Matthew B. Smith  
Sahand Negahban (Faculty Advisor)  
CPSC 490—Special Projects

Matrix Completion for Daily Fantasy Hockey  
Report and Documentation

Background

Sports analytics

Sixteen years ago, Oakland Athletics general manager Billy Beane, constrained by the budget of his small-market team, used advanced statistics to seek out undervalued players who could make an outsized impact on the field. Beane’s success soon captured the attention and respect of his counterparts in front offices around baseball. After Michael Lewis told Beane’s story in the 2003 bestseller Moneyball, sabermetrics turned from an esoteric niche to a popular hobby for fans, math enthusiasts, and gamblers.

The nature of the game makes baseball a ripe candidate for such an approach. Baseball consists of discrete, countable events that happen one at a time: pitches, which are either called as balls and strikes, or put into play as hits, ground-outs, or pop-outs. The game starts and stops without a clock or referees. Not considering the score, there is a finite number of configurations for a game—so many runners on base, this many balls and that many strikes, etc. By succeeding at the plate, a hitter does not interfere with the success of any of his teammates. Likewise, a pitcher either succeeds in quickly retiring the side, or he surrenders walks, hits and runs. On both offense and defense, individual performance is a good predictor of team performance.

Analytics exists for other sports with varying degrees of application and success. As a lifelong hockey fan, I came late to the sabermetrics movement when I read Moneyball earlier in college. Hockey in particular offers unique challenges for anyone looking to apply statistics to predict success. Unlike baseball, the game is fluid, with every player on the ice constantly interacting with and challenging other players. Combinations of players who play well together might yield great results that those players could not achieve individually. Events happen simultaneously and affect other events on the other side of the ice. Since the game play is so fast, statistics that rely on manual reporting—like hits, puck possession, assists, and time on the ice—are less reliable. Coaches, scouts, fans, and gamblers disagree on how to apply some of the newest metrics, like Fenwick and Corsi scores, which serve as proxies for puck possession.

The increased popularity of analytics largely coincided with the rise of the internet age, making sports gambling markets more efficient overnight. Sports bettors in 2016 look less like the Robert De Niro character in Casino and more like an accountant behind screens full of spreadsheets, with all manner of data tools at their disposal. The classic forms of betting still exist—gamblers can either take the “spread,” where they a team is handicapped by a
certain amount of points depending on their opponent, or they can take the "money line" and pick a team to win or lose, straight up or down. Others bet in pools on their own fantasy leagues, or make prop bets in online contests. But today's gamblers operate with far better information at their fingertips.

**Daily fantasy**

This project focuses on a new gambling application that is similar to Beane's original predicament. “Daily fantasy” differs from traditional fantasy in several important ways. Bettors play a new lineup every day instead of drafting at the beginning of the season. Instead of playing in leagues with a dozen friends, they compete against thousands of other contestants on the internet. When a bettor chooses a daily fantasy lineup, he can draft any player who has a game that night. Multiple contestants can have the same players on their rosters—if one player decides to draft Sidney Crosby and Steven Stamkos, that does not stop anybody (or everybody) else from doing the same. But there is a catch—each player costs some amount of (imaginary) money, and contestants are constrained by a salary cap, making it impossible to draft several blue-chip players and guarantee success. So, in this manner, daily fantasy is very much like running a small-market team with a limited budget like Billy Beane.

How “managers” (i.e. contestants) evaluate players is where daily fantasy differs the most from *Moneyball*. In general, game outcomes do not matter (except for goalies). Nor does sustained performance for any particular player, since contestants choose new rosters each night. Players win points for events such as scoring or assisting on goals and making other productive plays. But their performance on a given night may not coincide with their overall season or career trends. This makes looking at nightly matchups a natural starting point for a heuristic for picking the right players, and how I proposed to tackle the problem.

In this form of gambling, as in all gambling, the house always wins. But in daily fantasy, there are no odds to play, so the house never has a “bad” day relative to other days. In the popular FanDuel NHL Breakaway contest, roughly the top 15 percent of finishers (among over 10,000 contestants) are awarded fixed prizes, ranging from thousands of dollars for first place to barely above the $2 entry fee for 2,000th place. In this respect, daily fantasy is more like a state lottery than spread betting.

Unlike state lotteries, however, a small minority of contestants finds ways to win over and over again. According to the Washington Post, in the first half of the 2015 baseball season, the top 1% of players on DraftKings (a leading daily fantasy vendor) accounted for 41% of the entry fees and took home 91% of the profit (Harwell). This simple fact gives credence to the industry’s claim, during the legal battles with New York State this past winter, that their contests are indeed “games of skill.” Yet the same article cited a study that found that 70% of contestants either lost money or broke even in the previous year—which implies that 30% of contestants must have turned a profit. This statistic offers hope that a relative layman with some correct data might be successful over time at winning in daily fantasy.
This project proposes a machine-learning framework and offers an implementation to exploit lopsided matchups each night of the NHL season to predict daily fantasy success. The aforementioned FanDuel NHL Breakaway serves as the case, but the model and the reasoning behind it can be readily applied to any similar site. For each position—goalie, defenseman, centerman, left wing, right wing—the program implicitly generates a matrix of size num_players by num_teams, where the entries represent the anticipated value of player $i$ against opposing team $j$. When asked to generate a lineup for a specific date, the model trains itself on all the preceding games of the season, returns the information necessary to generate an up-to-date matrix, then approximates a solution to the mixed-integer linear programming problem of choosing the optimal lineup of one goalie and two skaters at each position, given the salary cap constraint of $55,000.

The Data

Identifying the correct data, scraping it from the web, and formatting it properly was the main unanticipated and underestimated challenge of this project. Before its focus became clearer (and narrower), the project underwent several stages in its approach towards exactly which data would best serve its purposes.

Structure

Initially, instead of a num_players by num_teams matrix, I considered the possibility of tracking num_players by num_players, with each entry representing a player’s strength against every other player. This approach would be impractical, since hockey players come onto the ice in the same lines, for specific shifts against specific opposing lines, and do not even face most of players on the opposing team. Thus, most entries in a num_players by num_players matrix would be empty, and attempting to complete such a sparse matrix would yield redundant entries. It became clear that to track each player’s performance against particular opposing lines would be equally impractical, after an exhaustive search for information organized in any way that would allow for such an approximation. On the other hand, a matrix that indexed columns by opposing teams would be less sparse, more manageable, and probably as predictive as one could hope for.

Identifying the correct data

The initial plan for the data itself was to choose several real-life hockey statistics and trends for each player, including some of the new proxies for puck possession, like Corsi and Fenwick, and combine them into a single number—player fitness score—that would ostensibly predict fantasy success. Toward this end, the site stats.hockeyanalysis.com maintains by far the most comprehensive information, going back nearly a decade, and offering the ability to examine data from highly specific scenarios—for example, in 5-on-5 situations while leading at home. This detailed approach might be more appropriate for pro scouts or traditional gamblers betting on teams to win or lose. But it offers little information that would allow us to approximate a player’s expected performance against specific teams.
It soon became clear that it makes little sense to attempt to predict fantasy points with something other than fantasy points themselves. The model would need to build its approximation of each player’s performance against each opposing teams from scratch. The only way to do this is to iterate through game logs, night by night, player by player.

**Scraping**

ESPN offers game logs for the recently-completed season as well as several recent seasons, so this seemed a natural place to start. The first scraping utility implemented (not used in the final version), called `scrape.py`, collected statistics for every player at every position for the entire season, sorted by opponent. It also reproduced the game logs for each player. Several Python libraries were involved in scraping data from the game logs at ESPN:

- `urllib.parse`—This library can break down a URL into constituent tags to help construct successive URLs for teams, players, and game logs.
- `urllib2`—Follow links and open web pages.
- `BeautifulSoup`—After a web page is “opened” by urllib2, the functions from this library parse a web page’s HTML and capture the text rendered on the screen.
- `openpyxl`—Creates, reads and writes Excel worksheets inside workbooks to store the output from the scraping.

The spreadsheet tracked real-life statistics and calculated the number of fantasy points that a player earned each night, tracking each player’s fantasy performance against each opponent. Each night, FanDuel awards points to skaters according to the following scheme:

- **Goal:** 3 points
- **Assist:** 2 points
- **On-target shot on goal:** 0.4 points
- **Being on the ice at even strength when one’s own team scores:** 1 point
- **Being on the ice at even strength when the opposing team scores:** -1 point
- **Bonus for scoring or assisting on a goal on a power play:** 0.5 points

And for goalies, points are awarded for the following events:

- **Win:** 3 points
- **Goals against:** -3 points
- **Saves:** 0.2 points
- **Shutout:** 0.2 points

The ESPN data was cumbersome to collect, but the fantasy scores calculated by the script were accurate. But, historical FanDuel “salary” information would still need to be located on another site, scraped with a separate utility, and imported into the rows corresponding to the correct players in the same spreadsheet.

The FantasyCruncher site—specifically, its Lineup Rewind feature—turned out to be the perfect source for this information. What is more, each Rewind page corresponds to a
single date of the season, and it includes a table of every player that was available to draft on that date. Each player row includes, among other figures, the player’s name, position, his own team, the opposing team, his FanDuel salary, and the number of fantasy points he scored on that day. All that information, combined with the date, is everything needed to build the matrix of players and track their performance. At this point, it had become clear that “real-life” statistics would not be included in the model. Since FanDuel points and salary are already included in the FantasyCruncher data, there is no need to re-invent the wheel and re-calculate anything from ESPN data. This rendered the ESPN scraping completely useless.

Each FantasyCruncher Rewind page has a “copy data” button that copies all the data for that day onto the clipboard. This opened up the possibility that the scraping would not need to rely on generating and parsing an HTML tree, an approach that would break if any of the web pages were constructed or formatted even slightly differently. Instead, once the web page is rendered, the data can be read all at once with the click of a button, moved from the clipboard to a dataframe, and copied to an Excel sheet. The new scraping utility, getFCinfo.py, iterates through each FantasyCruncher Rewind page and performs the following:

- Using the webdriver module from the Python library selenium, open and load the FantasyCruncher page in the Firefox browser.
- Using webdriver, click to close the login prompt window (one need not login to access the information).
- Using webdriver, click the “copy data” button on the page, which copies the entire table of FanDuel information for that date onto the clipboard.
- Using the Pandas data analysis Python library, read the contents of the clipboard to a Pandas dataframe.
- Using the Pandas module ExcelWriter, write the table to an Excel worksheet named by the date, inside a workbook named by the month (see next page).
The `getFCinfo.py` code is far cleaner, shorter, and less prone to subtle HTML-related bugs than the original ESPN scraping utility. The output is also better-organized and easier to read.

Additionally, the FantasyCruncher data provided other information that smoothed out some bugs in the model. Certain players are injured, or otherwise scratched from the lineup by their coaches, on any given night. In the same table that was scraped by `getFCinfo.py`, FantasyCruncher provides nearly real-time updates to players' injury/scratch status up until the time the puck drops. Before the model chooses players, it can run through the worksheet and, if the player is listed as injured or scratched from the lineup, set the salary for that day to $999,999.00, ensuring that the model can never select an injured player who will (most likely) score no fantasy points. The script `cleanFC.py` performs this task, converts players' names to the convention LASTNAME::FIRSTNAME, and, using the CSV Python library, writes the “cleaned” data to files named for each date of
the season, with strictly increasing numbers for later dates: 20151007.csv, 20151008.csv, etc.

**The Matrix Problem**

**Building the matrix**

For any approach to building the matrix, the information logged by the FantasyCruncher scraping utilities, for each position, must be read into a game log (covering every player’s games up to a certain point in the season). Additionally, we must create dictionaries of players and teams covering all the games in that log. In analyze.py, the function `gameLog` builds the game log. Information from each line in each of the `<date>.csv` files becomes an entry in the game log.

The game log—referred to as `d` by most functions—contains one entry for each game played by every player at a given position, for the entire season. Each game entry is formatted as a list to include `[name, position, team, opponent, salary, fantasy score, date]`. For example:

- ['RASK::TUUKKA', 'G', 'BOS', 'CBJ', '8600.0', '7.6', '20160216']
- ['LEHTONEN::KARI', 'G', 'DAL', 'STL', '8300.0', '2.0', '20160216']
- ['ANDERSEN::FREDERIK', 'G', 'ANA', 'EDM', '8200.0', '6.8', '20160216']
  ...
- ['RASK::TUUKKA', 'G', 'BOS', 'NSH', '8800.0', '2.4', '20160218']
- ['JONES::MARTIN', 'G', 'SJ', 'FLA', '8400.0', '5.8', '20160218']
- ['MILLER::RYAN', 'G', 'VAN', 'ANA', '8200.0', '1.6', '20160218']
  ...

Then, `createDictionary` iterates through the game log to construct dictionaries of teams and players covering every game in that log. The player dictionaries are formatted as follows:

```
{'SMITH::MIKE': 0, 'CRAWFORD::COREY': 1, 'HAMMOND::ANDREW': 2,
 ..., 'HOLTBY::BRADEN': 76, 'CONDON::MIKE': 17}
```

And the team dictionaries are formatted as follows:

```
{'MIN': 0, 'TOR': 1, 'CAR': 2, 'BOS': 3, 'DET': 4, ..., 'EDM': 26, 'SJ': 27, 'ARI': 28, 'OTT': 29}
```
The first, straightforward approach to building the matrix is to simply return the average historical performance for each player, at each position, against each team. The function `builda(d, playerdict, teamdict)` in `analyze.py` does this. For the position specified by the user—either goalie, defenseman, centerman, left wing, or right wing—`builda` returns a 2-tuple containing two NumPy matrices, `A` and `Acount`, each with dimensions `num_players by num_teams`. The average fantasy performance of player `i` against opponent `j` is given by `A[i,j]` (the total fantasy points he has earned against that opponent for the season so far) divided by `Acount[i,j]` (the number of times he has faced that opponent).

A more sophisticated approach to estimating fantasy value goes beyond looking at historical average. Every player has some inherent, underlying strength, independent of the team he faces on a particular night. Likewise, every team has some intrinsic baseline level of strength. The outcome of a game—and more importantly, the fantasy performance of the individual players on both teams—will be influenced by the individual strengths of players on both sides, and also by the collective strength of the opposing team.\(^1\) It still makes sense to envision, for each position, a `num_players by num_teams` matrix. But instead of explicitly generating and returning the entire matrix, the `coorddescents` function implements randomized coordinate descent to approximate the singular value decomposition (SVD) for the matrix that will contain an entry for each player's anticipated performance against each opponent. Given as inputs a game log, a player dictionary, a team dictionary, and some integer `k` (the rank of the matrix to be approximated), we generate the following:

- The `num_players`-length vector `va`, containing values corresponding to each player's strength;
- The `num_teams`-length vector `ub`, containing values corresponding to each team's strength;
- The `num_players by k` matrix `UU`, which stores player attributes;
- The `num_teams by k` matrix `VV`, which stores team attributes;
- The number `alpha`, which is the mean of all scores.

Given these outputs, the matrix entry `GUESS_{ij}` for player `i` and team `j` is approximated as:

\[
va[i] + ub[j] + alpha + \text{numpy.sum}(UU[i,:] * VV[j,:])
\]

If we are trying to estimate the performance of the players who are eligible for FanDuel drafting on date `X`, we seek vectors `va` and `ub`—which approximate player strength and team weakness, respectively—and `k`-rank matrices `UU` and `VV`—which store player

---

1 We are less concerned with the converse of that—that is, with how well the model can approximate a team's performance against an individual—since in daily fantasy, individual players from different teams are drafted.
attributes and team attributes, respectively—such that, for each player $i$ and opposing team $j$, tend to minimize the squared error:

$$E_{ij} = (GUESS_{ij} - ACTUAL_{ij})^2$$

where $GUESS_{ij}$ is our estimate of a player’s performance against his opponent given how he has performed through date $X - 1$, and $ACTUAL_{ij}$ is his observed FanDuel performance in each of the games he has already played against that opponent.

On any given day in the season, when a user seeks to pick a fantasy lineup, he would “train” the model with all the previous games in the season. This “training” is performed by either the `builda` or the `coorddescent` functions, described above. Then, the user would approximate the anticipated fantasy value for each of the players available to draft that night. The file `cleanercode.py` demonstrates the model being trained on each date during the final 42 days of the season, when a great many games have already been played. After this training is complete, the optimization code, discussed below, chooses a roster given the model's estimates of how each player that night will perform.

Somewhat surprisingly, higher $k$ was not correlated with lower squared error between the estimate and the actual fantasy outcomes on most days. The naïve approach of anticipating fantasy value based on historical averages with `builda`—without accounting for the underlying strengths of teams and players—performed only slightly worse than approximating the singular value decomposition with low $k$. This problem, and possible solutions for it, are addressed in the “Further Work” section.

The Optimal Roster

Given the ability to estimate how each player will perform against each team on some date $X$—based on both his own performance and that of his opponent through date $X - 1$—we are left with the task of picking the optimal fantasy lineup for date $X$. A FanDuel fantasy lineup consists of:

- One goalie (G)
- Two defenders (D)
- Two centermen (C)
- Two left wing forwards (LW)
- Two right wing forwards (RW)

Each individual player costs a certain amount between $3,000 and $10,000, and the total cost of the roster cannot exceed $55,000. Better players tend to be more expensive, so it would be impossible to draft an All-Star at every position. The goal is to maximize total anticipated fantasy performance, across all positions, given the constraint that the total cost of all nine players may not exceed the “salary cap.”

---

2 When relying on historical averages, it makes little sense to use a training set much smaller than half the season. A matrix completion approach can tolerate greater sparsity.
**Integer linear programming**

The problem of selecting a FanDuel roster can be stated as a mixed-integer linear programming problem. It is also equivalent to a version of the bounded knapsack problem, where the user is charged with filling a knapsack with the most valuable combination of items, given that:

- Each item weighs a certain amount;
- The knapsack has a maximum capacity weight $C$;
- There are several categories of items, and the user must choose some fixed amount from each category (an amount that can vary by category).

Although accurate, the knapsack and linear programming analogies are clinical and removed from everyday reality. The problem of planning a diet to maximize calories—given food group requirements and a budget—is more illustrative and relatable.

The optimization formulation of the diet problem—“What is the least expensive, nutritionally-valid selection of food items for the diet?”—is NP-hard. No known algorithm exists for verifying that a solution is optimal, and approximation algorithms are beyond the scope of this project. But various software implementations of those approximation algorithms, like Gurobi, arrive at good solutions with high probability. Gurobi offers a free academic license for its full suite of optimization tools, which are used in this project. Gurobi can be used from its own interactive shell, but all its modeling features are also implemented in the Python package gurobipy, which comes with the Gurobi installation and can be run in the user’s preferred Python environment.

Gurobi’s website describes an implementation of the diet problem that seeks to minimize the total cost of the food items chosen given acceptable ranges for each nutrient (calories, fat, protein, sodium, carbohydrates). When it generates a solution, Gurobi assigns each food item variable (milk, pizza, chicken, hot dog, salad, ice cream, fries, pasta) a non-negative integer value, indicating how many times they are to be purchased in the optimal diet.

This diet framework can be readily translated into a description of the daily fantasy problem. Each player available to draft on a particular day is analogous to one of the food item variables, except unlike the foods, he must be assigned a binary value (e.g., one can buy four or five or eight gallons of milk on a day, but only draft Anze Kopitar once). On a particular day, each player-instance has the following fields:

- Name
- Anticipated fantasy value
- Salary
- Is goalie? (0 or 1)
- Is defensemen? (0 or 1)
- Is centerman? (0 or 1)
- Is left wing? (0 or 1)
• Is right wing? (0 or 1)
• Weight (always equals 1, to ensure we draft exactly 9 players each day)

The constraints in this optimization problem serve four purposes:

1. Restricting the roster to exactly nine players
   a. $9 \leq \text{total weight} \leq 9$

2. Restricting the roster to exactly one goalie and two skaters at each position
   a. $1 \leq \text{total goalies} \leq 1$
   b. $2 \leq \text{total defenseman} \leq 2$
   c. $2 \leq \text{total centermen} \leq 2$
   d. $2 \leq \text{total left wings} \leq 2$
   e. $2 \leq \text{total right wings} \leq 2$

3. Ensuring that no variable (player) may be assigned a value other than 0 or 1
   a. Draft[player] == 0 or Draft[player] == 1

4. Ensuring that the total cost of the roster does not exceed the salary cap
   a. $0 \leq \text{total salary} \leq 55,000$

Once the variables and constraints have been specified, the call to optimize() returns an assignment of binary values to each player, instructing the user whether or not to draft the player on that day.
After being tested on each day of NHL games in March, the optimization yielded mixed results. Usually, the roster chosen by the model placed around the 75th percentile of fantasy scores for the day, scoring a number of fantasy points in the mid-20s. To win the lowest level of profit ($5 prize on a $2 entry), a roster must be in roughly the 85th percentile or better, which typically coincides with a fantasy score in the mid- to high-30s. The rosters improved toward the end of the testing set—games in April—with many scores in the mid 30s (hovering around the 85th percentile) and a high score of 56.2 fantasy points:

![Image of bash output](image)

This upward trajectory is unsurprising. However, it would be beneficial to incorporate noise into the anticipated fantasy values for each player-team combination in the interest of generating multiple lineups each night. This is addressed further below.

**Future work**

The daily fantasy discussion pages on Reddit reveal a great demand for more-comprehensive and better-formatted fantasy (and real-life) data, as well as more systematic methods for choosing daily rosters. I strongly believe that the framework described and implemented here has the potential to fill this niche at least until the daily fantasy hockey market becomes more efficient. Armed with the entire 2016-2017 NHL season as a training set, this model will begin picking daily fantasy rosters at the start of the
upcoming season in October 2016. For this to be effective, the following improvements should be made this summer.

**Inclusion of multiple seasons**

The sample size for all future training sets could be increased by widening the net of the scraping utilities. Instead of using performance in at most 3 or 4 games against a specific opponent to judge a player’s future performance, it would be helpful to consider a player’s entire career, which for veterans includes dozens of games against every opponent. It would make sense for this approach to be applied to a model that weights historical performance more heavily as opposed to “completing” the matrix to approximate the strength of teams and players.

**More intelligent error handling**

Very little error handling is implemented in either the scraping or the modeling codes. If certain values cannot be located or calculated, the default behavior is usually to set them equal to zero, which may not lead to optimal behavior. For example, if the model encounters a player in the test set that was not included in the training set, a better heuristic, instead of setting his expected performance to zero, would be to look at the expected performance of his linemates, which, from a hockey standpoint, can be expected to correlate with his own performance.

**Better approximation of fantasy value**

Initially, the randomized coordinate descent method to approximate player and team strength to complete the matrix initially predicted fantasy value only within 1.6 absolute deviations. As $k$ increased, this generally became worse. To improve on this, it would be helpful to try to alternative matrix completion methods, such as matrix factorization with a trace norm penalty, or to simply rely on aggregate historical performance with a very large sample size. It would also be worth combining the two methods and trying matrix completion on a matrix of long-term historical averages (which would be sparse if the data stretched back several years). Yet another idea, which was mentioned above, is the introduction of noise into the matrix of computed anticipated fantasy values. A small, random, zero-mean epsilon would be added to each anticipated fantasy value, so that $GUESS_{ij}$ would be computed as:

$$\varepsilon + va[i] + ub[j] + \alpha + \text{numpy.sum}(UU[i,:] * VV[j,:])$$

The introduction of such noise would allow the model to generate several plausible lineups for the day, evaluate them based on some criteria, and enter one or more into the Breakaway contest.
References

Articles


Documentation of Libraries

Gurobi Optimization: Python API Overview. 

Openpyxl: A Python Library to Read/Write Excel 2010 Files. 


SciPy.org: Linear Algebra. 


Stack Overflow Peer Help


“Understanding the map function.” June 11, 2012. 
