Fantasy Football Coach

Undergraduate Honors Thesis
May 10, 2006
Johns Hopkins University

Raaid Ahmad
The Johns Hopkins University
raaid@jhu.edu

Venkat Bhagavatula
The Johns Hopkins University
venkat@jhu.edu

Jeffrey Dvornek
The Johns Hopkins University
jdvornek@jhu.edu

Abstract

Millions of people play some fantasy sport and finding a way to create an optimal team to win the game is extremely difficult. Fantasy Football Coach (FFC) attempts to predict player performance and analyze scoring and league specifications to allow managers to draft the best teams possible. FFC employs three algorithms to enhance a manager's preparation for the fantasy football season. A pre-draft algorithm uses normalization and regression techniques to predict player performance and create a ranking of NFL players. A live-draft algorithm attempts to re-order the pre-draft lists as a live draft is occurring based on draft trends and other in-draft factors. Finally, a trade evaluation algorithm allows managers to determine the benefit they receive from a trade. The study concluded that the various algorithms employed yielded teams that were between 7% and 10% better than teams created using existing draft lists (ESPN, Yahoo, etc). The conclusions imply that statistical research and methods can better predict player performance than subjective measures.

I. Introduction

I.A. What are Fantasy Sports?

Fantasy Football is a game in which a group of players create and manage their own football team. Each player's team consists of real NFL (National Football League) athletes that the players draft to be on their team. Once a real NFL player has been selected to be on a team, he may not be a member of any other team. The goal is to create the best possible team and at the end of the season, the manager with the best team wins the league championship.

I.A.1. The Draft

The way each manager creates his/her team is through a process known as a fantasy draft. Every manager is assigned a "draft pick" at random. The draft pick denotes the position at which the manager may choose a player. The manager with the first pick in the draft has his/her choice of any player in the NFL. The second manager will then pick from any player that remains. This continues until the last manager has picked a player. At this point, every manager has selected one player and then the draft order reverses and continues. This means that the last manager to pick in round 1 receives another pick right after their first pick, i.e. the first pick in round 2. In this way, the draft "snakes" around as illustrated in Figure 1. The draft continues until a team's entire roster has been filled (typically between 15 and 21 players, depending on the settings of the league).

Figure 1 – Illustration of "Snaking" Draft Progression

I.A.2. Scoring

Each week, a manager selects which of the players on their roster will "start" for their team that week. The rest of the players are "benched". Only players who are started will earn points for their team during that given week. The manager must

select his/her starting lineup before the beginning of real NFL games that week. Once the lineup is set, it cannot be changed after live games have started. Figure 2 illustrates a sample roster of players that a manager drafts.

POSITION	PLAYER	TD	PASS YDS	RUSH YDS	REC YDS	FG	PAT
quarterback	Peyton Manning	39	3745	121	0	0	0
quarterback	Kerry Collins	26	3200	345	0	0	0
running back	Deuce McAllister	11	0	1134	373	0	0
running back	Clinton Portis	19	0	1674	2128	0	0
running back	Warrick Dunn	9	0	899	555	0	0
Wide receiver	Torry Holt	11	0	0	1100	0	0
Wide receiver	Anquan Boldin	9	0	0	1254	0	0
Wide receiver	Rod Smith	7	0	0	900	0	0
Wide receiver	Ashley Lelie	5	0	0	875	0	0
Wide receiver	Wayne Chrebet	4	0	0	993	0	0
tight end	Jerramy Stevens	6	0	0	751	0	0
tight end	Jeremy Shockey	5	0	0	627	0	0
kicker	David Akers	0	0	0	0	29	34
kicker	Martin Gramatica	0	0	0	0	18	28

Figure 2 – Illustration of a Sample Roster with Sample Statistical Performances (Simplified by reduced number of statistical categories)

Of all the players available in the roster, the manager must choose who to start based on league roster restrictions. A sample of a roster scheme is given in Figure 3. In the example league, since the manager can only start 1 quarterback and 2 running backs, he may choose to start Peyton Manning in lieu of Kerry Collins and Clinton Portis and Deuce McAllister in lieu of Warrick Dunn. Whenever an NFL player earns statistics in real life, (i.e. scores a touchdown, gains yards, throws a completed pass, fumbles the ball) these feats earn the player a varying number "fantasy points". Every player that a manager has designated as a starter will earn points for the manager's team based on their real-life performance. Figure 4 illustrates the conversion of real life statistics to points.

POSITION	STARTERS
quarterback	1
running back	2
wide receiver	3
tight end	1
kicker	1

Figure 3 – Listing of a Sample League's Starting Position Restrictions

STATISTIC	UNITS	PTS
TD	1	6
PASS YDS	25	1
RUSH YDS	10	1
REC YDS	10	1
FG	1	3
PAT	1	1

Figure 4 – Listing of a Sample League's Scoring Scheme

For example, if a quarterback passes for 250 yards, he will earn 1 point for every 25 yards passed, or according to the table, 10 points. Figure 5 shows an example of the calculations require in the conversion from statistical output to fantasy points.

	TD	PASS YDS	RUSH YDS	REC YDS	FG	PAT	
Peyton Manning	39	3745	121	0	0	0	
Fantasy Point Formula	6*39	3745/25	121/10	0/10	3*0	0*1	**Total**
Fantasy Point Total	234	149.8	12.1	0	0	0	395.9
David Akers	0	0	0	0	29	34	
Fantasy Point Formula	6*0	0/25	0/10	0/10	3*29	34*1	**Total**
Fantasy Point Total	0	0	0	0	87	34	121

Figure 5 – Calculation of fantasy score from real statistics, given scoring scheme from Figure 4.

Finally, each week, one manager's team will be pitted against another manager's team. At the conclusion of the games for the week, the fantasy team to have earned the most fantasy points from real life performances will be declared the winner for that week and will earn a win. The team's opponent will in turn, receive a loss. At the end of the season, (unless playoffs are used in the league) the fantasy team with the best win/loss record will be crowned the champion.

I.B. Handling Variation

A difficulty in analyzing and mastering fantasy sports lies in many sources of variance. This study deals primarily with 3 forms of variance and attempts to account for 2 forms of variance and attempted to predict, or minimize the 3rd source of variance. The three primary sources of variance that will be discussed in this study are variance in scoring scheme, roster restrictions and player performance. The scoring scheme plays a very important role in which team wins a game during a given week. Many different leagues have different scoring systems and this often changes the value of different players and different positions. A second and equally important variation between leagues comes from the league roster composition. The number of starters at any given position changes from league to league as well as the total number of roster slots. This changes the value of different positions and causes variations analyzing different leagues. Finally, perhaps the most prevalent source of variation is the player performance itself. While athlete performances can be projected, they are by no means certain.

I.C. Current Applications

There is no shortage of data available regarding player projections for upcoming years (ESPN.com, Yahoo.com, etc). There is also no shortage of listings that list players in terms of statistical output, by position. However, the main problem

with this data is that it does not allow the user to use any of their own judgment when making modifications to the projections. While in the specific case of fantasy sports, statistical analysis is necessary, it is necessarily true that a purely objective approach will almost never be optimal since the statistics cannot take into account player slumps or breakouts.

Additionally, a major shortcoming of the current approach is that while it may not be too difficult to analyze a league metric and rate various players in a given position, (yet very few applications do this) it is significantly more difficult to place players from every position into one comprehensive list, taking into account the relative values of different positions. This comprehensive list is necessary to give league managers an idea of which positions are more valuable than others.

Finally, and possibly most significantly, there is a huge void in the area of live draft applications. There are no applications which give players updated rankings during a player draft. A major reason for this is because algorithms for doing this are by no means trivial. The algorithm requires information regarding the players and information about the draft, such as the players that have been taken and what teams have taken which players.

I.D. Fantasy Sports Coach's Innovation

A major innovation offered by Fantasy Sports Coach is its ability to allow the user to determine what quantitative data to use and what subjective modifiers should be left to the user's judgment. To date, the few fantasy sports applications that exist use their own algorithms to create a player list based on statistics, projections, and all quantitative criteria and end the analysis at that stage. Fantasy Sports Coach offers users the ability to add modifiers to players based on their personal feelings, hunches, or extra information that they have about a player that cannot be captured in statistics alone. Given that a purely objective approach will never be optimal (due to the large statistical variance in player performances from year to year), allowing players to use their own extra information to modify a baseline ranking (which was created via quantitative data) creates the possibility for better rankings.

Fantasy Sports Coach brings unparalleled customization capability to the fantasy sports frontier. First, users can customize Fantasy Sports Coach to create player rankings based on the specific league format they play in. Users can specify the number of starting spots in a league, the number of bench spots, and every single detail of the scoring scheme of the league. Each of these pieces of information provides Fantasy Sports Coach valuable data that can be used to optimize player rankings. Second, users may manually add modifiers to players to move them up and down on the application-generated rankings. This allows intangibles and user based predictions to be used in the statistical rankings, something that does not exist in any status quo applications.

II. Algorithms

II.A. Player Rating Algorithm (Pre-Draft)

Pre-ranking players for a fantasy football league draft requires consideration of many factors. The first set of factors has to do primarily with the format of the league and the makeup of the teams. In our program, the user provides details concerning the scoring system of a league, the number of teams, and the positions included.

When a fantasy league is created, a certain point value is assigned to each type of major accomplishment in football (i.e. touchdowns, rushing yards, receiving yards, passing yards, tackles, field goals, etc.). For example, at 1 point for every 10 receiving yards and 6 points per touchdown, a wide receiver would gain 9.5 points if he caught a 35 yard pass for a touchdown. Our ranking algorithm therefore first reviews the statistics of current NFL players over the number of years specified by the user, and calculates the number of points that each player would have scored under the scoring system provided for each year. To do so, each player is sorted into a container for his respective position, and the scoring system is consulted to determine his **absolute output weight**. The "weight" of a player can be thought of as a scaled number representing his value.

$$Weight_{AO}(Player) = \sum_{i=1}^{\#Years} \%Year_i \sum Accomplishment * Value$$

Figure 6 – Absolute output weight formula

Above we see the "absolute output weight" (shown as $Weight_{AO}$) as the number of points that the player would have scored over a certain number of seasons, augmented by declining percentage significance for each season after the most recent one. The number of seasons taken into account and the percentage assigned to each season can be set by the user, but is defaulted to 75% for the most recent season, 20% for the one prior, and 5% for the season before that.

Once each position has its players ranked by absolute output weight, it is necessary to begin considering the more difficult problem of determining how best to compare a player in one position to a player in another. Unfortunately, this is not as simple as comparing the number of points achieved under the scoring system. Each position is unique, in that the offensive achievement of a player in one position could easily be more valuable to a team even though there are players in other positions that have more total points for a season. The reason for this boils down to averages: a player who produces significantly more than the average number of points for his position is usually more valuable to a team than a player who scores more points but whose production is similar to many others in his own position (this relates to the concept of "position scarcity" discussed in the Live Draft section). Because players are required for each position, it is important

to create a team that produces well across each one. In this way, drafting only Quarterbacks (because they tend to score the most overall points) is useless since a team can only garner statistics for a given number of Quarterback roster slots.

The most obvious solution to this issue is to take the average weight of each position and score each player in that position according to how well they do relative to the average. However, since there are only a certain number of players that actually start in each position for each team, taking the average across every player in a given position doesn't provide an accurate benchmark to judge fantasy players by (as there would be many players included in the average that are near the bottom of their position and would never be drafted in the first place). Because of this, our algorithm creates positional averages only across "drafted players". For a given position, this is equal to 1.5 times the number of starting slots (the extra 0.5 accounting for backup) multiplied by the number of teams in the league. Once the average is taken, all of the players in each position are assigned new weights based on the percentage of points they score above or below the average for their position. This is called the **drafted average weight** of a player. The formula for this new weight for the i^{th} player is below:

$$\eta_k = (1.5 * \text{\# Starters for position } k)$$

$$\mu_{p_k} = \text{Expected number of points scored by } \eta_k \text{ players for position } k$$

$$Weight_{DA}(Player_i) = Weight_{AOi}\left(1 + \frac{Weight_{AOi} - \frac{\mu_{p_k}}{\eta_k}}{\frac{\mu_{p_k}}{\eta_k}}\right)$$

Figure 7 – Drafted average weight formula

Though these new weights could conceivably be compared as is, the comparison would still not exactly provide a user with what he needs to have a good draft. Because different leagues assign a different number of slots for each position, each league has the potential to favor a different position. Thus, it is important to compute what percentage of a team's total production can be expected from a given position. This is done by taking the average number of points for a given position and multiplying it by the number of starting slots for that position. These values are then summed to determine the expected average production of a team in a given league. Once this is done, the percentage of total production for each position can be computed. Every player's drafted average weight is then multiplied by his respective position's percentage, providing the **relative average weight**.

$$\beta_k = \frac{\mu_{p_k}}{\sum_{k=1}^{n} \mu_{p_k}}$$

$$Weight_{RA}(Player_i) = Weight_{DAi} * \beta_k$$

Figure 8 – Relative average weight formula

Above, we define β_k as the percentage of a team's expected output that can be expected from position k (this is calculated for each of the n positions). This value is then multiplied by a player's drafted average weight to created his relative average weight ($Weight_{RAi}$ above). With the relative average weight, it is now possible to compare a player in one position to another. The algorithm combines:

- The total number of points a player would produce under the current scoring system
- How well the player does compared to the rest of the players in his position
- The expected total average output for a given player's position on a team

All of this is used to create a comprehensive ranking list of the players in NFL based which players would be most suited to a particular league's configuration.

For thoroughness, 4 different variations of the main algorithm were used in testing and a simplified version of our original algorithm was found to be superior. All four algorithms are summarized here:

Algorithm 1: The main algorithm discussed above.
Algorithm 2: Simplified algorithm which just ranks players based on their absolute output weight. (Biases positions which score more, i.e. quarterbacks)
Algorithm 3: Algorithm which rates players based on their relative average weight. The number of players used in this average is a static number.
Algorithm 4: Same as algorithm 3, except the number of players over which the average is taken is not static, but instead is dynamic and equal to the number of starting players of the given position (i.e. the drafted average weight is included as well).

II.B. Live Draft Algorithm

There are several issues to consider when re-ranking fantasy players in a live draft. The first is "position scarcity". The idea of position scarcity in fantasy football is best illustrated by the tight end position. A tight end is an offensive player whose primary role it is to block for the quarterback or running back.

Sometimes, however, the tight end will receive passes from the quarterback in either surprise plays or last-ditch efforts. Because of the scoring in fantasy leagues, the tight end is not judged by how good he is at playing his position (i.e. blocking effectively), but rather by his offensive production. Unfortunately, because of the position's blocking responsibility, there are very few tight ends that are able to become star receivers. This makes the tight end position "scarce" in a fantasy league: there are a small number of stars that are a great deal better than the rest.

After some analysis, it was decided that the best way to deal with the position scarcity issue during live drafting was to reprioritize the draft list every time a player in a "scarce" position is drafted to reflect the increased value of the remaining "good" players at that position. Here, a "good" but scarce player is defined as one that is several statistical deviations above his peers, with the rest of the players at his position having relatively similar statistics. To do this, a "drop score" was assigned to each player to reflect the percent difference between him and the next ranked player in his position. During a live draft, a bonus is assigned to scarce players by finding which player in each position in the rankings possessed the largest drop score. This player and every player in his position that is ranked above him is given a bonus that is scaled based on his drop score. Assuming p_i is the score of the i^{th} ranked player on the list, his drop score would be:

$$DropScore(p_i) = \frac{p_i - p_{i+1}}{p_{i+1}}$$

Figure 9 – Drop score formula

To demonstrate, we will look at an example of the scarcity algorithm in progress. Let's suppose that the algorithm is looking at the following eight quarterbacks, each with the given modified score and drop score for the 2004 season:

Name	Score	Drop Score
Peyton Manning	627	0.058
Daunte Culpepper	592	0.591
Donovan McNabb	372	0.489
Trent Green	250	0.136
Brett Favre	220	0.023
Jake Plummer	215	0.075
Jake Delhomme	200	----------

Figure 10 – Drop scores example

We find that the quarterback with the largest drop score is Daunte Culpepper, because his raw score is 1.591 times greater than Donovan McNabb's. However, the algorithm only actually applies a bonus after a pick has been made. Let us now suppose that Trent Green is the first Quarterback taken. The following table is the result:

Name	Score	Drop Score
Peyton Manning	627	0.058
Daunte Culpepper	592	0.591
Donovan McNabb	372	0.692
Brett Favre	220	0.023
Jake Plummer	215	0.075
Jake Delhomme	200	----------

Figure 11 – Drop Score after removal

Above, we now find that Donovan McNabb has the highest drop score. This means that he, Daunte Culpepper, *and* Peyton Manning will have their scores augmented by a factor of 1.692. This serves to better distinguish the great players in a given position than their original ranking scores would, so that a really excellent quarterback who might not score as many points as a running back still rises to the top. Below we see the results of the algorithm modification.

Name	Score	Drop Score
Peyton Manning	1061	0.058
Daunte Culpepper	1002	0.591
Donovan McNabb	630	1.863
Brett Favre	220	0.023
Jake Plummer	215	0.075
Jake Delhomme	200	----------

Figure 12 – Modified scores

The formula for this modification is as follows:

$$\textit{For all } p_i \textit{ where } i \leq \textit{indexOf}\,\{\max[\text{DropScore}(QB_1, QB_2, \ldots, QB_n)]\}\ldots$$

$$Bonus(p_i) = p_i * \frac{[1 + \max(DropScore)]}{PickNumber}$$

Figure 13 – Formula for Bonus based off drop scores

To avoid having certain players' scores spiral out of control, and to give large bonuses only to the truly "scarce" players, the amount of bonus that a given player receives is divided by the overall pick in the draft at which the algorithm is being applied. For example, if the above table was to be modified with the 10^{th} pick, the top three quarterbacks in the table would have their scores modified to $[1+(^1/_{10})(0.893)][(\text{Player Score})]$.

A problem that often occurs during a live draft is what is called a "run" on a particular position. This is essentially a psychological effect, whereby one manager will pick a position that has not yet been touched in the draft, and the

rest of the managers will follow suit. In football, this can often happen with tight ends, defensive players, or kickers – positions that are not heavily depended on but are nonetheless necessary to round out a team's production. To address this issue, our algorithm keeps track of how many players at a particular position have been drafted in the last 1½ rounds. If a particular position makes up more than a user-determined percentage, the ranking algorithm will augment the value of players at the position to account for the detected run. The increase in value is calculated based on the number of teams participating in the draft. The more teams there are, the more the value of a good player at the run-on position will increase. In order to increase the chances that a manager will be able to get in on a position run before it's too late, the algorithm also keeps track of when the first player at a given position is drafted. For a round after this event, all good players at this position are given a weight increase.

During a live draft, it is also incredibly important to keep track of the positions on a team's roster that have already been filled. If a team already has enough running backs to fill its starting positions, the value of running backs left on the board will be significantly less to that team. There are, however, a few factors that retain value for the remaining players at filled positions. In football, certain positions are particularly susceptible to season ending injuries (running backs, for instance). These positions must be backed up to avoid having to search the less qualified free agent pool later in the season. Another pro to having more than the required number of starters in a given position is that certain positions account for so much of a team's production that it is a good idea to draft multiple good players and decide during the season which ones will pan out best.

Due to their likelihood of injury and percentage of total team production the running back and quarterback positions are each set to require at least one backup in our algorithm. Therefore, the weight given to players in these positions will not begin to decrease until a team has at least one more player than is required at both quarterback and running back. A position's value is decreased with respect to how useful the position's production is to the team (determined earlier by the original ranking algorithm). If, for example, the average percentage of a team's total points produced by wide receivers is 20%, the players remaining on the board in the wide receiver position will receive an 80% weight decrease. This way, once all starting positions have been filled, the positions will remain ordered by their percentage of average total production for a team.

It is interesting to note the effect of other teams' compositions in a live draft. A smart fantasy manager keeps an eye on what positions on the other teams have been filled. Below is an illustration of the way a typical live draft is structured. Let us suppose that we are in the fourth position in a five team draft.

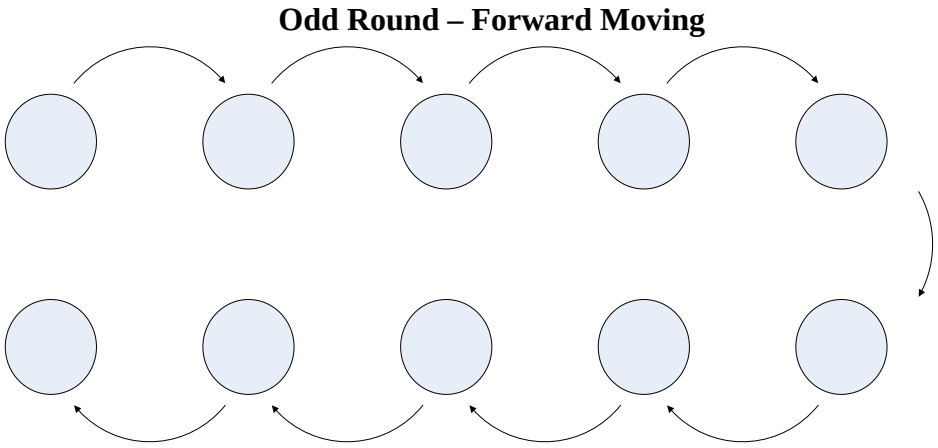

Figure 14 – Snake draft consequences

For the sake of fairness, live drafts usually proceed in a "snake style", with the team having the last pick in the first round receiving the first pick in the second round and so on. Suppose that our team still needs a running back, and we are considering drafting one in an odd round. This means that Team 5 will have the next two picks after us, and then we get to draft another player. If Team 5 already has his starting running back slots filled, it becomes very unlikely that he will draft a running back with his next two picks. This means that we can take a better player in another position before Team 5 does, and probably still get to take the running back that we wanted in the following round.

Team 1

To account for this, the Live Draft algorithm assigns a penalty to a position that is filled for each team following the user's team in a particular round. The penalty itself is reduced by the number of teams there are following the user in a given round. This is because the more teams there are remaining, the more likely it is that one of them will draft a backup. For example, the penalty assigned to running backs in an odd round if Team 5 has his running back slots filled will be less than if Teams 1, 2, and 3 have their running back slots filled in an even round.

To counter the decrease in a given round, there is also an increase to return a given position to its original value if teams on the other "side" of the user do not have their starting slots for that position filled. To illustrate, let us suppose that Team 5 has his running back slots filled, but Team 2 does not. Let us also suppose that the penalty assigned to running backs on the Team 5 side is that their scores are reduced to 80% of their original value.

Team 1

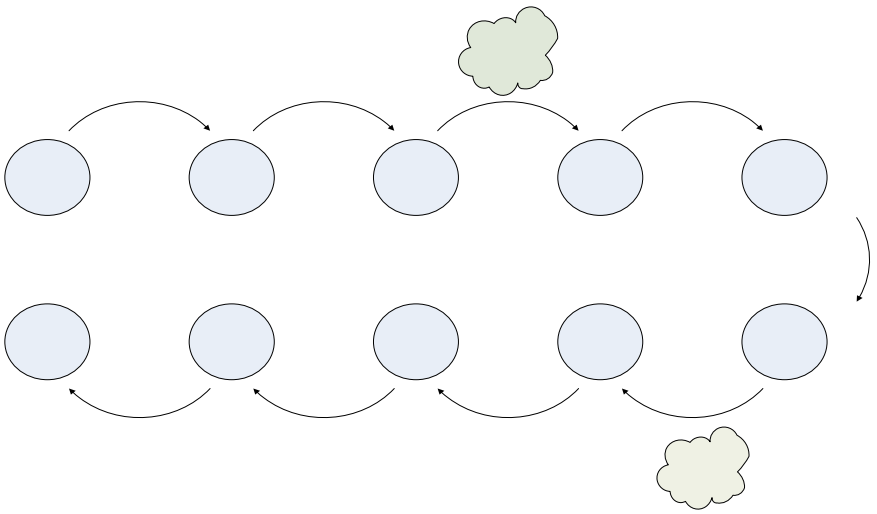

Figure 15 – Bonuses based on draft position

Here we see that just before our team has a pick in the odd round, the scores of all running backs are reduced to 80% of their original. However, because Team 2 does not have his running back slots filled, a bonus of 125% is assigned to the running back position to return it to normal. In this way, we are not likely to risk skipping a quality running back again, since Team 2 still needs one.

As it is above, the penalty assigned to a given position is determined by the number of teams in the draft, and how many there are following the user's team in a given round. The formula for the penalty is as follows:

Team 1

$$Penalty = (Position)\frac{\#TeamsLeft}{\#Teams}$$

Figure 16 - Penalty formula

If there are 5 teams in the draft, and 1 following the user in a round that has a given position filled, the position's value will decrease by $^1/_5$ (or, 20%).

II.C. Trade Evaluation

Team 1

An important algorithm developed by Fantasy Football Coach is its trade evaluator. This feature allows the utility of the application to transcend the draft day. The trade evaluator not only handles the trivial case of 1 player for 1 player trades, but also adeptly analyzes multi-player trades. When players are traded from one team to another, the sum or even the average of the absolute, total output of the players is not of much consequence. Instead, the correct methodology is to identify the total expected output of a roster before the trade,

and after the trade. These methods are quite different because some trades require shuffling of a roster due to a limited number of starting slots.

The concept of expected output is a very important one when it comes to trades. The goal of a trade is to increase the fantasy point output of one's team. The expected fantasy point output of any given player is calculated by taking the overall expected fantasy point output of the player, dividing it by 16 (the number of games in an NFL football season) and then multiplying this quantity by the number of games the player has left (adjusted for bye weeks when the player's team does not have a game). This new quantity is called a player's expected fantasy point output, for the remainder of the season. For the remainder of this section, we will refer to this quantity as the expected output of a player.

Now, to discuss the specifics of the trade evaluator, the first case will be the trivial case of a 1 for 1 trade. With players of the same position, there is no need to look any further than the expected output of the players being traded. The team which receives the player with the highest output is receiving the better side of the trade and it is not possible for the trade to, on face, be beneficial for both teams.

In the slightly more complex case, 1 player from each team is traded, with each player being of a different position. In this scenario, suppose team T_a is trading away player P_a who plays in roster slot R_a and T_b is trading away player P_b who plays R_b. For T_a, the benefit is calculated by taking the expected output of the team after the trade, and subtracting it from the expected output of the team after the trade. More specifically, the algorithm takes the roster for T_a and removes player P_a from the roster. Next, player P_b is added to the roster of T_a. Then the roster is sorted, first by position, then by expected output (descending order). Once the roster is sorted, the algorithm selects the top N_r players, for each position R, where N_R is equal to the number of starting players that play position R. Then the expected outputs of the ($N_{QB} + N_{RB} + N_{WR} + N_{TE} + N_K$) selected players are summed to yield the roster's total output. This is compared to the previous output to determine if the trade is beneficial to the team.

Below, the asterisked player is the player that is being traded away from each team. The figure shows that the player that T_b is trading away has an expected output (210.4) that is significantly higher than the player that T_a (167.3) is trading away. For T_a, it replaces it's traded player WR_S, with expected output 167.3, with the WR_B player with output 152.3 for a net loss of 15.0. Then, T_a replaces its old RB_S with expected output 149.5, with T_b's RB_S player with expected output 210.4. This is a net gain of 60.9, for a total gain of 45.9.

T_a
Pre-Trade

Roster Slots	Output
QB_S	*321.2*
RB_S	*149.5*
*WR_S **	*167.3*
QB_B	178.6
RB_B	132.4
WR_B	152.3
STARTER SUM	638.0

T_b
Pre-Trade

Roster Slots	Output
QB_S	*308.4*
*RB_S**	*210.4*
WR_S	*129.5*
QB_B	192.3
RB_B	185.3
WR_B	112.4
STARTER SUM	648.3

T_a
Post-Trade

Position	Output
QB_S	*321.2*
RB_S	*210.4*
WR_S	*152.3*
QB_B	178.6
RB_B	132.4
WR_B	149.5
STARTER SUM	683.9

T_b
Post-Trade

Position	Output
QB_S	*308.4*
RB_S	*185.3*
WR_S	*167.3*
QB_B	192.3
RB_B	129.5
WR_B	112.4
STARTER SUM	661.0

Figure 17 – Before and After Trade Algorithm

For T_b, the loss of *RB_S* requires the replacement with *RB_B*, with an output of 185.3, for a net loss of 25.1. Then, T_b's WR_Starting is replaced with the newly acquired *WR_S* from team A, for a net gain of 37.8. For T_b, this yields a total gain of 12.7 points.

Finally, the general case for the trade evaluator will be discussed. The general case is just an extension of the 1 for 1 case, with more calculations involved. The steps are outlined below.

Step 1:
For each roster, sort by position first, then by expected output, in descending order.

Player	Position	Output
B	QB	219.8
H	WR	122.8
I	WR	101.6
A	QB	327.3
D	RB	167.5
F	WR	138.9
C	RB	197.3
J	WR	96.2
E	RB	121.2
G	WR	132.9

Unsorted

Player	Position	Output
A	QB	327.3
B	QB	219.8
C	RB	197.3
D	RB	167.5
E	RB	121.2
F	WR	138.9
G	WR	132.9
H	WR	122.8
I	WR	101.6
J	WR	96.2

Sorted

Figure 18 – Sorting of Roster

Step 2:
For each position, select the top N_R players (where N_R = # of starting roster spots for position R). The following example assumes 1 QB, 2 RB, and 3 WR will start. The shaded players are selected to start.

Step 3:
Sum up the expected output for the selected players on each team and denote them O_{1B} (Output of Team 1 before trade) and O_{2B} (Output of Team 2 before trade).

Player	Position	Output
A	QB	327.3
B	QB	219.8
C	RB	197.3
D	RB	167.5
E	RB	121.2
F	WR	138.9
G	WR	132.9
H	WR	122.8
I	WR	101.6
J	WR	96.2
O_{1B}		**1086.7**

Figure 19 – Selection of starting players from sorted roster

Step 4:
For each of the teams, subtract the players being traded away. For the purposes of our example, we will be trading away players A and D for players Y and Z.

Step 5:
For both of these modified rosters, add the players being traded to the team. We are only following team 1, but the process is analogous for team 2, where we subtract players Y and Z, and players A and D.

Player	Position	Output
B	QB	219.8
C	RB	197.3
E	RB	121.2
F	WR	138.9
G	WR	132.9
H	WR	122.8
I	WR	101.6
J	WR	96.2
Y	QB	300.2
Z	RB	198.3

Figure 20 – Roster after subtraction of traded players and addition of new players

Step 6:
Sort the rosters first by position, then by expected output, in descending order.

Step 7:
For each position, select the top N_R players (where N_R = # of starting roster spots for position R).

Step 8:
Sum up the expected output for the selected players on each team and denote them O_{1A} (Output of Team 1 after trade) and O_{2A} (Output of Team 2 after trade). In figure 21 the final roster is shown, with shaded players denoting the starters and the output of the roster after a trade.

Player	Position	Output
Y	QB	300.2
B	QB	219.8
Z	RB	198.3
C	RB	197.3
E	RB	121.2
F	WR	138.9
G	WR	132.9
H	WR	122.8
I	WR	101.6
J	WR	96.2
O_{1A}		1090.4

Figure 21 – Final roster of team 1, after trade

Finally, the evaluation is done using 2 criteria. First, the improvement in one's own team is calculated, and second, the improvement in the other team is calculated. First, in order for Fantasy Football Coach to approve a trade, $O_{1A} > O_{1B}$ must be true. In other words, the trade must benefit one's own team. A secondary rule can be added by the user: $O_{1A} - O_{1B} > O_{2A} - O_{2B}$. In other words, in addition to the trade benefiting one's own team, it must benefit one's own team *more* than it benefits the opponent.

III. Evaluation

III.A. Pre-Draft Algorithms

Now that we have created an algorithm to create a draft list, it is our burden of proof to show that this algorithm creates a "better" draft list than the other draft lists offered in the status quo. The first task at hand is to find a way to "compare" two different draft lists to see which one is better than the other. Since a Fantasy Football team's success is based directly on the output of players, we need to find a mechanism to transform a static draft list into a team that is created, based on that list.

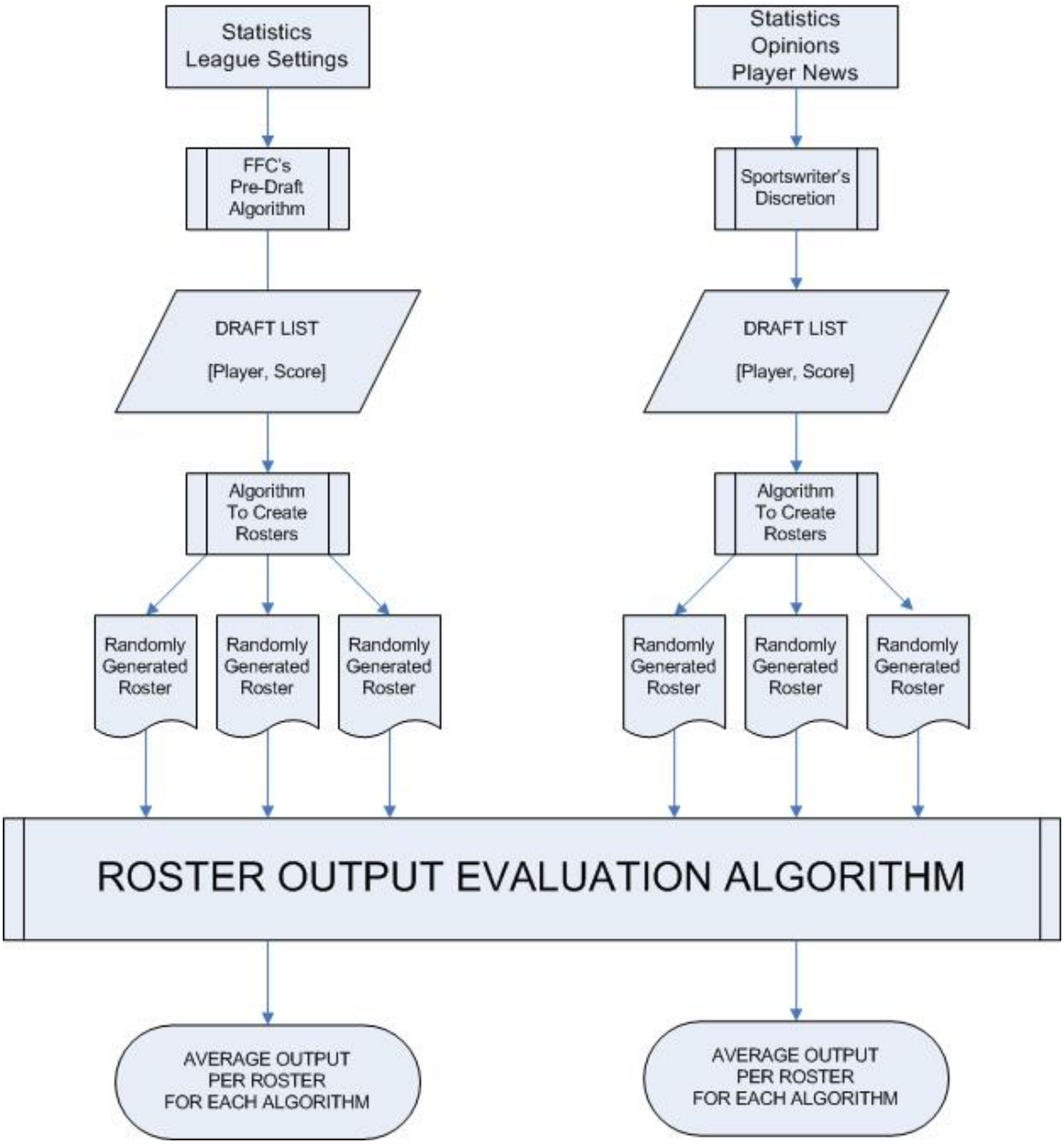

Figure 22 – Testing Methodology

Once a team is created for each draft list, we can compare the actual output of the given teams to find out which team is better. The draft list which consistently yields the best team can be deemed to be superior.

For the purposes of testing, it is not prudent to have a deterministic algorithm to create a team from a list. The reason for this is multi-pronged. First, a deterministic algorithm would mandate that a unique team be created from a draft list, which is definitely not the case, since during a draft, while players generally follow the list, some small deviations, and sometimes large deviations do occur. Second, if the deterministic algorithm happens to select a player that severely overperformed, underperformed, or was injured during the upcoming season, then the team that is created from that list, will have an unfair advantage, or disadvantage over an algorithm that might be better, in the general case. For this reason, a random element must be added to the algorithm.

The algorithm to transform a draft list into a team entails a few steps. The draft list is first split into tiers, or draft rounds. The size of the tiers is equal to the number of teams in the league. For example, if there are 5 teams in a league, players 1-5 are in tier 1, 6-10 are in tier 2, etc. This tier system is illustrated in Figure 23.

	Player	Relative Avg. Wgt.
Tier 1	P1	457
	P2	450
	P3	438
	P4	429
	P5	426
Tier 2	P6	420
	P7	418
	P8	392
	P9	390
	P10	385
Tier 3	P11	382
	P12	380
	P13	371
	P14	369
	P15	358
Tier 4	P16	347
	P17	334
	P18	329
	P19	312
	P20	303
	...	...

← Select a player at random from tier 1.

← Select a player at random from tier 2.
- If player selected is of a position already filled, find the closest player who occupies an unfilled position

← Continue until all roster spots are occupied.

Figure 23 – Tiered system used for creating random rosters from a draft list

A player is selected at random from a tier. This simulates a player drafting a player in round 1, 2, etc, once for each tier. This system proceeds until all roster

slots have been filled. However, this "dumb" algorithm has a flaw in that it does not account for intelligently filling roster slots correctly. To deal with this the algorithm makes sure that players to fill all *starting* roster slots are taken first. If the random player selected plays a position that the team does not need, a player closest to the selected player that plays a position that the team *does* need is selected. Once all starting slots are filled, the algorithm continues, making sure that all bench slots are filled.

Now that a roster has been created based on the draft lists to test, we must determine how to measure how "good" a roster is. To do this, we will take the actual performance of the players during the year for which this draft list was created. For example, if the draft list was created for the 2004 season, we would use actual statistics from the 2004 season to see how the teams actually did. Using the player statistics for the appropriate year, and using league settings to determine point values for statistics, we create the actual outputs of the players on each roster. The sum of each starting player's output is taken and then the sum of each bench player's output times one-half, is taken. Once these values are summed, we obtain the output of the roster. This will be the metric that is used to rate a roster. This is analogous to the formula in Figure 6, when the expected output was calculated for various players. The only difference here is that statistics that we already know to be accurate are used, this producing an actual output statistics, rather than an expected output statistic.

Now, to test the different draft lists, the algorithm was run multiple times on each list. Each of the roster's outputs was taken and recorded. The experimental data below is for a league of 10 teams, with each team comprised of 1 starting quarterback, 2 starting running backs, 3 starting wide receivers, 1 starting tight end, 1 starting kicker, and for bench players, 1 quarterback, 2 running backs, 2 wide receivers, 1 tight end, and 1 kicker. Expected player outputs were calculated using a declining historical weighting with 2004 weighted 75%, 2003 at 20% and 2002 at 5%. The true values were calculated using 2005 statistical performances. A sample of the experimental values are shown in the table below.

Trial #	Alg1	Alg2	Alg3	Alg4	ESPN
1	1317.3	1583.2	1030.0	1144.6	1199.3
2	1556.5	1290.9	1496.7	1107.4	1720.5
3	1506.8	1475.3	1190.2	1679.4	1387.3
…	…	…	…	…	…
999	1342.1	1431.6	1301.6	1481.3	1354.2
1000	1183.4	1614.7	1313.9	1291.7	1427.9
Mean:	1407.8	1450.6	1415.5	1436.9	1310.4
Std Dev:	179.0	170.8	180.0	180.4	174.2

Figure 24 – Test Results

As shown in the Figure 24 above, all 4 of the designed algorithms outperformed ESPN.com's draft list in the tests according to the mean output of the rosters. The following table shows the aggregate percentage improvement in using each of the algorithms, over ESPN.com's draft list. A p-value is also included, which represents the probability that the algorithms actually have the same performance. Obviously, smaller p-values denote a higher probability that the given algorithms outperform the ESPN list.

Algorithm	% Improvement	P-Value
1	7.43%	< 2.2E16
2	10.69%	< 2.2E16
3	8.02%	< 2.2E16
4	9.65%	< 2.2E16

Figure 25 – Improvement over ESPN Draft List

III.B. Live Draft Algorithm

Testing and tweaking of the Live Draft algorithm proved difficult, to say the least. This was mostly because there is no other application currently in existence that shifts rankings during the progress of a draft, so there wasn't anything to compare our algorithm iterations to. It was decided that since our goal was to prove our application superior to common drafting techniques, that we would obtain a list of the pre-ranked players for the 2005 season as determined by ESPN experts. Since most fantasy users use a similar ranked list to make their decisions on draft day, a set of rules was created so that every team except for the one that we designated would use the ESPN ranked list.

Our designated team would use the top ranked player assigned to him at the time by the Live Draft algorithm. To avoid having one of the "ESPN Teams" take too many players in one position, we assigned a number of designated starting and bench slots for each team. The team would then find the top ranked player remaining on the ESPN list who was in a position that was not already completely filled (starters and bench) with a random normalized probability. This was actually very representative of normal drafting strategy, and it gave us a good idea of how well our algorithm would perform in a real draft.

In order to evaluate success or failure, and the relative merits of the different parts of the Live Draft algorithm, a draft was designed with seven teams (in which our designated team was randomly assigned a draft position). After the draft, each team had its top players evaluated by finding the number of points each player actually scored during the 2005 season (our algorithms ran on pre-season stats). Each draft run was set with the same parameters, which were as follows:

Position	# Starters
Quarterback	1
Running Back	2
Wide Receiver	3
Tight End	1
Kicker	1

Accomplishment	Score
10 Receiving Yards	1
10 Rushing Yards	1
25 Passing Yards	1
Touchdown	6
Interception Thrown	-2
Fumble Lost	-2
FG 0-19 Yards	1
FG 20-29 Yards	2
FG 30-39 Yards	3
FG 40-49 Yards	4
FG 50+	5
Extra Point Made	1

Figure 26 – Draft Test Parameters

The first test of the Live Draft algorithm tested only took filled position bonuses/penalties into account for a user's team. This means that neither position scarcity nor the composition of other teams was considered. Below is an example of one of the runs, displaying the top 9 players taken by each team, and his raw score for the 2005 season. Here, the algorithm is running on Team 5:

TEAM 1	
LaDainian Tomlinson	301.2
Clinton Portis	235.20001
Corey Dillon	167.40001
Antonio Gates	170.1
Reggie Wayne	135.5
Roy Williams	116.7
Larry Fitzgerald	200.9
Derrick Mason	123.3
Chris Chambers	173.8

TEAM 2	
Priest Holmes	106.8
Terrell Owens	112.3
Marvin Harrison	186.6
Tony Gonzalez	102.5
Andre Johnson	78.8
Michael Bennett	81.700005
Carnell Williams	157.90001
Donald Driver	152.1
Jerry Porter	124.2

TEAM 3	
Deuce McAllister	63.2
Randy Moss	148.5
Ahman Green	40.2
Joe Horn	67.4
Hines Ward	161.5
LaMont Jordan	222.8
Jason Witten	111.7
Alge Crumpler	115.7
Plaxico Burress	161.4

TEAM 4	
Shaun Alexander	361.8
Kevin Jones	107.3
Julius Jones	147.1
Javon Walker	2.7
Darrell Jackson	66.2
Steve Smith	228.3
Jeremy Shockey	131.1
Ashley Lelie	83
Eric Moulds	105.6

TEAM 5	
Peyton Manning	298.38
Daunte Culpepper	89.259995
Curtis Martin	113.3
Ricky Williams	117.600006
Muhsin Muhammad	99
Domanick Davis	165.3
Drew Bennett	97.8
Adam Vinatieri	104
David Akers	78

TEAM 6	
Jamal Lewis	123.7
Edgerrin James	266.30002
Steven Jackson	190.6
Chad Johnson	197.2
Anquan Boldin	180.2
Michael Clayton	37.2
Dallas Clark	72.8
Todd Heap	125.5
Jimmy Smith	138.3

TEAM 7	
Tiki Barber	303
Willis McGahee	170.5
Rudi Johnson	226.8
Torry Holt	185.1
Nate Burleson	38.8
Laveranues Coles	114.5
Isaac Bruce	70.5
Randy McMichael	86.2
Rod Smith	142.5

Figure 27 – Team Compositions for Mock Live Draft

In this run, we see the way luck plays into fantasy sports. One of the algorithm's picks was Daunte Culpepper, who was fantastic in the 2004 season. Unfortunately, injury and the loss of his star receiver for the 2005 season severely reduced his numbers. This run teaches us the lesson that sports will always be unsure. Averaging the seven runs together, we find the first version of the algorithm somewhat wanting, as its expected results under perform the average team score by just under 10%.

1st Live Draft Avg. Expected Score	Avg. Score of Other Teams	Percentage Difference
1511.4	1673.9	-9.70%

P-Value: $2.2*10^{-16}$

The next set of runs of the Live Draft algorithm included the position scarcity bonus as determined by a player's "drop score". As discussed earlier, the scarcity bonus increases the value of players that are particularly excellent within their position by finding the point at which the biggest percentage drop occurs between positional player scores. Once this is done, all positional players above the "big drop" player receive a bonus relative to the high drop score. The results obtained after seven runs of the "scarcity bonus" algorithm are below:

2nd Live Draft Expected Score	Avg. Score of Other Teams	Percentage Difference
1700	1584	6.82%

P-Value: $2.2*10^{-16}$

We see that adding the scarcity bonus definitely adds an improvement over the previous algorithm. This is likely because overachieving players in less prominent positions are seeing the light of day in the draft list much earlier on.

Finally, we tested the third incarnation of the Live Draft algorithm. This included the consideration of other teams' compositions in making a decision (along with the scarcity bonus). Because of the way in which the draft was designed, this added feature helped a great deal. The automated players were very unlikely to draft positions that they had already taken, and the designated team profited from this immensely by holding off on certain high-value players that could be obtained in a later round. The results of the "team composition" algorithm test are below:

3rd Live Draft Expected Score	Avg. Score of Other Teams	Percentage Difference
1767.12	1592	10.00%

P-Value: $2.2*10^{-16}$

It is clear that the third algorithm, including consideration of other teams' compositions *and* position scarcity, is the best. A 10% difference in average score is an enormous advantage to look forward to on draft day.

IV. New User Interface Model

IV.A. GUI Model

Developing software requires that a choice be made regarding a user interaction model. The primary choice for this model is that of a desktop application. In this model, a program runs in a dedicated window drawn by a toolkit such as Java's Swing or GTK. Another choice is to run the application on a web server using tools such as Java's J2EE package or PHP. Both of these choices have positive and negative aspects to them which will be explored further.

IV.B. Application

The application model is the most familiar model to users. An application based GUI allows the interface to run from local code. Since this generally requires no data transfer across a network, this is the most responsive method in which to create a user interface. The fact that the GUI runs on a different machine than that on which the data is stored presents challenges in writing a potentially complex network subsystem to allow the GUI to communicate with the server in a secure manner. Also of concern is the fact that if the client application is written in Java, the compiled code is easily decompiled into source, exposing any proprietary algorithms that may have been developed. If it was decided that the client application had to include the algorithms (as opposed to having a server process the data as well as host it) this would likely have to be done in a purely compiled language such as C or C++. This is then less portable than Java and requires additional coding to support multiple operating systems. As well, creating a client-side application promotes situations in which different clients may be running different versions of the application. This can be solved through a system of automatic updates, but this still requires addition coding and raises privacy concerns.

IV.C. Web Application

One method to eliminate some of the shortcomings of the application model is to create software that is entirely based on a server, allowing users to connect via any web browser. By hiding all code behind a web page, it is far less likely that proprietary algorithms will be inadvertently made available to the public. This can be accomplished by using tools like PHP and its related Apache module or a Java Application Server like Tomcat or Jboss. This model has security benefits as well, because far less data is being transmitted over a network. Even though there has been a recent rise in the number of web applications available to the public (GMail, etc.), it is still less familiar to many users than a standard windowed application. Another possibility is to create a web application that uses a Java applet to do the windowing. This proves difficult, however, because there is still

the issue of communication with the data server to overcome. An applet, as well, requires a sizable download to the client browser before it displays, far longer than if the application had used standard lightweight markup to render directly in the browser window.

IV.D. Windowed Web Toolkit

To avoid the compromises that must be made to employ one of the existing user interface models, we devised and implemented a new model. Our user interface appears graphically as though it is a standard windowed application, but it exists solely in the context of a web browser instance. What differentiates this method from running an applet in a browser is that our windowing toolkit is implemented entirely in JavaScript, HTML, and CSS. This choice of lightweight markup and scripting languages allows our toolkit to render quickly and require a minimal amount of data to be transferred from the server to the client.

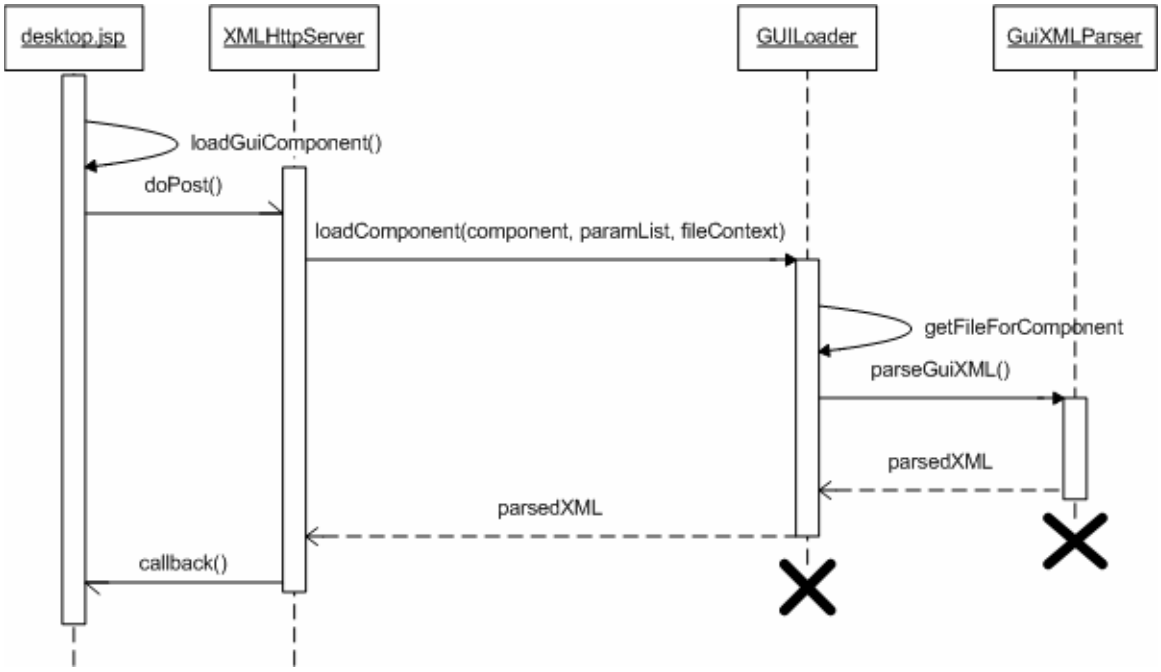

Figure 28 – Windowed Web Toolkit Specifications

IV.D.1. Markup Language Components

The additional markup required to use the windowing toolkit was consciously kept to a minimum to allow for greater ease of use. The only necessary overhead is loading the toolkit's JavaScript file for each page that will use the GUI components. After this script is loaded, each call to loadGuiComponent(*name, parameters*) will cause the server to return code representing the component in a

displayable format such as HTML. The loadGuiComponent function calls methods from a JavaScript XMLHttpRequest object. These calls set up and send a POST request to a web server. Our loadGuiComponent function does not wait for the server to complete the request; instead, all component loads set up a handler to receive the data in the form of a callback function. After getting a response, the callback function is in charge of parsing the response and inserting the markup language into the document in an appropriate location.

IV.D.2. Server Side Components

To implement this toolkit, several server-side components needed to be put in place. The first necessary piece is that which communicates with the clients. This communication channel is implemented as an extension of the J2EE HTTPServlet class. As the request is received, it is parsed to determine which component is being loaded. The loader then searches for the XML template corresponding to that component. Should the template not be found, an error is returned. Pending a successful loading of the template file, the necessary parameters are then inserted into the template and valid markup language is returned. The request handling and response are handled asynchronously to maximize the server's utilization and minimize the total wait time to all clients, not just one particular client.

We chose a hierarchical file format for our template because it more closely matched the target data than did a flat file. To illustrate this format, the following is a representation of the standard HTML anchor tag. While this example is not of any practical value, it is conceptually identical to larger components.

```
<content>
    <a>
        <href>$PARAM1</href>
        <text>$PARAM2</text>
    </a>
</content>
```

As this file is read, each content block signifies a new level in the hierarchy. Tags beginning with a '$' are read as variable and replaced in order with one of the parameters passed to the component loader. Text blocks are interpreted as visible text that would exist between the open and close tag and all other parameters are placed inside the open tag of the node. This yields:

```
<a href=$PARAM1>$PARAM2</a>
```

The process of loading a GUI component does not depend on any data that is held in server memory; each component load is done from the disk. This allows our toolkit to be dynamically extensible without requiring a server restart to add new components. The only required operations to add a component to the server's archive are the creation of the XML template and the modification of the markup

language from which the component will be loaded. In fact, though the XML templates are currently stored locally on the server, it is a simple matter to allow markup language to load remote template files, thus making it possible to use pre-constructed archives of GUI components for rapid site development.

IV.D.3. Usage

Using the toolkit's loadGuiComponent requires passing a component name and component parameters as arguments. The parameters string serves two functions. First, the end of the string is a comma separated list of whatever parameters need to be fed to a component to make it function. The first comma separated token of the string, however, is the id of a page element within which the component should be loaded. As an example, let us take this simple web document:

```
<html>
      <script type="text/javascript"
src="js/GUI.js"></script>
      <body>
            <table>
                  <tr>
                        <td>New Link</td>
                        <td id=”insertHere”></td>
                  </tr>
            </table>
      </body>
<html>
```

If we were to place a script to load a link from the template as specified above into this page:

```
      <script type="text/javascript">
            loadGuiComponent(“link”,
“insertHere,www.jhu.edu,JHU”);
      </script>
```

This load would cause the callback() function in the GUI JavaScript library to place the HTML version of the link inside the tag whose id is insertHere. This specification allows complex layouts to be managed in whatever method the user would demand. We feel that this allows for as much layout control as is possible in existing windowing toolkits like Swing.

The requirement of a component to have a parent tag raises the question of what to do about windows, as they have no logical parent in a layout. The solution to this is to create a division within the body of the page that will become the parent tags for all created windows. The bare document, before any windows are created, would be constructed as follows:

```
<html>
      <script type="text/javascript"
src="js/GUI.js"></script>
      <body>
            <div id=”windowSet”></div>
      </body>
<html>
```

The actual windowing implementation is accomplished using a wrapper for an inline frame. This method allows our windows to display any content that can be displayed in a standard web browser (this includes other uses of our windowing toolkit).

IV.D.4. Examples and Analysis

There are existing libraries that allow dynamically peaceable web content; we, however, feel that we have a distinct advantage over these solutions. Projects such as YouOS, Orca, EyeOS, and Fold offer users the look and feel of a desktop on a webpage, however, these solutions only provide lightweight windows which are limited to specialized content. Below is a screenshot from one such web desktop, Fold.

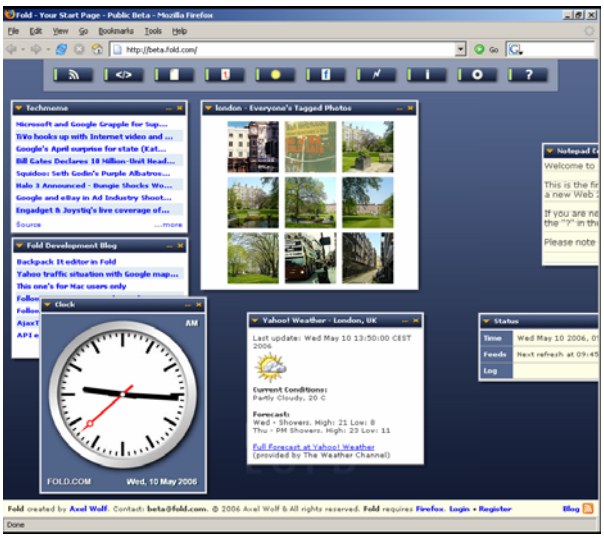

Figure 29 – Screenshot of Fold

When compared to our web windowing toolkit, shown below, Fold is certainly more visually impressive, however, it is in the background that our design differentiates itself. Our web windowing toolkit is capable of displaying any pre-

written web content as well as any code that will be developed for use with the toolkit. This heavier weight rendering capability allows dynamic content to be loaded directly into a window as if it were a new browser window or desktop application. Competitors such as fold are limited to simple layouts of images and text.

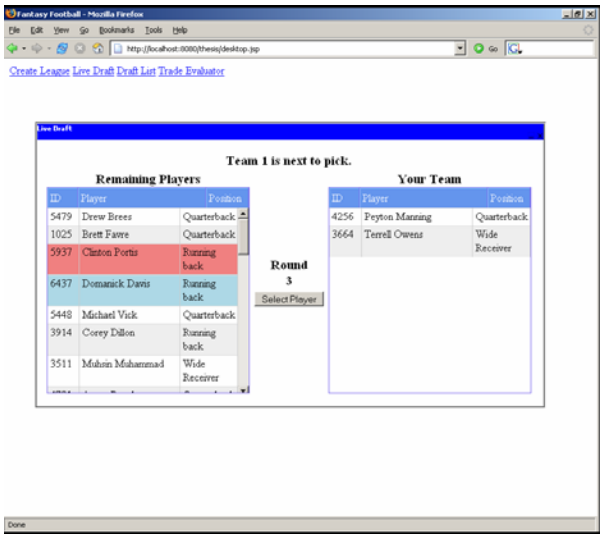

Figure 30 – Screenshot of Fantasy Football Coach

V. Conclusion

V.A. Findings

After rigorous testing of the algorithms, we have determined that there is a statistically significant (as shown by the p-values) difference in the performance of teams drafted with an algorithm generated list, as opposed to ESPN's list. The pre-draft algorithms offered a roster output that is 7.43, 10.69, 8.02, and 9.65% greater than rosters created by the ESPN list. While a subjective aspect of sports will always exist, this study sought to prove that a balance between user-defined subjective factors and quantitative analysis would provide results superior to that of subjective factors alone.

The second algorithm, which selected the player with the highest projected output performed the best as compared to the ESPN list. The reason for this most likely lies in the workings of the algorithm itself. Since the algorithm picks players based on highest absolute output, it takes relatively little information into account about position scarcity. Since the test league that was used has very loose roster restrictions, it did not hurt the second algorithm much. Given a more strict league, the performance of the second algorithm should decline considerably.

V.B. Advancements

We have used metrics that have yet to be considered quantitatively in the Fantasy Sports world. Position scarcity has always been a hot topic of debate and scarce position players always have premium value on draft day. However, there has yet to be discussion or study into how much of a premium should be placed on scarce position players or in the converse case, what discount should be placed on players who play a deep position. The pre-draft algorithm's use of an "average output over drafted players" illustrates a new concept that captures position scarcity and allows a drafter to determine what affect a player's position should have on his draft position.

V.C. The Future

Future plans for Fantasy Sports Coach center largely around expansion of its scope into the domain of other popular fantasy sports. The algorithms were designed to be modular, and as such, require merely the change of metric calculations to enable application to other sports.

Another method of analysis that we would wish to explore is the relationships between the performances of players on the same team. This has impact not only to trade evaluation, but also draft order. For example, if two players on a team generally performed well at the same times, drafting both players, even if one would have been ranked lower, may well prove to maximize total expected output.