# NLTHP - Technical Guide

# CA400 Year 4 Project


## **Members:** Immanuel Idelegbagbon (17393433) & Stefano Puzzuoli (17744421)


## **Project Title**: No-Limit Texas Hold'em Poker AI

## **Date Finished**: 01/05/2021


### Abstract

This project consists of an easy to use web application that allows users to test their Texas Hold'em Poker skills against 5 Artificial Intelligence Agents in a game of No-Limit Texas Hold'em Poker. Users are able to select from 4 different levels of difficulty and train and assess their skills according to their desired challenge magnitude.

The user interface consists of a user-friendly web UI which users can access through their web browsers. This UI allows the users of the application to visualise the Poker game, with the table, opponents, cards and all the possible accepted commands when playing each hand. The application also includes a sign up/log in functionality which allows users to keep track of their game statistics (e.g. winning percentage). Users which do not want to avail of this functionality have the option to play games as guests, which will not involve any statistics recording. Since with No-Limit Texas Hold'em Poker there are $10^{71}$ possible game states, the web application makes use of a personally implemented poker hand evaluation library, which is lightweight and fast. This evaluation library handles 5, 6 and 7 card hand lookups and all lookups are done with bit arithmetic and dictionary accesses.

The models (different models for different difficulties) consist of Reinforcement Machine learning models with Gradient Boosting Regressors that allow each Artificial Intelligence Agent to try to predict the maximum expected return value on each different game state throughout a game. These AI Agents are trained repeatedly with the final objective of generating interesting and viable poker winning strategies, allowing users to test and improve their Texas Hold'em Poker skills.

# Table of Contents

# 1. Introduction

## 1.1. Motivation

After successfully creating a Fantasy Football Point Predictor in our Third Year Project, both team members were curious to find out how Machine Learning and Artificial Intelligence could be applied to a completely different game which requires strategy, intuition, and mainly reasoning based on hidden information.

After some profound research and discovering that AI had repeatedly had success at beating humans in past years in games like Chess and Go (games that follow predefined rules and are not affected by random factors), we became aware that much more rare were the Artificial Intelligence Agents/Bots that had obtained the same success in games like No-Limit Texas Hold'em Poker. From this we decided to take on this challenge and try to create AI Agents that can give human poker players a real challenge.

Additionally, since there are countless Texas Hold'em Poker players of various different levels in the world, we thought that it was very important to allow users of the application to be able to select amongst different game difficulties, in order to allow Poker players of all levels to train and assess their skills according to their desired challenge degree.

## 1.2. Glossary

**Agent:** An Artificial Intelligence player.
**AI**: Artificial intelligence.
**API**: Application programming interface.
**Client**: A third-party web application used to play Texas Hold'em Poker.
**GUI**: Graphical User interface.
**Machine Learning**: Application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed.
**AI Model**: Mathematical algorithm that is "trained" using artificial data and human user inputs to predict the maximum expected return value on each different game state throughout a Poker game.
**NLTHP**: No-Limit Texas Hold'em Poker.
**No-Limit**: Rule in Poker that allows players to bet an unlimited amount of money when the action is on them.
**Reinforcement Learning**: Area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward.
**Texas Hold'em Poker**: One of the most popular variants of the card game of Poker. Two cards, known as hole cards, are dealt face down to each player, and then five community cards are dealt face up in three stages. Each player seeks the best five card poker hand from any combination of the seven cards; the five community cards and their two hole cards.
**UI**: User interface.

**UX**: User experience.

## 1.3. Research and Analysis

### 1.3.1. General

The research aspect of this project was very important. As mentioned, although AI has repeatedly had success at beating humans in past years, many of the games in which AI was able to achieve this are two-player zero-sum games (e.g. Chess, Checkers, Go, etc.). In such games, there are approaches (e.g. Nash equilibrium) that can guarantee statistically that the AI cannot lose, no matter what the opponent does. In six-player No-Limit Texas Hold'em Poker, such a solution is not applicable and creating AI models that can compete against humans is much more challenging. The reason for this is that No-Limit Texas Hold'em Poker contains an enormous strategy space, imperfect information and stochastic events, all elements that characterize most of the highest level challenging problems in multi-agent artificial intelligence systems. Since with No Limit Texas Hold'em Poker there are $10^{71}$ possible game states, extensive research was necessary to investigate the best approaches to tackle AI solutions for such a large strategy space game.

From the initial research, it was found that some of the main challenges in creating AI Poker Agents are the following:

- **Hand strength estimation**

  This is to estimate the winning potential of agents' hands as well as that of their opponents, based on the community cards on the table.

- **Opponent modelling**

  This involves estimating the probability for available actions (fold, check, call, raise) and the amount involved for each opponent.

- **Decision making and Risk management**

  This involves handling the aggressiveness of the decision of the AI Agents being modelled.

In order to tackle such problems, although there are various different approaches that can be adopted, we found that making use of the historic data of the AI Agents was adequate for the needs of our model.

### 1.3.2. Feature Selection

Once it was decided that the historic data was going to be the main concept on which our model was based, it was necessary to determine what statistics of the Agents should be generated, i.e. what features to store for accurate prediction of hands and/or future actions.

Features are the core elements of an AI Poker Agent created by a machine learning technique. They are the basic building blocks for determining how a Poker Agent plays and performs. In order to develop an accurate model, statistics and certain measures which are used as features, should be collected and analyzed very carefully. A model with redundant features may lead to high computational complexity and low performance, whereas missing features may lead to inaccurate predictions.

There are several factors that affect a Poker Agent's decision, with obvious ones like the Agent's private cards and community cards. Simply from analyzing these we have that there are C(52, 2) = 1326 combinations for private cards and C(52, 6) = 20358520 game configurations amongst a table of 6 players. Solely exposing card combinations to a machine learning model as a feature is very likely to fail because of this very high dimensionality. Thus, with our research we found that a preferable approach is to convert card information using hand evaluation algorithms, and for this purpose we implemented our own hand evaluator which is lightweight and fast, well at least in regards to the performance needs for this application. Our poker hand evaluator assesses the strength of the hand of the agent, assigns it to a class (Straight, Flush, Full House, etc.) and performs the evaluation with bit arithmetic and dictionary lookups.

Additionally to expected features to be selected for our model such as private and community cards, other interesting and viable statistics were analysed to improve the accuracy of predictions of hands and/or future actions of our AI Agents. Aside from the cards information, with some research and analysis it was found that the table context significantly affects the decision of Agents too. Information such as the order of play (e.g. the position of an Agent for a current hand) , the current stack of the Agent, or even the committed portion of stack (chips bet in current hand) was found to considerably affect the predictions and actions of the AI Agents. Furthermore, the possible actions allowed and their corresponding amounts/proportion of stack allowed to bet also seemed to have improved the prediction Poker models. It is important to note that one of the features selected, specifically "Proportion of Raise", is not applicable to all hands but only hands in which the Agent can select the Raise action. This is the only feature which is not applicable to all decisions of the AI. Aside from the ones mentioned above, our model was researched and analysed with a large variety of different features such as "Highest valued card in the revealed community cards", "Number of cards of same suit in the revealed community cards", "Last action selected" and much more, although it was verified that some features worsened the performance of the models, thus when this occurred said features were removed.

It is important to note that each AI Agent has access to this information for each of their opponents too (except hidden information such as Private Cards and Hand Strength) through which they attempt to predict game strategies and opponents' moves.

### 1.3.3. Model Selection

The development of competitive artificial Poker players is a challenge to Artificial Intelligence because the agent must deal with unreliable information and deception which make it essential to model the opponents to achieve good results. With the different features being used having been selected, the idea is that, throughout a game, after each iteration, each Agent is trained using the set number of selected features and labels while the remaining features and labels from the beginning of the Agent's “career” are discarded. The motivation for this discarding is that the expected return of an Agent’s action is a function of the Agent's future actions in any hand, so older hands become inaccurate as an Agent improves.

After experimenting with various machine learning models, more successful and accurate results were obtained with Linear and Ensemble models. With some research and investigation, it was found that this is due to their resistance to overfitting given the large amount of randomness that is present in No-Limit Texas Hold’em Poker. Linear models (e.g. Linear Regression) avoid overfitting via their simplicity. Alternatively, Ensemble models (e.g. Random Forest, Gradient Boosting) work well by fitting regressors to multiple random subsets of the training data, minimizing overfitting.

#### 1.3.3.1. Linear Regression

The first Linear model we attempted was a Linear Regression model. To motivate linear regression, we first checked for a correlation between our chosen features and hand strength. We found that each selected feature was indeed correlated with hand strength. With the evidence for feature correlation in hand, linear regression was a natural algorithmic choice. To perform linear regression we first expanded our feature set by incorporating not only a player’s total game actions, but their individual round actions. The thinking here was that a check during the flop may be weighted differently than a check during the river when trying to predict the river hand rank. To determine the most important features of our algorithm, we conducted a forward search through the feature set. We also found that as more features were added as some of the ones mentioned in the previous section ( “Highest valued card in the revealed community cards”, “Number of cards of same suit in the revealed community cards”, etc,), the prediction model accuracy dropped (verified with hand strength prediction).

We finally performed cross validation of the Linear regression model and obtained an $R^2$ value of 0.115220, which we deemed unsatisfactory for our application. This being said, the errors are relatively high as expected since with Poker, players tend to change tactics during the game, making it difficult to find a pattern with little error.

#### 1.3.3.2. Random Forest Regression

The second model, our first ensemble model attempted, was a Random Forest Regression model.We thought that this could improve in accuracy compared to the previous attempted Linear Regression model as it is a popular approach for low

overfitting and easy interpretability. This interpretability is given by the fact that it is straightforward to derive the importance of each variable on the tree decision. In other words, it is easy to compute how much each variable is contributing to the decision., which was also helpful for the feature selection process. When training our model, the Random Forest Regression model computed how much each feature decreased the impurity, the more a feature decreased the impurity, the more important the feature was. The impurity decrease from each feature was averaged across trees to determine the final importance of the variable, thus assisting us in finding the “best” features for this model.

We performed cross validation of the Random Forest Regression model and obtained an $R^2$ value of 0.370860, which was clearly a very significant and positive improvement compared to the Linear Regressor. Such an accuracy was sufficient for our model and we were quite happy with such a result, but before accepting this solution, we decided to further investigate one last model, which we believed could bring further improvements.

#### 1.3.3.3. Gradient Boosting Regression

Since an adequate accuracy had been achieved with the Random Forest Regression model, we believed that decision trees suited the characteristics of our problem. Before accepting such a model, we wanted to verify if we could further improve the accuracy of our AI Agent and so we investigated a Gradient Boosting Regression model. Gradient Boosting is an ensemble technique that learns from previous predictor mistakes to make better predictions in the future. The technique combines several weak base learners to form one strong learner, thus significantly improving the predictability of models. It works by arranging weak learners in a sequence, such that weak learners learn from the next learner in the sequence to create better predictive models. Gradient boosting adds predictors sequentially to the ensemble, where preceding predictors correct their successors, thereby increasing the accuracy of the model. New predictors are fit to counter the effects of errors in the previous predictors. The gradient of descent helps the gradient booster in identifying problems in learners’ predictions and countering them accordingly.

We performed cross validation of the Gradient Boosting Regression model and obtained an $R^2$ value of 0.468609, which was an outstanding result for us and which indicated that the features selected were quite well fitted for our problem and the model explains a sufficient variability of the response data around its mean.

#### 1.3.3.4. Outcome

All in all, from our experimentations, Ensemble models seemed to outperform Linear models. This is likely because Ensemble methods can capture the non-linearities present in No-Limit Texas Hold’em Poker with regressors like decision trees. As for specific models, best performance was observed with a Gradient Boosting

Regression Model with an $R^2$ value of 0.468609. Although Linear models performed quickly, their accuracy would have made it difficult to to achieve AI Agents that could compete with real humans.

We also experimented with other models and the overall results ($R^2$ values) found to measure how much each model fit our Poker data were the following:

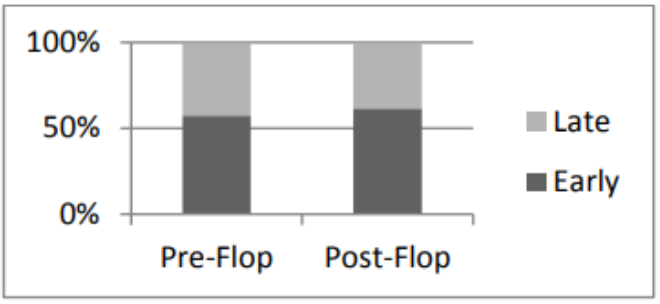
```
Cross-validating LinearRegression...
R-squared (R^2): 0.11522077424650523

Cross-validating Ridge...
R-squared (R^2): 0.13287444732695206

Cross-validating Lasso...
R-squared (R^2): 0.19658688224237508

Cross-validating RandomForestRegressor...
R-squared (R^2): 0.370870631754831

Cross-validating GradientBoostingRegressor...
R-squared (R^2): 0.4686094662377023
```

Models' $R^2$ values

### 1.3.4. Agent Actions Analysis

With our model and features having been selected, and with the model having also been trained (we discuss more in Section 4 on how this was done), we performed some research and analysis on the actions and their frequency of our obtained Agents, specifically on our Expert-level model (our second best performing model). For the purpose of this analysis we consider an action anything but a Fold, thus Check, Call and Raise (or Bet) are the considered actions.

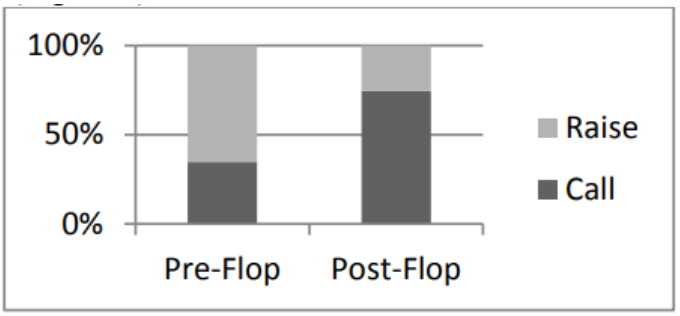
One element we were interested in was to verify how much the position of the player in the hand affected their performance and actions taken in the Pre-Flop and Post-Flop. Since "Position (Order of Play)" is one of the features used in our model and according to our initial analysis it decreased the impurity of the model, thus indicating that it was a variable which helped improve the model's accuracy, we thought it could be interesting to perform this check and investigate the AI Agents according to this feature.




Position Actions Distribution (Pre-Flop and Post-Flop)

Regarding position, there is no notable difference between the players' behavior in Pre-Flop and Post-Flop (Figure above). There are more actions in early positions, as was expected, because they are the first players to act.

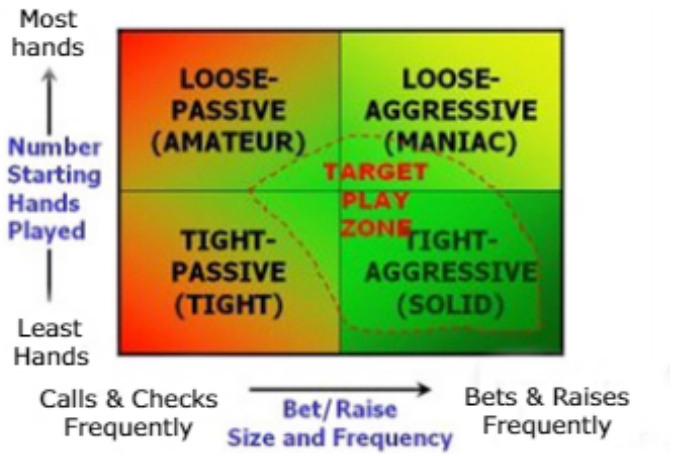
Another interesting area we thought was worth investigating was the aggressivity of the Agents in their plays in the Pre-Flop and in the Post-Flop.




Call and Raise Actions Distribution (Pre-Flop and Post-Flop)

We notice that in this case there is a clear difference of the types of actions. In Pre-Flop there are more raise actions than calls, and after the Flop there are much more calls than raises. This can be explained by the fact that there are more chips (larger gain) involved in Post-Flop actions which explains the higher frequency of more passive actions, because the Agents are more cautious to lose larger amounts (gain function returns lower value).

Having obtained such results, we performed some further investigating and found that Poker Agents can be classified under four categories of playing styles. Each style describes the frequency of play and how the player bets. Playing styles are loose/passive, tight/passive, loose/aggressive and tight/aggressive. It has been proven by previous research (Annija Rupeneite, (2014). Building Poker Agent Using Reinforcement Learning with Neural Networks , Faculty of Computing, University of Latvia, 19 Raina blvd., LV-1586 Riga, Latvia, https://www.scitepress.org/papers/2014/51489/51489.pdf), that the AI Poker Agents can be vary in aggressiveness and cautiousness with the "Target Play Zone" being highlighted in the following table:




Agent classification and target play zone for an AI Poker Agent

So from this research and the analysis of our model, we believed that our Expert-level AI Poker Agents were performing with an appropriate balance between Calls and Raises, especially since this varied according to the chips at stake.

### 1.3.5. AI Difficulties/Levels Analysis

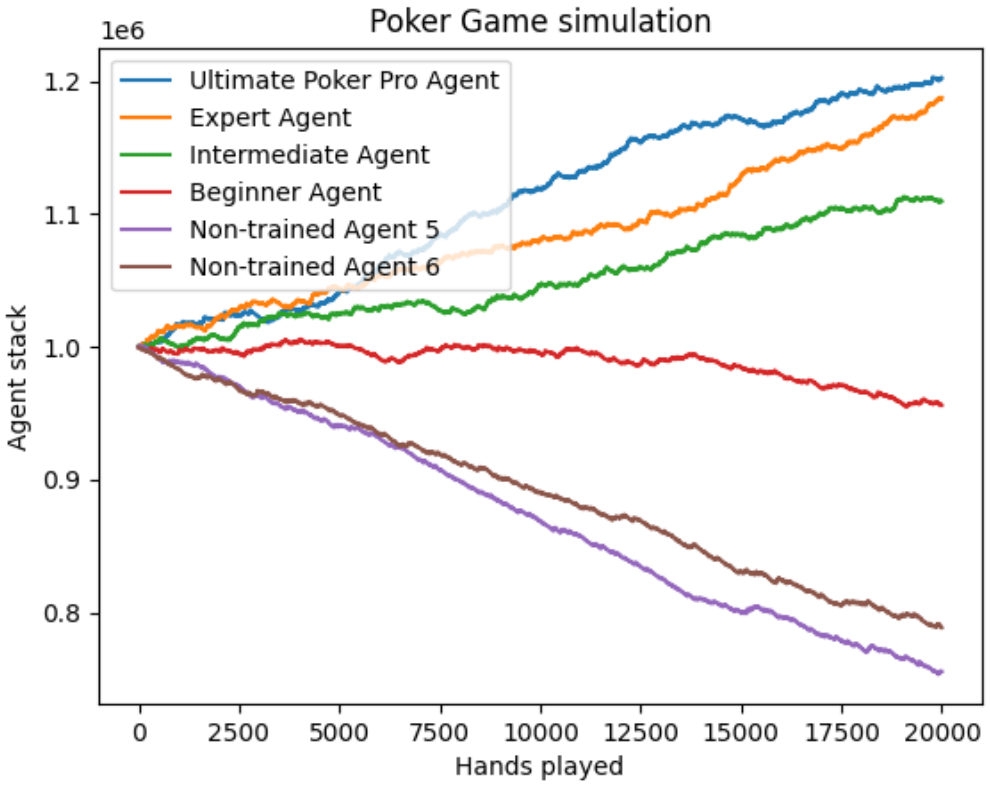
As mentioned earlier, one of the main purposes of this web application is to allow users of the application to be able to select amongst different game difficulties, in order to allow Poker players of all levels to train and assess their skills according to their desired challenge degree.

The models (different models for different difficulties) consist of Reinforcement Machine learning models with Gradient Boosting Regressors that allow each Artificial Intelligence Agent to try to predict the maximum expected return value on each different game state throughout a game. These AI Agents are trained repeatedly a different number of times according to their skill level with the final objective of generating interesting and viable poker winning strategies, allowing users to test and improve their Texas Hold'em Poker skills.

The different levels of AI Agents are trained accordingly:

- AI "Beginner" difficulty: trained with 10000 simulated hands.
- AI "Intermediate" difficulty: trained with 50000 simulated hands.
- AI "Expert" difficulty: trained with 100000 simulated hands.
- AI "Ultimate Poker Pro" difficulty: trained with 250000 simulated hands.

Once the different Agents were successfully trained, we ran a Poker Game simulation of 20000 hands with one type of each Agent along with 2 non-trained Agents to verify their performance against each other. The performance was measured via overall stack throughout the simulation and the result was the following:




Performance of differently trained AI Agents over 20000 hands

As expected and desired, Agents that have been trained over a larger number of hands tend to be more skilled and use more winning Poker strategies. It is clear from our simulation how the training significantly impacts performance with the order of the players from best to worst being:

1. Ultimate Poker Pro Agent
2. Expert Agent
3. Intermediate Agent
4. Beginner Agent
5. Non-Trained Agent 6
6. Non-Trained Agent 5

# 2. System Architecture

## 2.1. Languages, Compilers and Tools

### 2.1.1. Programming Languages

- Python
    - Version: 3.9.1
    - Usage: NLTHP backend, Flask Server

- JavaScript
    - Version: ES6
    - NLTHP User Interface, NLTHP Login, NLTHP Database

### 2.1.2. Tools

#### 2.1.2.1. Platforms

- Firebase
    - Version: 8.2.4
    - Usage: User Login Authentication, NLTHP Database

- Docker
    - Version: 20.10.5
    - Usage: Deployment of NLTHP

- DigitalOcean
    - Usage: Hosting of NLTHP

- Namecheap
    - Usage: Domain Registration

#### 2.1.2.2. Libraries

- Scikit-learn
    - Version: 0.24
    - Usage: Python Machine Learning Models & Rules

- Flask
    - Version: 1.1.2
    - Usage: Server to Connect Backend to Frontend

- Node
    - Usage: React Development of Frontend

- React
    - Version: 16.5.2
    - Usage: Development of Javascript Frontend

- Jest
    - Version: 26.6.3
    - Usage: Testing Javascript Frontend

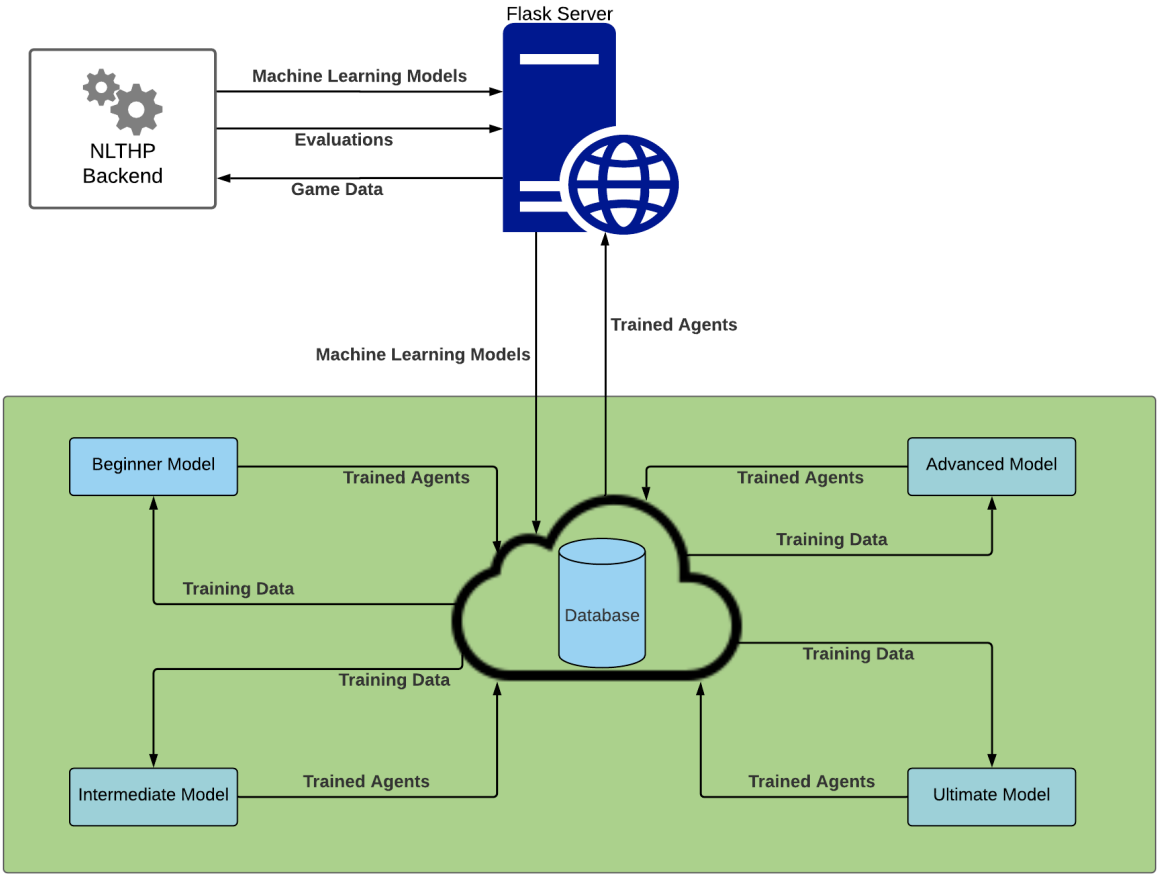
## 2.2. Dependencies

The list of the dependencies for the NLTHP app include:

- Python
- Scikit-Learn
- Flask
- Yarn
- JavaScript
- Node
- React

However, by use of Docker, a container was created for the NLTHP app. This container is present in a Docker Virtual Machine which runs on Linux. All dependencies required to run the NLTHP app are installed on the virtual machine. On deployment, the app is hosted from this virtual machine, eliminating the need for a user to have every dependency installed on their system.

## 2.3. Backend Architecture

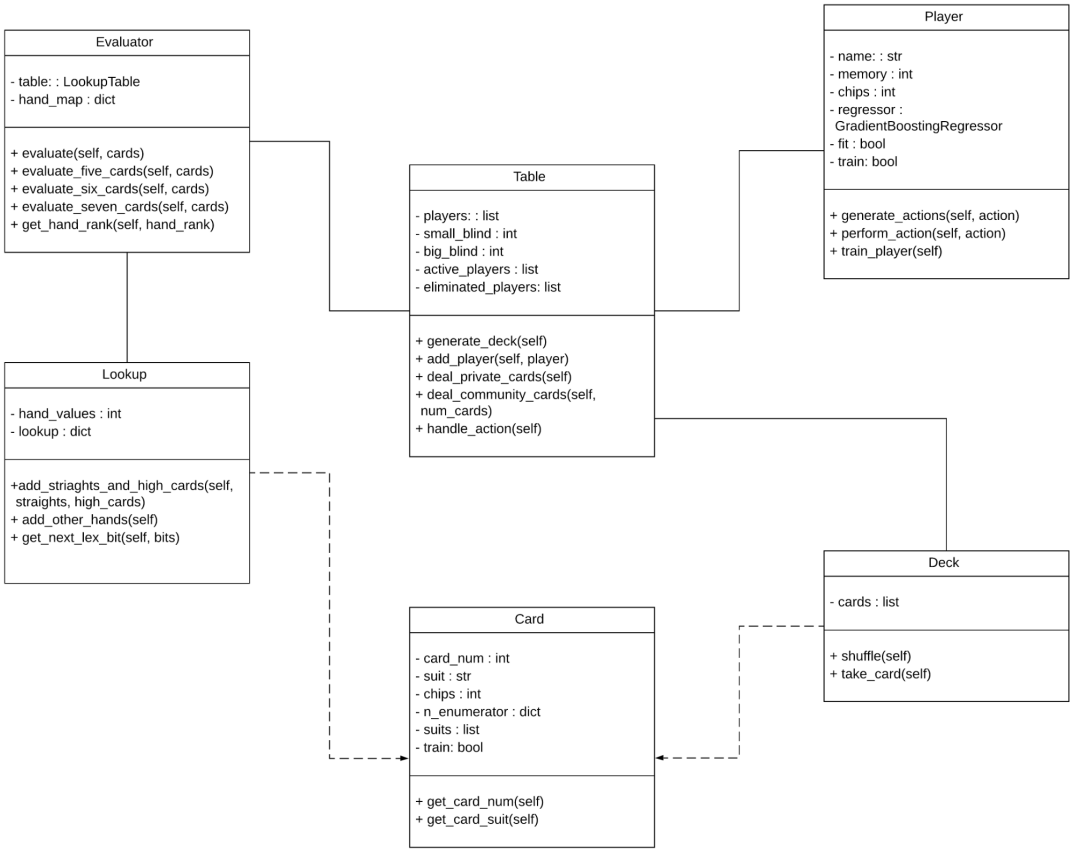



NLTHP Backend Architecture Diagram

Above is a high-level view of the architecture of the NLTHP backend. The diagram depicts the flow of data as the application backend interacts with its server and database. The architecture of the NLTHP comprises the NLTHP backend, Flask Web Server and a Firebase Database.

### 2.3.1. NLTHP Backend

The NLTHP backend is written in Python. It comprises classes that define and implement the logic of No-Limit Texas Hold'em Poker and accordingly, make up the poker game. These classes include a Card class, Player class and Table class. The backend also includes a powerful card hand evaluator. This evaluator is used to evaluate the rank of a player's hand. Finally, agents are trained by a script that simulates a magnitude of poker games. The results of these games are applied to a Gradient Boosting Regression model which produces trained agents. Once created, the model and trained agents are stored in the database.




NLTHP Backend Class Diagram

Above is a simple class diagram that represents the structure of the NLTHP backend. It shows the main classes of the NLTHP backend as entities. The relationship between classes is also apparent. A dotted arrow represents a class that is dependent on another, while the solid lines depict associations between classes.

#### 2.3.1.1. Player Class

The player class defines the Player object which represents a player in a game of poker. Both users and agents are regarded as players. Players are initialised with a set of attributes that are used to control their behaviour. One of these attributes is a regressor that can be fitted with simulated data. Another attribute players possess is the memory attribute. This attribute determines the amount of hands a player can remember when being trained. Higher level AI players have a greater memory, giving them more exposure to the possible hands they might come across. When an AI player encounters a hand it possesses data on, the AI player is able to make an informed decision on its turn. The Player object also provides players with core functionalities including betting and folding.

#### 2.3.1.2. Card Class

The Card class defines the Card object which represents a card in a game of poker. Just like in the Player class, Cards are initialised with a set of attributes that are used to control their behaviour, most importantly their suit and number. The Card object also provides cards with functions that allow them to reveal data relating to them.

A game of poker uses a list of Cards that is manipulated by the Deck Class. The Deck class defines the Deck object which is essentially a list of cards. The Deck object provides functionalities for the Deck like shuffling and handing out cards.

#### 2.3.1.3. Table Class

The table class defines the Table object which represents the table in a game of poker. The Table implements the logic of No Limit Texas Hold'Em Poker and controls the flow of the game. It is responsible for starting/ending rounds, adding/removing players from the table and handling the game state. Along with this, the Table also acts as the dealer by dealing cards and handling the allocations of money during rounds.

#### 2.3.1.4. Hand Evaluator

The strength of a card hand is evaluated using a lightweight Hand Evaluator. This Evaluator is implemented using bit arithmetic. Bit arithmetic involves using the bitwise operations that work with individual bits. Bits are the smallest units of data in a computer making their operations significantly faster and lighter in memory than more commonly used bytes. An object called a LookupTable is also implemented. This comprises a deck in the form of a dictionary of bitwise evaluations. All possible card hand combinations are calculated and added to the LookupTable again using bit arithmetic. The Evaluator can instantly look up a player's hand in the LookupTable and obtain its evaluation. Player hands are compared using bit arithmetic and ranked. The Evaluator handles lookups for 5 card hands, 6 card hands and 7 cards hands.

### 2.3.2. Flask Server

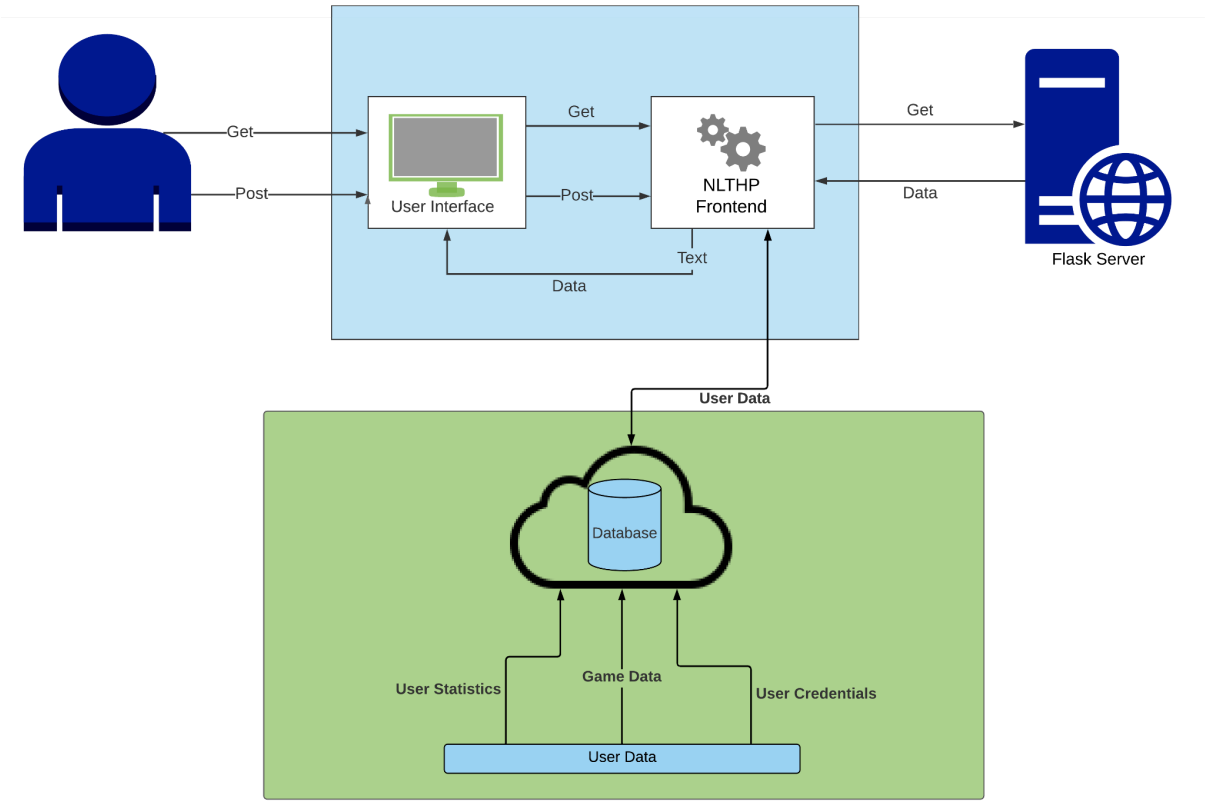
The Flask server is written in Python using Flask. Flask is a web framework that provides a library and tools to develop Python web applications. Using the Flask framework eliminates the need to write low-level code like thread management speeding up the development cycle. A Flask Server was set up to allow the React frontend to communicate with the Python backend. The server is set up within a

Python virtual environment. Once running, any requests not recognised by the frontend will be redirected to the backend.

### 2.3.3. Firebase Database

The NLTHP application uses one database for the frontend and backend, both of which are able to communicate with the database. The NLTHP backend stores the machine learning models in the database. There are four models, each for a different difficulty of AI agents. Once agents have been trained, they are also stored in the database. Upon the user loading the game, the agents for the selected difficulty are called. After every game, the model is used to make a new set of trained agents using user data. These trained agents replace the already existing agents in the database. The frontend's use of the database will be explained in section 2.4.3.

## 2.4. Frontend Architecture




NLTHP Frontend Architecture Diagram

Above is a high-level view of the architecture of the NLTHP frontend. The diagram depicts the flow of data as the user interacts with the user interface. The architecture of the NLTHP frontend comprises the User, NLTHP frontend, User Interface, Flask Server and a Firebase Database.

### 2.4.1. User

The user initiates the cycle of processes that occur within the NLTHP application. Upon accessing the NLTHP app URL, a get request is sent to the web server. This prompts the web server to load the NLTHP app. Users interact with the NLTHP app via the User Interface which controls the NLTHP Frontend. User input can either send data to the NLTHP app or change the state of the NLTHP app itself.

### 2.4.2. NLTHP Frontend

The NLTHP Frontend is written in React. It comprises components that implement NLTHP logic from classes in the NLTHP Backend. These components visually represent the objects from the NLTHP backend classes. When a user loads the NLTHP app and starts a game, the NLTHP frontend sends a Get request to the Flask Server. Data from the backend including the trained AI agents are retrieved. This data is then displayed to the user via the User Interface.

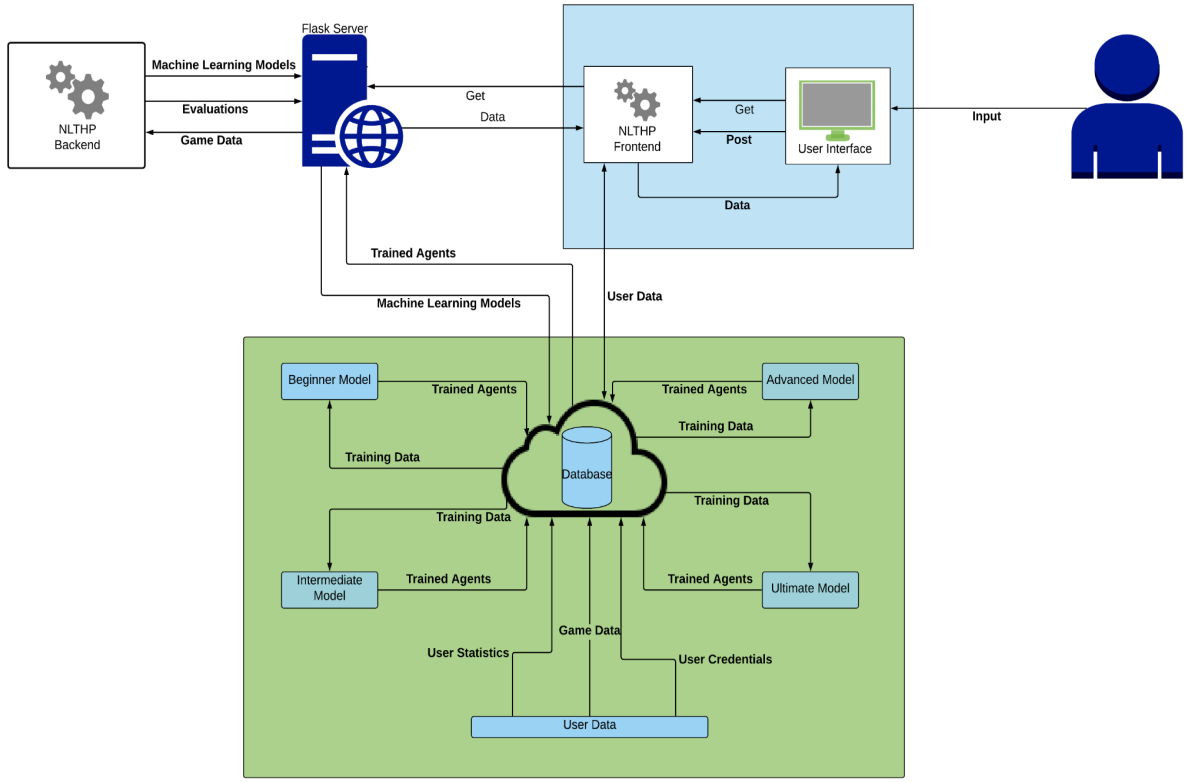
Also included in the NLTHP frontend is a login feature. This enables a user to create an account that tracks their game statistics. If a user avails of this feature, a post request with their data is sent to the web server. Once received by the app this data is added to the Firebase Database.

User authentication is also implemented in the NLTHP app using Firebase Authentication. Users are only able to access the app by logging in with verified credentials. To do this, users must create an account on the NLTHP app. Users who do not wish to create an account can still access the app using the guest feature. This allows users to access the app via a verified guest account, however, statistics are not recorded on the guest account.

### 2.4.3. Firebase Database

As mentioned in section 2.3.3, the NLTHP application uses one database for the frontend and backend, both of which are able to communicate with the database. The NLTHP frontend stores user data in the database. User data consists of user statistics, user game data and user credentials. User statistics are recorded after each game a user completes. These statistics can be retrieved and displayed to the user by the NLTHP frontend. User game data comprises data of user decisions during games. This data is used to retrain the AI agents with the intent that they adapt to user play. Finally, user credentials are stored in the database when a user creates an account. Firebase Authentication uses these credentials to verify users when they log in.

## 2.5. Complete System Architecture



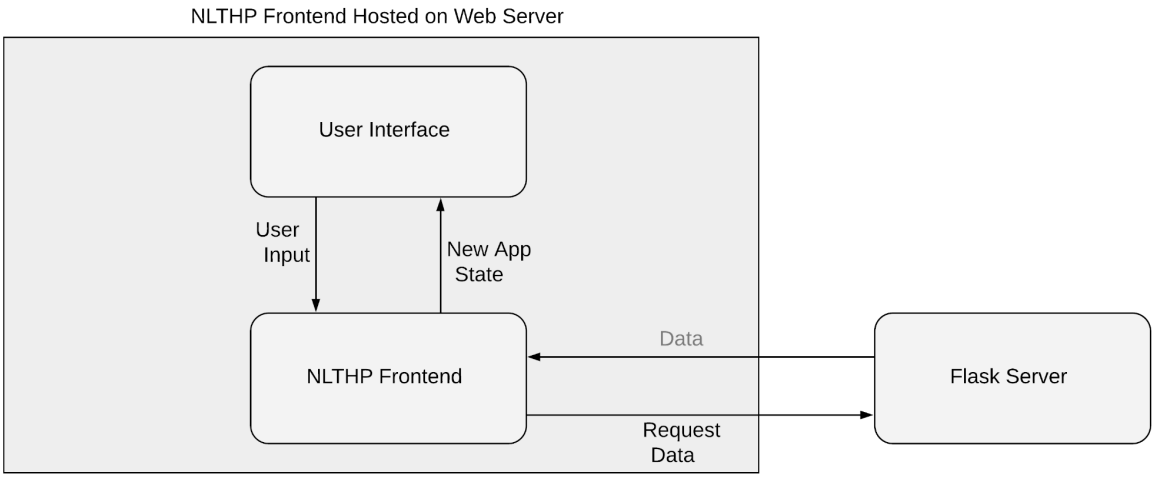

NLTHP Architecture Diagram

Above is a complete high-level architecture diagram of the NLTHP application. It shows the full interaction between the NLTHP frontend and NLTHP backend.

# 3. High-Level Design

Data flow diagrams were used to depict and visualize the flow of data during the various processes in the NLTHP application.

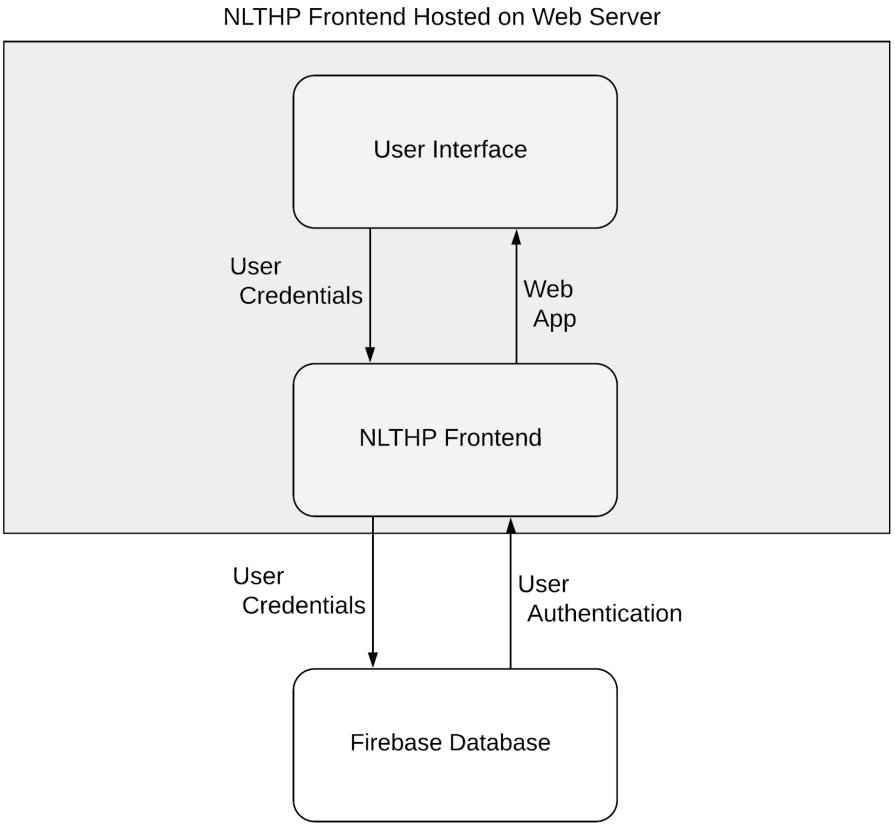
## 3.1. Data Flow Diagram - General Interaction




General App Interaction Data Flow Diagram

This data flow diagram shows the general interaction that takes place in the NLTHP application during a game. The app takes in user input via the user interface. If necessary, the NLTHP frontend will request data from the NLTHP backend via the Flask Server. The new app state is then loaded for the user. Usually, user input causes the app to change state. User input consists of any interaction the user makes with the NLTHP user interface.

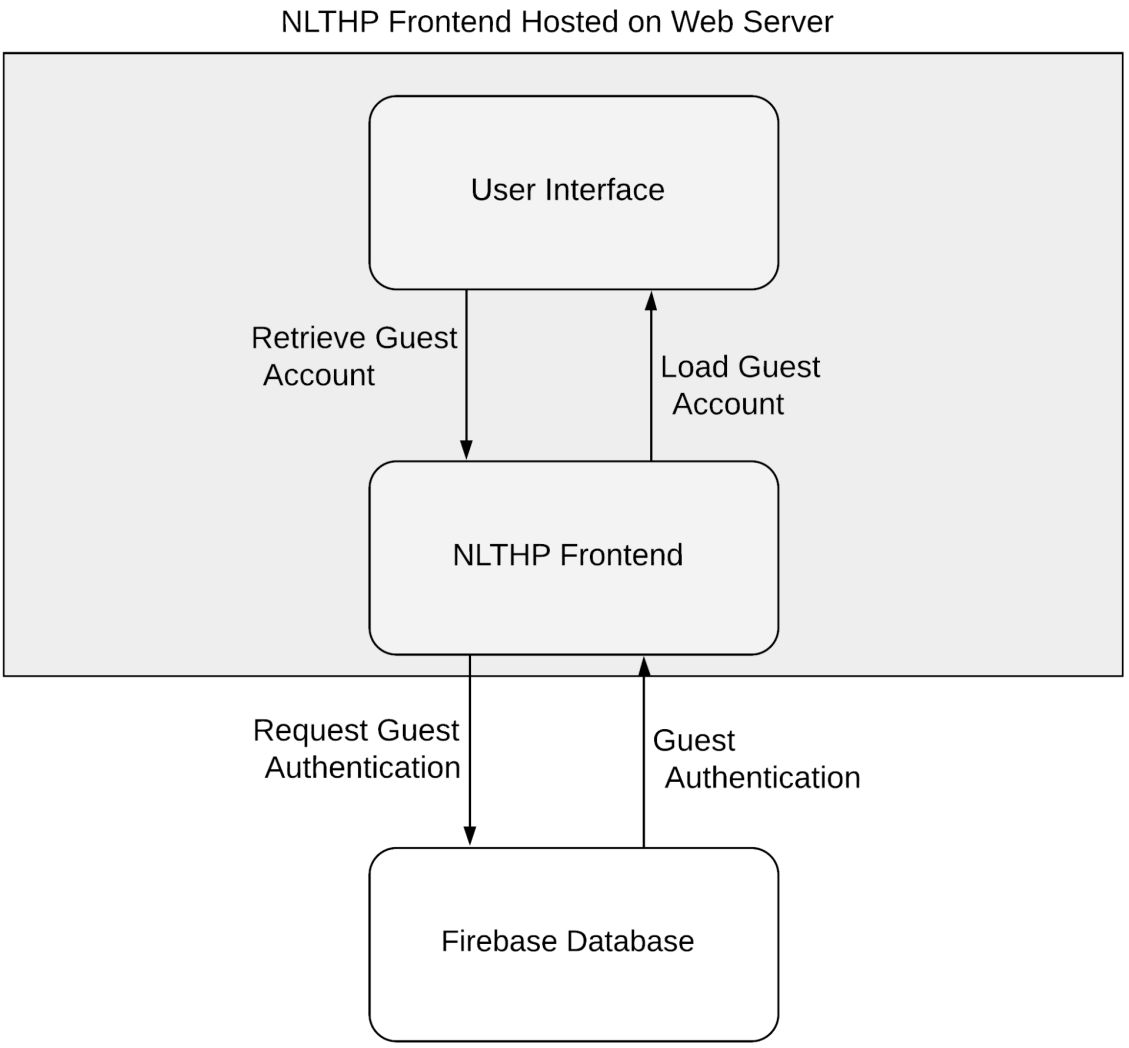
## 3.2. Data Flow Diagram - Sign Up/Log In




Sign Up/Log in Data Flow Diagram

This data flow diagram shows the flow of data as a user initiates the sign up or log in process. Upon loading the NLTHP application, the user will be prompted to enter their login details or to create an account. After the user enters their credentials via the NLTHP user interface, their credentials will be added to the app's database. This database is hosted on Firebase, which also provides user authentication. Once authentication is received from Firebase, the NLTHP dashboard will be loaded for the user.

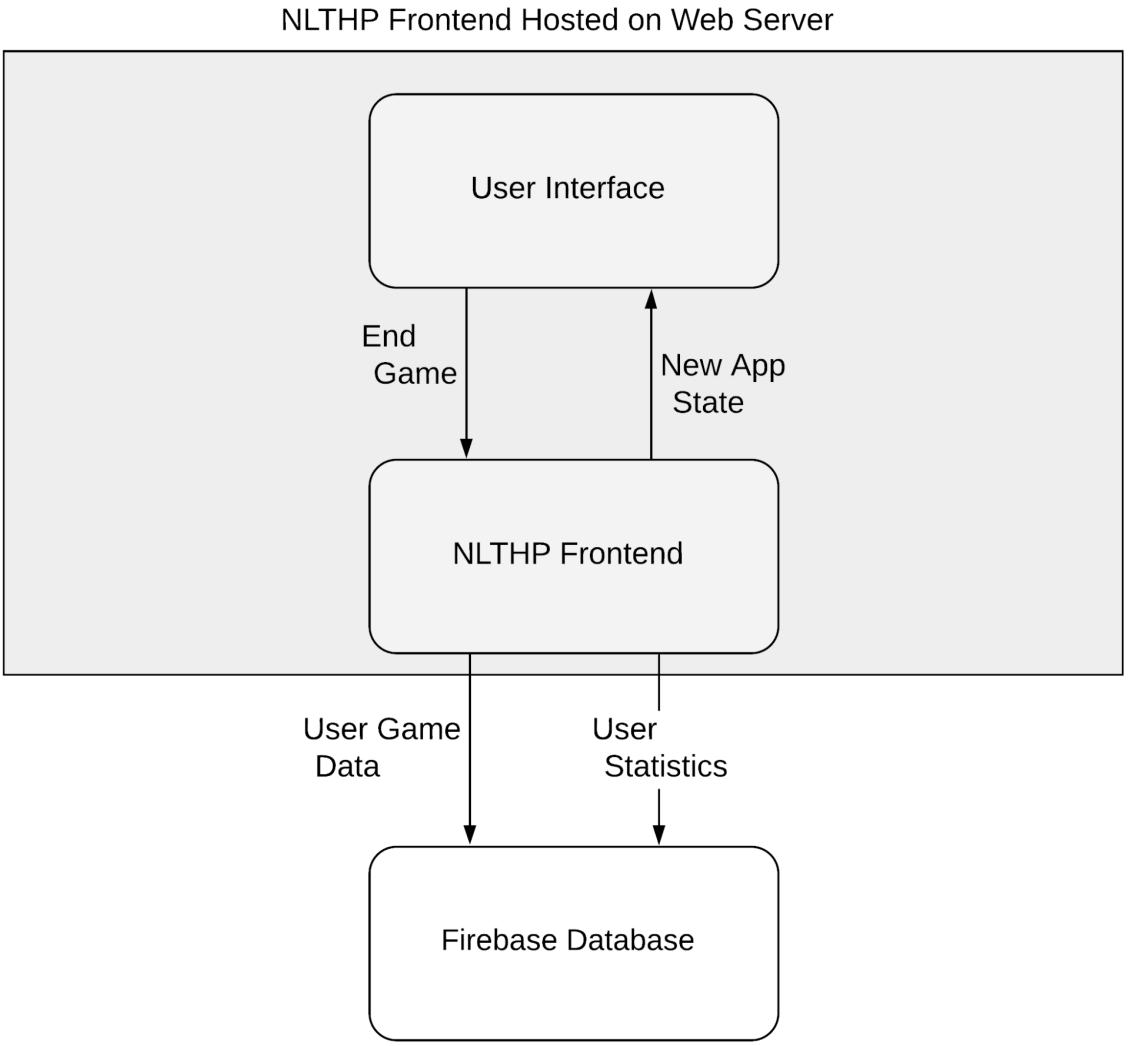
### 3.3. Data Flow Diagram - Guest Account




Guest Log In Data Flow Diagram

The above data flow diagram shows the flow of data when a user chooses to play NLTHP as a guest. The flow of data is similar to that of a regular log in. Upon selecting the guest account, the NLTHP frontend will request authentication from Firebase. After receiving authentication, the guest account presented to the user and the NLTHP dashboard is loaded. When using a guest account, user statistics are not recorded.

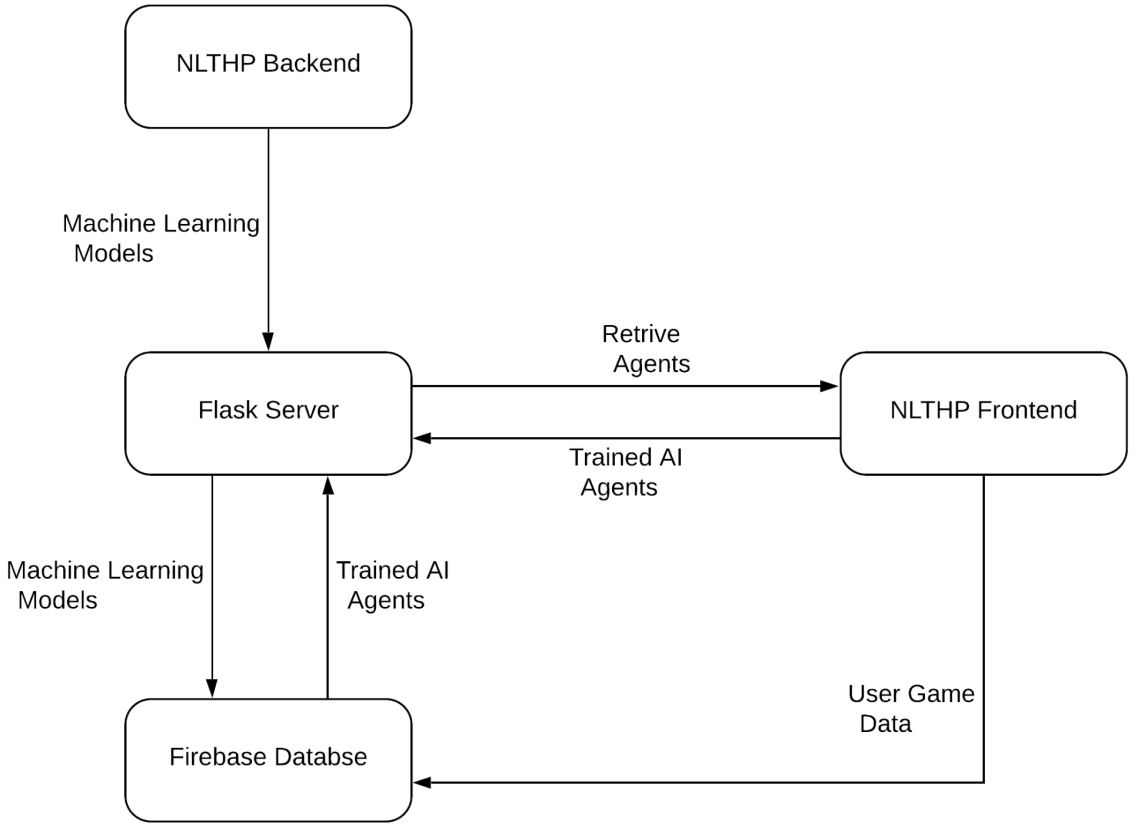
### 3.4. Data Flow Diagram - End Game




End Game Data Flow Diagram

The data flow diagram above depicts the flow of data at the end of a game of NLTHP. The NLTHP Frontend records user game data and user statistics. User game data consists of decisions the user makes during the game and the game's difficulty level. These decisions define the play style of the user. User game data is used to retrain AI agents that match the game difficulty of the user game data. The aim is for AI agents to adapt to the user's play style. User statistics include information about how many games the user has played at each difficulty level and how many games they have won. Both user game data and user statistics are sent to the app's database.

## 3.5. Data Flow Diagram - Backend Processes




Backend Data Flow Diagram

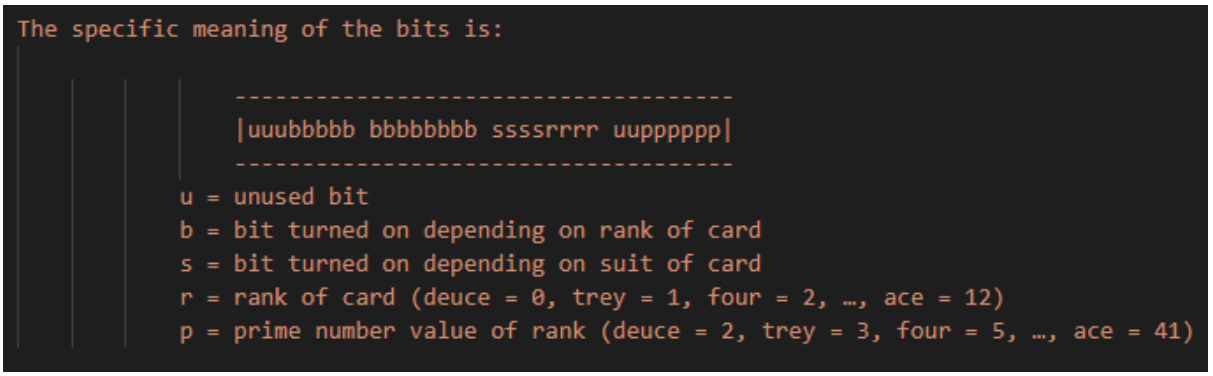
This data flow diagram describes the flow of data during processes that occur within the backend of the NLTHP application. Four machine learning models are created by the NLTHP backend and stored on the app's database. Each of these four machine learning models are for one of the difficulty levels in the NLTHP app. AI agents are trained for each difficulty level and also stored in the app's database. When a user starts a game of NLTHP, the app loads all its resources to run a game. One of these resources is the trained AI agents. A request is sent from the NLTHP to retrieve the AI agents. The AI agents are retrieved and sent to the NLTHP frontend via the Flask Server. At the end of each game of NLTHP, user game data is sent to the app's database. AI agents of matching difficulty to the user game data are retrained. The retrained AI agents are stored in the database in place of the original AI agents for that difficulty.

# 4. Problems Solved

## 4.1. Improving Hand Evaluator Performance

One of the main challenges of this project was to create a poker hand evaluation library, which had to be lightweight and fast in order to allow large volumes of training for the AI Agents. We needed a library that could handle 5, 6 and 7 card hand lookups by basically calculating the percentage of win given 2 private cards and either 3 (post flop), 4 (post turn), or 5 (post river) community cards. Based on our research, we had decided that a lookup table was the best approach to adopt. Unfortunately though, we made the mistake to think it was sufficient to store each hand that could be held by opponents given the community cards on the board, which instead ended up not being the case as this resulted in enormous lookup tables which could not be managed by standard machines.

It was clear that such a solution was not satisfactory, thus further research was conducted and a new improved approach was thought of. The idea was based off one of the most popular and performant approaches used for Poker hand evaluation libraries. Through the use of prime numbers evaluations and bitwise arithmetic, we represented cards as 32-bit integers, with each bit representing a specific characteristic of the card such as rank, suit, prime number evaluation, etc.




Hand Evaluator cards bit format

The general idea is based on the power of combinatorics, which from 2,598,960 possible poker hands (52 choose 5) identifies only 7462 distinct hand values that one needs to be concerned with in the hand evaluation. This originates from Cactus Kev's Poker Hand Evaluator which can take advantage of this fact by storing a card's representation in an efficient manner, through some bit manipulation, some multiplications, a few separate lookup tables, allowing to determine a hand's equivalence class value in a very quick and efficient manner.

The key concept here is that each of a card's possible 13 ranks (two through ace) is stored as a different prime number, and by multiplying a hand's five prime numbers together, we get a unique result that can then be used to determine that hand's overall value. Once certain

special cases are handled, it is simply a matter of performing binary search in the obtained tables.

This approach was inspired from https://elasticdog.com/2010/11/porting-a-poker-hand-evaluator-from-c-to-factor/ which used this approach to create a poker hand evaluator in Factor.

```python
def evaluate_seven_cards(self, cards):
    """
    Performs a hand evalution given a hand of 7 cards (after river), mapping them to
    a rank in the range [1, 7462], with lower ranks being better evaluations.
    """
    minimum = LookupTable.possible_high_card

    # generate combinations of five card hands
    five_card_combos = itertools.combinations(cards, 5)

    # calculate what best 5 card hand is (discards two cards)
    for combo in five_card_combos:
        score = self.evaluate_five_cards(combo)
        if score < minimum:
            minimum = score

    return minimum
```

Python 7-hand evaluation function

## 4.2. Improving Model Accuracy

The development of competitive artificial Poker agents is a challenge to Artificial Intelligence because the agents must deal with unreliable information and deception which make it essential to model the opponents to achieve good results. Creating an accurate and performing model was essential to this project as, if the agents could not compete against real life human players, the whole application would lose purpose. Thus, the selection of the model for our AI Agents was a very delicate process as it would significantly impact the entire outcome of the project.

After experimenting with various machine learning models, more successful and accurate results were obtained with Linear and Ensemble models. The main ones focused on were Linear Regression, Random Forest Regression and finally, the chosen model which was a Gradient Boosting Regression model.

The general idea is a machine learning Gradient Boosting Regressor is used to estimate a function from the set of selected and stored features to the set of saved labels. In order to predict the best action, the Agent executes this function according to the state of the game and the entire set of possible actions. The one that is estimated to be the maximum expected return value is picked. The Artificial Intelligence Agents are this way trained repeatedly with the final objective of generating interesting and viable poker winning strategies.

We performed cross validation of the Gradient Boosting Regression model and obtained an $R^2$ value of 0.468609, which was an outstanding result for us and which indicated that the features selected were quite well fitted for our problem and the model explains a sufficient variability of the response data around its mean.

```
# create Agent players
for i in range(NUM_AGENT_PLAYERS):

    # create Agent that uses GradientBoostingRegressor to be trained
    name = 'Agent ' + str(i + 1)
    player = Player(name=name, regressor=regressor, chips_amount=10**5,
                    raise_choices=20000, raise_increase=0.5, memory=10**5)
    players.append(player)
```

AI Players creation through Gradient Boosting Regressor (before training)

## 4.3. Keeping AI Agents from improving more than necessary

Having four different levels of difficulties for users of the NLTHP app to play against, one of the concerns was to have AI Agents of a certain level and not have them further improve as this would cause them to perform too well, which is not the desired objective as each level should match its corresponding difficulty. For example, it would be a problem if we have a “Beginner” AI Poker Agent that continues to learn as users make use of our application and stores more and more information, becoming better than an actual “Beginner” Agent. This would not have been an issue if the training of the AI Agents was only performed the one time and then stored in the database, but, as one of the objectives of the project was to create AI Poker Agents which can implement interesting and viable poker winning strategies against humans, our Agent models are updated after each game is played, thus we had to find a way to restrict them from “learning too much”.

The solution adopted was to limit the memory of our AI Agents. Throughout a game, after each iteration, each Agent is trained using a set number of selected features and labels while the remaining features and labels from the beginning of the Agent's “career” are discarded. This allows for each Agent to only remember a specified number of hands played

(which depends on the level of difficulty -> higher difficulty Agents remember more hands), forbidding lower level difficulty AI Agents to improve too much.

```python
def forget_old_strategies(self):
    """
    Discards features/labels older than Player memory (self.memory) and updates features/labels with
    the new updated strategies from end of hand.

    Allows Agents to adapt to human strategies but forbids them of learning too much in order
    to restrict lower level Agents from improving too much.
    """

    for i in range(len(self.labels), len(self.features)):
        self.labels.append(self.stack - self.stacks[i])

    # update features/labels
    self.features = self.features[-self.memory:]
    self.stacks = self.stacks[-self.memory:]
    self.labels = self.labels[-self.memory:]
```

Function for limiting amounts of hands AI Agents “remember”

## 4.4. Integrating Backend Python with Frontend React

As all the Poker logic and methods for storing features and training the AI Agents was implemented in Python, it was necessary to integrate these back-end components of the application with the front-end, in order to create a user-friendly interface and experience.
Before working on the UI, while the main focus was on implementing accurate models for our AI Poker Agents, a CLI version was developed.

```
Hand 1
Agent 4(200) dealt 6c and 6h
Agent 5(200) dealt 9d and 3s
Agent 3(200) dealt Qh and 5c
Agent 1(200) dealt Ks and 7c
Agent 2(200) dealt 6s and Td
User(200) dealt 4d and 9s

Agent 5 posts small blind of 10
Agent 3 posts big blind of 20
Agent 1 calls 20
Agent 2 calls 20
Possible moves:
[('call',), ('fold',)]
```

CLI Poker version

Note: The above version involves a human player but when the AI Agents are being trained for the first time with thousands of hands, the user is excluded from the game.

To integrate the backend implementation with the front-end, we then made use of the Flask web framework that provides a library and tools to perform exactly what we required. Our Flask Server was written in Python using Flask. The Flask Server was set up to allow the React frontend to communicate with the Python backend. The server was set up within a Python virtual environment and once running, any requests not recognised by the frontend are redirected to the backend.

As mentioned, the frontend is written in React. It comprises components that implement NLTHP logic from classes in the NLTHP backend. These components visually represent the objects from the NLTHP backend classes. When a user loads the NLTHP app and starts a game, the NLTHP frontend sends a Get request to the Flask Server. Data from the backend including the trained AI agents are retrieved. This data is then displayed to the user via the User Interface.

## 4.5. Storing User Statistics

As one of the main purposes of the NLTHP application is to allow human users to test and improve their Texas Hold'em Poker skills against AI Agents, we believed it was important to give users a way to store their statistics to keep track of their games and results.

This was implemented by allowing users to create an account on the NLTHP web application via a sign up functionality. Users are required to enter a username and a password to avail of this service and this allows their information to be stored in a cloud database to be retrieved when the user tries to log in with the same credentials, allowing to keep track of the player performances and statistics. Alternatively, if users of the app do not want to record their results, they can simply play games as guest users without creating an account.

The Authentication side of the app was implemented with the use of Google’s Firebase platform. Firebase Authentication allows to build secure authentication systems, while improving the sign-in and onboarding experience for end users. It provides an end-to-end identity solution, supporting email and password accounts and many other identification methods but we restricted our application to email and password, as we believed this satisfied our needs.

Additionally to Firebase Authentication, we also used this platform to store User statistics, allowing each user to keep track of their performances, by knowing their win percentages at each different level of difficulty.

The firebase authentication and statistics recording was implemented through React.

```
/**
 * Initialize Firebase cloud key values (for recording data on cloud DB)
 */
const app = firebase.initializeApp({
    apiKey: process.env.REACT_APP_FIREBASE_API_KEY,
    authDomain: process.env.REACT_APP_FIREBASE_AUTH_DOMAIN,
    databaseURL: process.env.REACT_APP_FIREBASE_DATABASE_URL,
    projectId: process.env.REACT_APP_FIREBASE_PROJECT_ID,
    storageBucket: process.env.REACT_APP_FIREBASE_STORAGE_BUCKET,
    messagingSenderId: process.env.REACT_APP_FIREBASE_MESSAGING_SENDER_ID,
    appId: process.env.REACT_APP_FIREBASE_APP_ID
})

export const auth = app.auth()
export const firebaseDb = app.database().ref()
export default app
```

Firebase cloud key values initialization

```
// Increase games played and/or wins for difficulty played
async function recordUserStatistics(e) {
    var userStatistics = new Object();
    firebaseDb.database().ref().child(currentUser.uid).once("value").then(function (snapshot) {
        var countValues = 0;
        snapshot.forEach(function (childSnapshot) {
            var key = childSnapshot.key;
            var childData = childSnapshot.val();
            userStatistics[key] = childData;
            countValues += 1;
        });
```

```
// Add results to DB
if (difficulty == "beginner") {
    userStatistics["num_beginner_games"] = userStatistics["num_beginner_games"] + 1
    if (props.winner.name == Dashboard.username)
        userStatistics["num_beginner_wins"] = userStatistics["num_beginner_wins"] + 1
}
```

Snippet of code to add User game result to Firebase Cloud DB

# 5. Results

## 5.1. Final Product

Overall, the NLTHP web application successfully performs exactly what it was devised to do, which is to allow Poker players of all levels to train and assess their Texas Hold'em Poker skills against Artificial Intelligent Agents.

The NLTHP app can challenge poker beginners and ultimate Poker players alike, as users are provided with the option to select the level of difficulty at which the AI Agents perform. The Agents will differ in skill due to the different levels of training that will be provided for them depending on the difficulty selected by the user. At higher levels, users will face Agents which have been trained much more intensely compared to Agents that users will face when selecting lower levels. The levels to select from, from easiest to hardest, are: Beginner, Intermediate, Expert, Ultimate Poker Pro.

To the best of the team's knowledge, there are not many free online No-Limit Texas Hold'em Poker web applications that allow players to assess their skills at multiple levels, against multiple Agents. Additionally, numerous applications found during the research only allow for Limit Texas Hold'em Poker, in which bets and raises during the first two rounds of betting (pre-flop and flop) must be equal to the big blind. This approach is often used, as the No-Limit alternative, such as our NLTHP app, significantly increases the strategy space of the game, making it more difficult to create competitive AI Agents.

Furthermore, an additional functionality that the NLTHP web application includes is the possibility to record player statistics, allowing each user to keep track of their performances, by knowing their win percentages at each different level of difficulty. To avail of such a functionality users are required to sign up and create accounts with a username and password which are credentials they will then have to use to log in to allow this performance tracking functionality. Alternatively, if users do not want their results saved, they can simply play games as guests which will not impact their record.

To summarise, the NLTHP app provides competitive Artificial Intelligence Poker Agents which, according to the difficulty selected by users, implement more or less viable Poker winning strategies, allowing users of all skill levels to test and push their Texas Hold'em Poker skills to the limit.

## 5.2. Testing

### 5.2.1. Testing Strategy

For the development of the NLTHP application, the team believed that an agile approach would work well due to the small size of the team and to reduce risks associated with a project of such scale with various unknowns. The approach adopted consisted of designing a sprint plan, with the objective to implement a feature or set of features for each sprint.

Additionally, this contributed towards the selection of our development strategy for the project which was a test-driven development (TDD) approach.
Test Driven Development (TDD) is a software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and, if the test fails, then the new code is written in order to pass the test making code simple and reducing possibilities of bugs.

The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfill them).

This strategy worked for the purposes of new features, but this was only the case regarding our unit testing. As the project was time sensitive, it was not possible to write extensive integration tests before developing each new feature, thus integration tests were written in the later stages of the development, which allowed us to speed up our development process.

We also implemented regression testing through the use of the Gitlab CI facility. Regression testing is generally considered to be a type of testing to confirm that a recent program or code change has not adversely affected existing features. Continuous integration was crucial to this process as it allowed us to use our Gitlab CI pipelines to re-run all our test suites and verify that our new changes were compatible and did not break any functionality that had been previously written and tested.

Finally, some Ad hoc testing was also applied with the attempt to capture unexpected defects or errors that we might have overlooked through our extensive formal testing approaches.

### 5.2.2. Scope of Testing

During the project development, we acknowledged the following areas of the project to be in the scope of testing:

- NLTHP Model - Accuracy Testing
- NLTHP Logic - Unit Testing
- NLTHP UI - Unit Testing
- Non-Functional Testing - Performance and Server Testing
- Integration and Regression Testing with Gitlab CI
- Ad Hoc Testing
- User Testing

### 5.2.3. Types of Testing

#### 5.2.3.1. NLTHP AI Model - Accuracy Testing

When starting the development of the NLTHP app, we knew that one of the most challenging aspects of the project was to guarantee a certain level of accuracy of our prediction model. The goal was to create models which would allow the Artificial Intelligence Agents to successfully predict the maximum expected return value on each different game state throughout a game (according to the level of difficulty).

The reason for which the accuracy level of our model was so important was due to the fact that accuracy is one of the key metrics for evaluating prediction models and depending on the level reached, we would be able to classify our NLTHP app as a success or a failure. For this reason, when creating our models, we were very careful and detailed when it came to analysing and selecting the features to make up the models, as they would make a significant impact in the final performance.

Although, the accuracy was a key factor, since the data used by the trained Agents varies continuously after each hand played or simulated, it was not possible to simply obtain a specific accuracy value for each model. Alternatively, we performed a cross validation of the “Ultimate Poker Pro" model, the highest trained and performing model. Cross validation is a model validation technique used for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice.

We performed a simulation of 20000 hands to cross validate the “Ultimate Poker Pro” model and obtained an $R^2$ value of 0.4686, which confirmed a satisfactory matching of the model from our selected features and data.

```
Cross-validating GradientBoostingRegressor...
R-squared (R^2): 0.4686094662377023
```

Cross validation of “Ultimate Poker Pro” model

```
#simulate 20000 hands
simulate(pokerTable, num_hands=20000, hands_between_buyin=100, hands_between_training=0, narrate_hands=False)

features = []
labels = []

for p in players:
    features.extend(p.get_features())
    labels.extend(p.get_labels())

features = np.array(features)
labels = np.array(labels)

#shuffle features/labels
index = np.arange(len(labels))
np.random.shuffle(index)
features = features[index]
labels = labels[index]

#initialize regressors with default parameters
regressors = {LinearRegression(): 'LinearRegression',
              Ridge(): 'Ridge',
              Lasso(): 'Lasso',
              RandomForestRegressor(): 'RandomForestRegressor',
              GradientBoostingRegressor(): 'GradientBoostingRegressor'}

for regressor in regressors:
    print('Cross-validating ' + regressors[regressor] + '...')
    print('R-squared(R^2):', np.mean(cross_val_score(regressor, features, labels)))
    print()
```

Model cross validation code snippet

Above a snippet of code used for the analysis of the AI models is shown.
It is important to note that this result is obtained according to the data that was part of the 20000 simulated hands, thus, when repeated we found that there was a variation that ranged between 0.37 and 0.49.

Additionally, the models were also verified by allowing them to compete against each other over 20000 hands too. The results can be seen in Section 1.3. As expected and desired, Agents that have been trained over a larger number of hands tend to be more skilled and use more winning Poker strategies.

#### 5.2.3.2. NLTHP Logic - Unit Testing

Regarding the testing of the NLTHP backend Python code which we used for the Poker game logic, the features and labels generation and storage, and the models creation and training, we implemented unit tests for each basic step that the script involved, testing each functionality separately and together.

To do so, we made use of Python's PyUnit unit testing framework which allowed us to include test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

These unit tests ensured that the core NLTHP functionalities persisted and were run as part of the continuous integration of code into the master branch.

```
class TestCard(unittest.TestCase):
    ''' Class for running unittests on functionalities of card.py '''

    def setUp(self):
        ''' SetUp Card object and a deck of Cards '''
        card = Card(10, 'c')
        self.card = card


        self.deck = []
        for suit in ['c', 'd', 'h', 's']:
            for value in range(2,15):
                self.deck.append(Card(value,suit))

    def test_deck_of_cards(self):
        ''' Test that a deck made up of Card objects is constructed as expected'''
        otherDeck = []
        for suit in ['c', 'd', 'h', 's']:
            for value in range(2,15):
                otherDeck.append(Card(value,suit))

        # check card values and suits
        for i in range(len(otherDeck)):
            self.assertEqual(self.deck[i]._card_num, otherDeck[i]._card_num)
            self.assertEqual(self.deck[i]._suit, otherDeck[i]._suit)
```

Card class PyUnit test code snippet

Above is a snippet from the PyUnit tests run to verify the Card class. It checks that when a deck is generated, each Card corresponds to the expected Card in the deck.

All these tests regarding the NLTHP backend Python code can be run using the "-*python -m unittest"* command in the src/test/ directory of the project.

#### 5.2.3.3. NLTHP UI - Unit Testing

With regards to the testing of the functionalities of the User Interface (UI) which were implemented in React, we made use of the React Testing Library, which builds on top of DOM Testing Library by adding APIs for working with React components.

It is a very light-weight solution for testing React components. It provides light utility functions on top of react-dom and react-dom/test-utils, in a way that encourages better testing practices.

So rather than dealing with instances of rendered React components, it allowed for our tests to work with actual DOM nodes. The utilities this library provides allowed us to query the DOM in the same way users would. Finding form elements by their label text, finding links and buttons from their text, etc.. It also exposed a recommended way to find elements by a data-test-id as an "escape hatch" for elements where the text content and label do not make sense or is not practical.

This library assisted us in making our applications more accessible and allowed us to test our components in a similar way users would make use of them, which allowed our tests to give us more confidence that your application would work as expected when real users used it.

Functionality and Components we verified through this type of testing were Game components, Signup components, Dashboard components, Statistics components and others we found were crucial for guaranteeing a smooth user experience for the NLTHP app.

```
/*
Test to verify that the Signup
component renders correctly.
*/
test('render user signup', () => {
    const component = renderer.create(
        <testSignup/>
    )
    let tree = component.toJSON()
    expect(tree).toMatchSnapshot()

})

/*
Test to verify that the credentials entered by the user
are consistent with the credentials rendered.
 */
test('match credentials', () => {
    expect(testSignup.emailRef).toBeUndefined()
    expect(testSignup.passwordRef).toBeUndefined()
})
```

Signup component test code snippet

Above is a snippet from the React unit tests run to verify the SignUp component. It checks that the Component renders as expected and additionally, it verifies that the credentials entered by the user are consistent with the credentials rendered.

All these tests regarding the NLTHP UI React code can be run using the "*yarn test"* command in the src/ui/src directory of the project.

#### 5.2.3.4. Non-Functional Testing - Performance and Server Testing

Additionally to testing the functional components of the NLTHP app, we believed it was important to verify non-functional aspects of the app too.

As the frontend is integrated with the backend and there are high amounts of features continuously being varied and stored as the Poker game goes on, we thought it would be important to verify rendering count and rendering times of certain components. The react-performance-testing library provides a solution for these cases as it provides monkey patches with an API that can count the number of renders and measure render time.

It is a library that is perfect for testing a React app's runtime performance.

```
test('render time should be less than 80ms', async () => {
  const Counter = () => {
    const [count, setCount] = React.useState(0);
    return (
      <div>
        <p>{count}</p>
        <button type="button" onClick={() => setCount((c) => c + 1)}>
          Raise
          </button>
      </div>
    );
  };

  const { renderTime } = perf(React);

  render(<Counter />);

  fireEvent.click(screen.getByRole('button', { name: /count/i }));

  await waitFor(() => {
    // 80ms is 300fps
    expect(renderTime.current.Counter.mount).toBeLessThan(80);
    expect(renderTime.current.Counter.updates[0]).toBeLessThan(80);
  });
});
```

Performance test code snippet

Above is a snippet of code that verifies that the Raise action button in the main Game page renders in a satisfactory amount of time (< 80 ms).

All these tests regarding the performance of the NLTHP UI code can be run using the "*yarn test .perf.test.js"* command in the src/ui/src directory of the project.

Furthermore, we also believed it would be useful to verify that a connection was successfully established with the NLTHP app and the server and that server errors were handled adequately with the expected error message.

```
const server = setupServer(
    rest.get('/', (req, res, ctx) => {
        return res(ctx.json({ connection: 'Connected' }))
    })
)

beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

test('loads and displays connection successful', async () => {
    render(<Fetch url="/" />)

    fireEvent.click(screen.getByText('Connection Established Successfully'))

    await waitFor(() => screen.getByRole('heading'))

    expect(screen.getByRole('heading')).toHaveTextContent('Connected')
    expect(screen.getByRole('button')).toHaveAttribute('disabled')
})
```

Server test code snippet

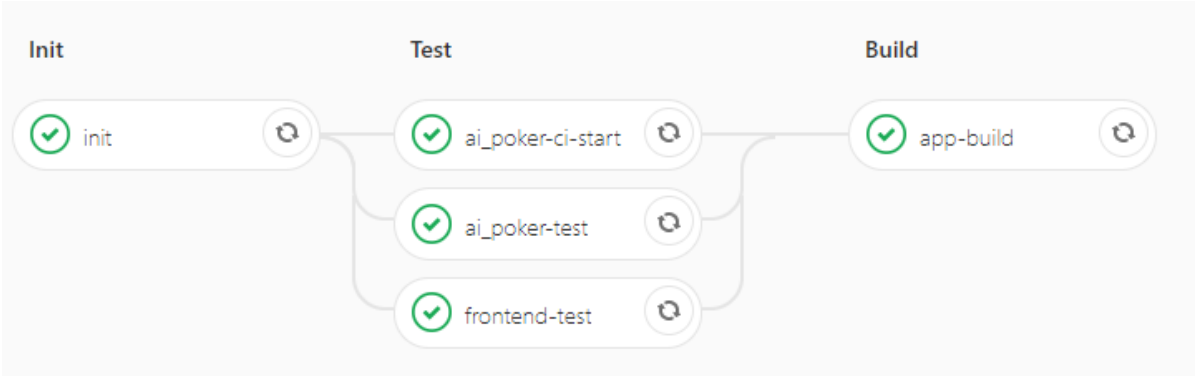
Above is a snippet of code that verifies that a connection is successfully established with the NLTHP web server.

All these tests regarding the performance of the NLTHP UI code can be run using the "*yarn test serverSetUp.test.js"* command in the src/ui/src directory of the project.

#### 5.2.3.5. Integration and Regression Testing with Gitlab CI

As previously mentioned in our testing strategy, the team made significant use of the Gitlab Continuous Integration pipelines to ensure new features introduced would not break any existing functionality

.




CI Pipeline

Our configured CI pipeline is shown above and it has 3 stages: Init, Test, and Build.

The Init stage sets up the nodeJS cache to be used for the frontend testing and NLTHP building.

Following, in the Test stage, all backend and frontend test jobs are run such as the NLTHP backend logic tests, the NLTHP user interface tests, along with the non-functional performance and server tests.

Finally, if all previous jobs are successful, a build of the NLTHP web app occurs in the Build stage.

Thus, the Regression testing covers the following areas of testing discussed above:

- NLTHP Logic - Unit Testing
- NLTHP UI - Unit Testing
- Non-Functional Testing - Performance and Server Testing

This ensured that changes to all of the above components were integrated as production-ready when they passed out pre-written tests.

```
app-build:
  stage: build
  image: node:10.15.3
  script:
    - cd ../ui/src
    - echo "Start building App..."
    - yarn install
    - yarn build
    - echo "Build successful!"
```

app-build job snippet

Above is a snippet of our *app-build* job which is executed in the Build stage.

All the jobs of the CI can be viewed in the *.gitlab-ci.yml* file in the root directory of the project repository.

The pipeline status of the NLTHP project may be viewed here:
https://gitlab.computing.dcu.ie/idelegi2/2021-ca400-idelegi2-puzzuos2/badges/master/pipeline.svg

#### 5.2.3.6. Ad Hoc Testing

As mentioned in Section 5.2.1 when discussing our testing strategy, some Ad Hoc testing was also applied. Ad hoc testing is an informal or unstructured software testing type that aims to break the testing process in order to find some possible defects or errors. Ad hoc testing is done randomly and it is usually an unplanned activity that does not follow any documentation and test design techniques to create test cases. This testing approach was applied through the testing technique called Error Guessing, which consists of attempting to "guess" the most likely source of errors, generally based on experience and knowledge.

This approach of testing consisted of both team members attempting to break the system in unconventional ways. Its purpose was to try to capture unexpected user behaviours that might have caused issues in the NLTHP app.

We believed that after a satisfactory amount of formal testing that this approach could help identify any areas we might have overlooked.

#### 5.2.3.7. User Testing

Finally, we decided to also conduct some user testing through user evaluations.

This user evaluation stage allowed us to classify the user Experience of our NLTHP app. By user experience, we refer to the aspect of the user interaction with the product, and more specifically how a user perceives the system before, during and after interacting with it. Since user experiences are subjective to the specific individuals, it was necessary that we conducted evaluations with the right extent, constructs, and methods to enable the data and information received from these studies to assist us in the dimension we required.

To evaluate our system, we decided that a heuristic evaluation and a user evaluation of our App through focus groups was sufficient to provide us with the information that we were looking for regarding the user experience.

By answering adopting such a method, we were able to get a general collection of opinions and thoughts about the App, highlighting areas in which improvements were required and areas which we had designed adequately which users were satisfied with.

Official documentation about our user studies is present in our repository in docs/ directory (e.g. Focus Group Questions).

# 6. Future Work

This section entails features that would be implemented into the NLTHP application if it was to be developed further. Although the NLTHP has been developed to a satisfactory level, there are various features that if added could improve user experience as well as the efficiency of the app.

## 6.1. Mobile Application

The NLTHP application is a web application that can be accessed via a web browser. This does not exclude mobile smartphones. The NLTHP app is also accessible and playable on any smartphone that has access to a browser. However, the nature of which the app is developed is tailored to desktops. A fully-fledged mobile application could greatly improve mobile user's experience. This application would have features tailored to smartphones such as notifications and offline play. The development of a mobile application would also eliminate the need for mobile users to access NLTHP through a browser. As the front end of the application is developed in React, the existing application could be adapted into a React Native environment. This would make it possible to develop mobile applications for both IOS and Android devices.

## 6.2. User Community

The development of a user community would greatly improve user experience. A feature to allow users to log in to the NLTHP app via various social media accounts could be integrated into the existing application. Early development of this feature could allow users to add friends in the NLTHP app. Users would then be able to view and compare statistics with their NLTHP friends while using the application. Further development of this feature could lead to the development of a feature that would enable users to play NLTHP online together.

## 6.3. Online PvP/Multiplayer

This would be the most exciting new feature that could be added to the NLTHP application. The development of this feature would allow users to play NLTHP against other users. For this feature, the NLTHP app would need a dedicated server that users could connect to while playing NLTHP. Users need to be connected to the same server so that changes in the game register the same for each user. With PvP (player vs player) multiplayer, tournaments could be organised to create a more exciting player experience. With this feature, users could create custom lobbies to play with their NLTHP friends, or enter public lobbies and play against other NLTHP users.

## 6.4. In-Game Currency/Real Money

The desire to add this feature is undecided as a feature like this could cause complications. It would make sense for an application like NLTHP to have some sort of in-game currency. As of the current build, users are assigned chips when they enter a NLTHP table. However, if the aforementioned features are implemented, the NLTHP application would evolve into an app that is driven by its user community. Users could have in-game cash assigned to their accounts which they would use to buy chips and play NLTHP. Chips won or lost in games would be converted to cash then added/deducted from user accounts. Users could also win prizes from tournaments such as cash. The issue arises when a user loses all their cash. A feature to allow users to purchase cash using real-life money could be added, however, this could cause ethical issues and introduce an element of gambling. A simple solution would be to automatically add cash to a user's account once their balance reaches zero.