March Madness Predictor

CS 428 Documentation

Adam Hebein
Brendan Caffrey
Dan Schweikert
John McConnell
Max Braun
Siddharth Shukramani

Table of Contents

I. Project Description

Description: The March Madness Predictor is a web application and statistics engine that predicts the winners of NCAA Mens Division 1 basketball games. The application has the ability to predict head to head match ups or an entire bracket for the current March Madness tournament. It also has the ability to calculate and display an overall team ranking for every team. The prediction engine utilizes two C++ machine learning algorithms, C5 (a decision tree) and Fast Artificial Neural Network (a neural network). The overall prediction routine returns the winner of a matchup by selecting the result of the algorithm which has the greatest confidence for that matchup. The algorithms use statistics from the previous season, which are scraped from ESPN's website. The scraper is part of a Ruby backend which populates a database with all the team's average statistics and the statistics for each game played during the season.

Motivation: Filling out March Madness brackets has become a popular form of gambling. People of all ages around the US form their own brackets based on their predictions and compete against their friends, co-workers and others online. Even President Obama makes a bracket. Our application will give users an edge over their peers. The goal of this project is to provide additional insight and enjoyment for the fans of the March Madness tournament.

II. Development Process

Process: We are following XP, but have made some slight modifications to the process. Below we will describe our process and our changes. We have divided this section between several of the crucial pillars of XP.

Iterative Development: The CS 428 project was designed to be completed in several 2 week iterations. This type of development schedule is referred to as iterative development. At the start of the project, and at the start of each subsequent iteration, we followed a practice called the planning game. During the planning game we would assign user stories to developers based on their strengths. We would also update our stories and expectations based on the performance of previous iterations. During each two week iteration we attempted to complete the assigned stories. At the end of every two week iteration, we met with Pranav (our TA). Following this pattern we continually added features to our project during each iteration. During our first two iterations we laid the foundation of our project. We did the simplest things first to get our basic components working. This included using mostly dummy data for our algorithms and web application. During the second two iterations we refactored our existing code and refined our components. By the end of these iterations we had a working web scraper, working algorithms,and working database connectors. In the final two iterations we focused on connecting all our components together into one cohesive application. We finished the socket connection and message interface between the frontend and backend. We also polished up our front end GUI. Using an iterative development process helped us to follow

several other XP pillars such as, continuous integration and small releases. The iterative development cycle was extremely productive for us.

Refactoring:

Refactoring was a key part of our development process. We refactored portions of our code every iteration, to make it easier to understand and easier to maintain. An example of refactoring can be seen in the ESPN scraper code. Initially, we coded the scraper in a fast and messy way, that quickly got the data we needed. We did this because in XP it is always recommended to do the simplest thing first. This also allowed us to quickly obtain real data to use with the machine learning algorithms. However, the code was incredibly hard to read, and as we learned later, hard to maintain when ESPN changed their page structure. To solve these issues we refactored. We extracted the basic functionality into smaller functions and stored the statistics in a smarter way in the database (it was the same data!). This made maintenance much easier. This example illustrates how we used the XP pillars of simplicity and refactoring. We used a similar thought process for all components of our system.

Automated Testing:

Automated testing ties in very closely to refactoring and continuous integration. Having many automated tests allowed us to catch bugs quickly and speed up refactoring. Also, when integrating different parts of our system (like the web app and back end algorithms) we were able to quickly tell where errors were coming from. We attempted to have automated tests for every one of our functions with complete code coverage. This included having multiple automated tests for each user story. Every iteration we increased the number of tests in our project. For C++ we used Gtest, an automated testing framework created by Google. For Ruby we used rspec, an automated testing framework for ruby. We also used some advanced testing, such as cucumber, to test our GUI.

Collaborative Development:

Our teams version of collaborative development was slightly different from the standard XP version. XP calls for teams to do pair programming. It had been hard for us to find time to schedule multiple pair programming sessions each week in CS 427, so we decided to change this part of the process. For CS 428, we decided to code individually and enforce peer reviews before submitting code. This ensured that at least two people on the team were familiar with each part of code. It also helped check for errors without wasting time. We feel that we learned more by having our code reviewed than by pair programming. Also, during almost every iteration we would have a team coding session. This put the whole team together (another aspect of XP) and could be considered the ultimate form of pair programming. We believe this increased collective code ownership and promoted team chemistry.

Coding Standards:

A commonly forgotten XP pillar is coding standards. We chose industry standards to follow before the project began and followed them. This helped the readability of our code and helped promote collective code ownership. We only applied the standards to new code written by us during the project and not to existing libraries used. The standards are as following:

- Ruby: http://www.caliban.org/ruby/rubyguide.shtml
- C++: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
- HTML / CSS: http://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml

III. Requirements & Specifications

Iteration 1 User Stories:

- write a proposal
- form the team
- compose a team contract
- discuss project and user stories
- spike code

Iteration 2 User Stories:

- connect to ESPN with API and grab data
- research an algorithm to derive an overall score for each team in a matchup
- web server connects to database
- create a basic head to head page on webapp

Iteration 3 User Stories:

- select a means to get data for database
- get teams game results from ESPN
- set up one or more machine learning algorithms with sample data
- clean up the front end design with twitter bootstrap and add basic integration tests
- use the database for the algorithm's training set

Iteration 4 User Stories:

- finish collecting game results from ESPN
- collect team statistics from ESPN
- parse real data from the database into algorithm format
- test algorithms by running every possible match-up with real data and printing the results
- set up initial front-end to back-end connection

Iteration 5 User Stories:

- scrape for more stats we deem influential
- use sleep and user agent manipulation to prevent 503 errors
- allow front-end user to submit a match-up to the back-end and see the result
- refine algorithm
- create page for full bracket
- create page for overall team rank

Iteration 6 User Stories:

- clean up database
- finish implementing bracket page on backend & frontend
- finish implementing team rank page on backend & frontend

- set up persistent hosting of app
- polish / refactor

IV. Architecture & Design

Use-Case Diagrams:

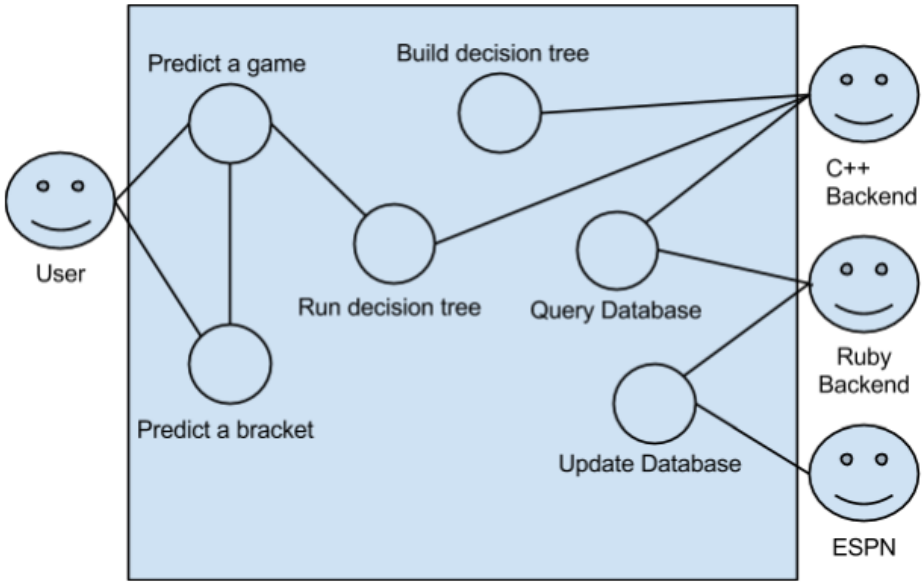

Figure 1: Use-Case Diagram

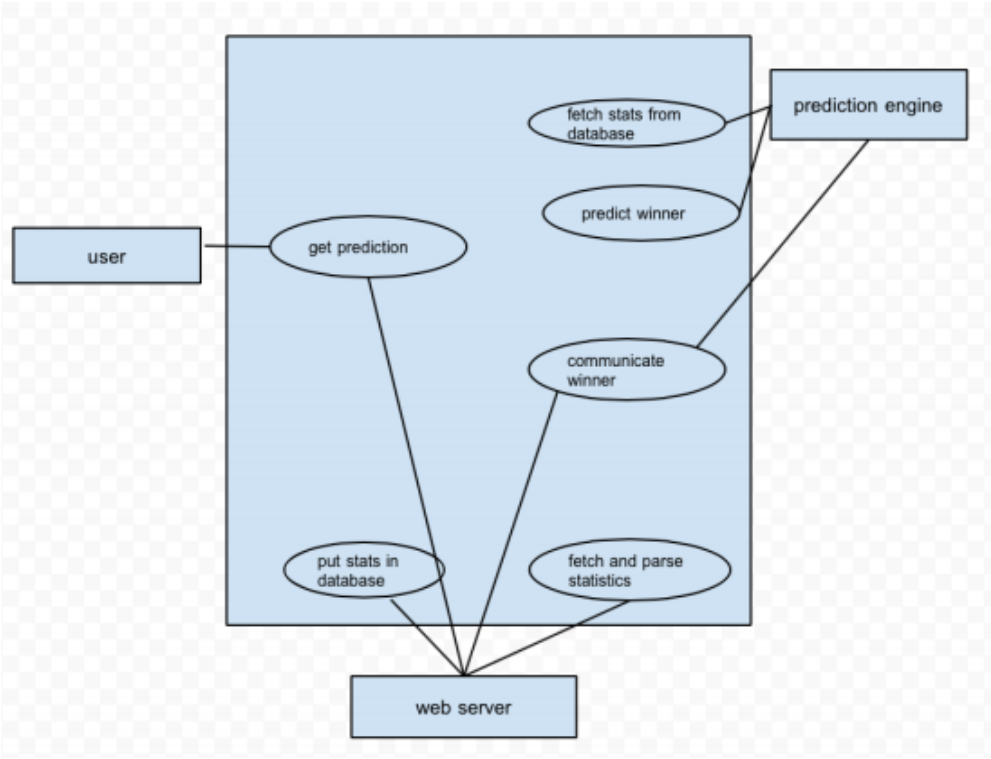

Figure 2: Use-Case Diagram

Class Diagrams:

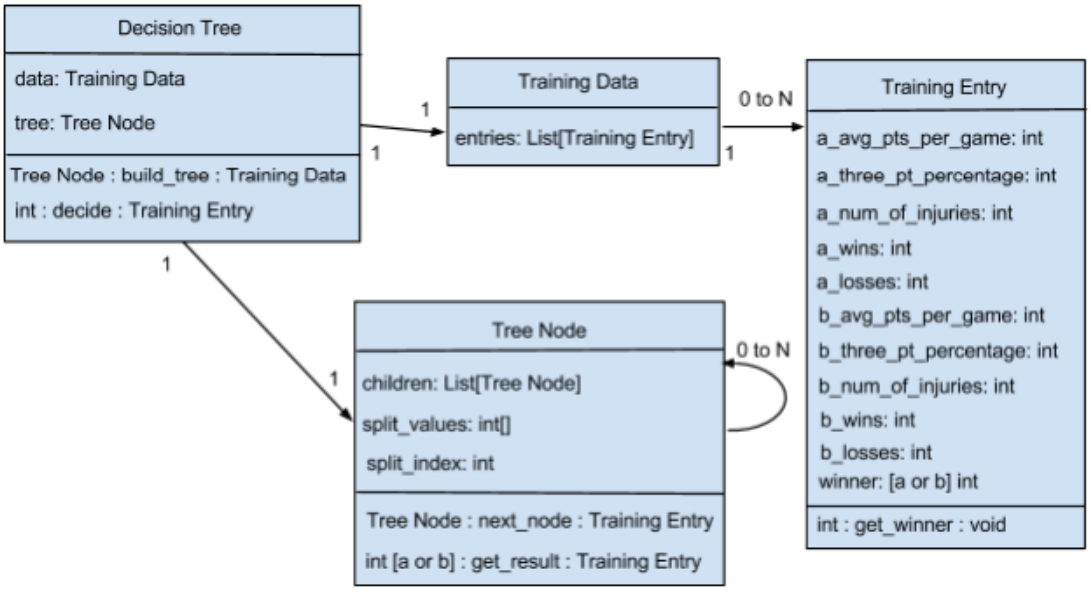

Figure 3: Class Diagram for Decision Tree

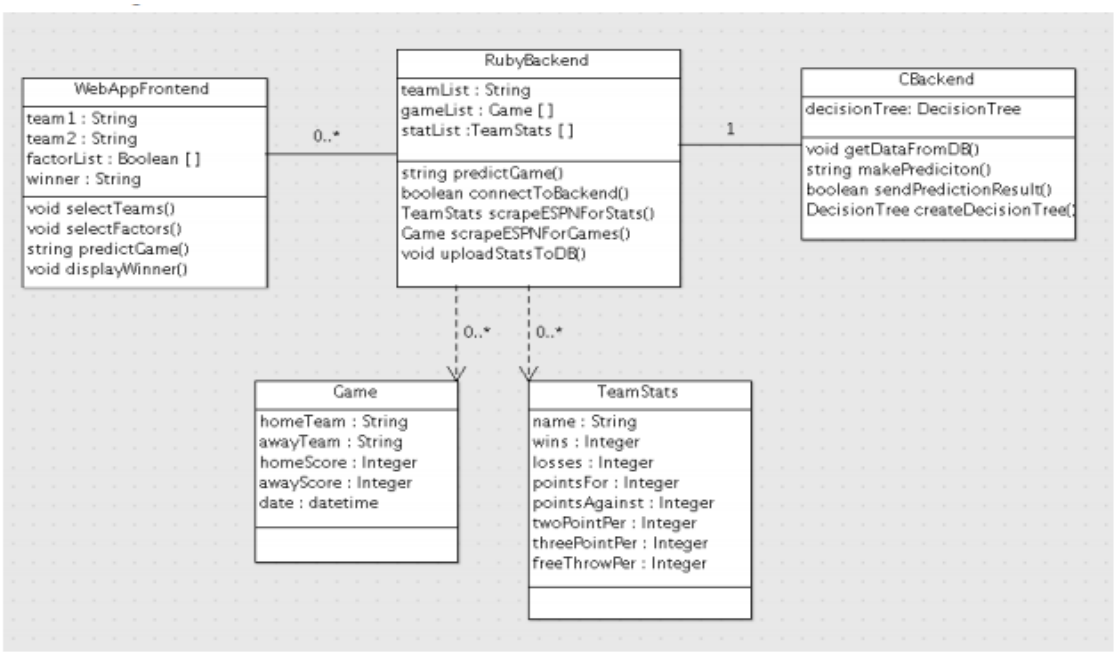

Figure 4: Class Diagram

Sequence Diagrams:

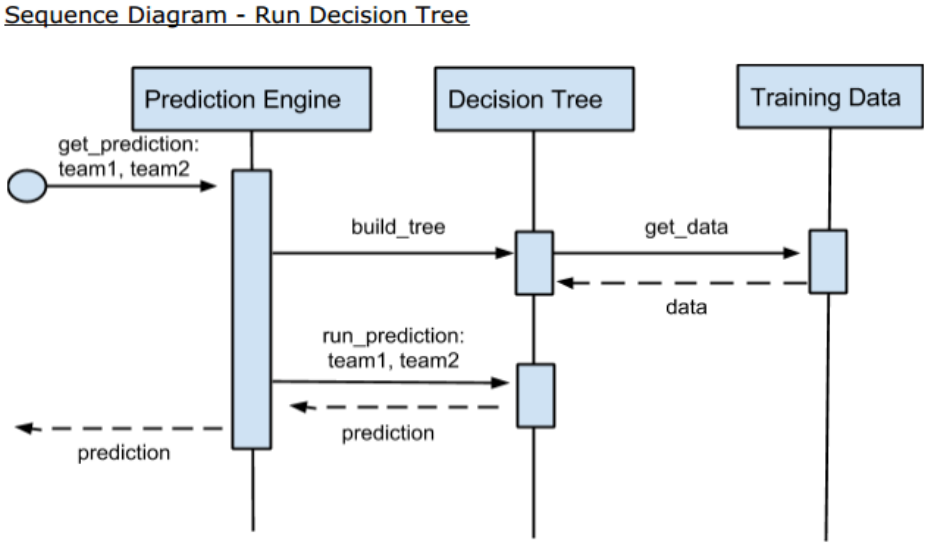

Figure 5: Sequence Diagram

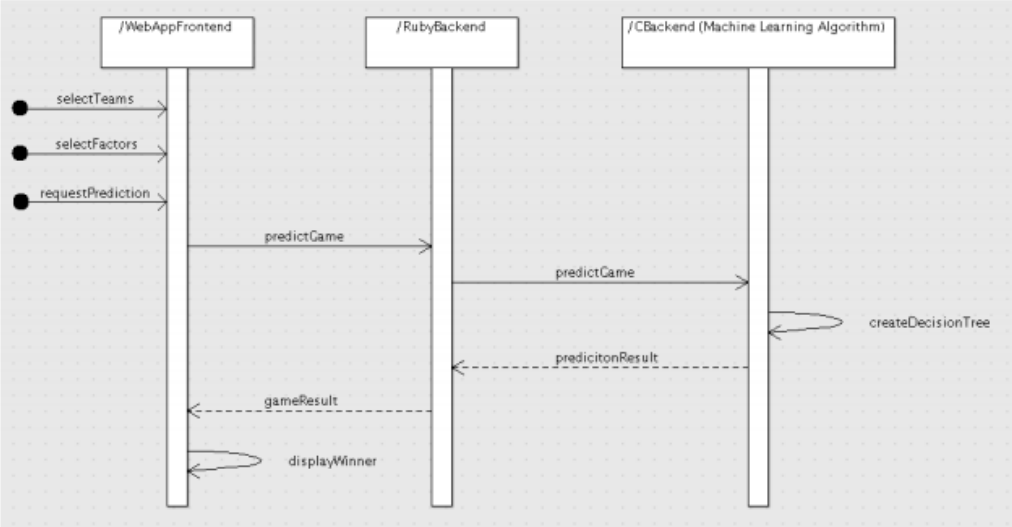

Figure 6: Sequence Diagram

Architecture:

ESPN Scraper: The ESPN Scraper is run in a rake job called lib/tasks/espn.rake. This rake job calls functions from app/helpers/espn_scraper_helper.rb. These functions loop through all the stats that need to be made and creates the given entries in the database. We scrape conferences first, then each team from each given conference, then each of the specified statistics for each given team. After that we create an entry for each game by looping through each teams game and identifying the game by its ESPN ID. These functions are tested by spec/tasks/espnScraper_spec using rspec. These tests make sure the fields are not null and percentages are in the correct range (0-1) since we do not know what the stats actual values are (constantly changing).

Database Schema:

- Conference: name, web extension, logo
- Team: name, conference id, rank, web extension, logo, conference wins, conference losses, wins, losses, home wins, home losses, away wins, away losses, offensive points per game, defensive points per game, field goals made per game, field goals attempted per game, field goal percentage, two pointers made per game, two pointers attempted per game, two point percentage, three pointers made per game, three pointers attempted per game, three point percentage, free throws made per game, free throws attempted per game, free throw percentage, offensive rebounds per game, defensive rebounds per game, total rebounds per game, points per shot, adjusted field goal percentage, assists per game, turnovers per game, assists per turnover ratio, steals per game, fouls per game, steals per turnover, steals per foul, blocks per game, blocks per foul
- Game: ESPN game id, date, home team, away team, home overall win percentage, home home win percentage, home points, home field goals made, home field goals attempted, home field goal percentage, home two pointers made, home two pointers attempted, home two pointer percentage, home three pointers made, home three pointers attempted, home three pointer percentage, home free throws made, home free throws attempted, home free throw percentage, home offensive rebounds, home defensive rebounds, home total rebounds, home points per shot, home adjusted field goal percentage, home assists, home turnovers, home assist per turnover ratio, home steals,

home fouls, home steals per turnover, home steals per foul, home blocks, home blocks per foul, away overall win percentage, away home win percentage, away points, away field goals made, away field goals attempted, away field goal percentage, away two pointers made, away two pointers attempted, away two pointer percentage, away three pointers made, away three pointers attempted, away three pointer percentage, away free throws made, away free throws attempted, away free throw percentage, away offensive rebounds, away defensive rebounds, away total rebounds, away points per shot, away adjusted field goal percentage, away assists, away turnovers, away assist per turnover ratio, away steals, away fouls, away steals per turnover, away steals per foul, away blocks, away blocks per foul

Web Application: The web application use the ruby on rails and sqlite stack. It host the gui and visual aspect of our application to the user. For example when you visit the website, you will be displayed a form which will have match up options for the teams you like to matchup. When you click the form the web application receives a ajax request and submits a query to the prediction engine. The prediction engine responds to the query and the web application forwards the result to the web page and displays the winner. All of this takes place in a couple of milliseconds. There is a separate section of the webpage that can run an entire bracket by making recursive calls to each individual match and finishing once only a single team is the winner. The web application was designed to be scalable and allow new functionality and queries to be sent to the prediction engine and new webpages to be added.

Database Connector: The database connector (found in dbc.cpp in the root directory of the march madness backend git repository) is responsible for translating what is in the sqlite database file into a format that the C5 and ANN algorithms can interpret. The functions performed by the database connector can be grouped into two categories: those responsible for reading information into memory from the database (`std::list<Game> retrieveGames(), std::list<Team> retrieveTeams()`)and those responsible for writing data from memory into formats readable by the C5 and ANN algorithms (`void writeC5(std::string filename), void writeANN(std::string filename), void writeC5Teams(std::string filename), void writeANNTeams(std::string filename)`). In most cases users simply make calls to the writing data functions, and they in turn make calls to the retrieval functions.

Decision Tree: The code for the C5 decision tree algorithm is in the c5_mmp folder. Inside this folder is another folder called c5_source. The contents of c5_source are fairly self explanatory. After these files are built an executable called c5.0 is produced. c5.0 is a program which can be run from the command line with various options. Running the program builds a decision tree and tests it on the supplied training data. The command to run it is `./c5.0 -f <file specifier>`. The file specifier is the filename of all the data files needed to construct and run the tree. The files needed are <file specifier>.names (defines the decision tree factors) and <file specifier>.data (lists the training data). An optional <file specifier>.cases file can be created which contains new data to run on an existing tree. After the program is run a <file specifier>.tree file is generated which can be reused after that point. After learning how to use this program it was necessary to create our own interface to run this program from within our application. The C5 webpage provided example code to run an existing tree on a .cases file. Using this example code we created our own file called mmp_c5_functions.cpp. In this file we defined three functions which would be used to link the C5 functionality into our own program. The first function `void buildTree(bool boost, std::string path_to_c5_mmp)` runs the c5.0 executable and produces a .tree file. The first argument boost can be used to build a boosted tree and the second argument is required to tell the program where the c5_mmp

folder is so it can run the executable. The next function is `std::string runMatchup(std::string T1, std::string T2, std::string path_to_c5_mmp)` which runs the existing decision tree on two teams T1 and T2. It also requires the path to the c5_mmp folder. This function utilizes a special file we created called <file specifier>.avgs which contains all the team average data. With this data we can construct a data set with the two team names to run on the decision tree. The final function is `std::vector<std::pair<std::string, int> > runAllMatchups(std::string path_to_c5_mmp)`. This function is similar to running a single match up except it instead runs all possible matchups between all possible teams. It returns a vector of pairs containing each team name and how many games they would win out of all their possible matchups. After writing these functions it was easy to integrate the algorithm into our general prediction routine.

Neural Network: The code for the ANN algorithm is in the neuralNetwork folder. The NCAA_ANN_src folder contains the code written for the project. The rest of the files are provided for the installation of FANN library. Inside the NCAA_ANN_src folder, there are files named ncaa_train (.h and .cpp) as well ncaa_test (.h and .cpp). The ncaa_train files have functions for training the data set. The `struct fann * createNetwork(unsigned int num_layers, unsigned int num_input, unsigned int num_neurons_hidden, unsigned int num_output)` function creates an untrained network with the given neurons in the given number of layers and the `int train_network()` function trains the network with the data given in ncaa.data file. The ncaa.data file starts with a line that specifies the number of data values given for training, the number of inputs and the number of outputs. After that, each data value takes 2 lines, the first one contains all the statistics for both teams separated by space and the second one has a 1 if the first team won and 0, otherwise. There is also a teams.txt file that contains the name of each team followed by it's average statistics. The ncaa_test files have functions that use the trained network to predict a matchup, rank teams as well as check accuracy of the network. The `string getPrediction(string team1, string team2)` function runs the prediction algorithm on the given teams and returns the name of the winner. The `void parse_teams(char * fileName)` takes in teams.txt as the filename and parses the average statistics for the ranking algorithm, which is then calculated by the `void calc_rank()` function. Lastly, the `int test_network()` function calculates the accuracy of the network by using cross validation, that is 20% of the size of ncaa.data stored in ncaaTest.data file (same format as ncaa.data). Also, the folder contains a file called annTest.cpp. This file uses the gtest framework to test the training and the testing of the network. The test can be ran by using the command `./ann_test` after running make.

Sockets / Message Interface: The web application and prediction engine communicate through a socket connection. The web application accepts a client connection from the C++ prediction engine and then they maintain the connection for the rest of the communication process. In when the server or client send bytes of information to each other they are scanned for the a "new message" character separator. When that occurs the programs records the message into a queue for future use. Example api looks like, "`void send(string message);`" which adds a message character separator to the string and sends it across the socket and "`string * getMessage();`" which checks for new bytes on the socket and scans for the character separator and adds the separated string to the queue and then returns the top element of the queue. Another layer of abstraction is broken up into the MessageInterface layer, which takes these messages and parses them for team names. Example API looks like, "`void sendRanking(vector<string> ranking);`" which takes a vector of strings a composes a message to send to the other program and "`vector<string> getTeams();`" which pops the top message off the queue and parses its content for a list of team names.

Prediction Process: Once the backend receives a message from the sockets/message interface the prediction process goes through a multi-step process in order to generate an accurate prediction. If it is the first time that the predictor is being called, the database will be read in from our sqlite file and then written out in an appropriate format for both the C5 and ANN algorithms. At this point both algorithms will be trained with the data we just read (again this is only true if this is the first time the predictor is being called). For every request depending on whether a ranking or single matchup is being requested, the algorithm will be run and then return a message either saying which team wins in a matchup or a complete ranking of teams. Note that the code by default selects to use the C5 algorithm but the ANN can be easily substituted.

Frameworks Used:

Rspec: We used rspec to help create unit tests for the ruby and rails web application. It uses has built in helper functions to make unit tests more readable and help provide feedback. For example, code like "`expect(my_var) to_equal(my_var2)`" is common. The tests also provide feedback on the runtime of the functions.

Cucumber: Cucumber is our integration testing framework. It is really powerful with its ability to write integration test in plain human readable language. Then using the framework you can use regular expressions to match the human readable code to actual ruby tests. Another great benefit is the ability to define input and output using the human readable code parse it into the ruby code and use it to check input and and output. This allows the integration tests to scale very well.

Google test: We decided to use gtest as our testing framework for our C++ prediction engine. It allowed us to create unit tests that could be compiled and run at once. Accordingly, it uses build expected and actual value help statements so that we would not have to write them ourselves.

C5 decision tree: We decided to use existing machine learning algorithms instead of writing our own from scratch. This allowed us to quickly produce a more powerful prediction engine than what we would have been able to produce in a semester by ourselves. We had to write our own interface to connect the C5 decision tree algorithm into the rest of our application.

FANN: This library, which also stands for Fast Artificial Neural Network, was used to create the neural network. It provided us with easy to use training as well as testing functions that made it easy for us to plugin our statistics for prediction. It also had a feature that allowed us to save the neural network generated into a .net file that could be used later when predicting a matchup between two college teams.

Installation & Distribution:

Rails Web Application

- Clone the repository (`git clone git@github.com:brendanjcaffrey/march-madness.git`)
- Run `bundle install` to install dependencies (and the correct version of Rails)
- Run `rake db:migrate` to create the needed database tables
- Run `rake espn:scrape` to scrape team stats and game results from ESPN (or just load the database from the email in db/)
- Use `bundle exec rails s` to start the server on port 3000
- This will start a development server that does not attempt to connect to the backend at all

C++ Prediction Engine

- Clone the repository (`git clone git@github.com:hebein1/march-madness-backend.git`)
- For the C5 algorithm:
    - `cd c5_mmp/c5_source/ && make` # builds c5.0 executable
    - `cd .. && make init` # compiles gtest
    - `make` # compiles c5_mmp code and unit tests
    - `./mmp_c5_tests` # run c5_mmp automated test cases
- For the ANN algorithm:
    - `cd neuralNetwork`
    - `cmake .` # installation step 1 for FANN
    - `sudo make install` # installation step 2 for FANN
    - `cd NCAA_ANN_src`
    - `make && make init` # compiles algorithm code, unit tests and gtest
    - `./ann_test` # run annTest automated test cases
- `cd ..` # return to root directory
- `make init` # compiles gtest, you only need to run this once
- `cd build && make` # compiles both the engine and the unit tests
- `./bin/mmetest` to run the tests (start the rails integration test first)
- `make mme`
- `./mme` to run the backend server

V. Future Plans

Risks & Challenges Faced:

Our biggest challenge during the project was getting data from ESPN. At the start of the project, we thought we would be able to use the ESPN API. The ESPN website makes it appear simple to get a key and start using the API. That was not the case. We requested a key from ESPN for weeks, with no luck. We tried email, web requests, stackoverflow, and even Twitter. None of these worked to become a strategic partner. We concluded that ESPN no longer supports their API. Our solution to this issue was to scrape the stats from ESPN. No one on our team had scraped before so we knew it was going to be a learning process. For the first few iterations the scraping went smoothly. However, once we attempted to scrape a large amount of data at once, we ran into 503 errors. This is a "Service Unavailable" error. It means that that ESPN detected that we were scraping and started blocking our requests. To solve this, we introduced User Agent String Spoofing and randomized delays. This prevented the errors, but did bring up ethical issues (we chose to ignore these issues since ESPN was so unresponsive about getting a key). To add more problems to information retrieval, ESPN's website broke for a few weeks. Some of their tables dropped teams, which gave us incomplete data for our database. To solve this we had to find the data in other places on the website. Getting data was challenging, but very rewarding, since in the end we had the freedom and knowledge to get any stat we wanted. When we started our project we thought that most of our risks would have to do with real time threading, real time stat updating, and security risks. These risks turned out to be less crucial than we thought and some were ignored completely. We solved the real time stat updating issue by fetching stats once a day after all the games for that day were over. This allowed for complete game stats in the database and prevented errors while updating the database. The other issues were largely ignored because functionality became our primary concern as

opposed to performance and security. Another issue was that many of us were using Ruby for the first time. This required some learning curve during the project, but wasn't too hard of a challenge.

Future Plans:

This semester we created a working algorithm, but it isn't the most effective. Statisticians have been working on perfecting game prediction for many years. We plan on tweaking our algorithm, and the statistics used in it, to make our predictor as effective as possible. This will involve looking at games and researching what analysts and statisticians believe are the best factors to predict the outcome of a game. This might also involve scraping more stats from ESPN, such as player injury, player efficiency, coach history, RPI, BPI, and tournament history. We learned during this project that creating a decent predictor is easy. But as it becomes more accurate, it becomes more than linearly hard to improve. Another goal is to make the website run persistently, so anyone can use it. This would require someone to invest some money on a web address and we would need to spend more time on the web app. It is functional and polished now, but we could add extra features to entice more users. If we were to do this, we would also need to make the scraper more intelligent. It would be best to update the stats every night at the same time. Also we would like the scraper to add only new games and update only the teams that played the previous day. All of these extra features are not necessary for the app to work, but would provide the most benefit if we were to actually publish the site.

Personal Reflections:

Dan: I spent the majority of my time writing the scraper. During this process I learned a lot about ruby and how to efficiently scrape HTML. The first few pages that I scraped were done messily, and took hours to figure out. Then something clicked and made much more sense. I was the forced to refactor the database into a much better format and essentially start a brand new scraper. I'm glad I was able to work on this, since it really piqued my interest in web scraping, something I plan on dabbling with in the summer. However, the fact that this was necessary was frustrating. It could have been done with a few web calls to the ESPN API. This took away from time I could have spent making our algorithm more efficient, or working on the web interface. Since I had to get the stats, I learned a good amount about the machine learning algorithms we used. While it wasn't enough to replace a machine learning class, I think it helped broaden my education at this school. I also learned a great deal about sports prediction algorithms from all the research I did on them. I find it really interesting and a niche of Computer Science I want to investigate more.

Adam: I spent most of my time working with the C5 decision tree algorithm used in our project. I was a bit rusty with C++ so I had to brush up on my coding skills in the language. At first we had considered writing our own prediction algorithm from scratch. We soon realized that there were many existing solutions that would likely be more effective than something we could produce in a semester. One of these existing solutions we decided to use was the C5 decision tree algorithm. Overall the library was very easy to work with and was very powerful. It was very interesting to play around with all the features it included. It did however require a decent amount of work to write an interface to connect the algorithm into our application. There is still a long way we could go to improving the accuracy of our algorithm but I think we have made a very good first attempt. I enjoyed the modified XP process that we used this year. I found it much simpler to accommodate into my schedule and I believe our team had more productivity than my team from CS 427. I learned a lot about software development and basketball prediction during this project.

Siddharth: My major contribution to this project was the implementation of the Artificial Neural Network (ANN) algorithm. After extensive research on quantitative sports analysis, particularly for basketball, we found out that neural networks (along with decision trees) have a good accuracy for prediction, that is, better than 65%, which is what professional sports analysts have. As a result, I decided to use the FANN (Fast Artificial Neural Network) library to plugin to our C++ application by using the statistics we scraped from ESPN. Prior to this project, I was not familiar with the neural network concept. But after using the library, I learned a lot and found the algorithm very interesting, particularly due to the fact that it is based of the human brain. Also, for our software development process, we used XP without pair programming. I found the iterative process very helpful as we all knew what he had to work on every week and by focussing on small user stories each iteration, we were successfully able to have a deliverable by the end of the semester. Lastly, I found the fact that we had the chance to work with the March Madness bracket while the NCAA tournament was going on very helpful.

Brendan: My main area of focus in this project was the front end. I have a lot of experience with Rails, so I was glad to be able to use my expertise there, but I also branched out by trying Cucumber, an integration test framework I hadn't used before. A lot of interaction with the server happens via AJAX and custom Javascript, so a lot of time was spent writing integration tests that spin up a browser instance and interacts with the application as a user would. I had never done testing in this fashion before, so I learned a lot through this project. We decided to use Twitter Bootstrap to make the design process easier, and we pretty much stuck with the default theme as most of us aren't skilled designers. While most of the work was straight forward on the application, I was still able to learn a lot about full stack integration testing, which should helpful in the future.

Max: The primary concern for the project was creating a way for the backend to effectively communicate with the database. This entailed not only creating a way to translate from what was stored in our sqlite database to an in memory representation, but then to save this information in text files in such a way that it could be read by both the C5 algorithm as well as the ANN algorithm. Getting the sqlite connector to work with C++ was fairly straightforward, so I did not find myself having that many explicitly technical issues. However the iterative nature of XP led to a lot of time not only going to figuring out new features for the database connector, but also retooling the way the current database connector was working to incorporate changes in algorithms along with changes in the actual statistics we were collecting. I'd say that because of this most of my learning was focused on finding ways to make my database connector as simple to change as possible so that when new statistics were incorporated it was with minimal friction.

John: I spent the majority of my time writing the in the code that allowed communication between the frontend and the backend. Each iteration built upon the communication process a little bit at a time. First let's try to open a socket and get the two to connect. Next I started sending the bytes over the wire. Third, I parsed the bytes and recorded them in a queue structure. Lastly, I abstracted the messages and queue behind a team api and query methods. I am glad I was able to work on it. It opened my eyes to the benefit of test driven development and expanded my knowledge. I was able to use some great testing frameworks, specifically, cucumber and gtest. I enjoyed my final implementation. There are issues if someone sends in a characters that need to be escaped but because we knew team names did not use those characters we agreed to a simplistic (value of XP) implementation. I would like to eventually explore other inter-process communication libraries and designs.