work.

Every game generally consists of two distinct phases, a ”laning phase” and a ”late game”. Due to resources being spread across the map (over three ”lanes” and an area called ”jungle” in between them), participants are usually set to five pre-determined roles that occupy specific areas of the map. Laning phase consists both teams' respective roles facing off in either 1on1 or 2on2 setting on their respective area of the map, with teammates being able to affect each other's fighting at the expense of their own resource gathering. Late game on the other hand refers to a later point of time where teams' participants have accumulated enough resources to work more effectively in a group of 4 or 5 (depending slightly on their character choices) and fighting is carried out in 4-5 on 4-5 setting. This is a very high-level abstraction of a real scenario but allows us track the temporal match timeline more effectively (since some people may be better in one context rather than the other). Typically some roles have much more resources than others, and they are expected to ”carry” their teammates throughout the ”late game”.

Additionally the game publisher has further supported this generalisation of games by displaying each participant's ”dedicated role” in character selection since two years ago in 2016. Disagreements over role sometimes lead to participants leaving the character selection, effectively cancelling the preparation of a match (while punishing the leaver with a penalty to their ranked position that is however less penalizing than a lost match would be). An interesting variable is the number of participants known as ”autofills” (players who are not in their primary nor secondary choice of role), but that effect is also likely to depend on the person's capability to play various roles (i.e. some are better doing that than others, so the metric alone wouldn't be sufficient for predictions).

A standard team setup consists of the following roles: toplaner, jungler, midlaner, carry, and support. We can use these roles (that are also supported by the game publisher as indicated in the game client) as unique identifiers per each team (to correlate between each other and respective enemy roles). The downside of this approach is that the

game publisher's role data on historical matches' roles is often inaccurate and must be determined based on heuristics when parsing each roles' variables. An example of this inaccuracy is depicted in the roles from a few random games, listed in figure 27.

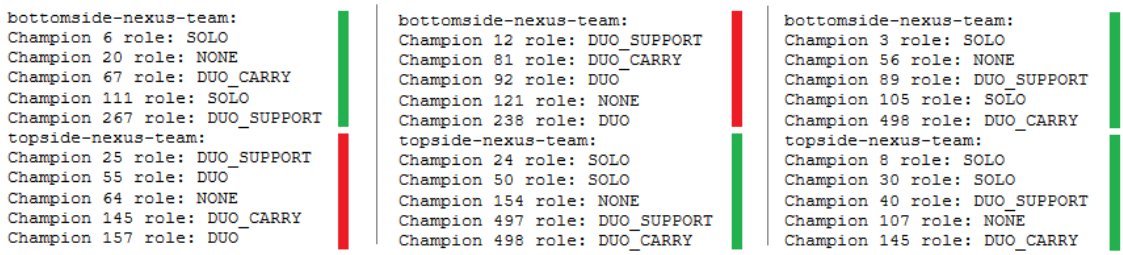


Figure 27. Three randomly selected games' roles of each champion in both teams, assigned by the game publisher in the dataset they provide. Red marks the cases where role assignment was very likely inaccurate since having two duos (DUO+DUO with DUO_CARRY+DUO_SUPPORT) would leave one part of map entirely uninhabited. Role "None" indicates a jungler. Toplane and midlane are both solo roles.

The heuristics we used to determine each participant's team-specific real role were based on the numerical aggregate data, location data, and summonerspell choices (since one of the roles involves a mandatory specific summonerspell). The jungle role was indicated by the combination of the presence of summonerspell "Smite" and the highest number of jungle non-playable characters killed (the dedicated source of resources for that role). At first minute the participants are still gathering as the gathering of resources begins after 1 minute 30 seconds has passed, second minute is the first observable location, and finally sixth minute (when toplane and midlane acquire their "ultimate" abilities) is the moment after which participants may be "roaming" (moving) around the map, trying to obtain objectives instead of their dedicated resources. From the remaining four participants we determined toplane as the one that spends the most or the second most of the five minutes (between 2 and 6 inclusive) in the topside of the map (as indicated by red circles in figure 28) and has the highest amount of resources gathered. The second most in case there is additionally a support. From the remaining three we consider carry as one of two that spends most of the five minutes in the bottomside and additionally has the highest amount of resources gathered (since the "support" role is intended to pass all those resources to the "carry").

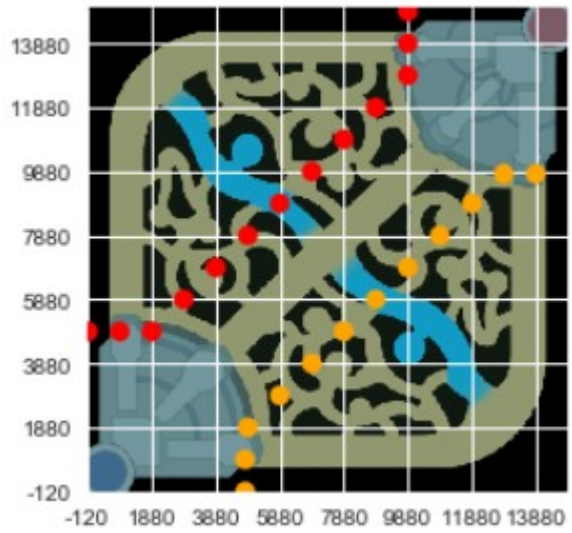


Figure 28. Three lane sections of the map. ”Topside”, ”midlane”, and ”bottomside”. Topside border marked by red circles. Bottomside border marked by yellow circles. Midlane in between the two.

Finally, we distinguished the ”midlane” role from the ”support” role by which one has the highest amount of resources gathered. We did this as opposed to a location based heuristic due to a trend of some supports spending more time moving around the map (i.e. helping multiple teammates) and therefore they may not be located in the bottomside during those specific minutes.

Having plotted some locations over the map to test the effectiveness of location bounds, proved it is working as intended, as seen in figure 29.

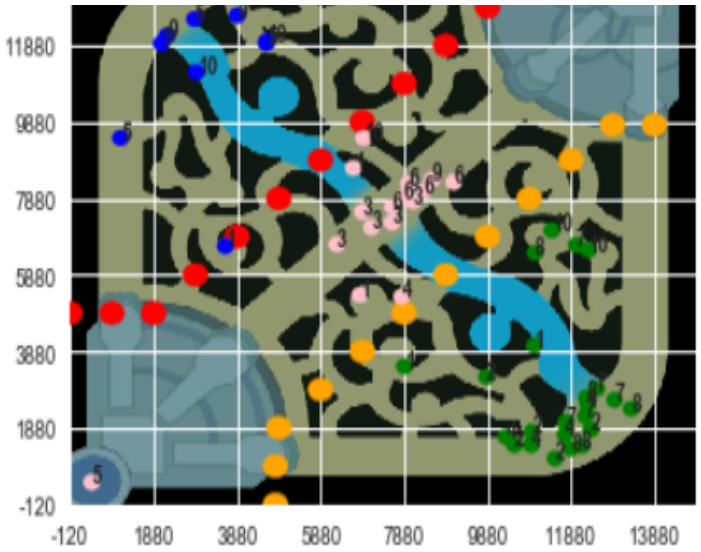


Figure 29. Testing map area filtering, sectioned to topside, bottomside, and the remaining area elsewhere (in between). Locations shown from minute 2 to minute 6. Participants 3 and 6 seem to be the teams' midlaners, and participants 5 and 9 the toplaners.

Initially this rule set seems to work, but a closer manual review on the interpreted games reveals that it was sometimes misinterpreting an early roaming midlaner as the carry, due to the fact we pick ”two champions most prevalent to the bottomside and choose the one of those with the most resources gathered”. This is a very realistic scenario and was fixed via recognising the support participant right after the jungler participant, so instead we set a rule ”the participant that spends the most time of all in the bottomside and is not (previously ruled out) support participant”. After manually reviewing the misinterpreted games, they were interpreted correctly after the fix. The incorrect and correct cases can be seen in the printout shown in figure 30. We'll be satisfied with this reworked set of rules, since (considering the entire game record) it is the most accurate possible (ruling out the two obvious roles, and splitting the final three based on their locations).

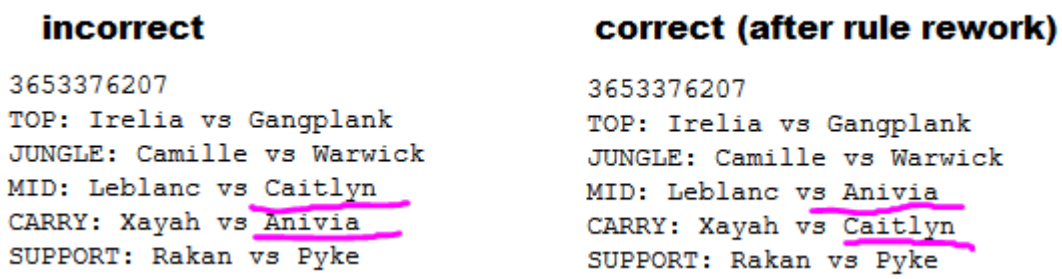


Figure 30. Misinterpreted role mappings using the initial rule set, and corrected role mappings after reworking the role recognition rule set. Match #3653376207.

Minimizing sparsity

Since character choices alone were not sufficient to predict the outcome of a match, we resorted to attempting to predict the outcome using historical post-game data that describes any recurring player behaviour. These metrics include general aggregate averages (such as the average number of kills, deaths, assists, healing, and damage dealt per game) as well as feature engineered averages (such as the average number of positive solo fights per game).

Although we concluded character choices alone not affecting the outcome in a predictable manner, the player behaviour metrics are highly correlated to character choices (e.g. some characters can heal much more than others, and some other characters can deal more damage than others). This combined with the previously defined set of 5 roles, would allow us to reduce the sparsity that all 141 selectable characters otherwise bring (2 teams with 5 members each with one of 141-character choices). Therefore we could build the classification mode separately for each team setup that exists in our data set, while concluding the rest of team setups ”undefined” in order to see how we can predict based on previously encountered team compositions.

Calculating the possible combinations of all the currently existing 141 champions (times the two sides teams may take), we obtain 1’234’899’493’035’516 possible combinations, and if we disregard the side of map each team is on we still have 617’449’746’517’758 possible combinations, as shown in shown in eq. (8).

$$\binom{141}{10}\times 2=\frac{141!}{(141-10)!}\times 2=617449746517758\times 2=1234899493035516 \qquad (8)$$

This is far more than is realistically being played (as most of the characters are only played in specific roles). While we could rely on gathering statistics from a third-party database such as champion.gg (Champion.gg, 2018), we instead relied on the gathered data alone for sake of concluding this work in a reasonable period of time. Calculating the number of unique match settings (specific 5 characters in the topside team together with specific other 5 characters in the bottomside team) we find how extremely sparse our data is when it comes to the character picks, each and every game has a unique setting as seen in the figure 35. Additionally, disregarding the side of the map and the team doesn't reduce the sparsity at all since the combinations of 10 characters in a single match are also all unique in our samples (as shown in figure 35). Taking the set of 5 characters from both teams in every match brings us some repeated settings, but still very few (at the very most 8 as shown in figure 36).

```
Loading matches from game version 8.7.1
Total of 88934 distinct side specific team setups
Total of 88934 distinct side irrespective team setups
Total of 174081 distinct team irrespective team setups
Loading matches from game version 8.8.1
Total of 76829 distinct side specific team setups
Total of 76829 distinct side irrespective team setups
Total of 149765 distinct team irrespective team setups
Loading matches from game version 8.9.1
Total of 57292 distinct side specific team setups
Total of 57292 distinct side irrespective team setups
Total of 112619 distinct team irrespective team setups
Loading matches from game version 8.10.1
Total of 241435 distinct side specific team setups
Total of 241435 distinct side irrespective team setups
Total of 458422 distinct team irrespective team setups
Loading matches from game version 8.11.1
Total of 150993 distinct side specific team setups
Total of 150993 distinct side irrespective team setups
Total of 294077 distinct team irrespective team setups
```

Figure 35. Total numbers of combinations found in each version (with varying number of matches). Only team irrespective setups have repeated combinations, all other are unique.

```
8.7.1
[('BOTTOM_005,BOTTOM_039,BOTTOM_042,BOTTOM_267,BOTTOM_498,TOP_037,TOP_064,TOP_098,TOP_134,TOP_145', 1), ('BOTTOM_027,BOTTOM
_028,BOTTOM_040,BOTTOM_134,BOTTOM_498,TOP_022,TOP_025,TOP_107,TOP_142,TOP_164', 1), ('BOTTOM_021,BOTTOM_069,BOTTOM_072,BOTT
OM_131,BOTTOM_497,TOP_018,TOP_025,TOP_041,TOP_104,TOP_142', 1)]
[('5,37,39,42,64,98,134,145,267,498', 1), ('22,25,27,28,40,107,134,142,164,498', 1), ('18,21,25,41,69,72,104,131,142,497',
1)]
[('13,14,121,497,498', 6), ('25,39,51,64,163', 5), ('13,121,164,497,498', 5)]

8.8.1
[('BOTTOM_034,BOTTOM_040,BOTTOM_041,BOTTOM_059,BOTTOM_222,TOP_015,TOP_048,TOP_101,TOP_150,TOP_267', 1), ('BOTTOM_007,BOTTOM
_072,BOTTOM_112,BOTTOM_412,BOTTOM_498,TOP_012,TOP_050,TOP_121,TOP_157,TOP_202', 1), ('BOTTOM_012,BOTTOM_048,BOTTOM_050,BOTT
OM_145,BOTTOM_164,TOP_043,TOP_081,TOP_104,TOP_114,TOP_432', 1)]
[('15,34,40,41,48,59,101,150,222,267', 1), ('7,12,50,72,112,121,157,202,412,498', 1), ('12,43,48,50,81,104,114,145,164,432'
, 1)]
[('7,104,202,412,516', 5), ('25,51,103,104,164', 5), ('41,103,104,497,498', 5)]

8.9.1
[('BOTTOM_028,BOTTOM_040,BOTTOM_043,BOTTOM_058,BOTTOM_081,TOP_015,TOP_016,TOP_056,TOP_105,TOP_114', 1), ('BOTTOM_036,BOTTOM
_051,BOTTOM_062,BOTTOM_136,BOTTOM_412,TOP_029,TOP_040,TOP_058,TOP_103,TOP_104', 1), ('BOTTOM_011,BOTTOM_053,BOTTOM_058,BOTT
OM_090,BOTTOM_222,TOP_002,TOP_005,TOP_015,TOP_245,TOP_497', 1)]
[('15,16,28,40,43,56,58,81,105,114', 1), ('29,36,40,51,58,62,103,104,136,412', 1), ('2,5,11,15,53,58,90,222,245,497', 1)]
[('54,104,157,497,498', 5), ('24,25,51,107,142', 5), ('39,40,81,104,105', 4)]

8.10.1
[('BOTTOM_026,BOTTOM_092,BOTTOM_104,BOTTOM_131,BOTTOM_145,TOP_028,TOP_043,TOP_081,TOP_105,TOP_157', 1), ('BOTTOM_025,BOTTOM
_051,BOTTOM_131,BOTTOM_150,BOTTOM_164,TOP_042,TOP_081,TOP_104,TOP_117,TOP_122', 1), ('BOTTOM_043,BOTTOM_054,BOTTOM_104,BOTT
OM_131,BOTTOM_498,TOP_007,TOP_025,TOP_051,TOP_056,TOP_150', 1)]
[('26,28,43,81,92,104,105,131,145,157', 1), ('25,42,51,81,104,117,122,131,150,164', 1), ('7,25,43,51,54,56,104,131,150,498'
, 1)]
[('39,64,142,497,498', 8), ('25,81,104,142,164', 7), ('8,39,40,48,81', 7)]

8.11.1
[('BOTTOM_027,BOTTOM_045,BOTTOM_056,BOTTOM_202,BOTTOM_412,TOP_064,TOP_076,TOP_161,TOP_429,TOP_555', 1), ('BOTTOM_008,BOTTOM
_045,BOTTOM_119,BOTTOM_427,BOTTOM_555,TOP_030,TOP_034,TOP_067,TOP_076,TOP_497', 1), ('BOTTOM_055,BOTTOM_080,BOTTOM_081,BOTT
OM_089,BOTTOM_121,TOP_074,TOP_077,TOP_114,TOP_236,TOP_432', 1)]
[('27,45,56,64,76,161,202,412,429,555', 1), ('8,30,34,45,67,76,119,427,497,555', 1), ('55,74,77,80,81,89,114,121,236,432',
1)]
[('39,48,81,238,267', 5), ('8,24,104,201,236', 5), ('5,99,122,236,555', 5)]
```

Figure 36. Maximum numbers of repetition for each combination type for each version.

Gathering historical mid-game data and modelling it

We could have gathered historical mid-game data from several points of view, namely from over 6 following categories of a player's historical matches: All (i.e. without limitations), while in a specific role, while as a specific character, while using a specific set of summonerspells, while using a specific set of runes, or any mixture of these.

Additionally we limited the lookup of historical matches using a maximum time span (since the statistics from games played years ago have virtually no correlation to the player's capabilities today) and a maximum number of matches (due to some people playing much more than the others and it'd otherwise make data gathering occasionally extremely slow and data possibly skewed).

To find out the relevant point of view, as well as the relevant limit on the number of games, we gathered the number of available historical matches in all 5 aforementioned categories, then determined the available parameters. As a maximum time span we used 3 weeks.

Using data from 885 matches (including 10 participants each and thus 8850 sets of match histories), we obtained a summary shown in figure 37. Based on this summary together with kernel density plots from figure 38, we concluded that globally (across all participants) a sweet spot would be 50 matches in total, including additionally the matches played in the current role. It includes well over half the participants (first quartile being 38, and median being 76). While matches played as the current character would be an interesting feature, they're far too limited to be included (only containing a few samples each with median value of 7).

	history_total	history_in_role	history_as_character	history_with_summonerspells	history_with_runes
count	8850.000000	8850.000000	8850.000000	8850.000000	8850.000000
mean	89.802826	35.491188	17.195141	47.462711	11.329378
std	66.768501	46.050713	27.885807	48.112877	20.751976
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	38.000000	6.000000	2.000000	11.000000	0.000000
50%	76.000000	18.000000	7.000000	32.000000	3.000000
75%	126.000000	46.000000	20.000000	71.000000	13.000000
max	515.000000	397.000000	309.000000	411.000000	279.000000

Figure 37. Summary of available historical matches for 8850 participants.

To confirm that there are no role-specific variations in available data, we plotted the distributions over each role. Based on kernel density estimation plots shown in figure 38, as well as box-plots in figure 39, we established that distributions are indeed the same across all roles.

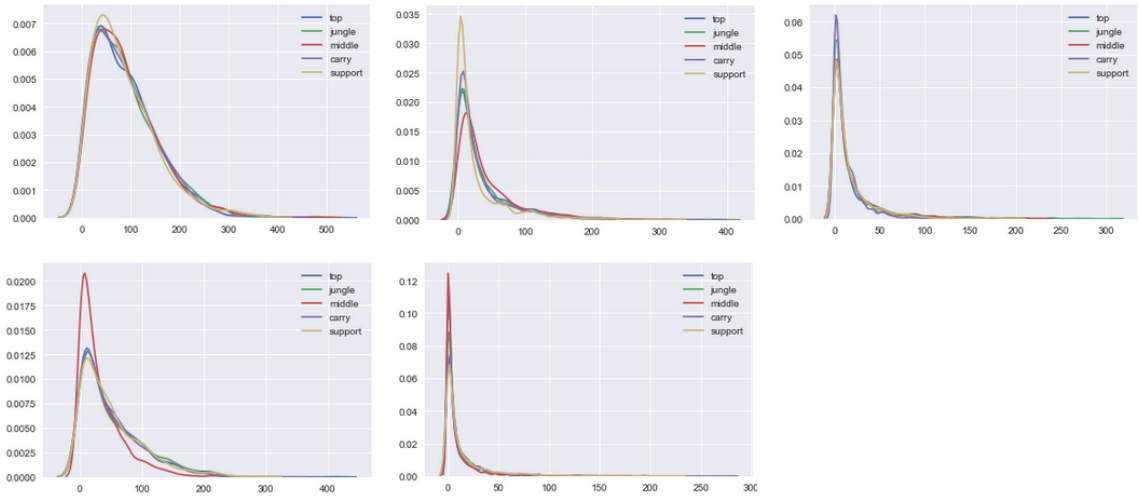


Figure 38. Kernel density estimation plots of the 5 categories in respective order of figure 37.

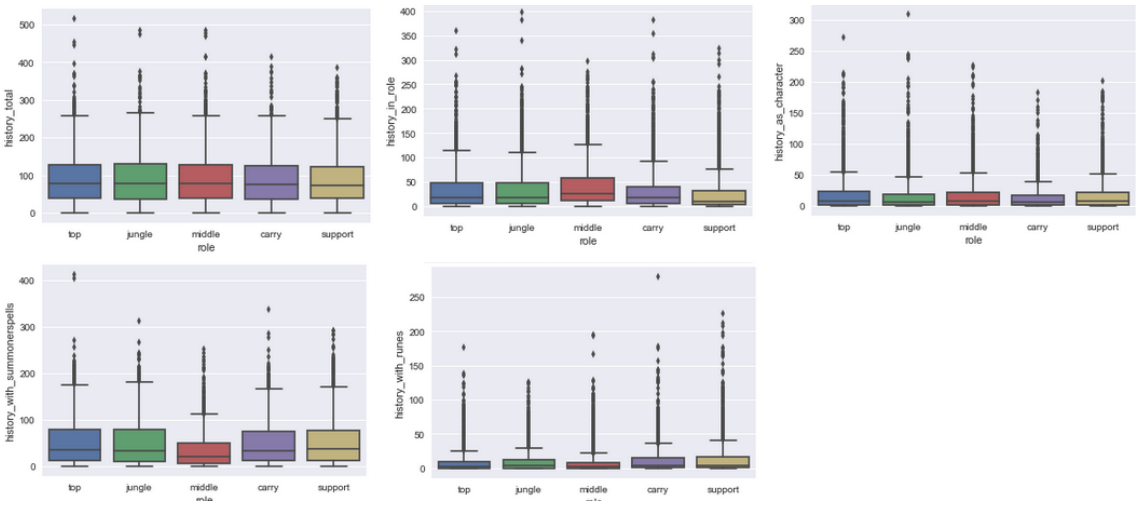


Figure 39. Box-plots of the 5 categories in respective order of figure 37.

We started off gathering the following attributes of each role (in the context of a team):

- Kills, deaths, and assists (of games played on the current lane)
- Number of consecutive wins (of games played on the current lane)
- Number of consecutive losses (of games played on the current lane)

Using these with exactly the same setup as described previously (in the beginning of chapter 4.4), we obtained accuracies varying around 0.51 and 0.52 (across the different neural network architectures).

To verify the system is still operating as intended, we momentarily added knowledge of "win" amongst the data points, and as expected the neural nets resulted in 1.0 accuracy ubiquitously. This proved we have a working model, but the attributes we had were not enough of data to draw conclusions on win (or loss).

Finally we performed the same training and validation on data points consisting of the following attributes, all of games played on the current lane:

- Character choice
- Kills, deaths, and assists
- Number of consecutive wins
- Number of consecutive losses
- Damage taken during the first 10 minutes
- Gold acquired during the first 10 minutes
- Minions killed during the first 10 minutes

This final set of data points resulted in models with validation accuracies around 0.53 as shown in figure 40. This concludes we are unlikely to obtain higher accuracy by simply adding more historical data.

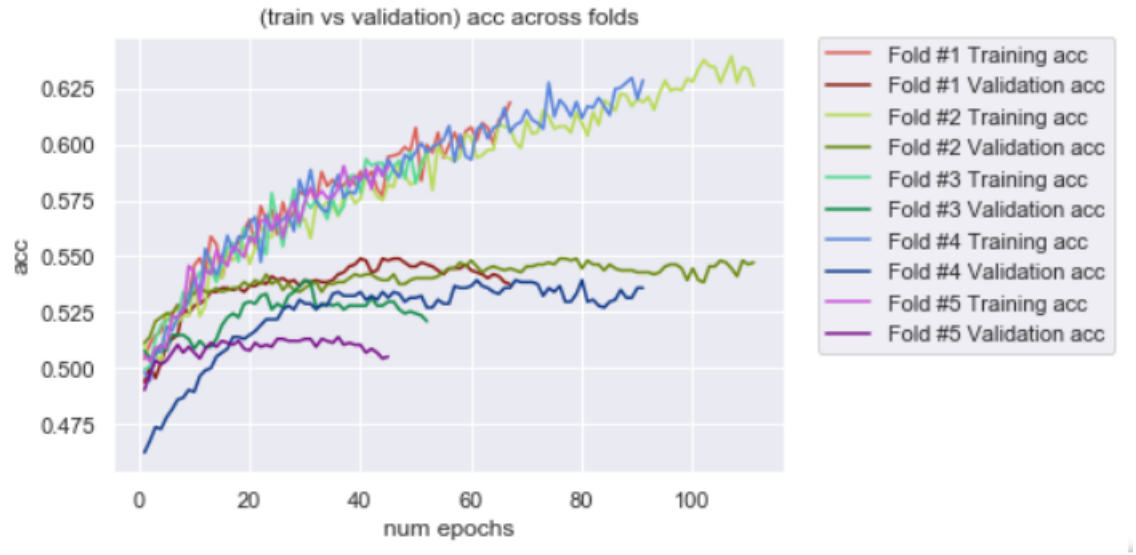


Figure 40. Both training and validation accuracies from one of the utilised deep neural network models. Validation accuracy around 0.53.

5 Findings

5.1 Results

Over the research, we gathered data from over a million historical matches. To put this into perspective, each match took an average of 25 minutes playtime with data points from 10 distinct players, comparable to approximately 173 611 days (or 475 years) of play time. Our methods for extracting and transforming the data proved to be efficient, however the loading of data for analytical purposes could have been more streamlined.

The monitoring user interface (that was developed after initial data gathering pipeline) turned out to be an invaluable tool, minimizing manual maintenance work and giving a holistic view on the available data at will. After its initial conception it did not require any further modifications for the purposes of this research.

Exploratory data analysis on the gathered data pointed out how relatively balanced the available character choices are, with win ratios of all characters converging towards the range between 45 and 55 per cent with more matches played. The same effect applied to combinations of pre-match choices (such as a combination of a certain character with a certain set of runes on a certain lane) as well as team compositions. These were summarised in figures 22, 23, and 24.

Utilising machine learning to try determine whether a certain team composition is going to win or lose a game against a certain other team composition, was unsuccessful in finding correlations, and at the best resulted in a prediction accuracy of 54 per cent.

Adding (various alternative) sets of refined historical data on each participants' past games, did not improve the results, even though availability of the historical data itself was highly consistent across different roles of the players (top, jungle, mid, carry, and support), as seen in figures 38 and 39.

5.2 Discussion

While the conclusion of the research was unfavourable, a lot of insight was found from the intermediate steps taken.

The data gathering pipeline was relatively painless to set up (namely, it worked exactly as its documentation described, without any hiccups), and it lasted all the way until the end of the research. Same architecture could be utilised in similar research projects, wherever there is a need to extract and transform data to a relational form (via ORM abstraction). Key frameworks in the system were Django and React (for user interface).

In the exploratory data analysis, we found that win ratios (around 45 to 55 per cent) matched the (1 and 2 dimensions of) aggregate data listed on the fan-sites mentioned in preface. This manual summarising of data points also enables us to tell whether a combination of an individual participant's choices is good or bad, and whether a combination of specific teammates is good or bad (even if it does not determine the outcome). This information could be utilised in an automated expert system to give a player advice on his or her choices. Such an expert system would be interesting to see in action, and it would be useful to many players of the video game who nowadays rely on the low (1 and 2) dimension aggregate information provided by fan-sites and on asking other players over the internet. This research could be used as a base for such a system.

Finally our conclusion on the predictability of matches' outcome was, that based on the pre-game choices of each participant as well as their past game information, we cannot predict the outcome of a given match with higher accuracy than 55 per cent at most (which is not sufficient to justify gambling over the outcome or giving up in the beginning of a match).

Further research could be performed also on a smaller scale, for example on trying to predict an individual player's playing behaviour. Such research could use the same system architecture as in this research.

References

1 Volk, P. (2016). League of Legends now boasts over 100 million monthly active players worldwide. [online] The Rift Herald.
Available at: https://www.riftherald.com/2016/9/13/12865314/monthly-lol-players-2016-active-worldwide [Accessed 6 Feb. 2018].

2 BBC News. (2011). Finnish football in fix scandal. [online]
Available at: http://www.bbc.com/news/world-europe-13736085 [Accessed 21 Dec. 2017].

3 PCGamesN. (2013). roX.KIS team banned from Starladder Dota 2 tournament over allegations of betting fraud. [online] PCGamesN.
Available at: https://www.pcgamesn.com/dota/roxkis-team-banned-starladder-dota-2-tournament-betting-fraud [Accessed 21 Dec. 2017].

4 Valve. (2015). Integrity and Fair Play. [online]
Available at: http://blog.counter-strike.net/index.php/2015/01/11261/ [Accessed 21 Dec. 2017].

5 Panetta, K. (2017). Top Trends in the Gartner Hype Cycle for Emerging Technologies, 2017. [online] Smarter With Gartner.
Available at: https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/ [Accessed 6 Feb. 2018].

6 Developer.riotgames.com. (2018a). Riot Developer Portal. [online]
Available at: https://developer.riotgames.com/api-methods/ [Accessed 6 Feb. 2018].

7 Signup.leagueoflegends.com. (2018). League of Legends - Signup Index - MOBA mention. [online]
Available at: https://signup.leagueoflegends.com/en/overview/index [Accessed 8 Feb. 2018].

8 Leaguepedia. (2017). 2017 Season World Championship. [online]
Available at: https://lol.gamepedia.com/2017_Season_World_Championship [Accessed 8 Feb. 2018].

9 Liquipedia Dota2 Wiki. (2017). The International 2017. [online]
Available at: http://liquipedia.net/dota2/The_International/2017#Prize_Pool [Accessed 8 Feb. 2018].

10 Blitz Esports. (2017). MarkZ explains why LCS teams often recruit from solo queue instead of the Challenger leagues. [online]
Available at: https://blitzesports.com/lol/video/715/markz-explains-why-lcs-teams-often-recruit-solo-queue-instea [Accessed 30 Dec. 2017].

11 Twitch. (2017). League of Legends - Live Streams - Twitch. [online]
Available at: https://www.twitch.tv/directory/game/League%20of%20Legends [Accessed 30 Dec. 2017].

12 Na.leagueoflegends.com. (2018a). New Player Guide. [online]
Available at: https://na.leagueoflegends.com/en/featured/new-player-guide [Accessed 10 Feb. 2018].

13 Boards.na.leagueoflegends.com. (2017). Player Behaviour Forum rant. [online]
Available at: https://boards.na.leagueoflegends.com/en/c/player-behavior-moderation/LmUoMPb3-riot-your-automated-system-is-broken-bring-back-tribunals?comment=00040001000000000001 [Accessed 14 Feb. 2018].

14 Developer.riotgames.com. (2018b). Riot Developer Portal - MatchDto. [online]
Available at: https://developer.riotgames.com/api-methods/#match-v3/GET_getMatch [Accessed 14 Feb. 2018].

15 Developer.riotgames.com. (2018c). Riot Developer Portal - Static Data. [online]
Available at: https://developer.riotgames.com/api-methods/#lol-static-data-v3 [Accessed 14 Feb. 2018].

16 Developer.riotgames.com. (2018d). Riot Developer Portal - MatchTimelineDto. [online]
Available at: https://developer.riotgames.com/api-methods/#match-v3/GET_getMatchTimeline [Accessed 14 Feb. 2018].

17 Postgresql.org. (2018a). PostgreSQL: Documentation: 9.5: Numeric Types - 8.1.4. Serial Types. [online]
Available at: https://www.postgresql.org/docs/9.5/static/datatype-numeric.html#DATATYPE-SERIAL [Accessed 25 Feb. 2018].

18 Postgresql.org. (2018b). PostgreSQL: About. [online]
Available at: https://www.postgresql.org/about/ [Accessed 25 Feb. 2018].

19 Cybertec. (2017). Why favor PostgreSQL over MariaDB / MySQL. [online]
Available at: https://www.cybertec-postgresql.com/en/why-favor-postgresql-over-mariadb-mysql/ [Accessed 25 Feb. 2018].

20 Developer.riotgames.com. (2018e). Riot Developer Portal - Getting Started. [online]
Available at: https://developer.riotgames.com/getting-started.html [Accessed 25 Feb. 2018].

21 Docs.djangoproject.com. (2018). Models | Django documentation | Django. [online]
Available at: https://docs.djangoproject.com/en/2.0/topics/db/models/ [Accessed 30 Mar. 2018].

22 GitHub. (2018a). Mew-www/lol-data-collection-system. [online]
Available at: https://github.com/Mew-www/lol-data-collection-system/blob/master/dj_lol_dcs/lolapi/models.py [Accessed 30 Mar. 2018].

23 GitHub. (2018b). Mew-www/lol-data-collection-system. [online]
Available at: https://github.com/Mew-www/lol-data-collection-system/blob/master/dj_lol_dcs/proto_data_gathering.py [Accessed 3 Apr. 2018].

24 Mysqlclient.readthedocs.io. (2018). MySQLdb User's Guide — MySQLdb 1.2.4b4 documentation. [online]
Available at: http://mysqlclient.readthedocs.io/user_guide.html#introduction [Accessed 6 Apr. 2018].

25 Man7.org. (2018). top(1) - Linux manual page. [online]
Available at: http://man7.org/linux/man-pages/man1/top.1.html [Accessed 7 Apr. 2018].

26 Fetch.spec.whatwg.org. (2018). Fetch Standard. [online]
Available at: https://fetch.spec.whatwg.org/ [Accessed 9 Apr. 2018].

27 GitHub. (2018c). Mew-www/lol-dcs-monitorpage. [online]
Available at: https://github.com/Mew-www/lol-dcs-monitorpage [Accessed 9 Apr. 2018].

28 Meltdownattack.com. (2018). Meltdown and Spectre. [online]
Available at: https://meltdownattack.com/ [Accessed 11 Apr. 2018].

29 GitHub. (2018d). Mew-www/lol-data-collection-system. [online]
Available at: https://github.com/Mew-www/lol-data-collection-system/blob/master/dj_lol_dcs/periodical_data_repair.py [Accessed 11 Apr. 2018].

30 Postgresql.org. (2018c). PostgreSQL: Documentation: 9.6: SQL Dump. [online]
Available at: https://www.postgresql.org/docs/9.6/static/backup-dump.html [Accessed 11 Apr. 2018].

31 Postgresql.org. (2018d). PostgreSQL: Documentation: 10: 33.14. Environment Variables. [online]
Available at: https://www.postgresql.org/docs/current/static/libpq-envars.html [Accessed 16 Apr. 2018].

32 Linux.die.net. (2018). ssh-keygen(1) - Linux man page. [online]
Available at: https://linux.die.net/man/1/ssh-keygen [Accessed 20 Apr. 2018].

33 R-project.org. (2018). R: What is R?. [online] Available at: https://www.r-project.org/about.html [Accessed 22 Apr. 2018].

34 Docs.python.org. (2018). General Python FAQ — Python 3.6.5 documentation. [online]
Available at: https://docs.python.org/3/faq/general.html#what-is-python [Accessed 22 Apr. 2018].

35 Kaggle.com. (2018). Learn | Kaggle. [online]
Available at: https://www.kaggle.com/learn/overview [Accessed 22 Apr. 2018].

36 Pandas.pydata.org. (2018a). Package overview — pandas 0.22.0 documentation. [online]
Available at: http://pandas.pydata.org/pandas-docs/stable/overview.html [Accessed 22 Apr. 2018].

37 Docs.scipy.org. (2018). About NumPy — NumPy v1.14 Manual. [online]
Available at: https://docs.scipy.org/doc/numpy-1.14.0/about.html [Accessed 22 Apr. 2018].

38 Matplotlib.org. (2018). Matplotlib: Python plotting — Matplotlib 2.2.2 documentation. [online]
Available at: https://matplotlib.org/ [Accessed 22 Apr. 2018].

39 Seaborn.pydata.org. (2018). An introduction to seaborn — seaborn 0.8.1 documentation. [online]
Available at: http://seaborn.pydata.org/introduction.html#introduction [Accessed 22 Apr. 2018].

40 Pandas.pydata.org. (2018b). pandas.io.json.json_normalize — pandas 0.22.0 documentation. [online]
Available at: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.io.json.json_normalize.html [Accessed 1 May 2018].

41 Leagueofgraphs.com. (2018). Champions stats - League of Legends. [online]
Available at: https://www.leagueofgraphs.com/champions/stats [Accessed 3 May 2018].

42 OP.GG Europe West. (2018a). Game stats by champion - League of Legends. [online]
Available at: http://euw.op.gg/statistics/champion/ [Accessed 7 May 2018].

43 Champion.gg. (2018). Champion.gg - LoL Champion Stats, Builds, Runes, Masteries, Counters and Matchups!. [online]
Available at: http://champion.gg/ [Accessed 7 May 2018].

44 Na.leagueoflegends.com. (2018b). Champions | League of Legends. [online]
Available at: https://euw.leagueoflegends.com/en/game-info/champions/ [Accessed 7 May 2018].

45 Ddragon.leagueoflegends.com. (2018). [online]
Available at: http://ddragon.leagueoflegends.com/cdn/8.8.1/data/en_US/item.json [Accessed 7 May 2018].

46 OP.GG Europe West. (2018b). Véigâr v3 - Summoner Stats - League of Legends. [online]
Available at: http://euw.op.gg/summoner/userName=v%C3%A9ig%C3%A2rv3 [Accessed 9 May 2018].

47 reddit. (2018). Meta right now • r/summonerschool. [online]
Available at:
https://www.reddit.com/r/summonerschool/comments/83egac/meta_right_now/ [Accessed 19 May 2018].

48 Google Developers. (2018). Machine Learning Glossary. [online]
Available at: https://developers.google.com/machine-learning/glossary/#hyperparameter
[Accessed 20 Jul. 2018].

49 Hao Yi Ong, Sunil Deolalikar, Mark Peng. (2015) Player Behavior and Optimal Team Composition for Online Multiplayer Games.
Available at: https://arxiv.org/abs/1503.02230
[Accessed: 29th June 2018]

50 TensorFlow. (2018). TensorBoard: Visualizing Learning | TensorFlow. [online] Available at: https://www.tensorflow.org/guide/summaries_and_tensorboard [Accessed 20 Jul. 2018].

Appendix 1 - Various neural network architectures trained on selected characters versus the outcome of a match – IPython printout

Loading matches from game version 8.7.1
Flattened dataframes of (88606, 281)
Shuffled and reformatted 88606 * 280(x)+1(y) dataframe suitable for tensors
Crossvalidating matches from game version 8.7.1 over various models

Starting 5-fold crossvalidation for the following architecture:

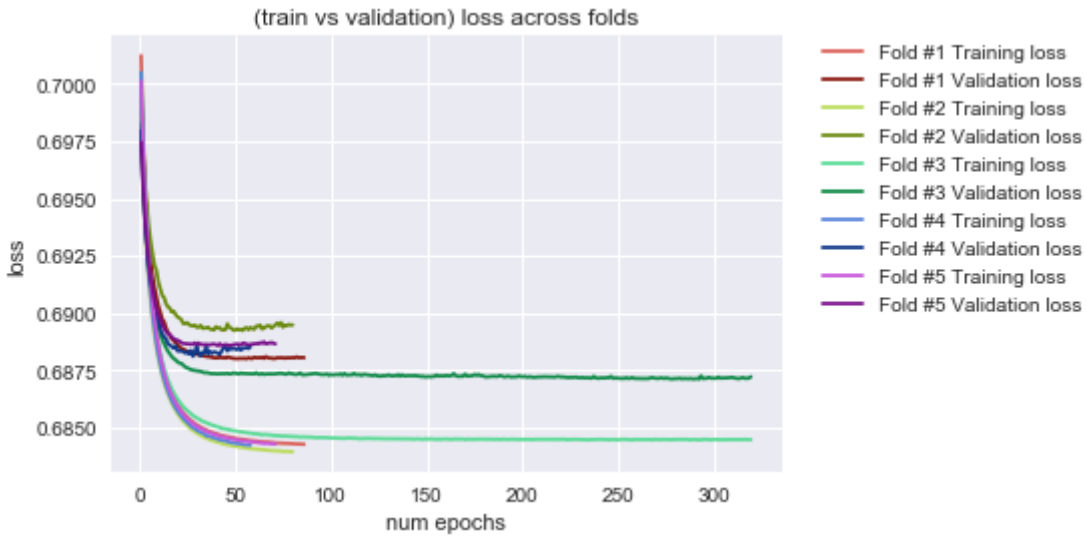
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 1)                 281
=================================================================
Total params: 281
Trainable params: 281
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
```

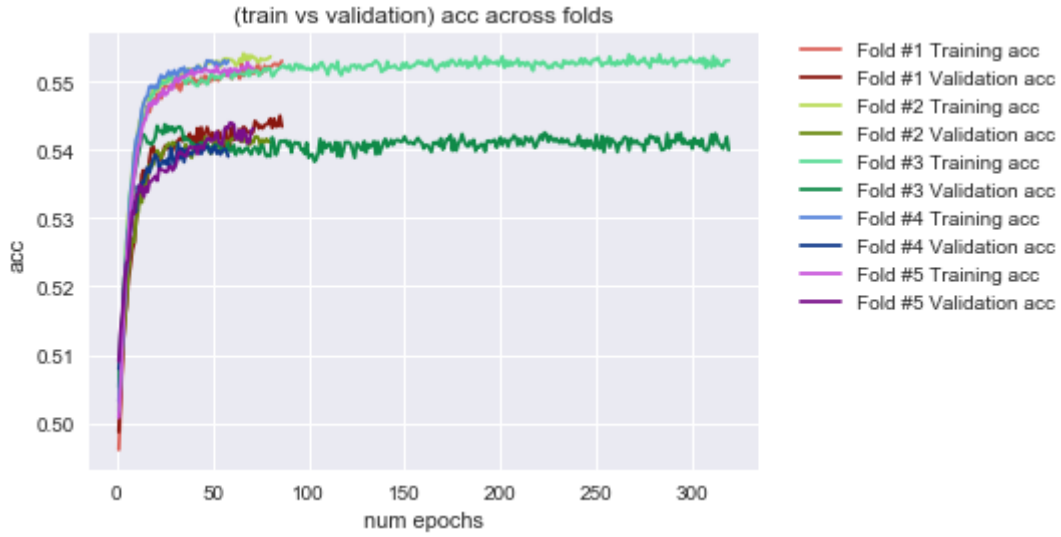
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 0s 34us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





17722/17722 [==============================] - 1s 33us/step

Model #1 on test set: [0.6882519406371509, 0.5400631982913446]

17722/17722 [==============================] - 1s 33us/step

Model #2 on test set: [0.6884685864621826, 0.5411353120448936]

17722/17722 [==============================] - 1s 32us/step

Model #3 on test set: [0.6877203224246382, 0.545141631880082]

17722/17722 [==============================] - 1s 33us/step

Model #4 on test set: [0.6878062817598625, 0.5451416318767187]

17722/17722 [==============================] - 1s 33us/step

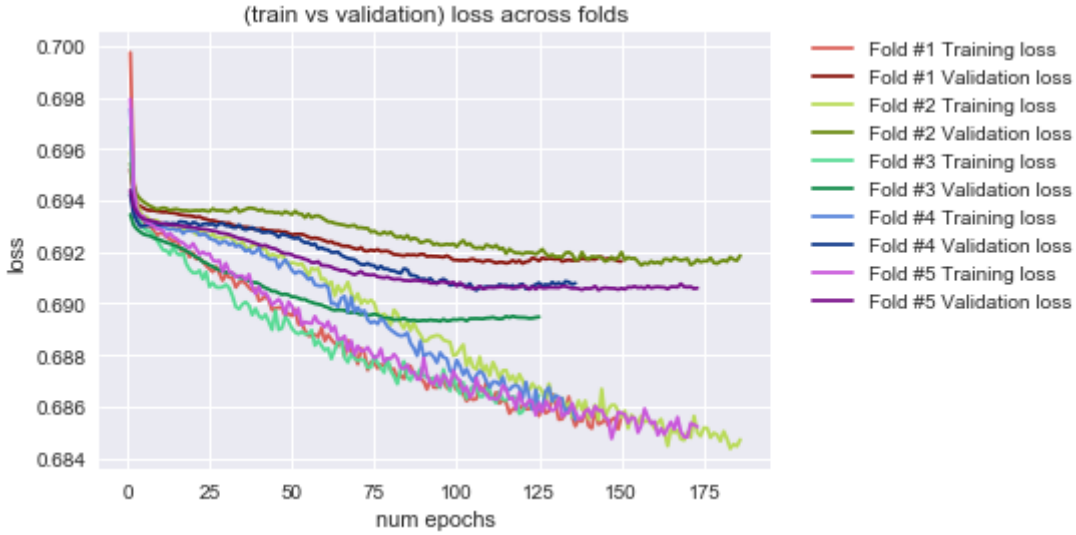
Model #5 on test set: [0.6885693046231663, 0.5420381446890201]

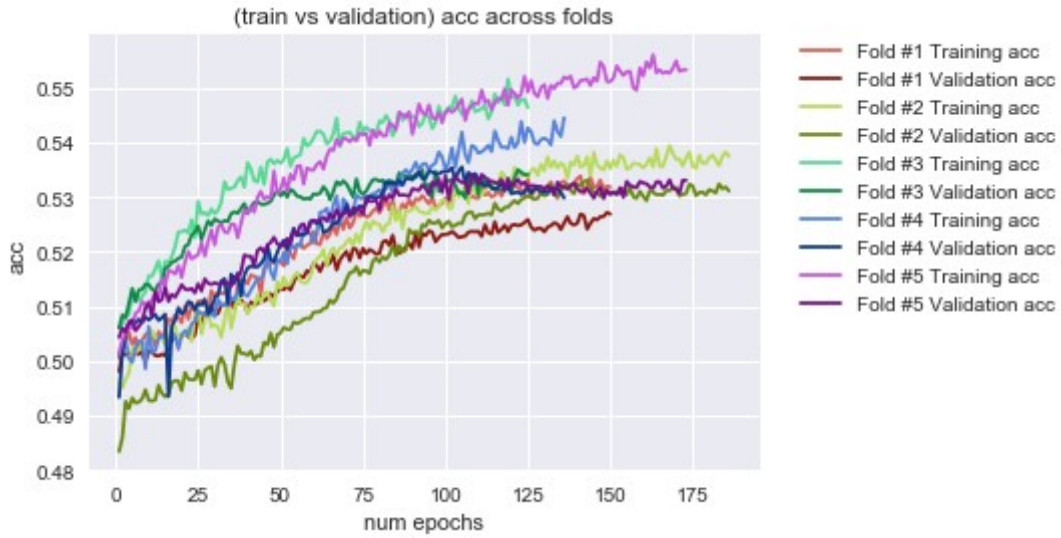
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 3)                 843
_________________________________________________________________
dropout_1 (Dropout)          (None, 3)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 4
=================================================================
Total params: 847
Trainable params: 847
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 1s 35us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 1s 36us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 1s 37us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 1s 37us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 36us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.53 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.53





17722/17722 [==============================] - 1s 38us/step
Model #1 on test set: [0.6919508540136606, 0.5235865026554342]
17722/17722 [==============================] - 1s 35us/step
Model #2 on test set: [0.6911368309449241, 0.5287213632772825]

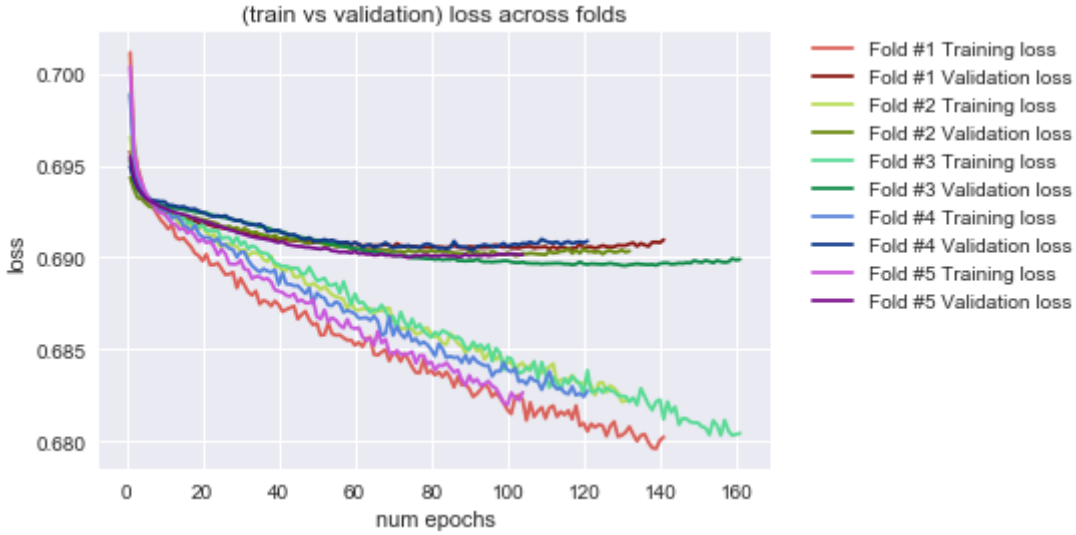
17722/17722 [==============================] - 1s 36us/step
Model #3 on test set: [0.6896914413074596, 0.5385960952589331]
17722/17722 [==============================] - 1s 35us/step
Model #4 on test set: [0.6906281576954664, 0.5350976187856454]
17722/17722 [==============================] - 1s 36us/step
Model #5 on test set: [0.6903175621360436, 0.531711996405486]

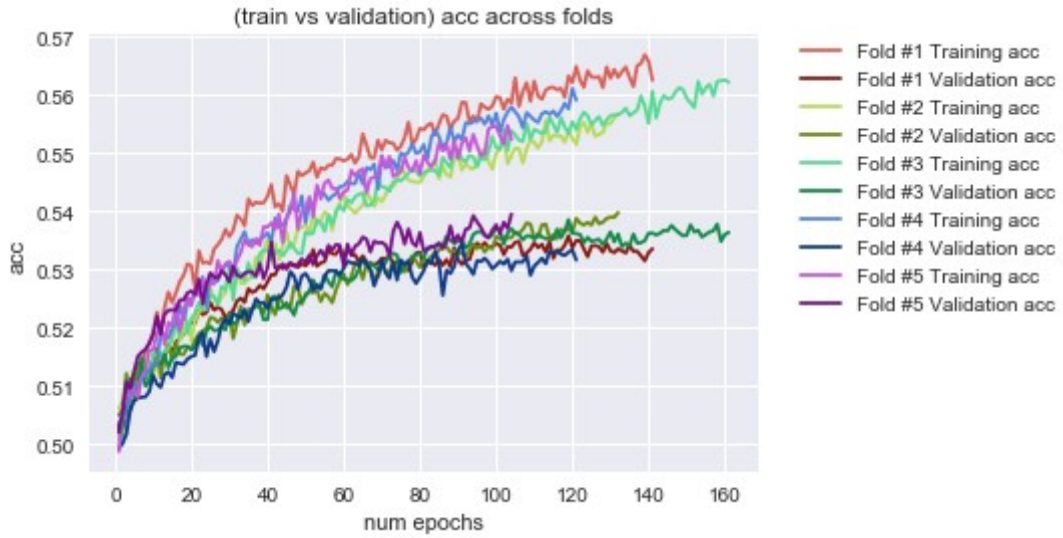
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 9)                 2529
_________________________________________________________________
dropout_6 (Dropout)          (None, 9)                 0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 10
=================================================================
Total params: 2,539
Trainable params: 2,539
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 1s 43us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 0s 35us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 0s 35us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 38us/step

loss average 0.69 (+-0.000 stdev)
```

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.54





17722/17722 [==============================] - 1s 38us/step
Model #1 on test set: [0.689810672371103, 0.5350411917491713]
17722/17722 [==============================] - 1s 37us/step
Model #2 on test set: [0.6893365600754587, 0.5400067712447805]

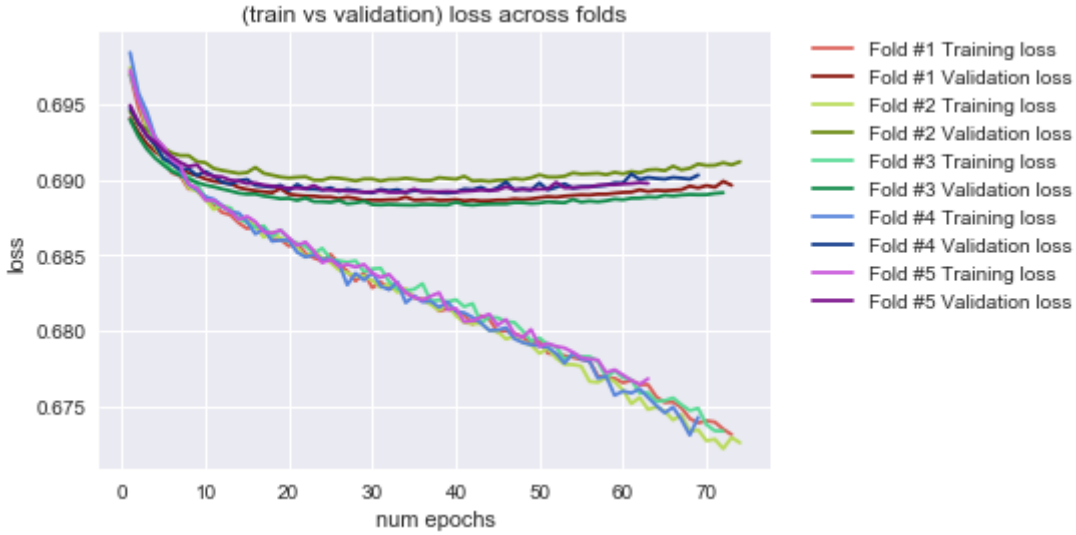
17722/17722 [==============================] - 1s 39us/step
Model #3 on test set: [0.6902208970707246, 0.5409660309623776]
17722/17722 [==============================] - 1s 36us/step
Model #4 on test set: [0.6895212171820617, 0.5314298611995719]
17722/17722 [==============================] - 1s 36us/step
Model #5 on test set: [0.6898255679155632, 0.5361697325459212]

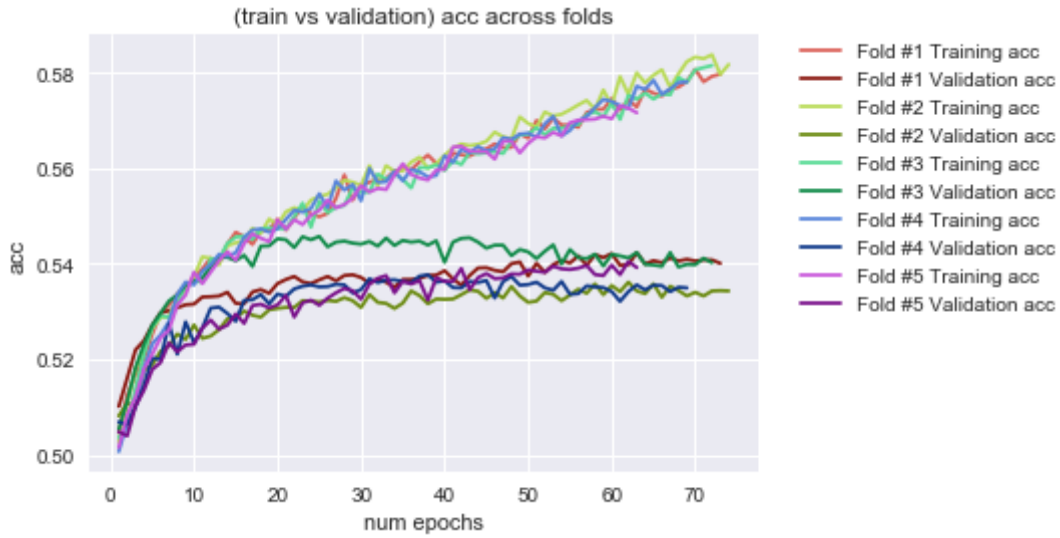
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 280)               78680
_________________________________________________________________
dropout_6 (Dropout)          (None, 280)               0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 281
=================================================================
Total params: 78,961
Trainable params: 78,961
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 1s 35us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 1s 55us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 1s 37us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 39us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.54





17722/17722 [==============================] - 1s 39us/step
Model #1 on test set: [0.6895810030690839, 0.5405146146066813]
17722/17722 [==============================] - 1s 36us/step
Model #2 on test set: [0.690092451348287, 0.5378625437343192]

17722/17722 [==============================] - 1s 36us/step
Model #3 on test set: [0.6891565280834556, 0.5433923936484831]
17722/17722 [==============================] - 1s 40us/step
Model #4 on test set: [0.6892162769269411, 0.5402324794108571]
17722/17722 [==============================] - 1s 37us/step
Model #5 on test set: [0.6893617534069724, 0.5392732197302564]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 3)	843
dropout_6 (Dropout)	(None, 3)	0
dense_12 (Dense)	(None, 3)	12
dropout_7 (Dropout)	(None, 3)	0
dense_13 (Dense)	(None, 1)	4

Total params: 859
Trainable params: 859
Non-trainable params: 0

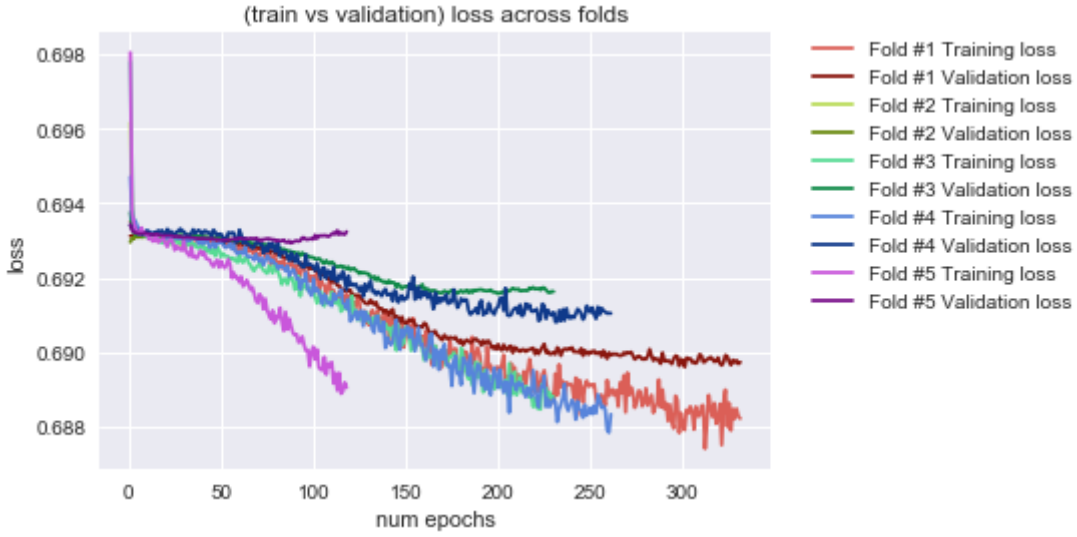
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 0s 34us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 0s 35us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 1s 38us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 1s 39us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples

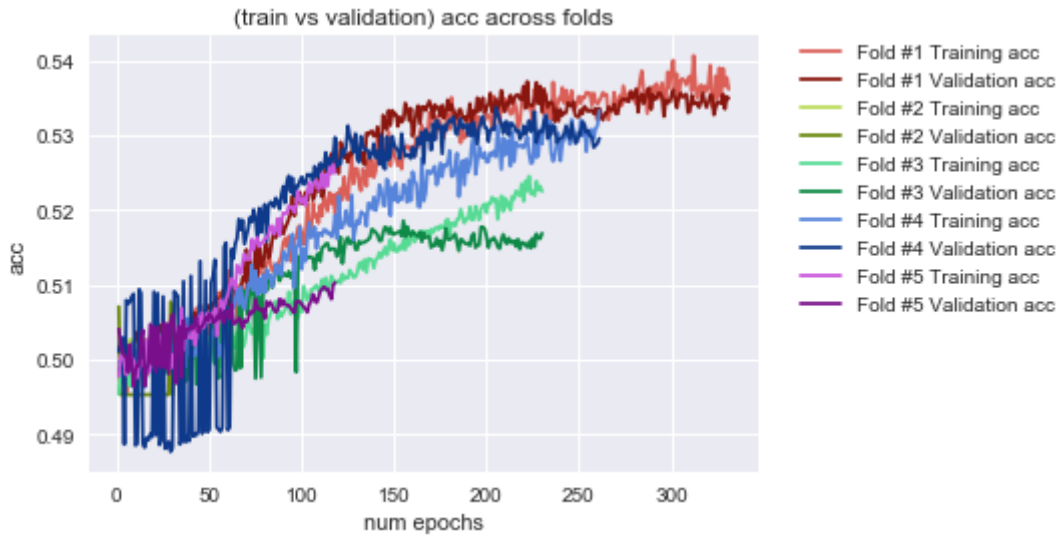
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 39us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.52 (+-0.011 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.51
Scores #3 loss: 0.69 acc: 0.52
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.51





```
17722/17722 [==============================] - 1s 40us/step
Model #1 on test set: [0.6904155935206325, 0.5339690779754425]
17722/17722 [==============================] - 1s 41us/step
Model #2 on test set: [0.6932149763678256, 0.501918519347748]
17722/17722 [==============================] - 1s 40us/step
Model #3 on test set: [0.6926718031885177, 0.5172666742306318]
17722/17722 [==============================] - 1s 39us/step
Model #4 on test set: [0.6902811530279179, 0.5315991423089945]
17722/17722 [==============================] - 1s 38us/step
Model #5 on test set: [0.6930948122335252, 0.513034646189007]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 32)	8992
dropout_11 (Dropout)	(None, 32)	0
dense_17 (Dense)	(None, 16)	528
dropout_12 (Dropout)	(None, 16)	0
dense_18 (Dense)	(None, 1)	17

Total params: 9,537
Trainable params: 9,537
Non-trainable params: 0

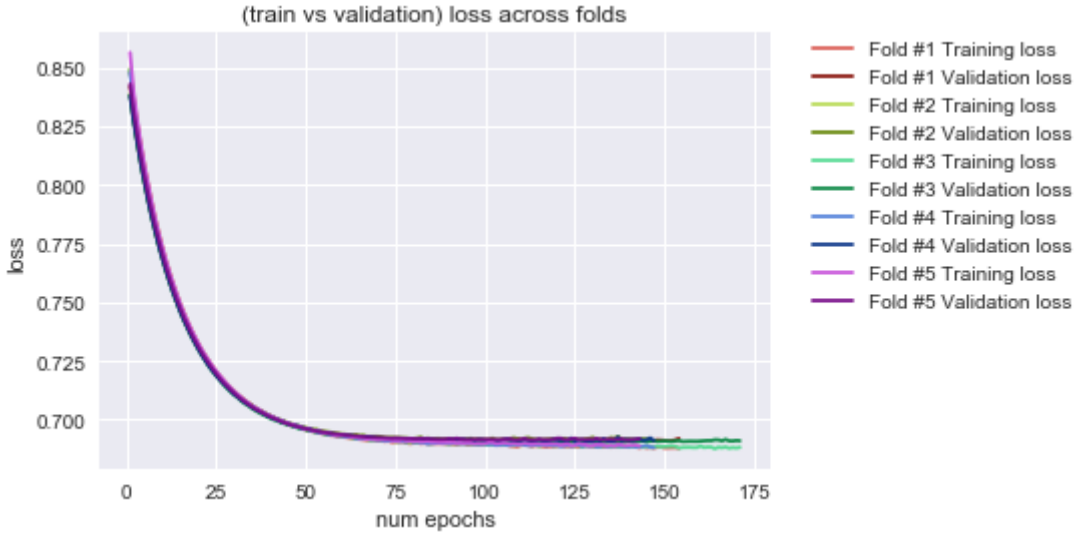
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 1s 38us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 1s 42us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 1s 41us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 1s 63us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples

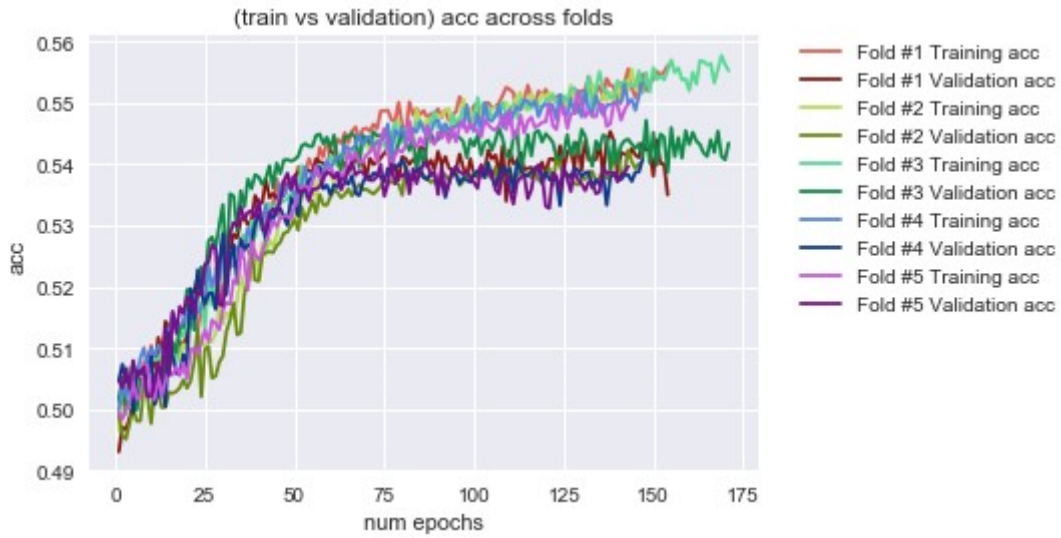
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 44us/step

loss average 0.69 (+-0.000 stdev)

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





```
17722/17722 [==============================] - 1s 43us/step
Model #1 on test set: [0.6927360853348665, 0.534194786151609]
17722/17722 [==============================] - 1s 40us/step
Model #2 on test set: [0.6912956460982463, 0.5411917390813678]
17722/17722 [==============================] - 1s 40us/step
Model #3 on test set: [0.6907925565615961, 0.5426588421238692]
17722/17722 [==============================] - 1s 40us/step
Model #4 on test set: [0.6907933682771693, 0.541756009483106]
17722/17722 [==============================] - 1s 40us/step
Model #5 on test set: [0.6919572457760294, 0.533856223912584]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 280)	78680
dropout_11 (Dropout)	(None, 280)	0
dense_17 (Dense)	(None, 120)	33720
dropout_12 (Dropout)	(None, 120)	0
dense_18 (Dense)	(None, 1)	121

Total params: 112,521
Trainable params: 112,521
Non-trainable params: 0

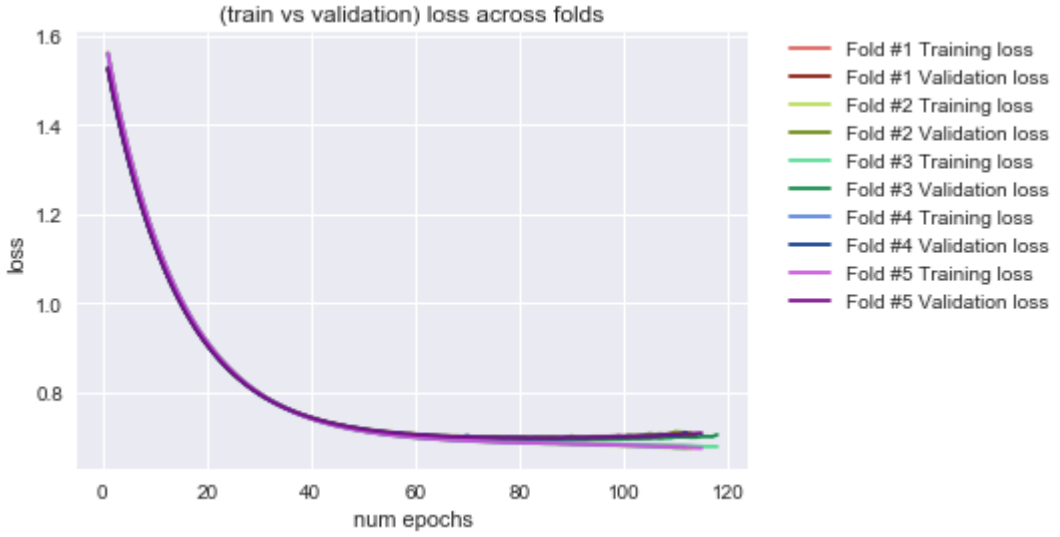
Training fold 1 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 1 of 5 . . .
14177/14177 [==============================] - 1s 44us/step
Training fold 2 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 2 of 5 . . .
14177/14177 [==============================] - 1s 40us/step
Training fold 3 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 3 of 5 . . .
14177/14177 [==============================] - 1s 39us/step
Training fold 4 of 5 . . .
Training using 56707 samples and validating using 14177 samples
Evaluating fold 4 of 5 . . .
14177/14177 [==============================] - 1s 40us/step
Training fold 5 of 5 . . .
Training using 56708 samples and validating using 14176 samples

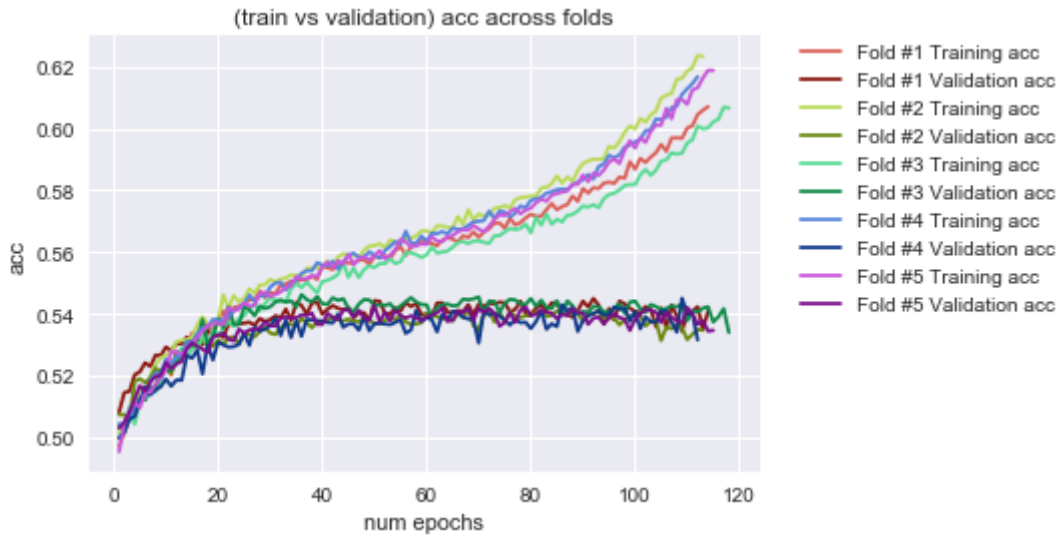
Evaluating fold 5 of 5 . . .
14176/14176 [==============================] - 1s 57us/step

loss average 0.71 (+-0.003 stdev)

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.70 acc: 0.54
Scores #2 loss: 0.71 acc: 0.53
Scores #3 loss: 0.71 acc: 0.53
Scores #4 loss: 0.71 acc: 0.53
Scores #5 loss: 0.71 acc: 0.53





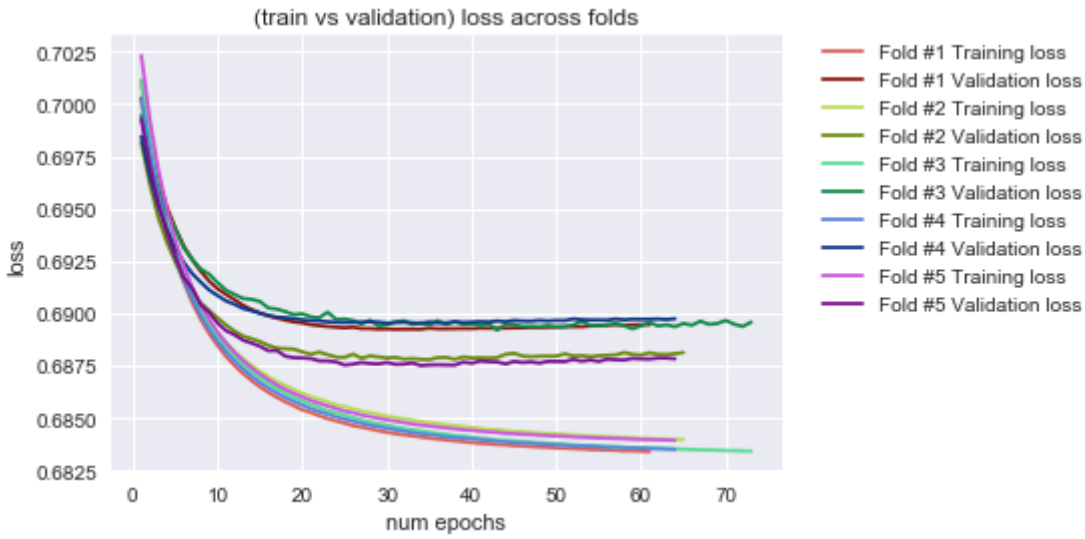
17722/17722 [==============================] - 1s 47us/step
Model #1 on test set: [0.7027847137600234, 0.5392732197302564]
17722/17722 [==============================] - 1s 45us/step
Model #2 on test set: [0.7094788777277978, 0.5370161381434835]
17722/17722 [==============================] - 1s 53us/step
Model #3 on test set: [0.7063202704469556, 0.5384268141394207]
17722/17722 [==============================] - 1s 55us/step
Model #4 on test set: [0.7082646665381007, 0.5361697325761909]
17722/17722 [==============================] - 1s 56us/step
Model #5 on test set: [0.7083782402189143, 0.5382011059800708]

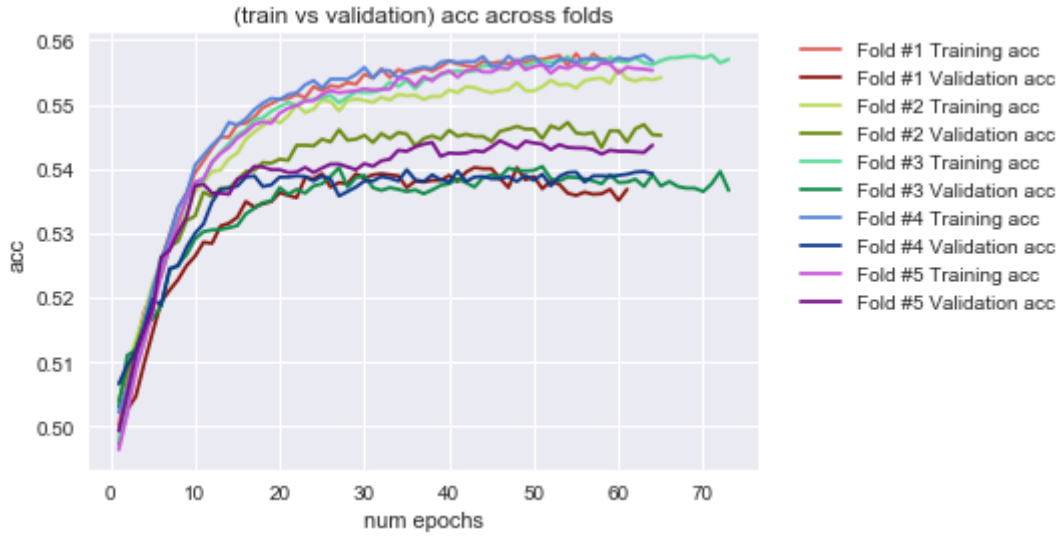
```
Loading matches from game version 8.8.1
Flattened dataframes of (72323, 281)
Shuffled and reformatted 72323 * 280(x)+1(y) dataframe suitable for tensors
Crossvalidating matches from game version 8.8.1 over various models

Starting 5-fold crossvalidation for the following architecture:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 1)                 281
=================================================================
Total params: 281
Trainable params: 281
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 0s 35us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 0s 37us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 0s 35us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 0s 36us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 0s 34us/step
```

loss average 0.69 (+-0.001 stdev)

acc average 0.54 (+-0.004 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.55
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





14467/14467 [==============================] - 1s 35us/step
Model #1 on test set: [0.6885786412036962, 0.5424068569869221]
14467/14467 [==============================] - 0s 33us/step

Model #2 on test set: [0.6887398942920822, 0.5391580839171772]
14467/14467 [==============================] - 0s 34us/step
Model #3 on test set: [0.6877931853389496, 0.5444805419250572]
14467/14467 [==============================] - 0s 33us/step
Model #4 on test set: [0.6887703362592, 0.5421303656638975]
14467/14467 [==============================] - 0s 33us/step

Model #5 on test set: [0.6878360498748324, 0.542752471143278]
Starting 5-fold crossvalidation for the following architecture:

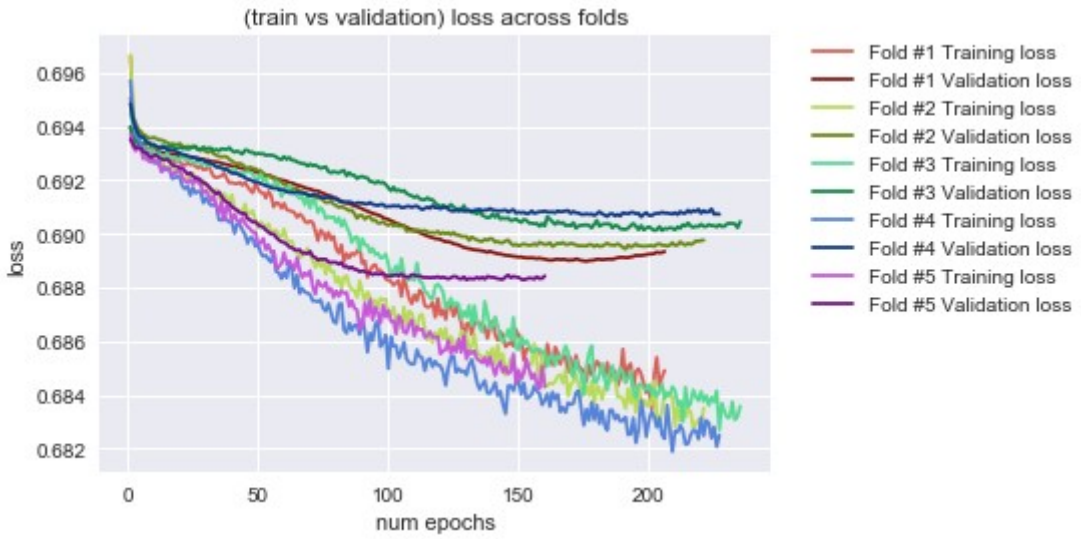
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 3)                 843
_________________________________________________________________
dropout_1 (Dropout)          (None, 3)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 4
=================================================================
Total params: 847
Trainable params: 847
Non-trainable params: 0
_________________________________________________________________
```

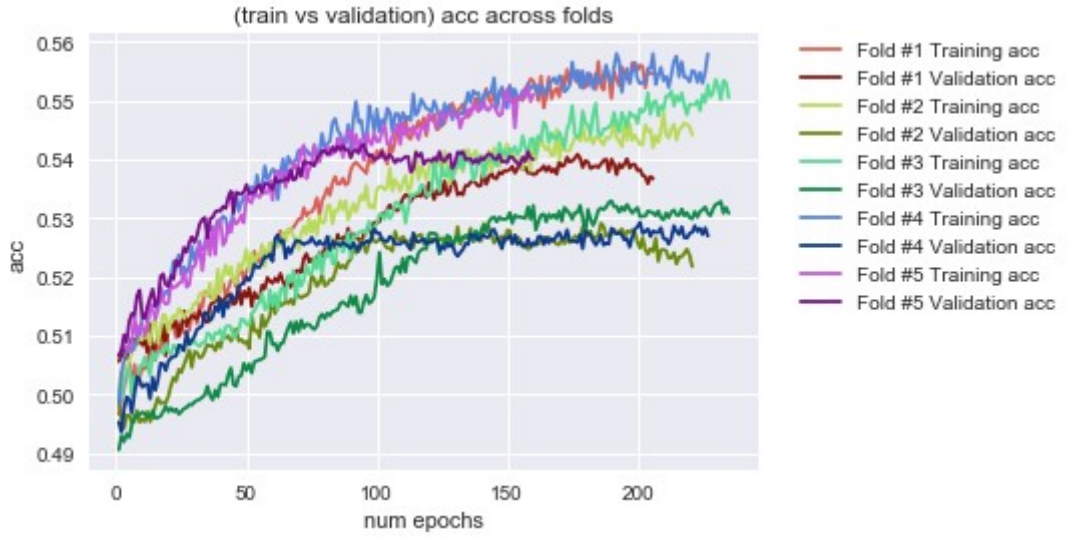
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 0s 38us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 0s 37us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 1s 58us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 1s 62us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 1s 59us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.53 (+-0.007 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.52
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.54





14467/14467 [==============================] - 1s 57us/step
Model #1 on test set: [0.6903527206035395, 0.5336282574175437]
14467/14467 [==============================] - 1s 57us/step

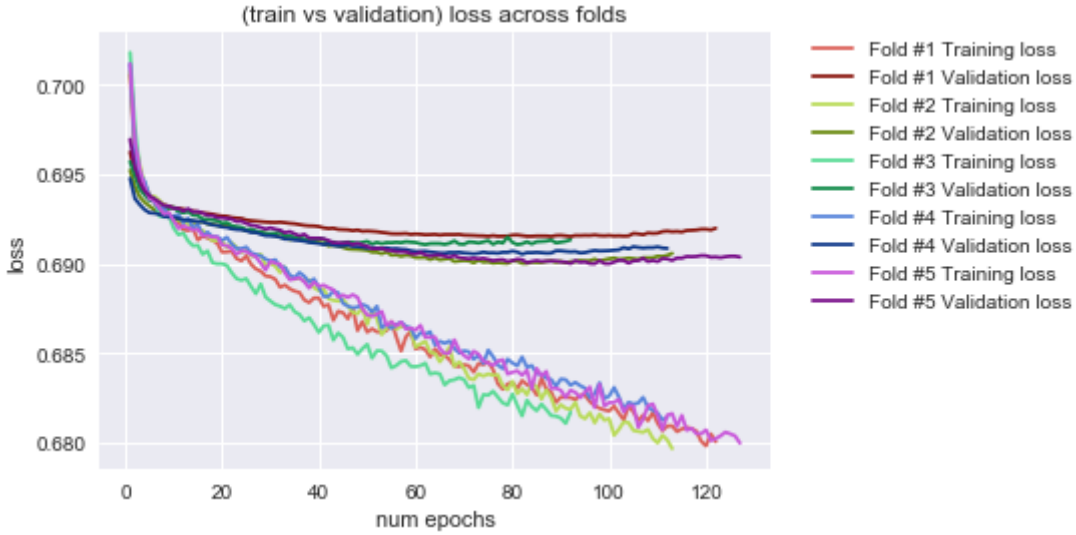
Model #2 on test set: [0.6897854512840511, 0.5271307112759938]
14467/14467 [==============================] - 1s 64us/step
Model #3 on test set: [0.6885627255481288, 0.5361858021725169]
14467/14467 [==============================] - 1s 62us/step
Model #4 on test set: [0.6896841429746117, 0.5360475565099746]
14467/14467 [==============================] - 1s 63us/step
Model #5 on test set: [0.6888221398025819, 0.5392963295817795]

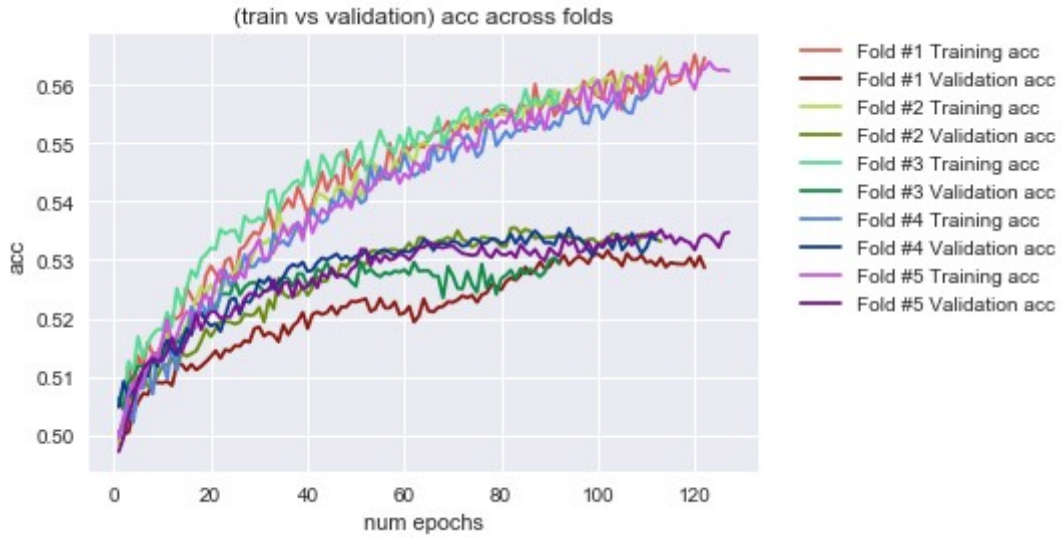
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 9)                 2529
_________________________________________________________________
dropout_6 (Dropout)          (None, 9)                 0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 10
=================================================================
Total params: 2,539
Trainable params: 2,539
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 0s 37us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 0s 40us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 0s 38us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 1s 56us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 0s 42us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.53 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.53





14467/14467 [==============================] - 1s 42us/step
Model #1 on test set: [0.6906572496659475, 0.5334208889216702]
14467/14467 [==============================] - 1s 39us/step
Model #2 on test set: [0.6896049465010489, 0.5338356259072372]

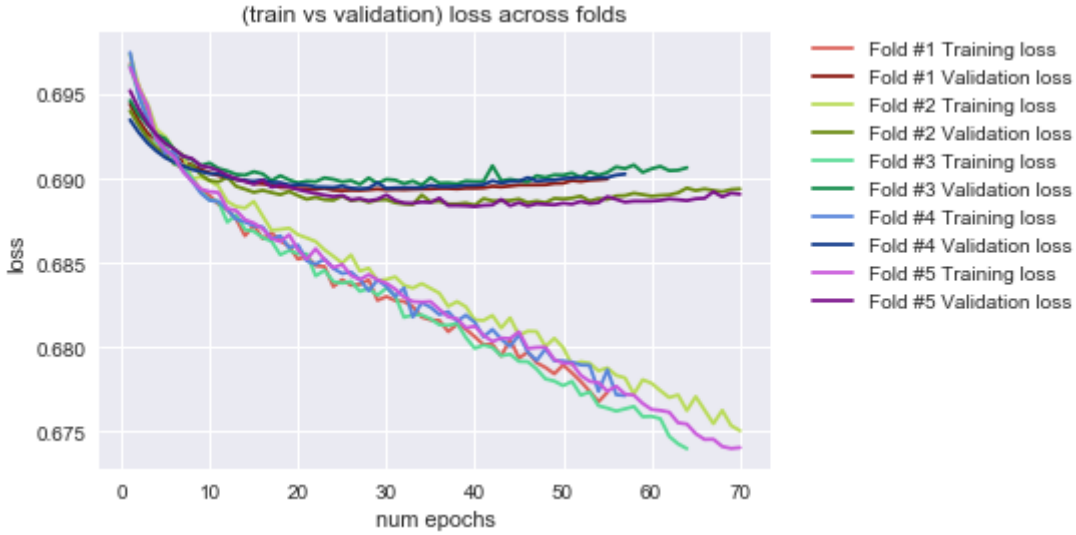
14467/14467 [==============================] - 1s 41us/step
Model #3 on test set: [0.6885855968725115, 0.5389507154254237]
14467/14467 [==============================] - 1s 40us/step
Model #4 on test set: [0.690291008709823, 0.5312089583230528]
14467/14467 [==============================] - 1s 43us/step
Model #5 on test set: [0.6895077618631701, 0.5361858021745769]

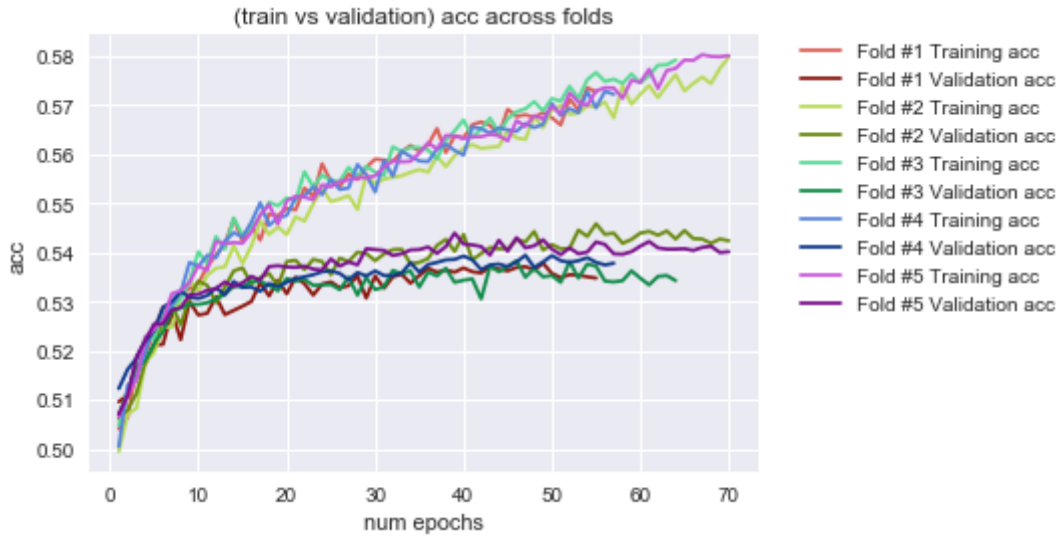
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 280)               78680
_________________________________________________________________
dropout_6 (Dropout)          (None, 280)               0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 281
=================================================================
Total params: 78,961
Trainable params: 78,961
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 1s 50us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 0s 41us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 0s 41us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 0s 39us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 1s 65us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





14467/14467 [==============================] - 1s 45us/step
Model #1 on test set: [0.6893514739813633, 0.5413008916886435]
14467/14467 [==============================] - 1s 55us/step
Model #2 on test set: [0.6897107359821962, 0.5354254510305941]

14467/14467 [==============================] - 1s 59us/step
Model #3 on test set: [0.6884629935749236, 0.5416465058449993]
14467/14467 [==============================] - 1s 59us/step
Model #4 on test set: [0.6897238360671156, 0.5395728209027442]
14467/14467 [==============================] - 1s 62us/step
Model #5 on test set: [0.6886898531954628, 0.53991843506116]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 3)	843
dropout_6 (Dropout)	(None, 3)	0
dense_12 (Dense)	(None, 3)	12
dropout_7 (Dropout)	(None, 3)	0
dense_13 (Dense)	(None, 1)	4

Total params: 859
Trainable params: 859
Non-trainable params: 0

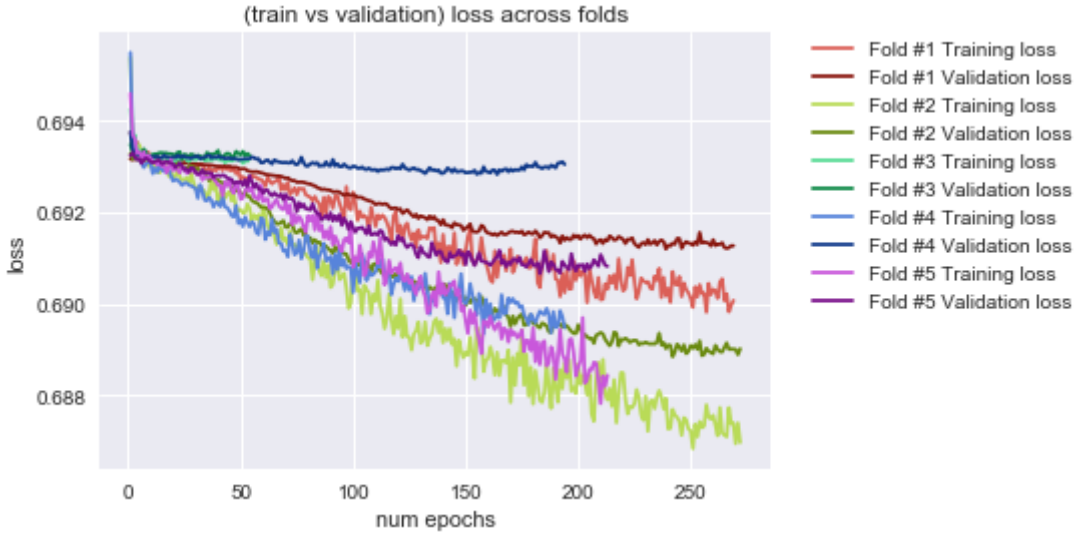
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 1s 56us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 1s 60us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 1s 58us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 1s 54us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples

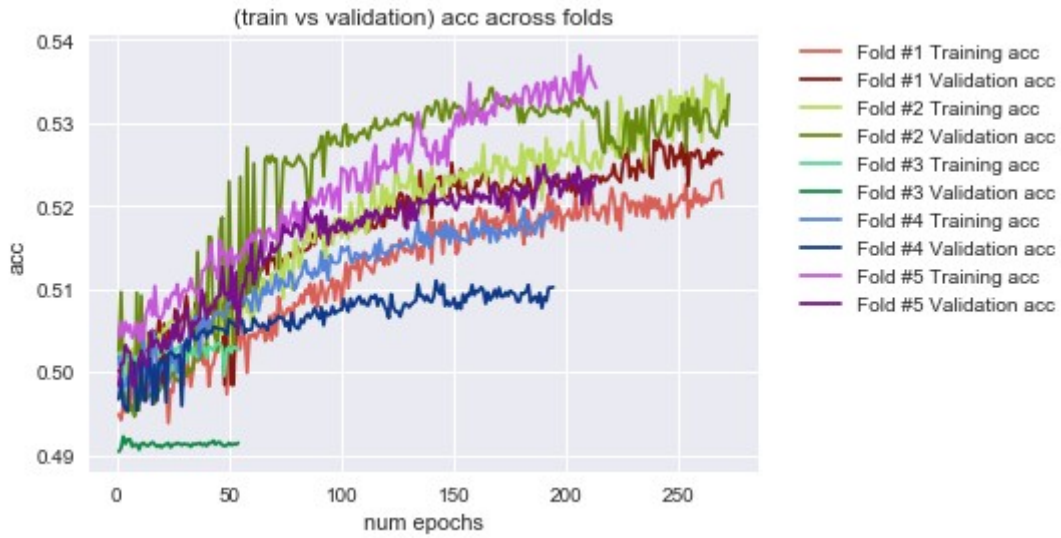
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 1s 63us/step

loss average 0.69 (+-0.002 stdev)

acc average 0.52 (+-0.015 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.49
Scores #4 loss: 0.69 acc: 0.51
Scores #5 loss: 0.69 acc: 0.52





```
14467/14467 [==============================] - 1s 71us/step
Model #1 on test set: [0.691349995431256, 0.5234672012186219]
14467/14467 [==============================] - 1s 60us/step
Model #2 on test set: [0.6900832883726795, 0.528858782055713]
14467/14467 [==============================] - 1s 61us/step
Model #3 on test set: [0.6931533029309006, 0.5015552637036013]
14467/14467 [==============================] - 1s 64us/step
Model #4 on test set: [0.6919113033510172, 0.5141356189949541]
14467/14467 [==============================] - 1s 68us/step
Model #5 on test set: [0.6908399653622628, 0.5265086057924933]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 32)	8992
dropout_11 (Dropout)	(None, 32)	0
dense_17 (Dense)	(None, 16)	528
dropout_12 (Dropout)	(None, 16)	0
dense_18 (Dense)	(None, 1)	17

Total params: 9,537
Trainable params: 9,537
Non-trainable params: 0

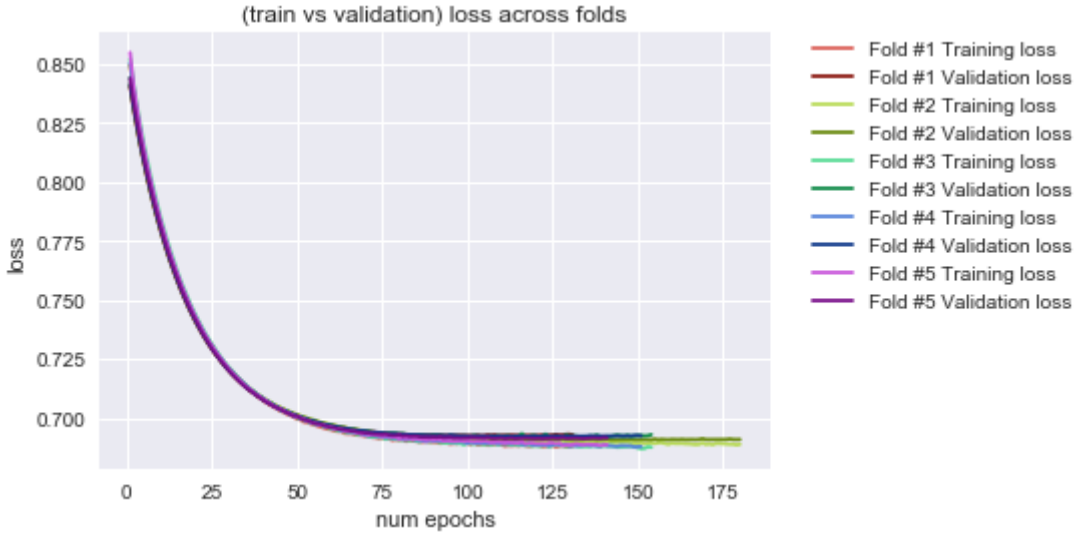
Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 1s 63us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 1s 49us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 0s 41us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 0s 40us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples

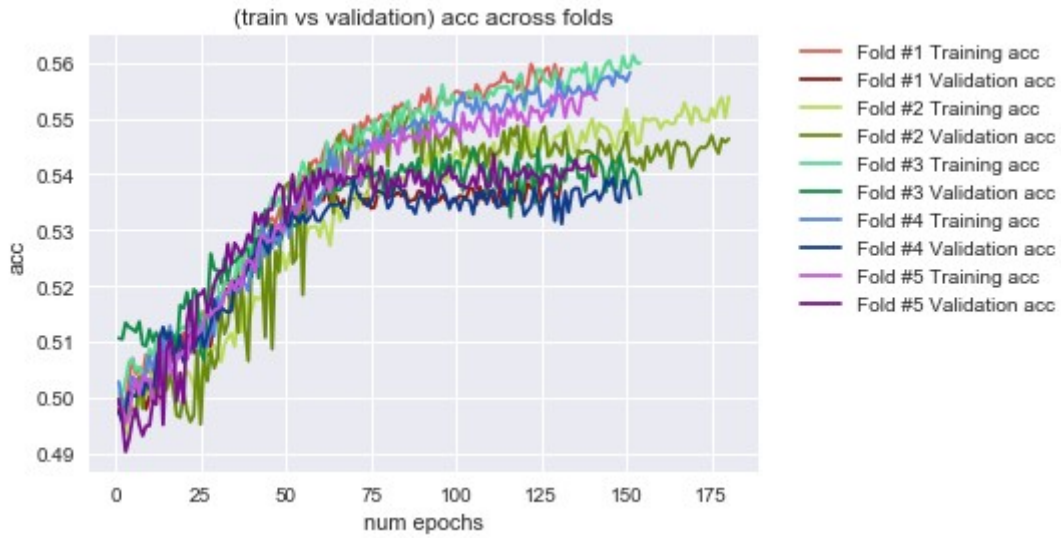
Evaluating fold 5 of 5 . . .
11568/11568 [==============================] - 0s 43us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.54 (+-0.004 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.55
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





```
14467/14467 [==============================] - 1s 42us/step
Model #1 on test set: [0.6920470754929021, 0.5389507154233637]
14467/14467 [==============================] - 1s 44us/step
Model #2 on test set: [0.691345767155954, 0.5414391373491257]
14467/14467 [==============================] - 1s 42us/step
Model #3 on test set: [0.6911472505595958, 0.5417847515034215]
14467/14467 [==============================] - 1s 41us/step
Model #4 on test set: [0.6913564680169297, 0.5407479090364141]
14467/14467 [==============================] - 1s 40us/step
Model #5 on test set: [0.6907805979412543, 0.5437893136123455]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 280)	78680
dropout_11 (Dropout)	(None, 280)	0
dense_17 (Dense)	(None, 120)	33720
dropout_12 (Dropout)	(None, 120)	0
dense_18 (Dense)	(None, 1)	121

Total params: 112,521
Trainable params: 112,521
Non-trainable params: 0

Training fold 1 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 1 of 5 . . .
11572/11572 [==============================] - 0s 43us/step
Training fold 2 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 2 of 5 . . .
11572/11572 [==============================] - 0s 40us/step
Training fold 3 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 3 of 5 . . .
11572/11572 [==============================] - 0s 40us/step
Training fold 4 of 5 . . .
Training using 46284 samples and validating using 11572 samples
Evaluating fold 4 of 5 . . .
11572/11572 [==============================] - 0s 38us/step
Training fold 5 of 5 . . .
Training using 46288 samples and validating using 11568 samples

Evaluating fold 5 of 5 . . .

11568/11568 [==============================] - 0s 38us/step

loss average 0.72 (+-0.003 stdev)

acc average 0.54 (+-0.005 stdev)

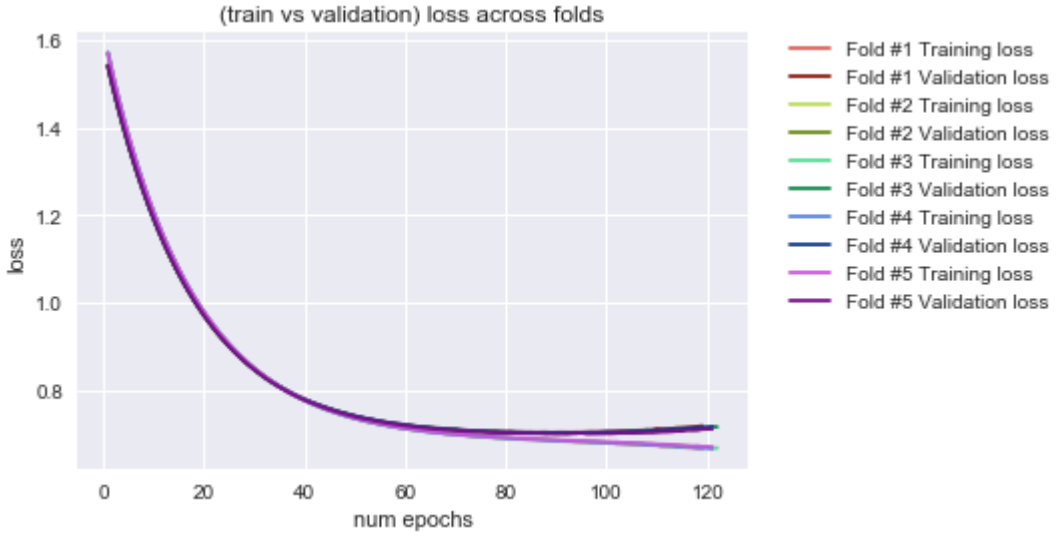
Scores #1 loss: 0.72 acc: 0.53

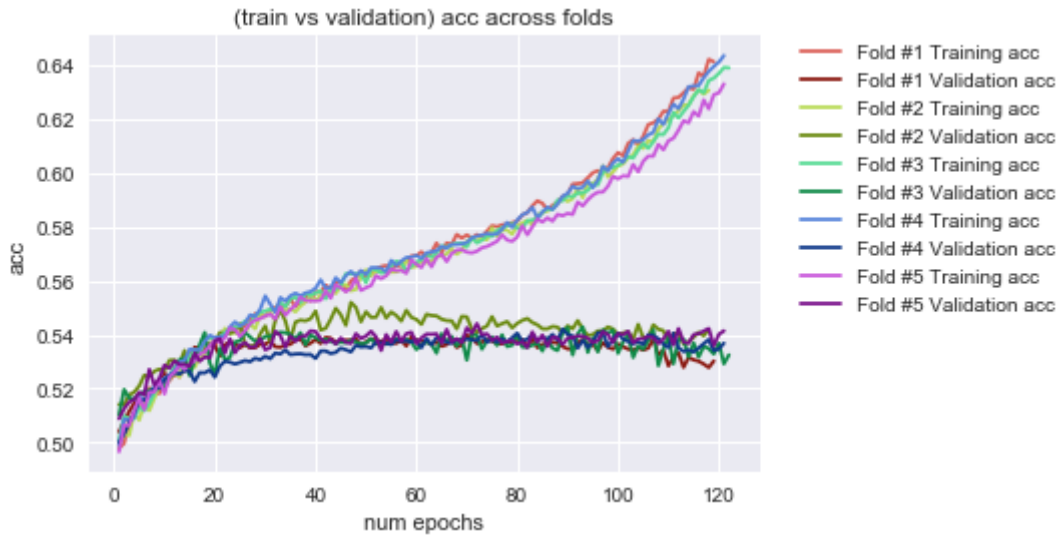
Scores #2 loss: 0.71 acc: 0.54

Scores #3 loss: 0.72 acc: 0.53

Scores #4 loss: 0.72 acc: 0.54

Scores #5 loss: 0.71 acc: 0.54





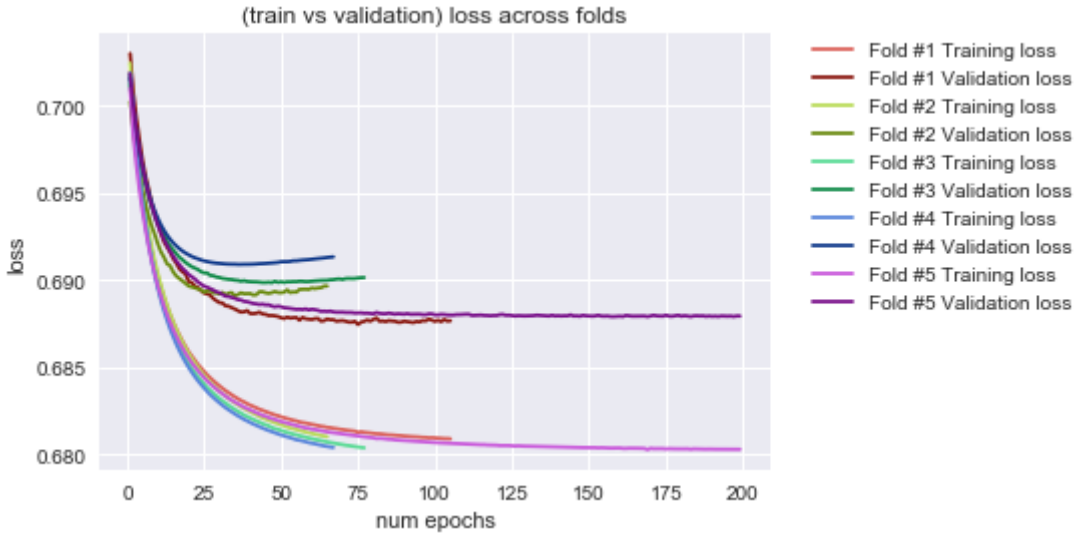
```
14467/14467 [==============================] - 1s 40us/step
Model #1 on test set: [0.7175834371742517, 0.5330061519361031]
14467/14467 [==============================] - 1s 37us/step
Model #2 on test set: [0.7121819310266739, 0.5341812400677131]
14467/14467 [==============================] - 1s 38us/step
Model #3 on test set: [0.7152297940318154, 0.5362549250017281]
14467/14467 [==============================] - 1s 37us/step
Model #4 on test set: [0.7168505937901348, 0.5345268542240689]
14467/14467 [==============================] - 1s 37us/step
Model #5 on test set: [0.7116046045697814, 0.5386051012690678]
```

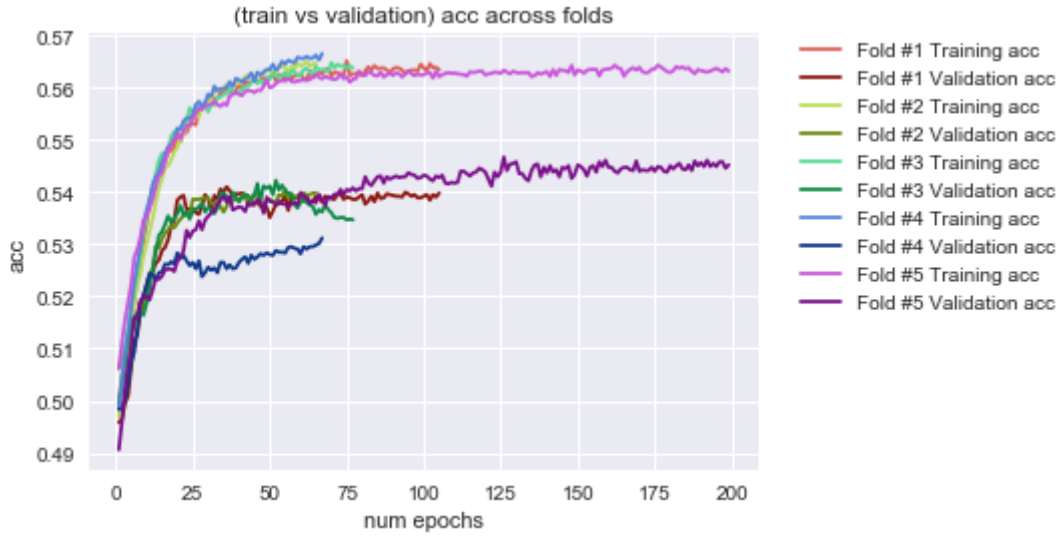
```
Loading matches from game version 8.9.1
Flattened dataframes of (41393, 281)
Shuffled and reformatted 41393 * 280(x)+1(y) dataframe suitable for tensors
Crossvalidating matches from game version 8.9.1 over various models

Starting 5-fold crossvalidation for the following architecture:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 1)                 281
=================================================================
Total params: 281
Trainable params: 281
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 33us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 32us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 32us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 32us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 32us/step
```

loss average 0.69 (+-0.001 stdev)

acc average 0.54 (+-0.005 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.55





8281/8281 [==============================] - 0s 34us/step
Model #1 on test set: [0.689384152046826, 0.538461538504725]
8281/8281 [==============================] - 0s 33us/step

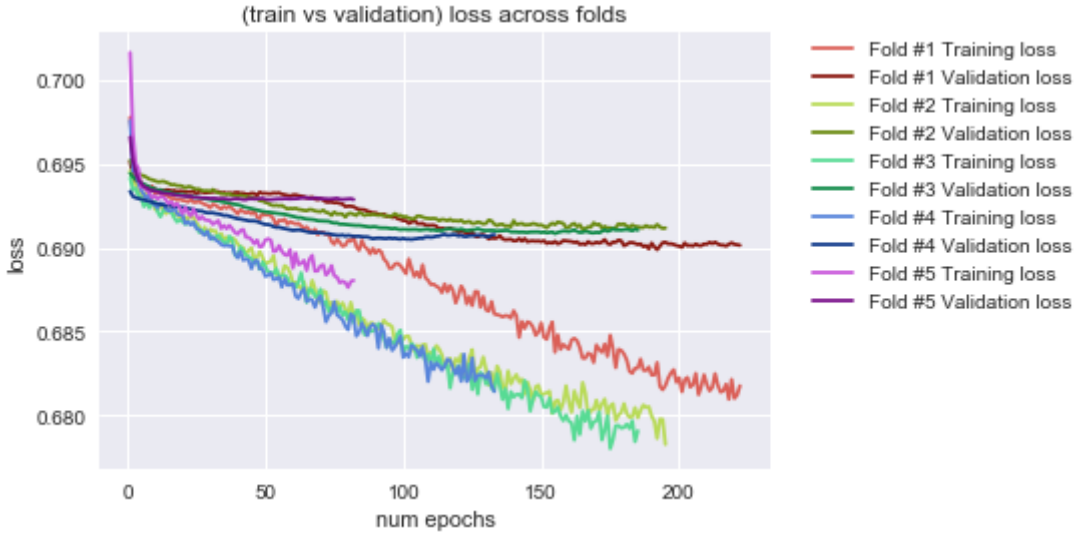
Model #2 on test set: [0.6894775619398582, 0.539789880391639]
8281/8281 [==============================] - 0s 32us/step
Model #3 on test set: [0.6889567860168128, 0.5424465644173886]
8281/8281 [==============================] - 0s 33us/step
Model #4 on test set: [0.6896474403166681, 0.5419635309421301]
8281/8281 [==============================] - 0s 34us/step
Model #5 on test set: [0.6892955627379218, 0.5412389807922225]

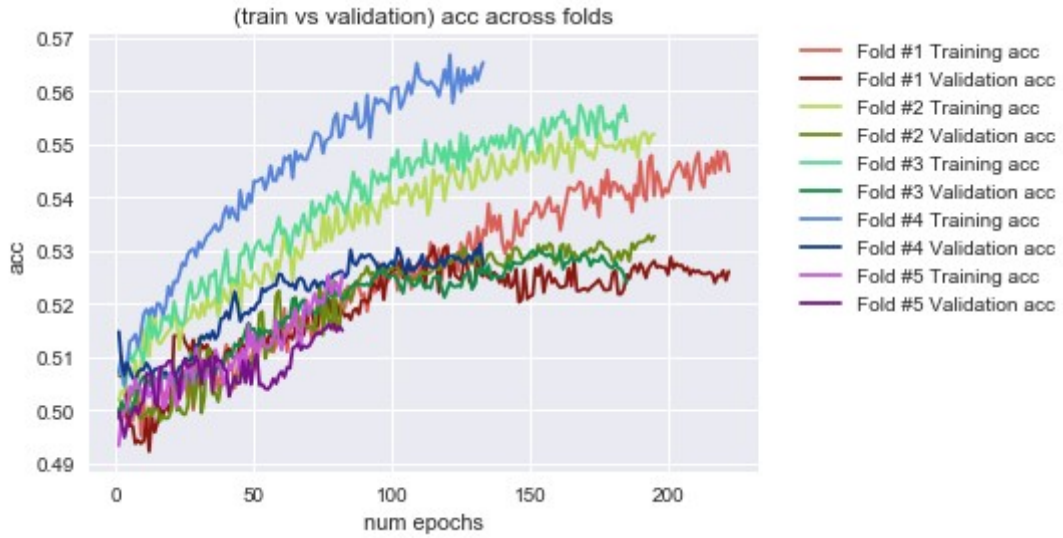
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 3)                 843
_________________________________________________________________
dropout_1 (Dropout)          (None, 3)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 4
=================================================================
Total params: 847
Trainable params: 847
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 35us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 35us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 36us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.52 (+-0.006 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.52
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.51





8281/8281 [==============================] - 0s 38us/step
Model #1 on test set: [0.690626767231292, 0.5325443787162193]
8281/8281 [==============================] - 0s 35us/step
Model #2 on test set: [0.6896139945676975, 0.5314575534283776]

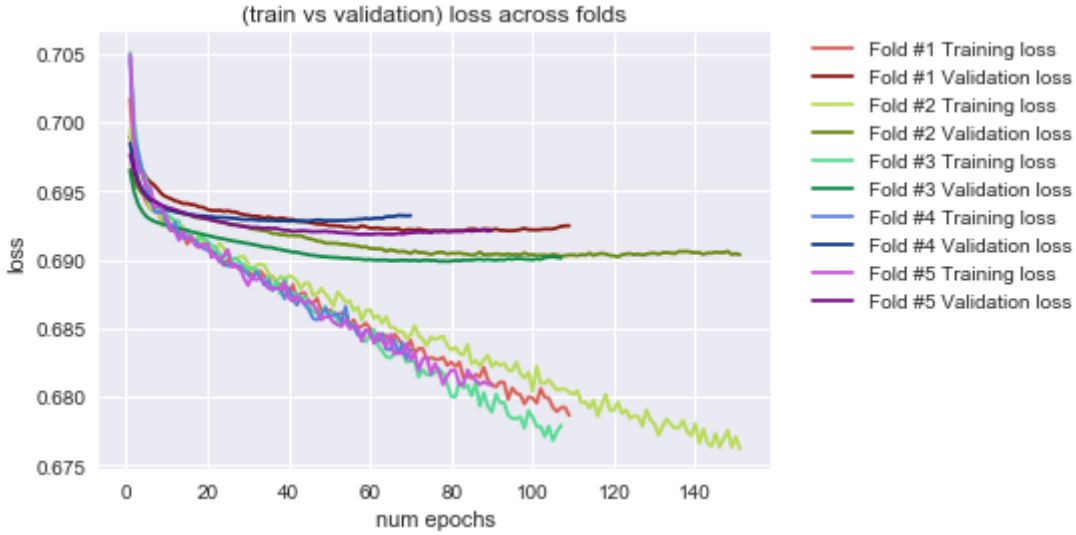
8281/8281 [==============================] - 0s 36us/step
Model #3 on test set: [0.690797702282354, 0.5257819103649051]
8281/8281 [==============================] - 0s 34us/step
Model #4 on test set: [0.6904503486680749, 0.5333896872034511]
8281/8281 [==============================] - 0s 35us/step
Model #5 on test set: [0.6927927063291969, 0.5204685424141745]

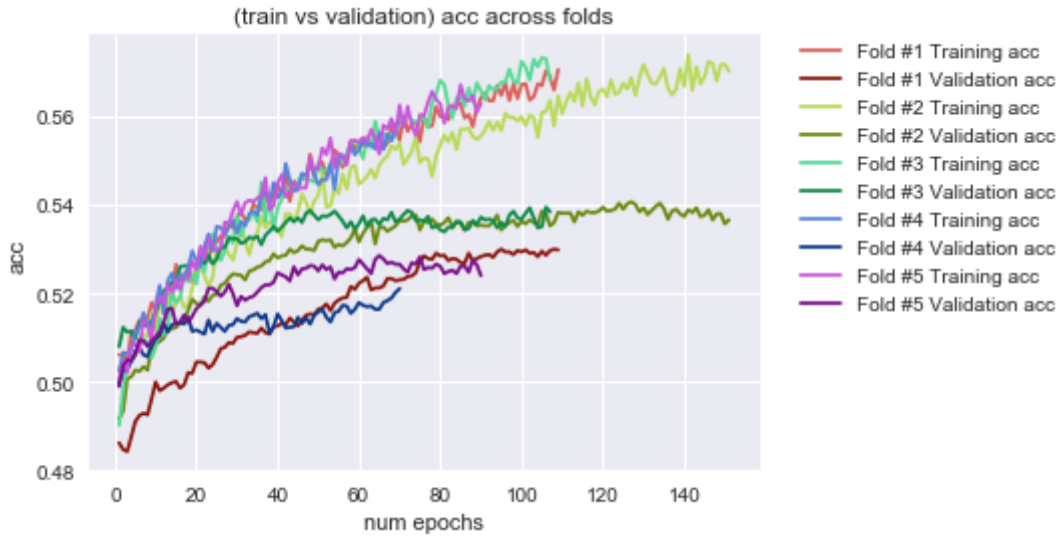
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 9)                 2529
_________________________________________________________________
dropout_6 (Dropout)          (None, 9)                 0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 10
=================================================================
Total params: 2,539
Trainable params: 2,539
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 33us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 36us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 34us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.53 (+-0.007 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.52
Scores #5 loss: 0.69 acc: 0.52





8281/8281 [==============================] - 0s 37us/step
Model #1 on test set: [0.6913077510608133, 0.5332689288661268]
8281/8281 [==============================] - 0s 36us/step
Model #2 on test set: [0.6913448947162073, 0.5337519623413853]

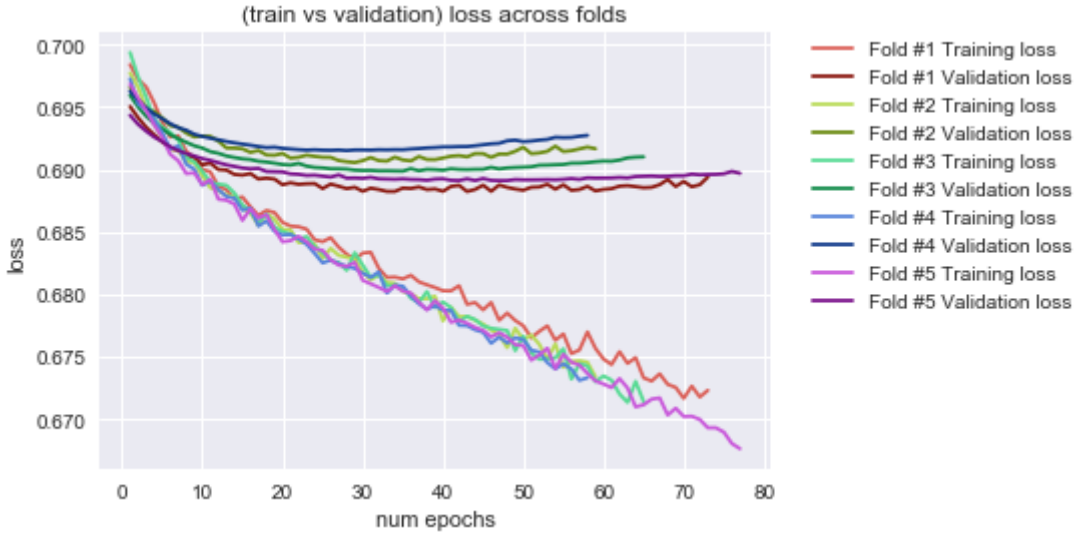
```
8281/8281 [==============================] - 0s 35us/step
Model #3 on test set: [0.6904954253338829, 0.5354425794706009]
8281/8281 [==============================] - 0s 36us/step
Model #4 on test set: [0.6920302734543715, 0.5243328100650901]
8281/8281 [==============================] - 0s 34us/step
Model #5 on test set: [0.6903424701559963, 0.5292839028526944]
```

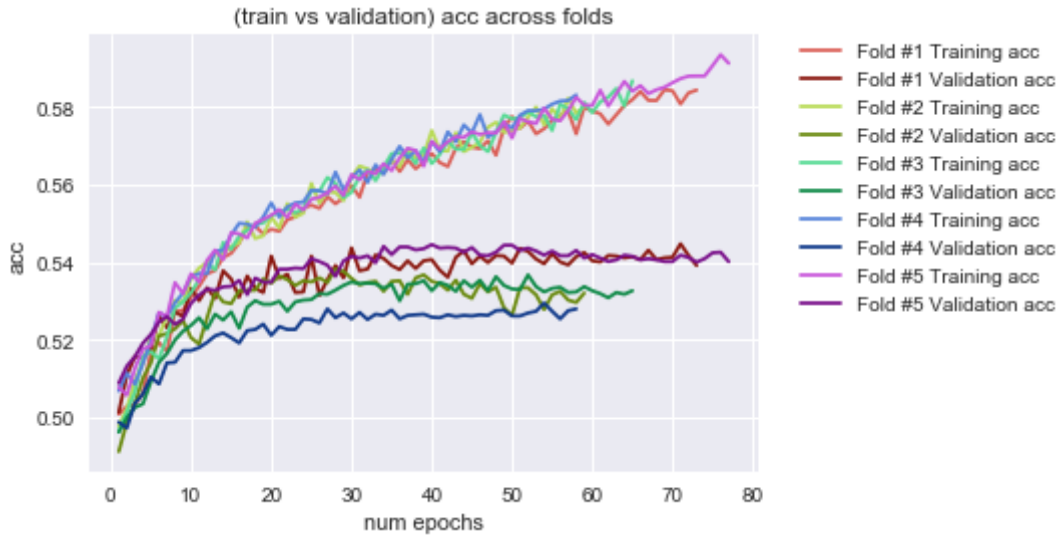
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 280)               78680
_________________________________________________________________
dropout_6 (Dropout)          (None, 280)               0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 281
=================================================================
Total params: 78,961
Trainable params: 78,961
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 33us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 34us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 35us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 34us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.53 (+-0.005 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.53
Scores #5 loss: 0.69 acc: 0.54





8281/8281 [==============================] - 0s 38us/step
Model #1 on test set: [0.690162137688797, 0.5385822967916651]
8281/8281 [==============================] - 0s 35us/step
Model #2 on test set: [0.6896986938547849, 0.5350803042534915]

```
8281/8281 [==============================] - 0s 34us/step
Model #3 on test set: [0.690068850603934, 0.5362878879038496]
8281/8281 [==============================] - 0s 35us/step
Model #4 on test set: [0.6893431977896914, 0.5407559473565517]
8281/8281 [==============================] - 0s 34us/step
Model #5 on test set: [0.6902210529537798, 0.5416012558545803]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 3)	843
dropout_6 (Dropout)	(None, 3)	0
dense_12 (Dense)	(None, 3)	12
dropout_7 (Dropout)	(None, 3)	0
dense_13 (Dense)	(None, 1)	4

Total params: 859
Trainable params: 859
Non-trainable params: 0

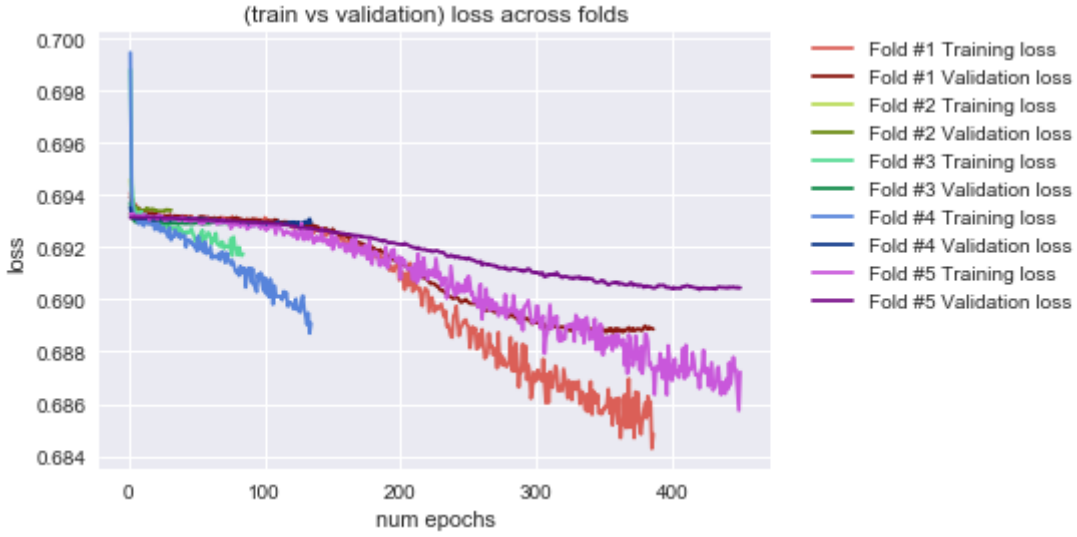
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 35us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 37us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 35us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 36us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples

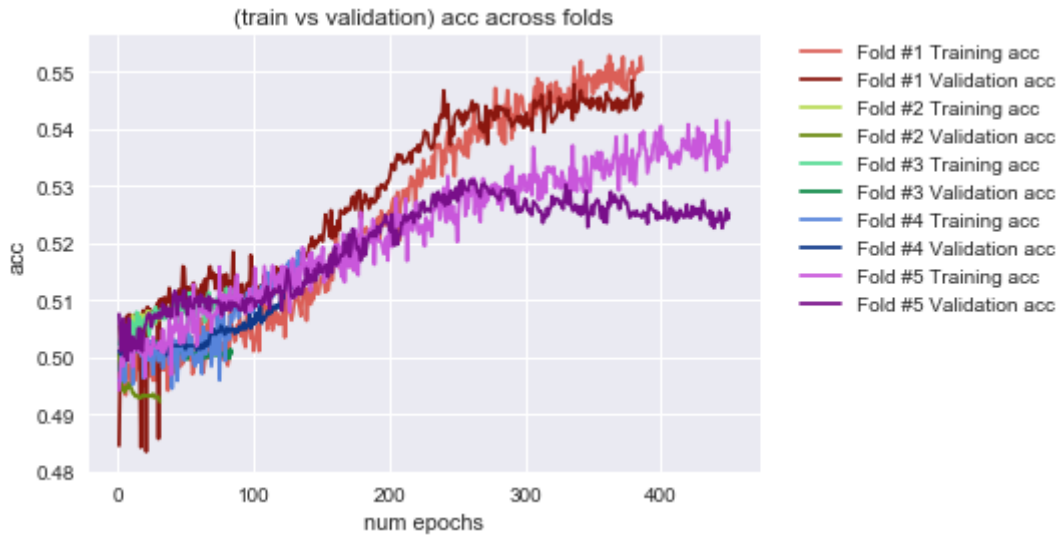
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 37us/step

loss average 0.69 (+-0.002 stdev)

acc average 0.52 (+-0.019 stdev)

Scores #1 loss: 0.69 acc: 0.55
Scores #2 loss: 0.69 acc: 0.49
Scores #3 loss: 0.69 acc: 0.50
Scores #4 loss: 0.69 acc: 0.51
Scores #5 loss: 0.69 acc: 0.53





8281/8281 [==============================] - 0s 37us/step
Model #1 on test set: [0.6910135831727511, 0.5277140441651708]
8281/8281 [==============================] - 0s 37us/step
Model #2 on test set: [0.6932251190177601, 0.5001811375509727]
8281/8281 [==============================] - 0s 35us/step
Model #3 on test set: [0.6925525196277207, 0.5056152639289997]
8281/8281 [==============================] - 0s 36us/step
Model #4 on test set: [0.6930696065119825, 0.5118946986754956]
8281/8281 [==============================] - 0s 36us/step
Model #5 on test set: [0.6912115276932299, 0.5262649438401636]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 32)	8992
dropout_11 (Dropout)	(None, 32)	0
dense_17 (Dense)	(None, 16)	528
dropout_12 (Dropout)	(None, 16)	0
dense_18 (Dense)	(None, 1)	17

Total params: 9,537
Trainable params: 9,537
Non-trainable params: 0

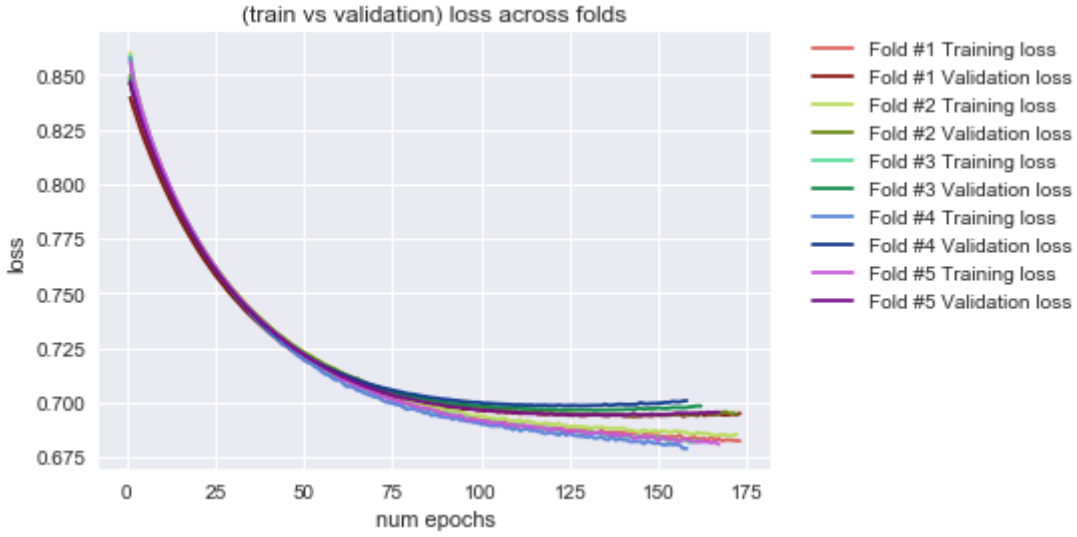
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 37us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 37us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 37us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 37us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples

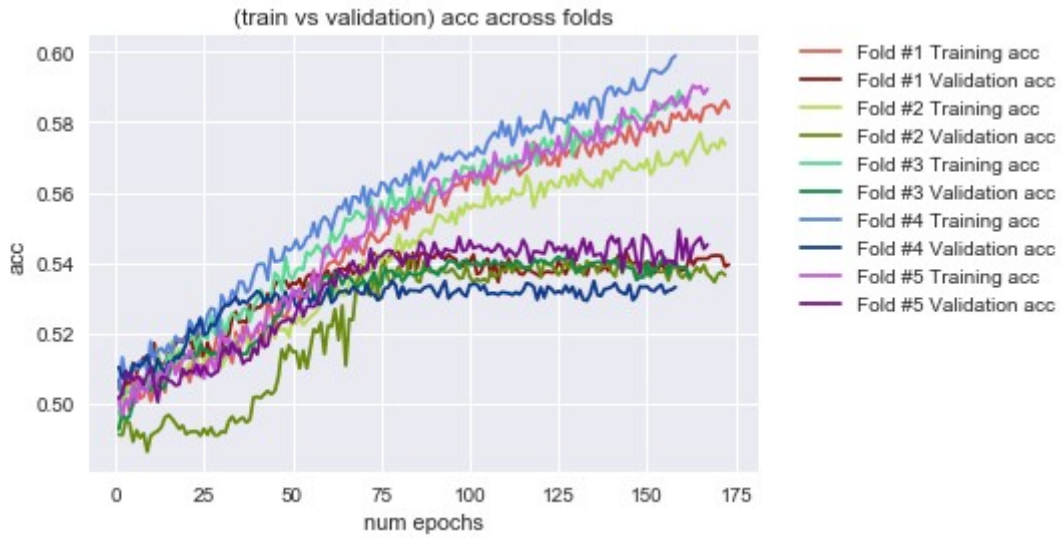
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 36us/step

loss average 0.70 (+-0.002 stdev)

acc average 0.54 (+-0.004 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.70 acc: 0.54
Scores #4 loss: 0.70 acc: 0.53
Scores #5 loss: 0.70 acc: 0.55





```
8281/8281 [==============================] - 0s 39us/step
Model #1 on test set: [0.6953280883672631, 0.5432918729550049]
8281/8281 [==============================] - 0s 37us/step
Model #2 on test set: [0.6941532050839169, 0.5379785050042742]
8281/8281 [==============================] - 0s 38us/step
Model #3 on test set: [0.6962127383600872, 0.5402729139424741]
8281/8281 [==============================] - 0s 38us/step
Model #4 on test set: [0.698353732097208, 0.5426880811424218]
8281/8281 [==============================] - 0s 36us/step
Model #5 on test set: [0.6974349286357298, 0.5406351890048318]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 280)	78680
dropout_11 (Dropout)	(None, 280)	0
dense_17 (Dense)	(None, 120)	33720
dropout_12 (Dropout)	(None, 120)	0
dense_18 (Dense)	(None, 1)	121

Total params: 112,521
Trainable params: 112,521
Non-trainable params: 0

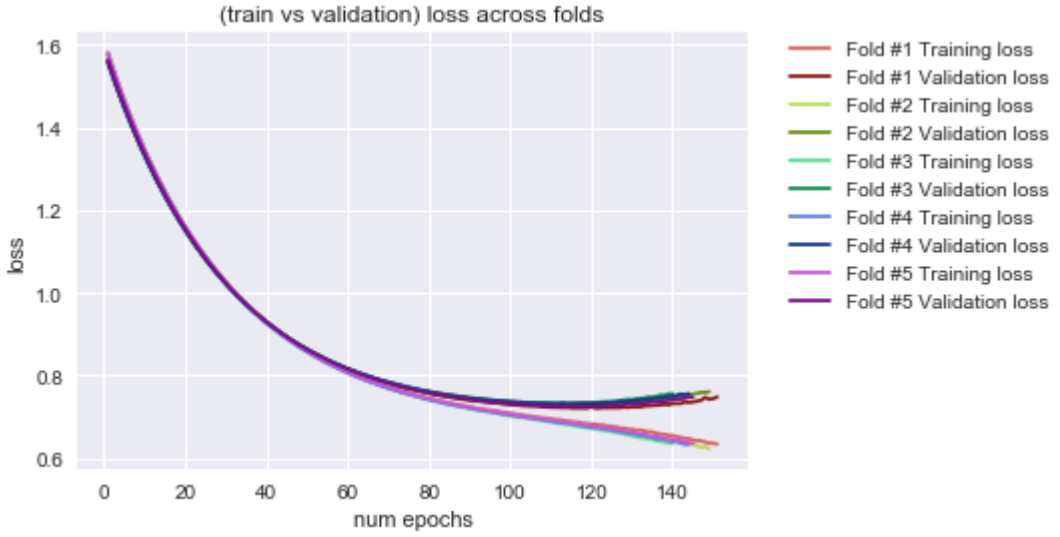
Training fold 1 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 1 of 5 . . .
6623/6623 [==============================] - 0s 44us/step
Training fold 2 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 2 of 5 . . .
6623/6623 [==============================] - 0s 45us/step
Training fold 3 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 3 of 5 . . .
6623/6623 [==============================] - 0s 44us/step
Training fold 4 of 5 . . .
Training using 26489 samples and validating using 6623 samples
Evaluating fold 4 of 5 . . .
6623/6623 [==============================] - 0s 40us/step
Training fold 5 of 5 . . .
Training using 26492 samples and validating using 6620 samples

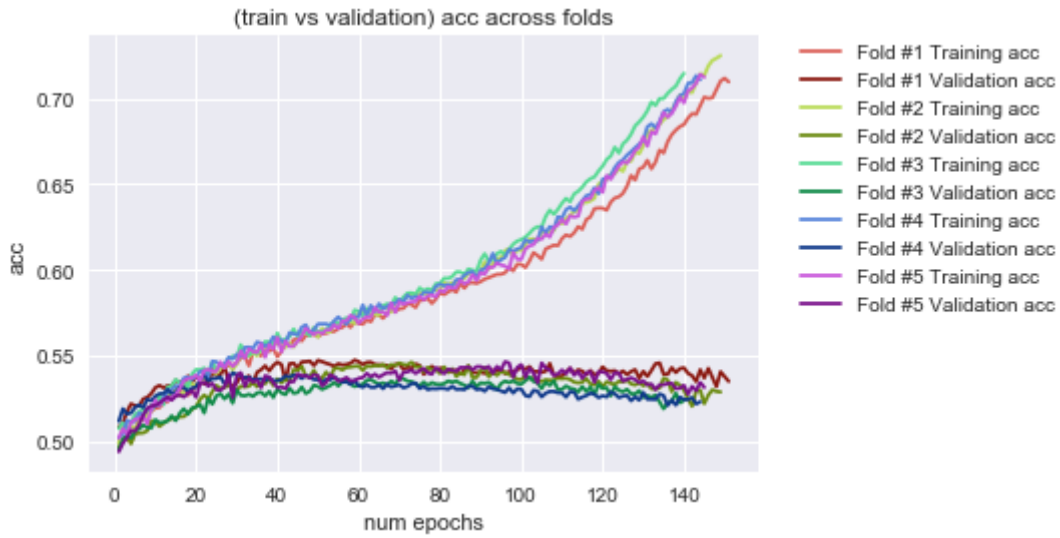
Evaluating fold 5 of 5 . . .
6620/6620 [==============================] - 0s 39us/step

loss average 0.75 (+-0.005 stdev)

acc average 0.53 (+-0.005 stdev)

Scores #1 loss: 0.75 acc: 0.54
Scores #2 loss: 0.76 acc: 0.53
Scores #3 loss: 0.76 acc: 0.52
Scores #4 loss: 0.76 acc: 0.52
Scores #5 loss: 0.75 acc: 0.53





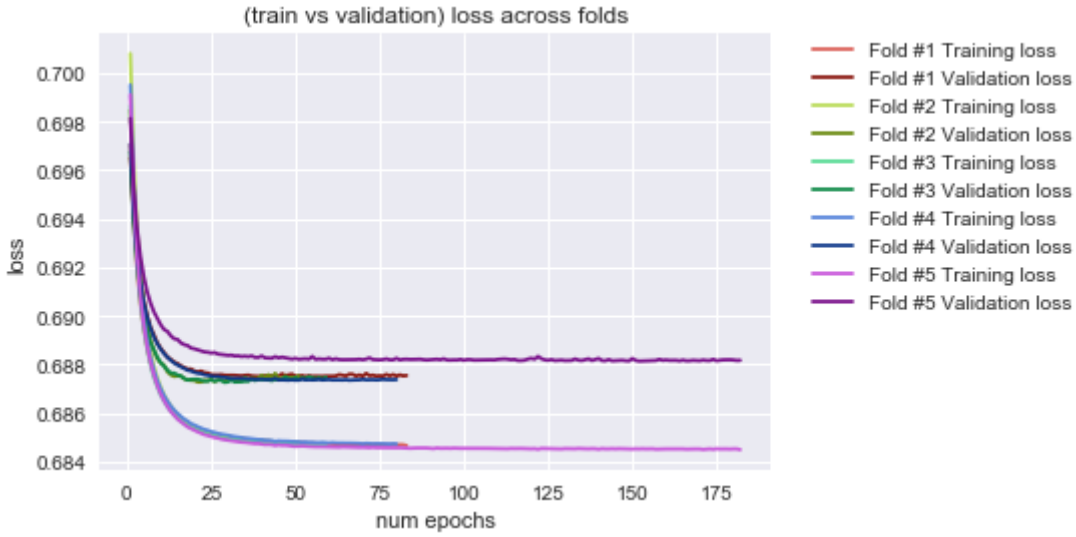
```
8281/8281 [==============================] - 0s 40us/step
Model #1 on test set: [0.7496217399990338, 0.5356840960660745]
8281/8281 [==============================] - 0s 39us/step
Model #2 on test set: [0.7576325012713062, 0.5312160367285366]
8281/8281 [==============================] - 0s 39us/step
Model #3 on test set: [0.7526112697202505, 0.5316990701534109]
8281/8281 [==============================] - 0s 38us/step
Model #4 on test set: [0.7527128551234055, 0.531819828530323]
8281/8281 [==============================] - 0s 38us/step
Model #5 on test set: [0.7483263569614649, 0.5384615384039564]
```

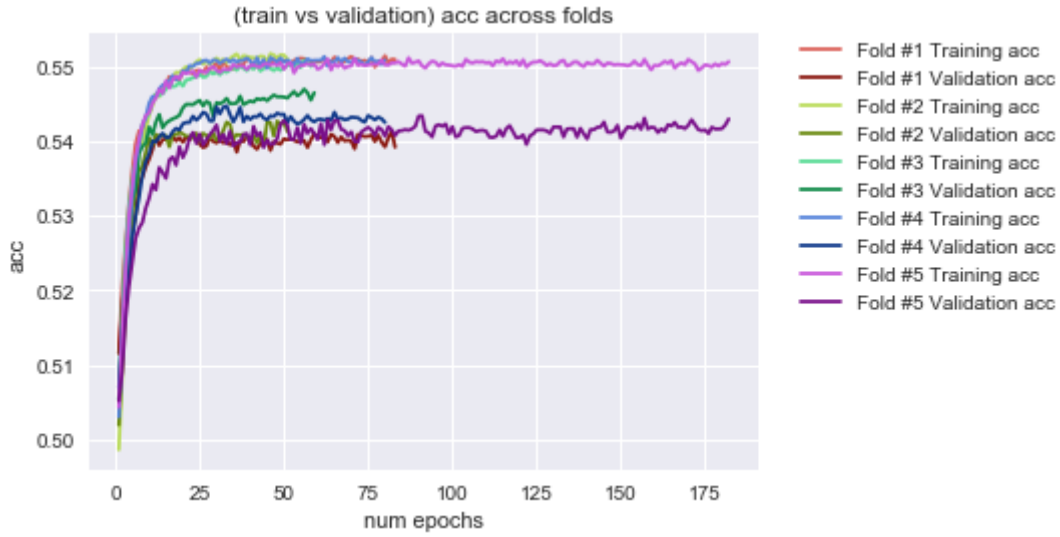
```
Loading matches from game version 8.10.1
Flattened dataframes of (141673, 281)
Shuffled and reformatted 141673 * 280(x)+1(y) dataframe suitable for tensors
Crossvalidating matches from game version 8.10.1 over various models

Starting 5-fold crossvalidation for the following architecture:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 1)                 281
=================================================================
Total params: 281
Trainable params: 281
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 34us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 34us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 34us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 34us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 34us/step
```

loss average 0.69 (+-0.000 stdev)

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.55
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





28337/28337 [==============================] - 1s 35us/step
Model #1 on test set: [0.6876443912959634, 0.5403536013133794]
28337/28337 [==============================] - 1s 34us/step

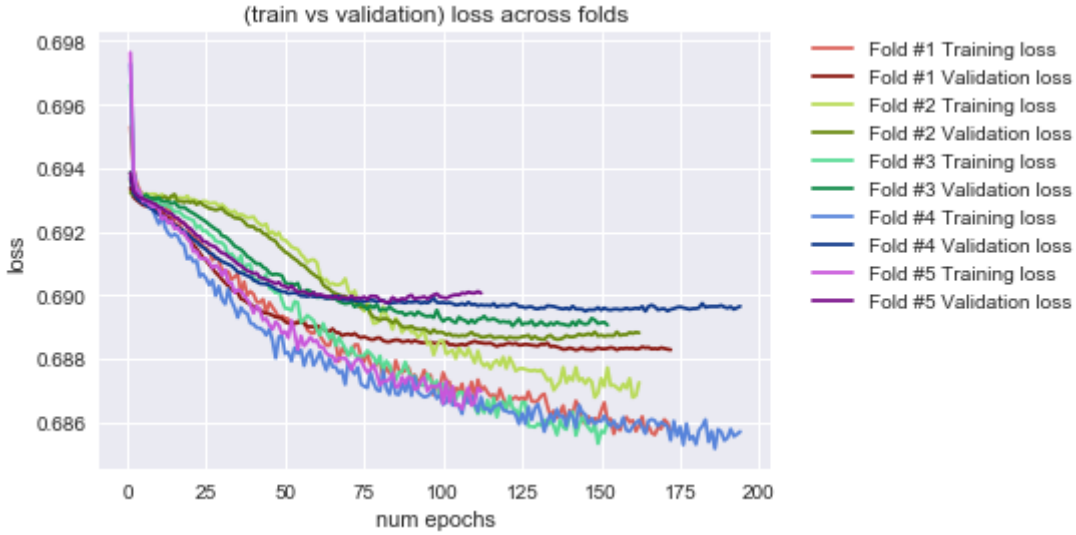
Model #2 on test set: [0.6877259589408555, 0.5415887355901201]
28337/28337 [==============================] - 1s 34us/step
Model #3 on test set: [0.687547294673357, 0.5383420969198304]
28337/28337 [==============================] - 1s 34us/step
Model #4 on test set: [0.6874808442509746, 0.5420122101992882]
28337/28337 [==============================] - 1s 34us/step
Model #5 on test set: [0.6877088943920909, 0.5385185446757539]

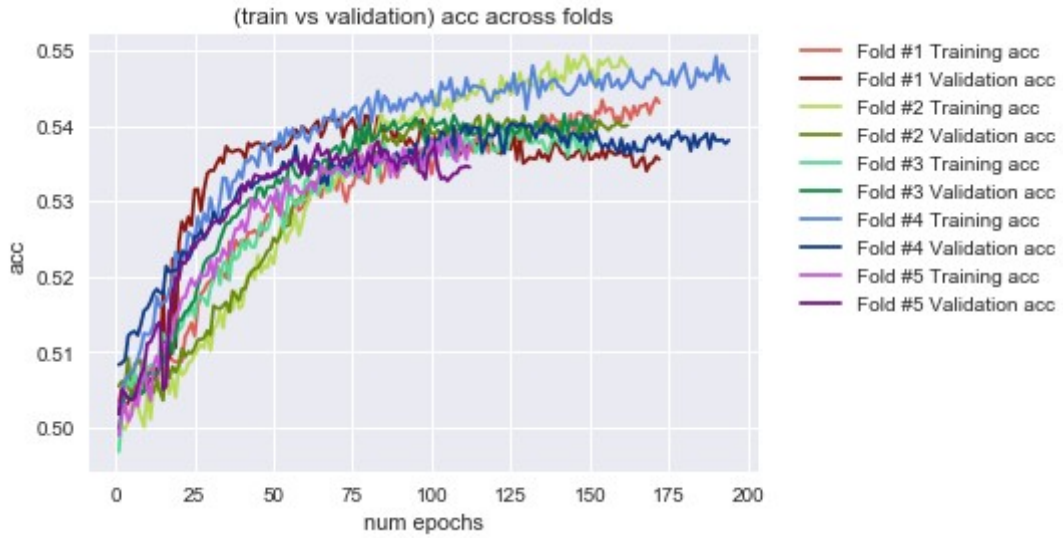
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 3)                 843
_________________________________________________________________
dropout_1 (Dropout)          (None, 3)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 4
=================================================================
Total params: 847
Trainable params: 847
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 35us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 38us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.53





28337/28337 [==============================] - 1s 37us/step
Model #1 on test set: [0.6886192673885836, 0.5378833327620015]
28337/28337 [==============================] - 1s 36us/step
Model #2 on test set: [0.6893837668166507, 0.533648586664009]

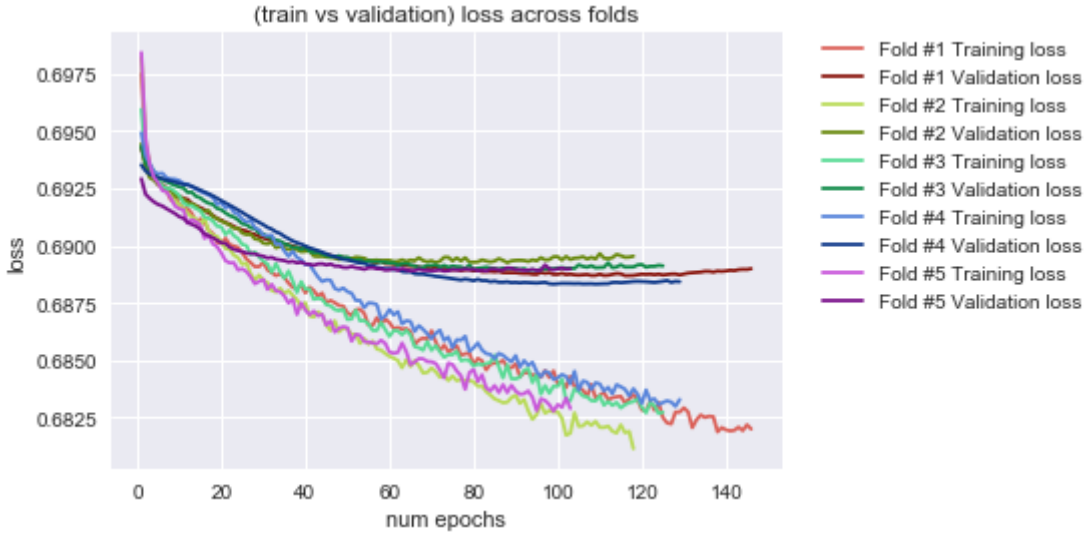
28337/28337 [==============================] - 1s 36us/step
Model #3 on test set: [0.6893628576915233, 0.535695380612529]
28337/28337 [==============================] - 1s 37us/step
Model #4 on test set: [0.689530406289091, 0.5368599357877416]
28337/28337 [==============================] - 1s 36us/step
Model #5 on test set: [0.6891141813122847, 0.5386244133259425]

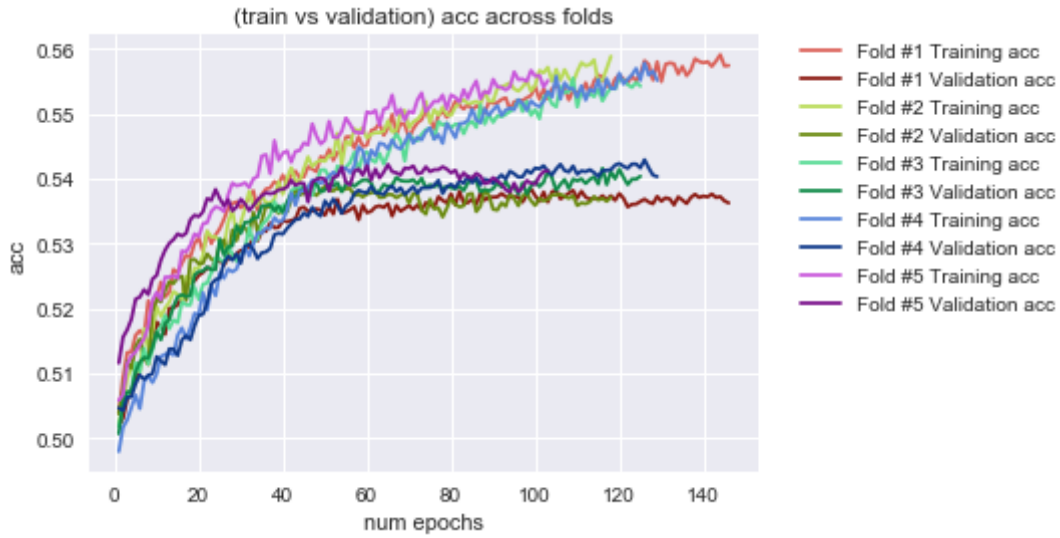
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 9)                 2529
_________________________________________________________________
dropout_6 (Dropout)          (None, 9)                 0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 10
=================================================================
Total params: 2,539
Trainable params: 2,539
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 38us/step

loss average 0.69 (+-0.000 stdev)
```

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





28337/28337 [==============================] - 1s 37us/step
Model #1 on test set: [0.6891622934192677, 0.5354836433058414]
28337/28337 [==============================] - 1s 36us/step
Model #2 on test set: [0.6892127513919153, 0.5387302819761312]

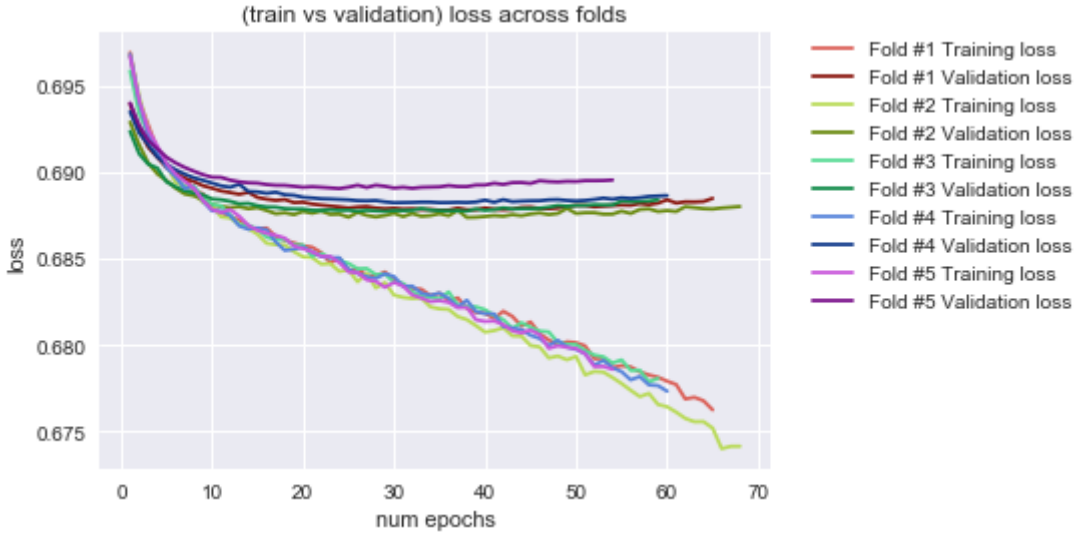
28337/28337 [==============================] - 1s 36us/step
Model #3 on test set: [0.6890842563099627, 0.5340720612752806]
28337/28337 [==============================] - 1s 36us/step
Model #4 on test set: [0.688941865542085, 0.53558951195603]
28337/28337 [==============================] - 1s 36us/step
Model #5 on test set: [0.6899901481380988, 0.5320252673309676]

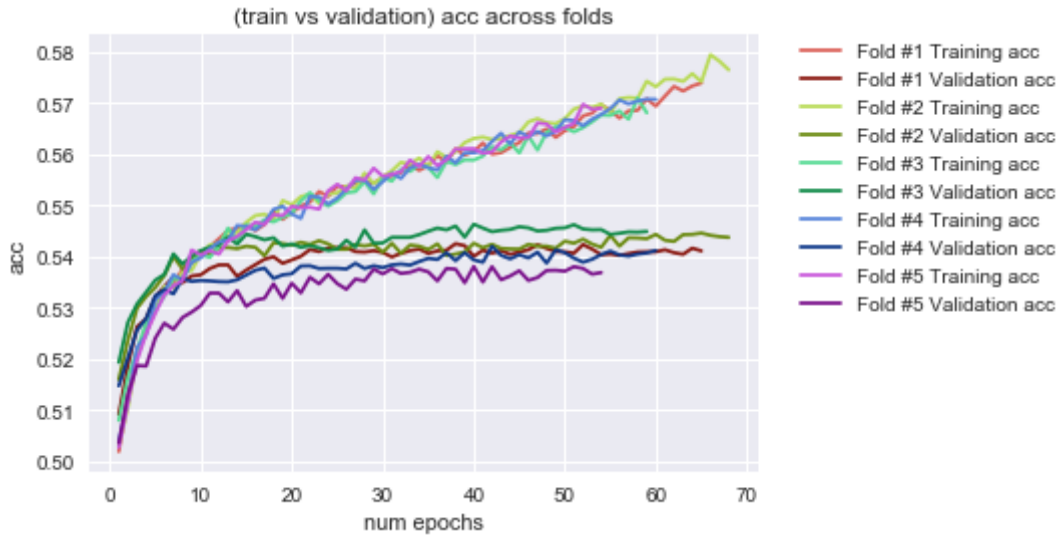
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 280)               78680
_________________________________________________________________
dropout_6 (Dropout)          (None, 280)               0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 281
=================================================================
Total params: 78,961
Trainable params: 78,961
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 35us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 36us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 36us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 38us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.54 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





28337/28337 [==============================] - 1s 37us/step
Model #1 on test set: [0.6890239710864509, 0.5402124431040131]
28337/28337 [==============================] - 1s 36us/step
Model #2 on test set: [0.68947752925593, 0.5383773864663876]

28337/28337 [==============================] - 1s 36us/step
Model #3 on test set: [0.6881588411767847, 0.5380597805137182]
28337/28337 [==============================] - 1s 37us/step
Model #4 on test set: [0.6884419736913674, 0.537565726803022]
28337/28337 [==============================] - 1s 36us/step
Model #5 on test set: [0.6884446603428078, 0.5401065744580312]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 3)	843
dropout_6 (Dropout)	(None, 3)	0
dense_12 (Dense)	(None, 3)	12
dropout_7 (Dropout)	(None, 3)	0
dense_13 (Dense)	(None, 1)	4

Total params: 859
Trainable params: 859
Non-trainable params: 0

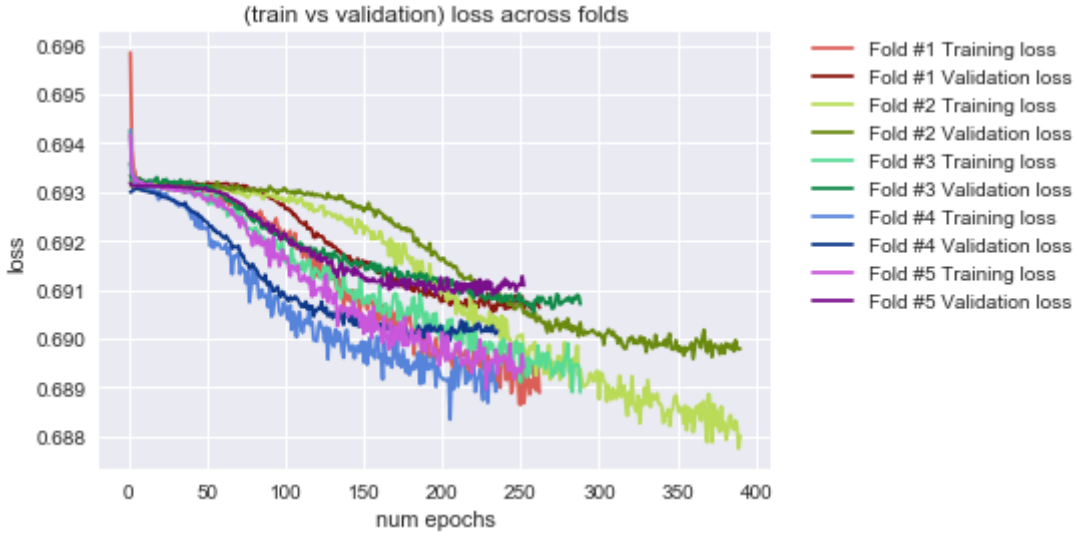
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 38us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 38us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 38us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 35us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples

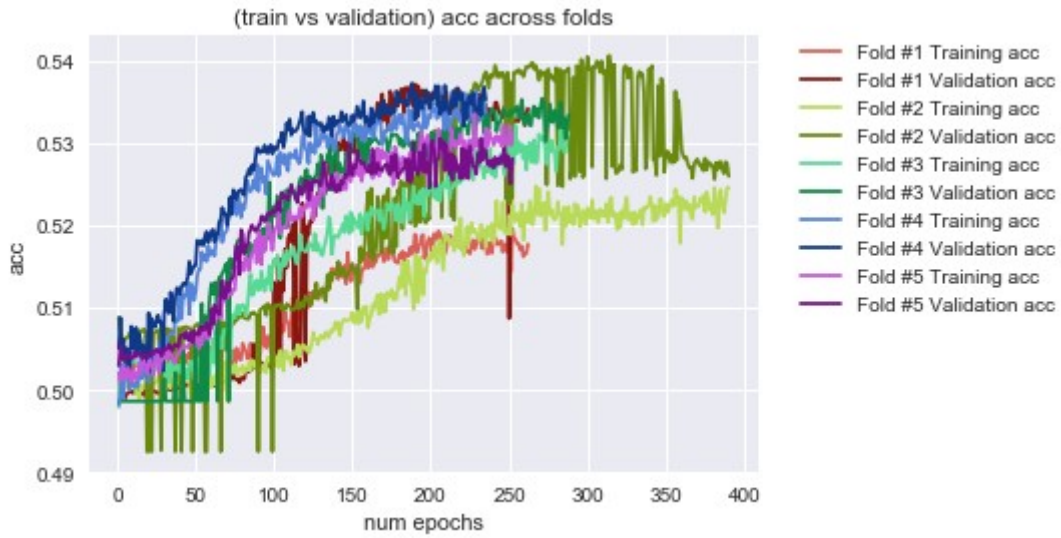
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 40us/step

loss average 0.69 (+-0.000 stdev)

acc average 0.53 (+-0.004 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.53
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.53





28337/28337 [==============================] - 1s 41us/step
Model #1 on test set: [0.6907358985598928, 0.5336132971037796]
28337/28337 [==============================] - 1s 40us/step
Model #2 on test set: [0.6899341521941433, 0.523167590084992]
28337/28337 [==============================] - 1s 41us/step
Model #3 on test set: [0.6903377790055943, 0.5313194763041184]
28337/28337 [==============================] - 1s 40us/step
Model #4 on test set: [0.6896365136460275, 0.5378833327462259]
28337/28337 [==============================] - 1s 40us/step
Model #5 on test set: [0.6912884196944162, 0.5276846525912001]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 32)	8992
dropout_11 (Dropout)	(None, 32)	0
dense_17 (Dense)	(None, 16)	528
dropout_12 (Dropout)	(None, 16)	0
dense_18 (Dense)	(None, 1)	17

Total params: 9,537
Trainable params: 9,537
Non-trainable params: 0

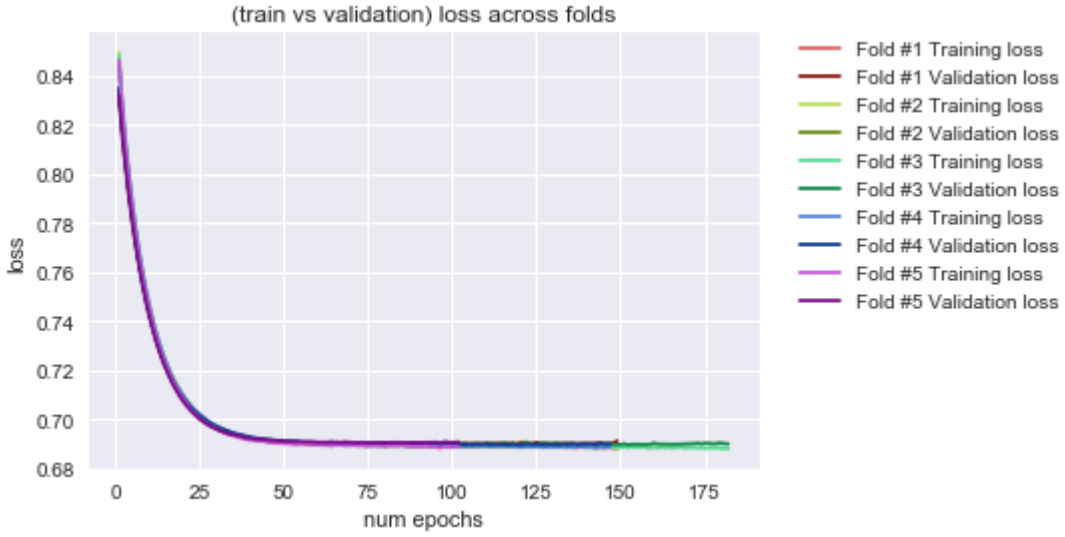
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 36us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples

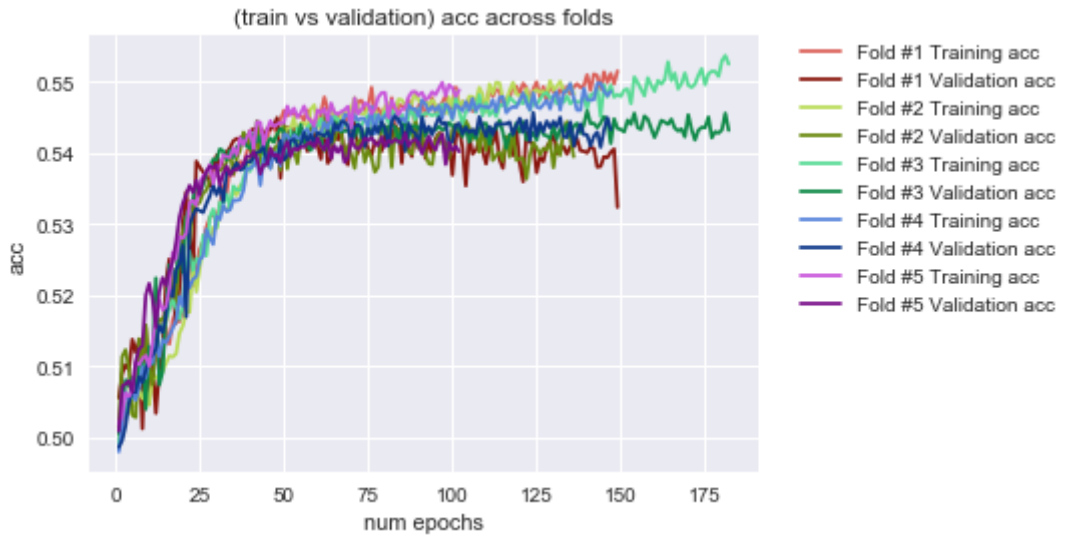
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 37us/step

loss average 0.69 (+-0.000 stdev)

acc average 0.54 (+-0.004 stdev)

Scores #1 loss: 0.69 acc: 0.53
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





28337/28337 [==============================] - 1s 38us/step
Model #1 on test set: [0.6913457120504385, 0.5348837209449565]
28337/28337 [==============================] - 1s 37us/step
Model #2 on test set: [0.6904765932339613, 0.5396830998488631]
28337/28337 [==============================] - 1s 37us/step
Model #3 on test set: [0.690352076876158, 0.5385891237772819]
28337/28337 [==============================] - 1s 37us/step
Model #4 on test set: [0.6902734651921283, 0.5390478879351107]
28337/28337 [==============================] - 1s 37us/step
Model #5 on test set: [0.6904625956499026, 0.5399301267042111]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 280)	78680
dropout_11 (Dropout)	(None, 280)	0
dense_17 (Dense)	(None, 120)	33720
dropout_12 (Dropout)	(None, 120)	0
dense_18 (Dense)	(None, 1)	121

Total params: 112,521
Trainable params: 112,521
Non-trainable params: 0

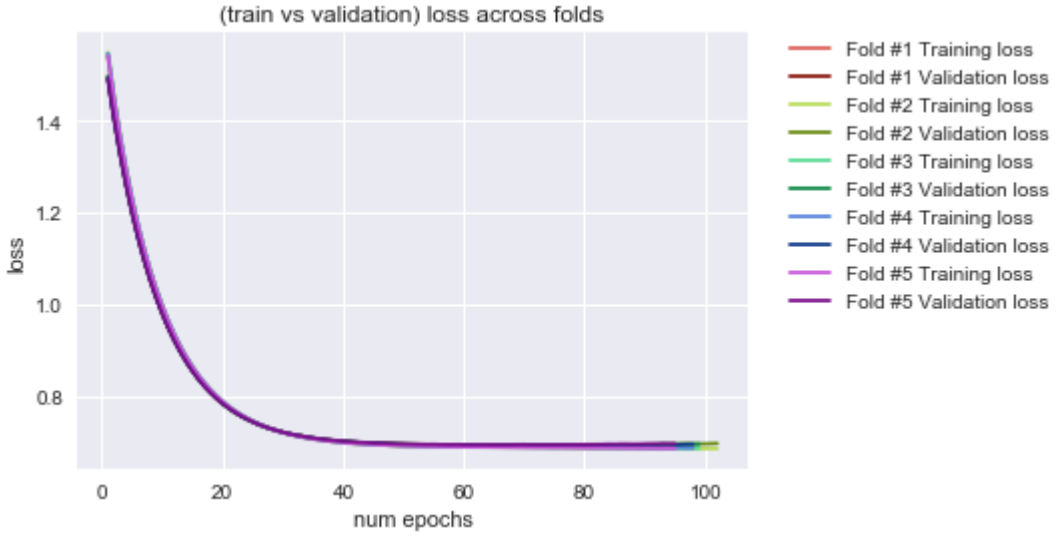
Training fold 1 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 1 of 5 . . .
22668/22668 [==============================] - 1s 37us/step
Training fold 2 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 2 of 5 . . .
22668/22668 [==============================] - 1s 41us/step
Training fold 3 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 3 of 5 . . .
22668/22668 [==============================] - 1s 40us/step
Training fold 4 of 5 . . .
Training using 90668 samples and validating using 22668 samples
Evaluating fold 4 of 5 . . .
22668/22668 [==============================] - 1s 40us/step
Training fold 5 of 5 . . .
Training using 90672 samples and validating using 22664 samples

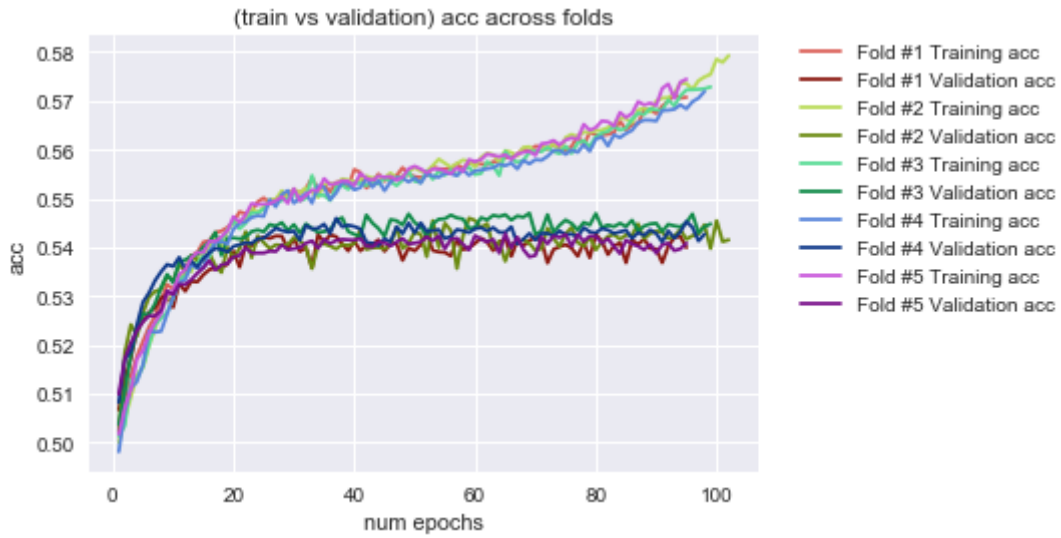
Evaluating fold 5 of 5 . . .
22664/22664 [==============================] - 1s 40us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.70 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.70 acc: 0.54





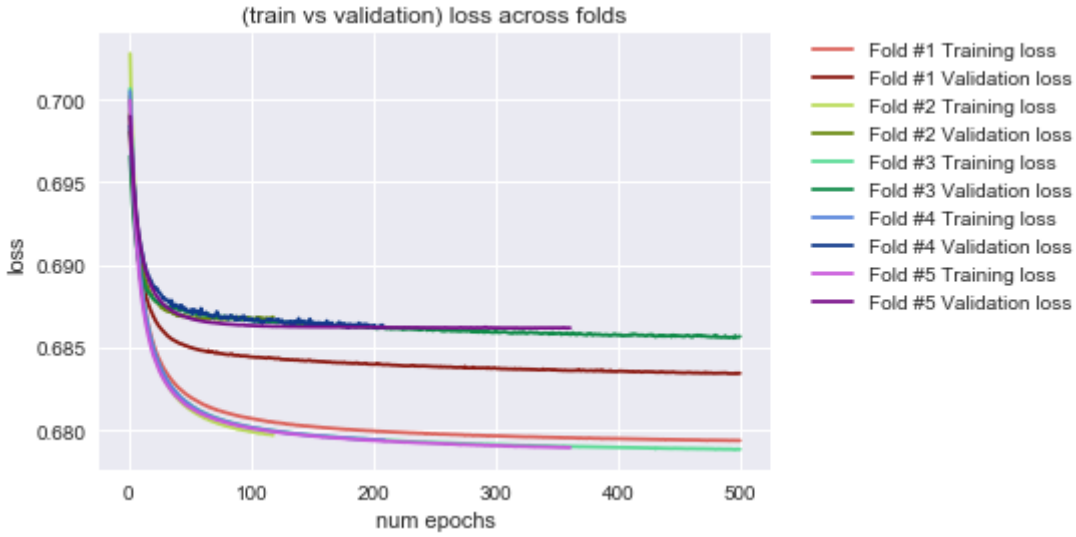
28337/28337 [==============================] - 1s 39us/step
Model #1 on test set: [0.6941743106387197, 0.5399654162549752]
28337/28337 [==============================] - 1s 38us/step
Model #2 on test set: [0.6967392006518455, 0.5377774641055026]
28337/28337 [==============================] - 1s 39us/step
Model #3 on test set: [0.694734418472532, 0.5402830222139547]
28337/28337 [==============================] - 1s 39us/step
Model #4 on test set: [0.6942105159727717, 0.5418710519983356]
28337/28337 [==============================] - 1s 38us/step
Model #5 on test set: [0.6949026786700913, 0.5378480432070306]

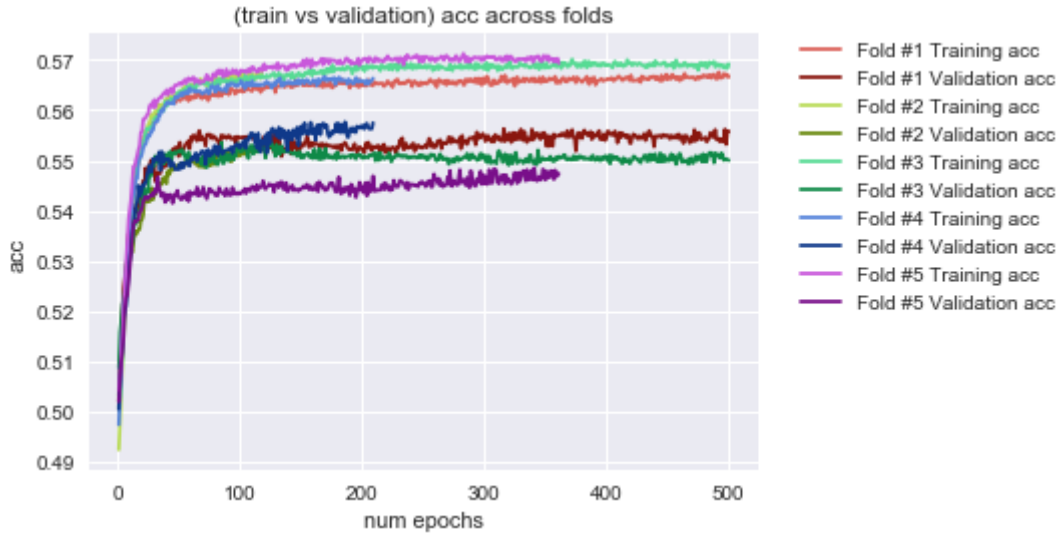
```
Loading matches from game version 8.11.1
Flattened dataframes of (64683, 281)
Shuffled and reformatted 64683 * 280(x)+1(y) dataframe suitable for tensors
Crossvalidating matches from game version 8.11.1 over various models

Starting 5-fold crossvalidation for the following architecture:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 1)                 281
=================================================================
Total params: 281
Trainable params: 281
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 35us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 34us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 34us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 34us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples
Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 34us/step
```

loss average 0.69 (+-0.001 stdev)

acc average 0.55 (+-0.004 stdev)

Scores #1 loss: 0.68 acc: 0.56
Scores #2 loss: 0.69 acc: 0.55
Scores #3 loss: 0.69 acc: 0.55
Scores #4 loss: 0.69 acc: 0.56
Scores #5 loss: 0.69 acc: 0.55





12939/12939 [==============================] - 0s 34us/step
Model #1 on test set: [0.6844344807330806, 0.5563799366441315]
12939/12939 [==============================] - 0s 35us/step

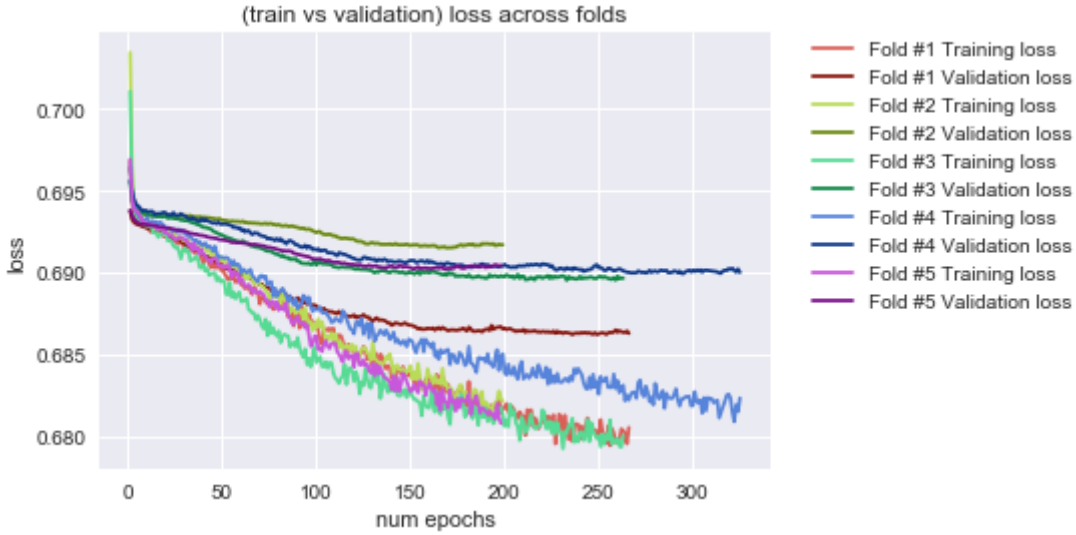
Model #2 on test set: [0.6859987138762099, 0.5471829353072413]
12939/12939 [==============================] - 0s 34us/step
Model #3 on test set: [0.6855254550420185, 0.5517427931245397]
12939/12939 [==============================] - 0s 34us/step
Model #4 on test set: [0.6850022528315081, 0.5532112219057438]
12939/12939 [==============================] - 0s 35us/step
Model #5 on test set: [0.6853518498046144, 0.5517427930969001]

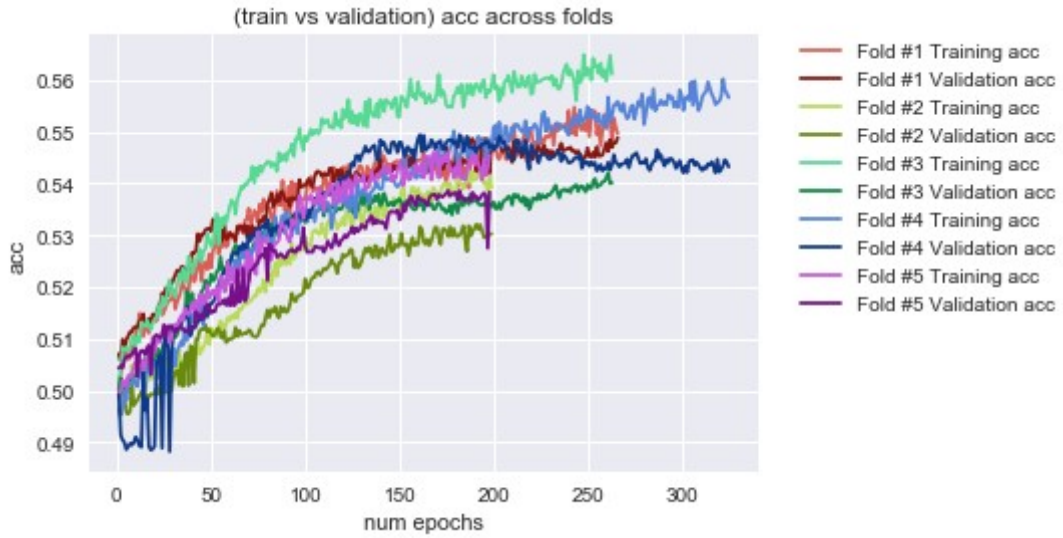
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 3)                 843
_________________________________________________________________
dropout_1 (Dropout)          (None, 3)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 4
=================================================================
Total params: 847
Trainable params: 847
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 41us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 35us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples
Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 37us/step

loss average 0.69 (+-0.002 stdev)
```

acc average 0.54 (+-0.006 stdev)

Scores #1 loss: 0.69 acc: 0.55
Scores #2 loss: 0.69 acc: 0.53
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.54
Scores #5 loss: 0.69 acc: 0.54





12939/12939 [==============================] - 0s 38us/step
Model #1 on test set: [0.6870942508172875, 0.5461782208779964]
12939/12939 [==============================] - 0s 36us/step
Model #2 on test set: [0.6920143440717044, 0.5294072185097785]

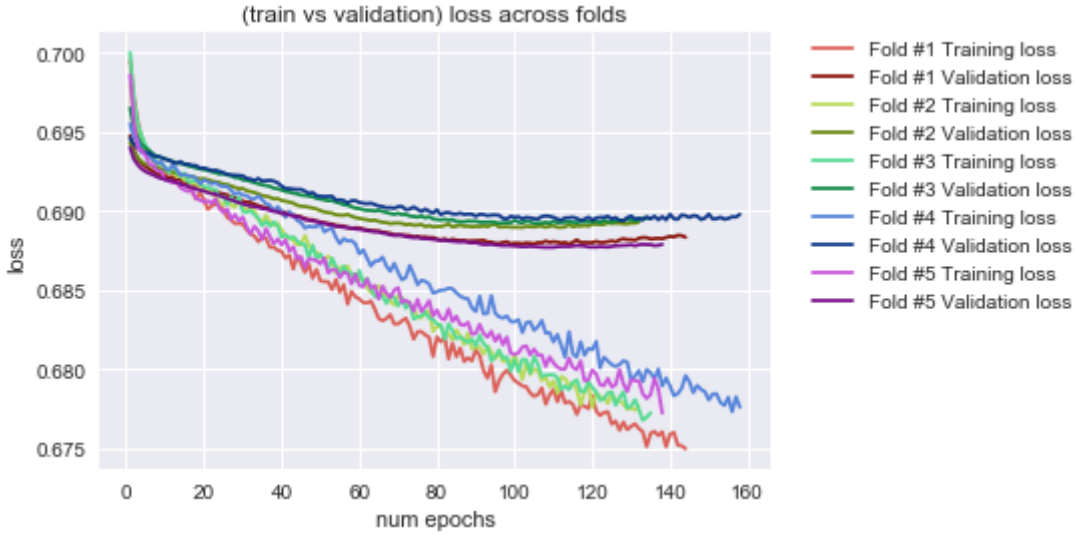
```
12939/12939 [==============================] - 0s 36us/step
Model #3 on test set: [0.6902032562538225, 0.5379086482634509]
12939/12939 [==============================] - 0s 35us/step
Model #4 on test set: [0.6882999113081633, 0.5430867918603285]
12939/12939 [==============================] - 0s 36us/step
Model #5 on test set: [0.6891211056434737, 0.5433959347662413]
```

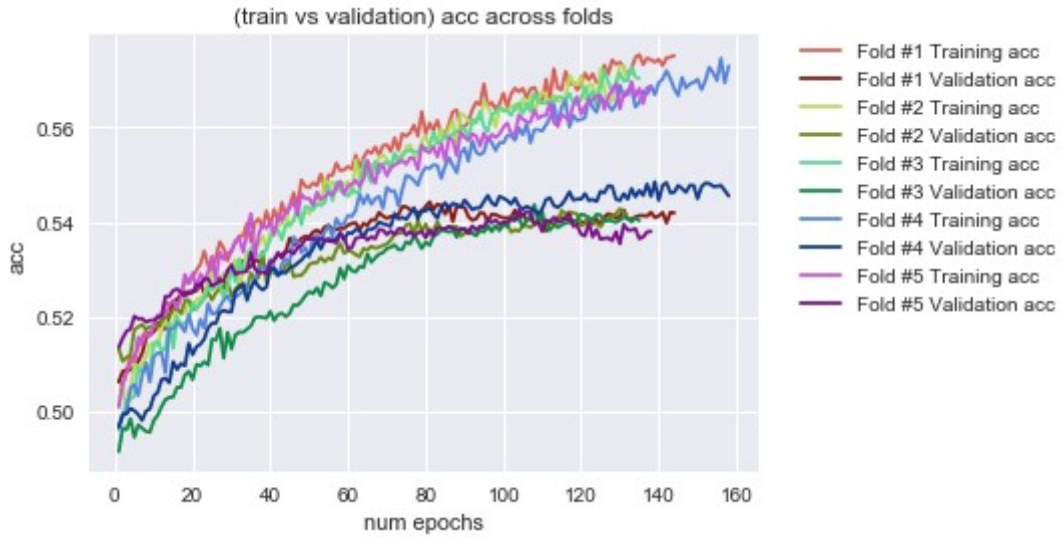
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 9)                 2529
_________________________________________________________________
dropout_6 (Dropout)          (None, 9)                 0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 10
=================================================================
Total params: 2,539
Trainable params: 2,539
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 36us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 36us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 36us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 41us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples
Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 37us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.54 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.54
Scores #2 loss: 0.69 acc: 0.54
Scores #3 loss: 0.69 acc: 0.54
Scores #4 loss: 0.69 acc: 0.55
Scores #5 loss: 0.69 acc: 0.54





12939/12939 [==============================] - 0s 38us/step
Model #1 on test set: [0.6874934629749428, 0.5457145065260373]
12939/12939 [==============================] - 0s 36us/step
Model #2 on test set: [0.6889716846762877, 0.5403045057531799]

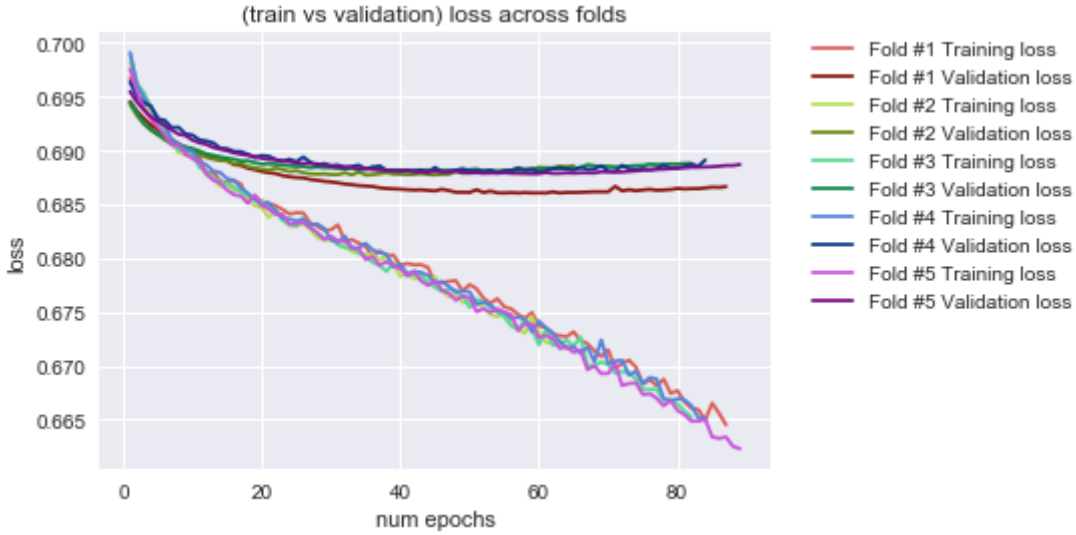
12939/12939 [==============================] - 0s 36us/step
Model #3 on test set: [0.6885085596313376, 0.5410773630064453]
12939/12939 [==============================] - 0s 35us/step
Model #4 on test set: [0.6874651182093091, 0.5434732205099941]
12939/12939 [==============================] - 0s 36us/step
Model #5 on test set: [0.688508251515079, 0.5360537908832537]

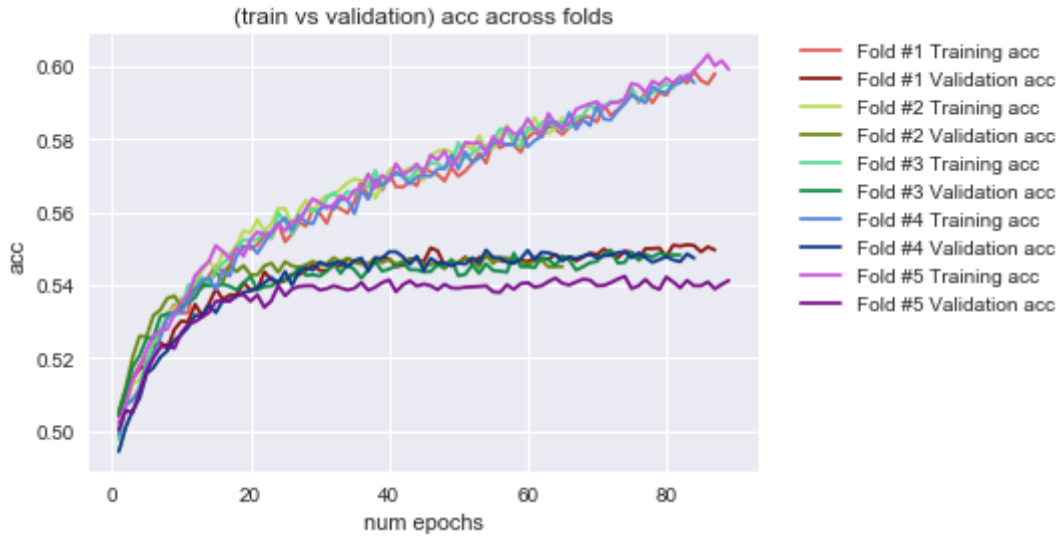
Starting 5-fold crossvalidation for the following architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 280)               78680
_________________________________________________________________
dropout_6 (Dropout)          (None, 280)               0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 281
=================================================================
Total params: 78,961
Trainable params: 78,961
Non-trainable params: 0
_________________________________________________________________
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 1s 49us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 44us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 42us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples
Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 39us/step

loss average 0.69 (+-0.001 stdev)
```

acc average 0.55 (+-0.003 stdev)

Scores #1 loss: 0.69 acc: 0.55
Scores #2 loss: 0.69 acc: 0.55
Scores #3 loss: 0.69 acc: 0.55
Scores #4 loss: 0.69 acc: 0.55
Scores #5 loss: 0.69 acc: 0.54





12939/12939 [==============================] - 0s 38us/step
Model #1 on test set: [0.6871147740982684, 0.5485740783861518]
12939/12939 [==============================] - 0s 37us/step
Model #2 on test set: [0.6877090074159381, 0.5419275059850371]

```
12939/12939 [==============================] - 0s 37us/step
Model #3 on test set: [0.6888112414333734, 0.5446325063668591]
12939/12939 [==============================] - 0s 37us/step
Model #4 on test set: [0.6869506710269774, 0.546564649504629]
12939/12939 [==============================] - 0s 37us/step
Model #5 on test set: [0.6892165110840736, 0.5396862199736007]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 3)	843
dropout_6 (Dropout)	(None, 3)	0
dense_12 (Dense)	(None, 3)	12
dropout_7 (Dropout)	(None, 3)	0
dense_13 (Dense)	(None, 1)	4

Total params: 859
Trainable params: 859
Non-trainable params: 0

Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 38us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples

Evaluating fold 5 of 5 . . .

10348/10348 [==============================] - 0s 39us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.52 (+-0.012 stdev)

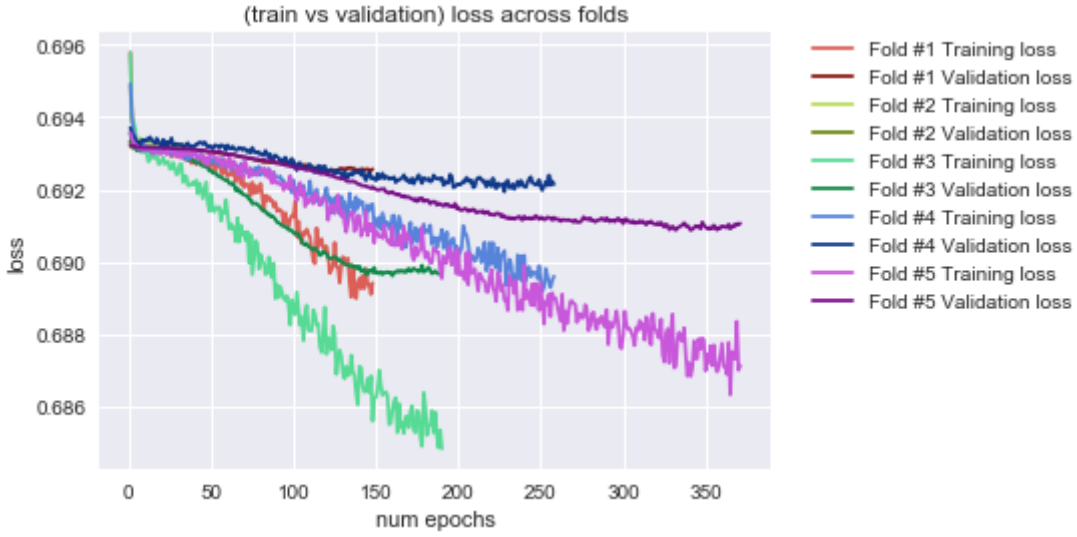
Scores #1 loss: 0.69 acc: 0.52

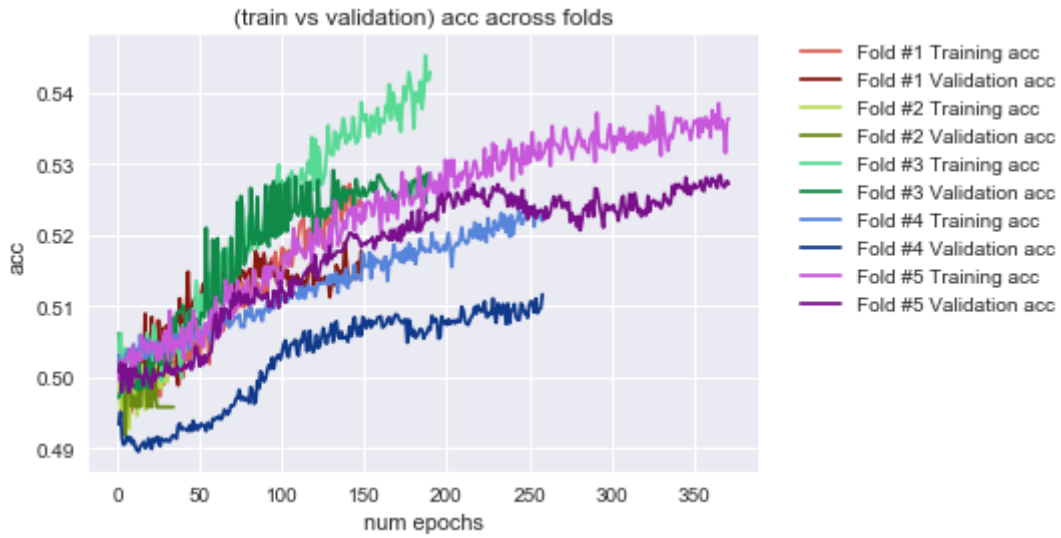
Scores #2 loss: 0.69 acc: 0.50

Scores #3 loss: 0.69 acc: 0.53

Scores #4 loss: 0.69 acc: 0.51

Scores #5 loss: 0.69 acc: 0.53





```
12939/12939 [==============================] - 1s 40us/step
Model #1 on test set: [0.6921716250484404, 0.5280160754492943]
12939/12939 [==============================] - 1s 39us/step
Model #2 on test set: [0.6932125679860114, 0.49091892727873904]
12939/12939 [==============================] - 0s 38us/step
Model #3 on test set: [0.6890839760127894, 0.532189504616927]
12939/12939 [==============================] - 1s 39us/step
Model #4 on test set: [0.6916103612069268, 0.5296390756857581]
12939/12939 [==============================] - 0s 38us/step
Model #5 on test set: [0.6904811586144443, 0.5309529330117025]
```

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 32)	8992
dropout_11 (Dropout)	(None, 32)	0
dense_17 (Dense)	(None, 16)	528
dropout_12 (Dropout)	(None, 16)	0
dense_18 (Dense)	(None, 1)	17

Total params: 9,537
Trainable params: 9,537
Non-trainable params: 0

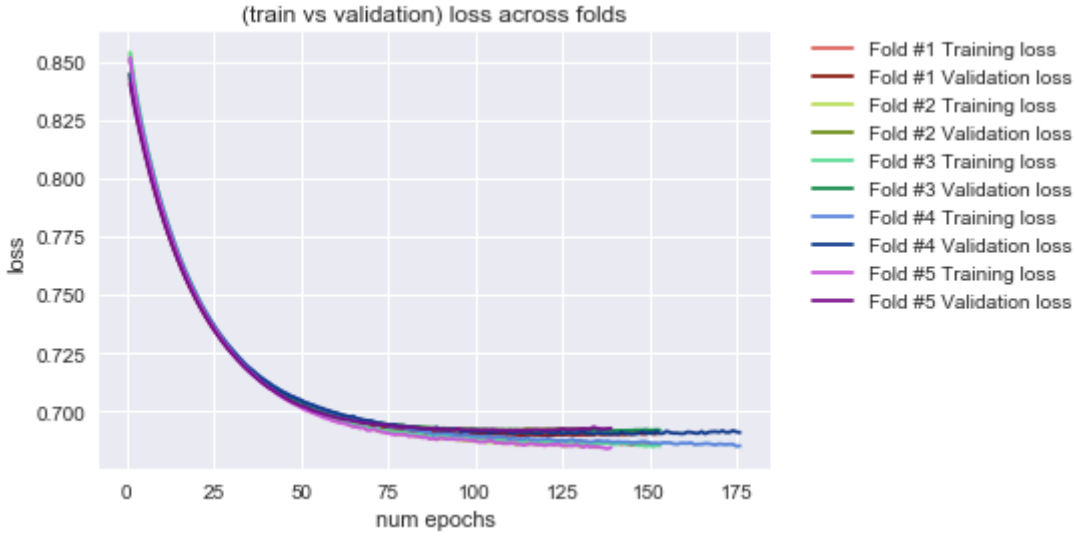
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 39us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 41us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 40us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 40us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples

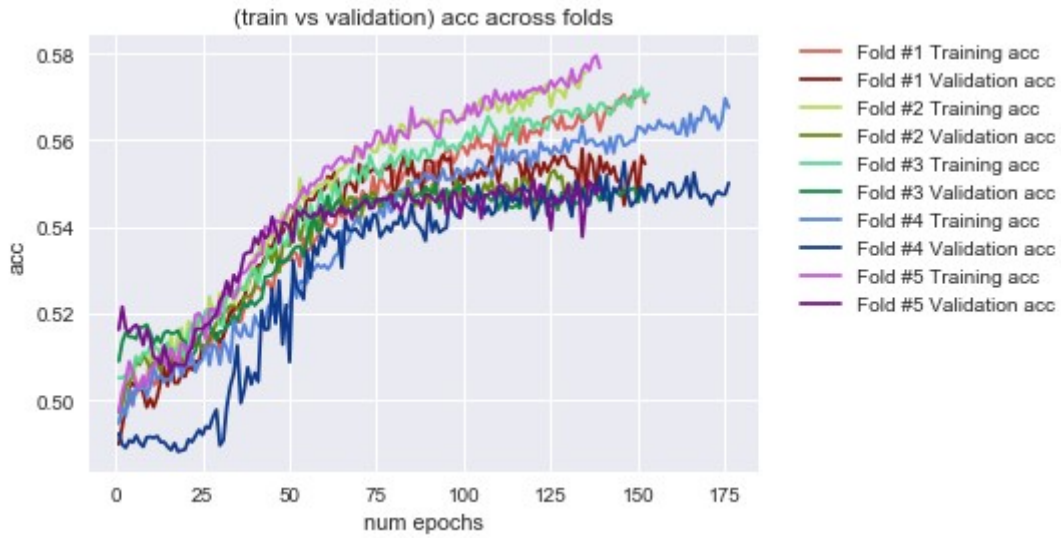
Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 40us/step

loss average 0.69 (+-0.001 stdev)

acc average 0.55 (+-0.002 stdev)

Scores #1 loss: 0.69 acc: 0.55
Scores #2 loss: 0.69 acc: 0.55
Scores #3 loss: 0.69 acc: 0.55
Scores #4 loss: 0.69 acc: 0.55
Scores #5 loss: 0.69 acc: 0.55





12939/12939 [==============================] - 1s 42us/step
Model #1 on test set: [0.6903840488283405, 0.5554525079171803]
12939/12939 [==============================] - 1s 40us/step
Model #2 on test set: [0.692430361454689, 0.5449416492727719]
12939/12939 [==============================] - 1s 41us/step
Model #3 on test set: [0.6919050158401464, 0.5435505062445338]
12939/12939 [==============================] - 1s 40us/step
Model #4 on test set: [0.6904771559361338, 0.5494242213417108]
12939/12939 [==============================] - 1s 41us/step
Model #5 on test set: [0.6923394806326629, 0.5464873638023354]

Starting 5-fold crossvalidation for the following architecture:

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 280)	78680
dropout_11 (Dropout)	(None, 280)	0
dense_17 (Dense)	(None, 120)	33720
dropout_12 (Dropout)	(None, 120)	0
dense_18 (Dense)	(None, 1)	121

Total params: 112,521
Trainable params: 112,521
Non-trainable params: 0

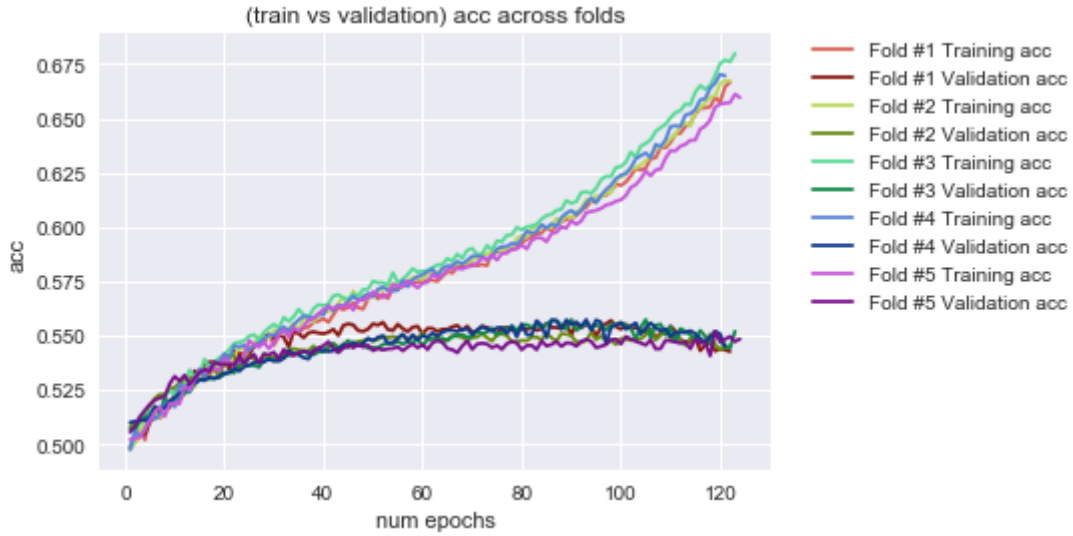
Training fold 1 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 1 of 5 . . .
10349/10349 [==============================] - 0s 42us/step
Training fold 2 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 2 of 5 . . .
10349/10349 [==============================] - 0s 43us/step
Training fold 3 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 3 of 5 . . .
10349/10349 [==============================] - 0s 42us/step
Training fold 4 of 5 . . .
Training using 41395 samples and validating using 10349 samples
Evaluating fold 4 of 5 . . .
10349/10349 [==============================] - 0s 43us/step
Training fold 5 of 5 . . .
Training using 41396 samples and validating using 10348 samples

Evaluating fold 5 of 5 . . .
10348/10348 [==============================] - 0s 44us/step

loss average 0.72 (+-0.004 stdev)

acc average 0.55 (+-0.003 stdev)

Scores #1 loss: 0.72 acc: 0.54
Scores #2 loss: 0.72 acc: 0.55
Scores #3 loss: 0.73 acc: 0.55
Scores #4 loss: 0.73 acc: 0.54
Scores #5 loss: 0.72 acc: 0.55



12939/12939 [==============================] - 1s 45us/step
Model #1 on test set: [0.7200243613692117, 0.5483422212055656]
12939/12939 [==============================] - 1s 42us/step
Model #2 on test set: [0.7235977322390629, 0.5392225056216414]
12939/12939 [==============================] - 1s 43us/step
Model #3 on test set: [0.7335676187866023, 0.5388360769673692]
12939/12939 [==============================] - 1s 43us/step
Model #4 on test set: [0.7242328046748144, 0.5461009351757028]
12939/12939 [==============================] - 1s 43us/step
Model #5 on test set: [0.7214204334468283, 0.5374449339345246]