Daily Fantasy Basketball Picker

Automated Decision Systems
Dec 20 2015
William Cai, David Hatch, Evan Green

1 Introduction

Daily Fantasy Sports, or DFS, is a new type of sports betting which has recently come into vogue. Because there is some element of skill, prosecutors have been unable to shut it down in the majority of the country. In this paper, we will detail an automated decision system that we have constructed, using various paradigms of decision making, to play Daily Fantasy sports. Section 2 gives a detailed explanation of the fantasy sports problem along with a mathematical problem statement. Section 3 breaks down our various data sources and methods for scraping them. It also explains the database we use to hold the data. Section 4 contains our methodology for cleaning the data into the final features we base the model on. Section 5 explains how are linear regression algorithm of choice works, and gives a human-friendly explanation of the model. In section 6 we talk about how we generate our lineups using our projections. Finally, in section 7 we put our money where our mouths are and bet on our lineups.

2 Problem Statement

blah blah double up, high expected value low variance blah blah

3 Getting the Data

yo

4 Cleaning the Data

hi

5 Leaning the models

The code related to this section is in the regression folder in the regress.py and predict.py files.

$$\beta_{elasticnet} = \arg\min_{\beta}(||y - X\beta||^2 + \lambda_2||\beta||^2 + \lambda_1||\beta||_1)$$

$$\beta_{LASSO} = \arg\min_{\beta}(\frac{1}{N}||y - X\beta||_2^2 + \lambda||\beta||_1)$$

$$\beta_{ridge} = \arg\min_{\beta}(||y - X\beta||^2 + \lambda||\beta||^2)$$

Figure 1: The cost functions of various regression methods in the regularized linear regression family. Note how elastic net contains the terms from both the LASSO and ridge regression.

Our linear regression algorithm of choice was the elastic net, a form of regularized linear regression. It was important to choose from the family of regularized linear regression because we have a pretty small ratio of data points to features, and we wanted the ability to add as many features as we wanted. Doing linear regression on a dataset with comparable amounts of features to data points quickly becomes an exercise in creating ridiculous models because of the danger of overfitting. Regularization combats this trend by adding in a term based on the coefficients of the model to the loss function of linear regression, which is simply the residual sum of squares.

There are several regression methods in the regularized linear regression family. The most popular of these are the LASSO, ridge regression, and elastic net. Our choice, elastic net, is an interpolation between the other two and has several properties which lend themselves to our project. For one, given a group of correlated variables, of which we have many, the elastic net will give all of them a coefficient. This is in contrast to the LASSO, which will arbitrarily select one and give it a coefficient and set the rest to 0. Using the elastic net allows us to better understand our program's explanation of it's decisions. The advantage that the elastic net has over ridge regression is that it will zero out some variables, which will allow us to say which of our features give no signal - another useful way for us to understand which features are important in predicting fantasy scores. On the other hand, ridge regression will almost never set a coefficient to zero, making interpretation more difficult.

In our code, we read in each player-game for each position from a csv file which was produced from our data cleaning. Each line of the csv is of form
$x_1, x_2, ..., x_n, y$,player-name We then load that into numpy matrices X and Y, where X is $num_{player-games}$by n and Y is $num_{player-games}$ by 1, and use numpy's elastic net, ElasticNetCV, to find a linear model β which solves

$$\beta = \arg\min_{\beta}(||y - X\beta||^2 + \lambda_2||\beta||^2 + \lambda_1||\beta||_1)$$

The λs are learned through three-fold cross validation, which splits the data into thirds and learns it on two thirds at a time, using the last third for validation. The elastic nets are made for a variety of λs, and the λs which have the lowest error are used. Furthermore, note that all features are regularized before using the elastic net to allow for the coefficients to be penalized equally. Mathematically, this means that for each feature x_i we subtract the

mean of the x_i such that

$$\sum_{j=1}^{n} x_j(i) = 0$$

Where $x_j(i)$ denotes the ith feature of the jth player-game. We then divide each $x_j(i)$ by the standard deviation of the x_is.
The result of this is a linear model for each position, which predicts how many daily fantasy points. The models, along with an interpretation, are given in files TODO TODO. They are not included here because there are over 100 features so it would take the majority of the paper.
Given these models, we can generate the expected value of fantasy points for each player by plugging their scraped data into the model. We can further generate their variance by going through their projections for previous games and seeing how much they deviate from the model. This allows us to generate our expectedvalues and residPOSITION csv files. Each row of expectedvalues is of form

Name, position, expected daily fantasy points, salary

and each row of residPOSITION is of form

Name, standard deviation

Our lineup generator will read in these files and use them to generate lineups with high expected value and low variance.

6 Generating the lineups

The code referenced in this section is found in the lineup folder, in set_lineups.py
Now that we have the projections, we need to generate some lineups to submit that we hope will make money! A formalization of this problem is as follows:

Given a list P of players, we would like to generate a set of lineups L s.t. given a lineup L_i in L, the sum of the expected values of the players of L_i is close to maximized under the constraint that the the sum of the salaries of the players of L_i do not exceed a maximum salary.

Any computer scientist will see this as a variation of the weighted knapsack problem, which is a classic NP-hard problem. However, our problem has several constraints which make it easier to solve, along with some other twists.

1)We are only interested in solving this problem for a budget of $50,000, because that's what every contest on DraftKings(a popular DFS website) uses.
2)Player salaries are all contained within the range from $3000 to $11000.
3)Players have variances along with their expected scores (we are using a gaussian model). Because we are specifically interested in double-up games, our ideal lineup is one which is high expected value and low variance, because it maximizes the chance of getting into the

money.
4)Because players can be used across many lineups, we must make sure that our set of lineups L does not depend too heavily on the performance of a small number of players. Otherwise, if those players had a bad game our bankroll would be entirely wiped out.
5)It's not entirely clear what the threshold for finishing in the money is - this varies quite a bit between days depending on how many players which are used in other people's lineups have good days.

The algorithm we came up with has three separate stages which address these issues.

6.1 Thresholding

The code that does the process described in this subsection is located at the beginning of the main() submodule in set_lineup.py.
Before we begin generating lineups, we need to figure out our threshold for expected value for the lineups (we will reject lineups whose expected value is below this). We decide to do this by generating lineups using our lineup generator (discussed in the next section, section 6.2) and taking the minimum expected value of lineups in the top X%.

X was tuned using past data by selecting an X such that on average, our threshold was above the money. Note that we can't just lower X to arbitrarily low numbers, because then we may be unable to find lineups and they will heavily depend on the same players. Furthermore, by getting some breathing room with a slightly higher X, we are able to focus more on variance, increasing the chances that our lineups make money. We will call this threshold T.

6.2 Lineup Generator

The code described in this subsection is in the get_lineup() submodule in set_lineup.py
Given this threshold T, we want to write a submodule which can generate a single lineup with an expected value above T but with a budget under our salary cap. We use a greedy randomized algorithm:

1)Randomize the order of the positions to be picked out of PG, SG, SF, PF, C, G, F, UTIL. This will allow us to get a healthy selection of players.
2)For each position, randomly select a player, where each player has $\frac{expectedvalue}{salary}$ chance of being chosen, making sure we have enough money left to fill out our roster. Then adjust the remaining salary.
3)After having chosen a player for each position, we will have some amount of money left over. Re-randomize the positions.
4)For each position, pick a random player according to the same probability distribution. If that player has higher expected value and fits our salary constraints, replace the current player for that position with him and adjust the salary accordingly.
5)Repeat steps 3-4 until there are no more possible improvements.

Through experimentation, we found that this typically generates a lineup that uses almost all of the salary and has close to maximum expected-value/salary ratio, meaning that we have succeeded in generating lineups with expected value above T. Note that this algorithm works well because of the small range in salaries (constraints 1 and 2 in our list).

6.3 List of Lineups generator

The code which contains the algorithm described in this section is in the main() submodule in set_lineup.py

Now that we have a tool for generating a single lineup, we would like to be able to generate a list of lineups. However, as mentioned in our list of constraints, we want to make sure the list is not too dependent on certain players. We invented a novel algorithm to solve this problem, which we call the box algorithm (it is a flavor of Monte Carlo):

1) Fill a box with n lineups, where n is the number of lineups you would like to generate.
2) Shake the box k times, where a box shake is defined as follows:
a) For each player, assign the player a score by sampling a number from a normal distribution centered at his expected value with standard deviation equal to his past standard deviation.
b) For each lineup in the box, add the scores of the players. Compute the median score.
c) For each lineup in the box below the median score, flip a coin. If it's heads, remove the lineup from the box. d) For each lineup that was removed from the box, add a new lineup from get_lineup into the box. e) If a lineup has survived for 10 box shakes, but was not in the initial box lineup, save it.
After k shakes, all the saved lineups are those that we would submit. We order these in the probability that the lineup will score below T using the normal distribution formula, where both the expected value and variance of the lineup is simply the sum of those values of its players (this is a well known fact for normal variables).
The rationale behind the box shaking is that if a single player has a bad game, many of the lineups depending on it will be removed from the box, which will remove dependencies on certain players.

7 Taking it out for a spin!

hi