Not all shots are created equal

Adjusting NHL Corsi statistics for quality of opponent

DP Rooney

February 6, 2016

Introduction and Background

There has been considerable recent growth in advanced statistics for professional hockey. Besides the traditional stats such as goals, assists, and shots, analysts have expanded their approach by considering so-called Fenwick stats (shots on goal + shots missed) and Corsi statistics (shots on goal + shots missed + shots blocked = shot attempts). Additionally, instead of looking at shots a player himself attempts, one considers his on-ice statistics. That is, the events that occur when he is on the ice, both for his team, and against his team. One can define the "Corsi-for percentage", or CF%, as follows:

$$\text{CF\%(player X)} \;=\; \frac{\text{No. shot attempts for X's team} \mid \text{X on-ice}}{\text{No. shot attempts for both teams} \mid \text{X on-ice}}$$

This stat evaluates how a player contributes to producing shot attempts for his team while preventing shot attempts against his team. Some of the attractive features of this statistic are:

1. It combines offensive and defensive merit, so that players with flashy scoring and frequent defensive lapses are not over-rated.
2. It credits players that contribute indirectly to an offensive play, such as with canny passes from the defensive zone, tight fore-checking, and smart spacing.
3. It eliminates some of the randomness inherent to turning shot attempts into goals.
4. The data set is larger (there are around twice as many shot attempts as shots on goal per game).

One major drawback with this stat is that it does not take into account the quality of opposition. A fundamental phenomenon in hockey is line-matching. Each team has four different forward-lines (and three defense-pairings) that generally differ in abilities, and coaches will strategize as how to optimize the match-ups between opposing lines. For example, a good defensive forward line will often be matched against the opposition's best offensive line. Therefore, comparing the CF% of a first-line and a fourth-line player is unfair to the better player, since he has to work harder to generate a shot attempt, and prevent opposing shot attempts.

For example, this reddit post points out that Jake Virtanen, a young, raw prospect for the Vancouver Canucks, has a better CF% than Jonathan Toews, who is seen as one of the best players in the league:

https://www.reddit.com/r/canucks/comments/4omaqp/why_jake_virtanen_is_better_than_jonathan_toews/ (https://www.reddit.com/r/canucks/comments/4omaqp/why_jake_virtanen_is_better_than_jonathan_toews/)

While Virtanen may develop into one of the best players in the league, it is unlikely that his performance was on par with Toews' last season.

The goal of my project is to develop a model that is more nuanced. Instead of modeling the ratio of shot-attempts as a function of a single player, we will consider the probability of shot-attempts as a function of all players on the ice. In other words, we want to model the probability

$$P(\text{ next SA is for home-team} \mid \text{home-players } X_1, \ldots X_6 \text{ on-ice; away-players } Y_1, \ldots, Y_6 \text{ on-ice })$$

My prospective client would be the management of a hockey team that wants to evaluate players in ways that go past the common wisdom. It may want to track the performance of its own players, and also evaluate players that are becoming free agents in the off-season. A predictive model that describes shot-attempt probability relative to opposing players would allow a team to directly compare players and assess their relative value.

Some practical issues:

- I will be considering shot attempts not just in 5-on-5 situations, but also for power play opportunities. Shot attempt difficulty is radically different for the latter situations, but these will be factored into my models. Penalty shots of course, both in-game and in shootouts, are excluded.
- Corsi stats are sometimes separated by "zone starts". Defensive players who usually start their shifts in their own end will be hurt considerably by this. I was unable to obtain the data that separated shot attempts by zone start unfortunately, so my models ignore this factor.
- Goalies will be excluded. Some goalies are better passers and rebound-handlers than others and arguably would affect shot attempts, but I decided to ignore this aspect. 6-on-5 situations were the goalie is pulled will be considered power play situations.

Scraping and Wrangling

The data was scraped from three types of game reports from the NHL: game rosters, event summaries, and play-by-play reports. The URL's for the first game are here:

http://www.nhl.com/scores/htmlreports/20152106/RO020001.HTM
(http://www.nhl.com/scores/htmlreports/20152106/RO020001.HTM)

http://www.nhl.com/scores/htmlreports/20152106/ES020001.HTM
(http://www.nhl.com/scores/htmlreports/20152106/ES020001.HTM)

http://www.nhl.com/scores/htmlreports/20152106/PL020001.HTM
(http://www.nhl.com/scores/htmlreports/20152106/PL020001.HTM)

The two-letter codes RO, ES and PL indicate the report type, and the last four digits of the six-digit sequences indicate the game number (the first two digits distinguish pre-season (01), regular season (02) and post-season (03)). There were 1230 game in the 2015-2016 regular season (30 teams and 82 games each). These reports are all in pure HTML (no CSS or JSON), so I had to go through a lot of nested `<table>`'s to get the necessary data.

I have decided not to include any code in this report, as I used quite a lot of Python. In the appendix I have listed all the files with Python code contained in the github repository:

https://github.com/darraghrooney/Springboard_Capstone
(https://github.com/darraghrooney/Springboard_Capstone)

In this section I will give an overview of the files I used to form my data sets.

In [1]:
```
import os
import numpy as np
import scipy.stats as sps
import pandas as pd

from sklearn.linear_model import LinearRegression, LogisticRegression

%matplotlib
%matplotlib inline

import matplotlib.pyplot as plt
import plotting.plotting as myplot
```

```
Using matplotlib backend: TkAgg
```

The `scraping` folder contains six files for scraping the data:

1. The file `roster_scrape.py` contains code for scraping the roster reports. It includes a class `RosterParse` which extracts the forty players that dressed (eighteen players and two goalies for each team). This includes their team, the player name, the position (center, left wing, right wing, defense or goalie) and their jersey number. A second class, `RosterBuilder` assembled rosters for all 1230 games and saved them in a 49,200 line file `Big_Roster.csv`.
2. The file `directory_build.py` contains code for consolidating the big roster into a player directory, saved in `Directory.csv`. It includes a class `SalaryParse` that extracts the salary for each player from the web-site www.capfriendly.com. The class `DirectoryBuilder` constructs the player directory and adds the salary information. The player directory contains 1,011 players, of which 111 are goalies.
3. The file `salary_fill.py` contains code for filling in some missing salary information that was not immediately available from CapFriendly. 25 players did not have 2015-2016 salaries available, so I used their 2016-2017 salaries. Salaries are only being used to estimate perception of player quality, so using salary from two different years is not such a big deal.
4. The file `report_downloader.py` contains code for downloading the play-by-play and event summary reports. I used this because I wanted to work on scraping while off-line.
5. Besides salary information, I also wanted to use time-on-ice (TOI) information. The file `es_scrape.py` includes code for doing this. It includes a class `EventParse` that looks at the event-summary reports and extracts TOI for each player for each game, and a class `TOIBuilder` which totals season TOI and adds it to the player directory. It also includes a function `Dir_process` which adds a column `paTOI/G` to the directory. This statistic is the per-game TOI for each player, multiplied by 0.75 if the player is a defensemen. Because there are 3 defense lines to 4 forward lines, defensemen play approximately 4/3 as much, so TOI/G should be adjusted.
6. The file `attempt_scrape.py` contains code for extracting the shot-attempt data for each game and saving it as a table. These tables include 14 columns: the event (shot, missed shot, goal, blocked shot), a boolean stating whether the attempt was for the home teams, the jersey numbers for the six home player and six away players (some of which would be listed as `None` if there were penalties). Note: these tables are not in the repo, as I combined them in the wrangling process and discarded them.

Here is a sample of the player directory in `Directory.csv`. Note goalies do not have a TOI, so there is a NaN recorded there:

```
In [5]: df = pd.read_csv('data/Directory.csv')
        df.head()
```

Out[5]:

	ID	Player	Position	Games Dressed	Salary	TOI	paTOI/G
0	1	AARON DELL	G	2	575000	NaN	NaN
1	2	AARON EKBLAD	D	78	925000	1690.816667	16.257853
2	3	AARON NESS	D	8	575000	99.100000	9.290625
3	4	ADAM CLENDENING	D	29	761250	429.083333	11.096983
4	5	ADAM CRACKNELL	R	52	575000	636.016667	12.231090

5 rows × 7 columns

There is still work to be done however. In a separate folder `wrangling`, I have three files for further data handling. I'll talk about the third in the next section. The other two are:

1. `attempt_manager.py` is responsible for consolidating the shot-attempt data into one data set. It contains a class `attempt_manager` that does this. It saves a number of objects into the file `Attempts.npz`. First and foremost, it contains a 0-1 sparse matrix (a `csc_matrix` object from the module `scipy.sparse`) with 1800 columns and 136,530 rows. Each row represents one shot attempt, and each column represents one player, and each element is `True` if and only if that player was on the ice for that shot attempt. There are 1800 columns because there are 900 players (we exclude the goalies) and we consider a player at-home to be distinct from a player away-from-home, so columns 1-900 are home players and 901-1800 away players.

   The sparse matrix is our main data set, but some other objects are also contained in the file:

   - the list of non-goalie names so that we can match the columns to players
   - game counts: the number of shot attempts in each game
   - a list of indices indicating which of the shot attempts was for the home team. This is our indicator variable.
   - a list of attempt type (whether an attempt was a goal, shot, missed shot, or blocked shot).
   - four lists indicating the average salary and average playing time of the players on-ice for the home and away sides

     The class also contains a method `compute_Corsi` which computes the season CF for any player.
2. The file `summary_manager.py` is for looking at attempt data on a game-by-game basis. It includes a class `summary_manager` which adds the goals, shots, missed shots, blocked shots and total shot attempts for each team in each game and saves it in the file `Summary.csv` as a 1230 by 10 table.

Here is a sample of the Summary data:

In [6]:
```
df = pd.read_csv('data/Summary.csv')
df.head()
```

Out[6]:

	Home goals	Home shots	Home misses	Home blocks	Home Corsi	Away goals	Away shots	Away misses	Away blocks	Away Corsi
0	1	36	14	15	66	3	26	16	9	54
1	2	32	14	15	63	3	24	11	3	41
2	1	29	11	13	54	5	39	11	10	65
3	1	19	11	13	44	5	27	12	10	54
4	2	29	19	13	63	6	26	9	9	50

5 rows × 10 columns

A first look at the data

First note that we have lots of data to work with. There were 136,530 shot attempts in the 2015-2016 season and 900 players (i.e. 1800 features):

In [7]:
```
import wrangling.attempt_manager as am

AM = am.attempt_manager()
AM.Load()
print 'Number of non-goalies that dressed in 2015-16:
{}'.format(len(AM.NGs))

t_dict = {'G': 'Goals', 'S': 'Shots saved', 'M': 'Shot missed', 'B':
'Shots blocked'}
for k in t_dict.keys():
    print t_dict[k] + ': ' + str(AM.attempt_type.count(k))
print 'Total shot attempts: {}'.format( AM.no_att )
```

```
Number of non-goalies that dressed in 2015-16: 900
Shots saved: 66601
Shots blocked: 34845
Shot missed: 28519
Goals: 6565
Total shot attempts: 136530
```

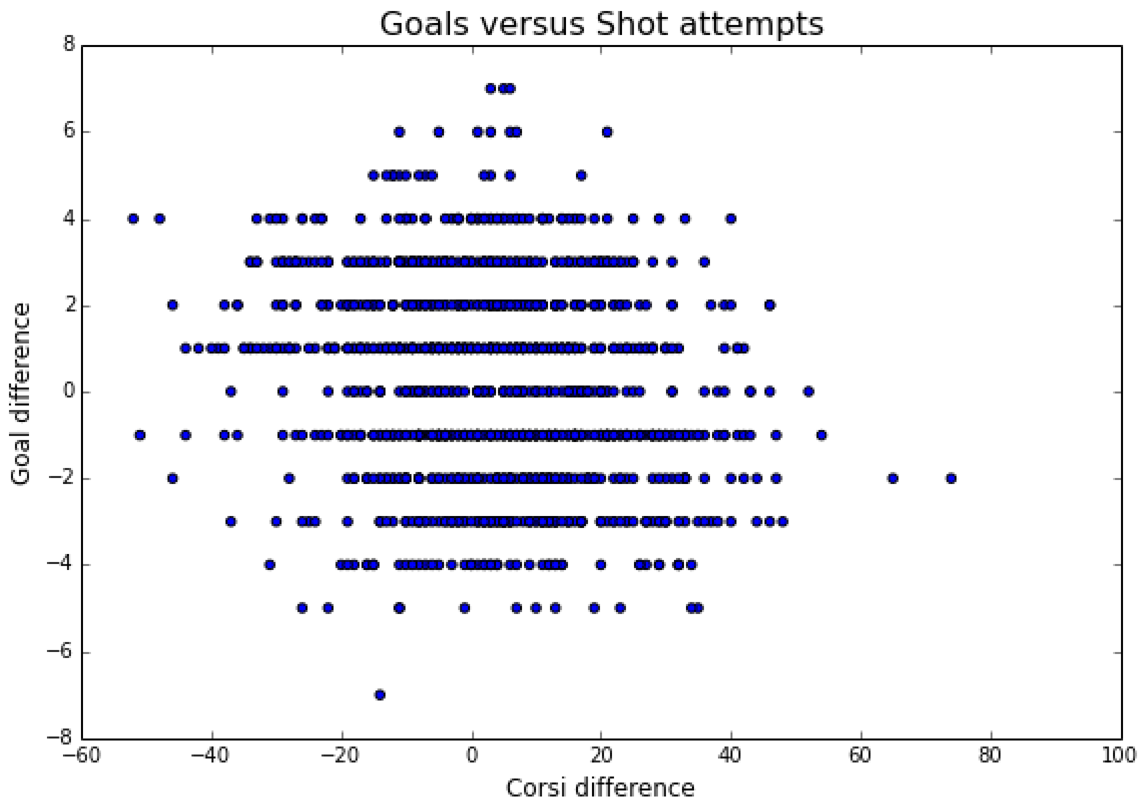
We can see there is a clear home-ice advantage, in terms of goals and shot attempts:

```
In [8]: print 'Home goals: {}'.format( sum( pd.Series( AM.attempt_type)\
                [AM.home_indices] == 'G') )
        print 'Away goals: {}'.format(AM.attempt_type.count('G') - \
                sum( (pd.Series( AM.attempt_type)[AM.home_indices] == 'G') ))

        print 'Home shot attempts: {}'.format(len(AM.home_indices))
        print 'Away shot attempts: {}'.format( AM.no_att - len(AM.home_indice
        s) )
```

```
Home goals: 3404
Away goals: 3161
Home shot attempts: 70468
Away shot attempts: 66062
```

Goals, not shot attempts, are what win games. So we would should examine the correlation between them. As it turns out, the correlation is actually slightly negative!

```
In [9]: import wrangling.summary_manager as sm

        SM = sm.summary_manager()
        SM.Load()
        goal_diff = SM.summary['Home goals'] - SM.summary['Away goals']
        Corsi_diff = SM.summary['Home Corsi'] - SM.summary['Away Corsi']

        fig = plt.figure()
        ax = fig.add_axes([.1,.1,1.2,1.2])
        plt.xlabel('Corsi difference', fontsize = 12)
        plt.ylabel('Goal difference', fontsize = 12)
        plt.title('Goals versus Shot attempts', fontsize = 16)
        ax.scatter(Corsi_diff, goal_diff)
        plt.show()
```

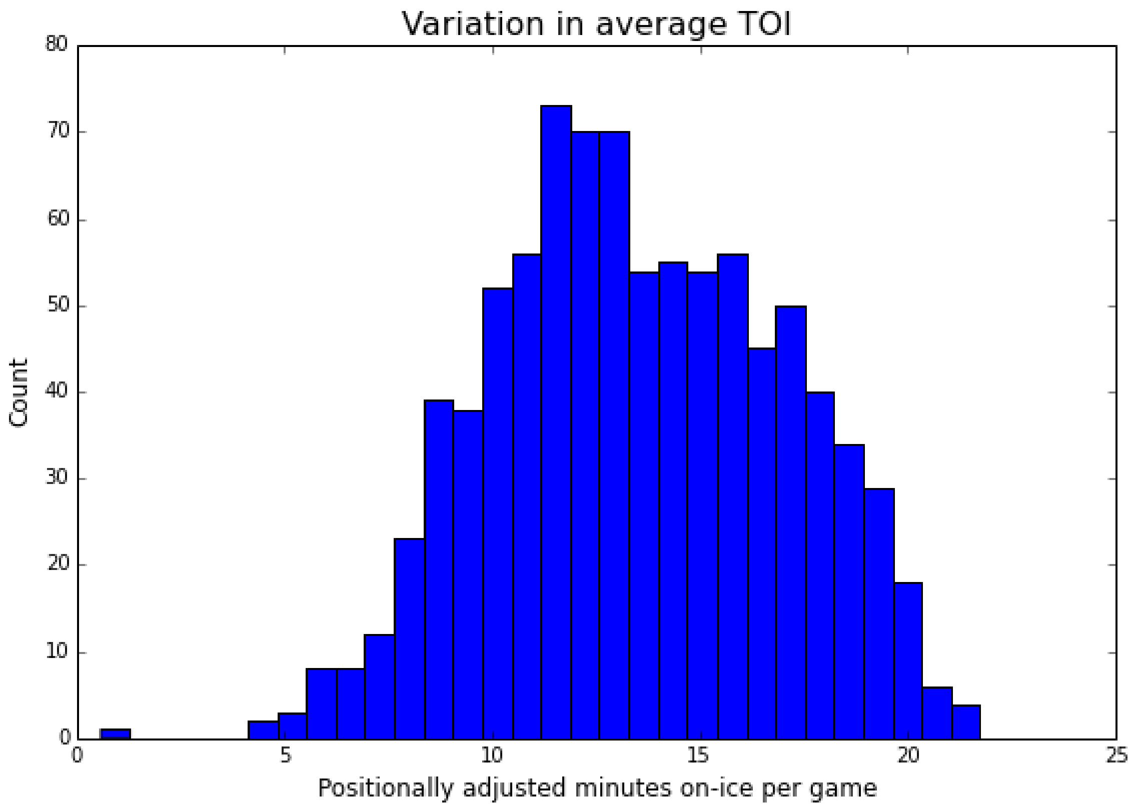


```
In [10]: LR = LinearRegression()
         LR.fit( Corsi_diff.reshape(-1,1),goal_diff.reshape(-1,1))
         print 'Regression coefficient: {}'.format(LR.coef_[0,0])

         Regression coefficient: -0.026606973457
```

There are ways to explain this. First of all, goalie skill is of course a factor in converting shot attempts into goals. More importantly, teams that score early and obtain a lead often sit back and allow their opponent to take poor-quality shots. A more nuanced analysis would take a deeper look at this effect. We are more concerned with using the stat as a indicator of player performance, not of team performance, so we shall move on.

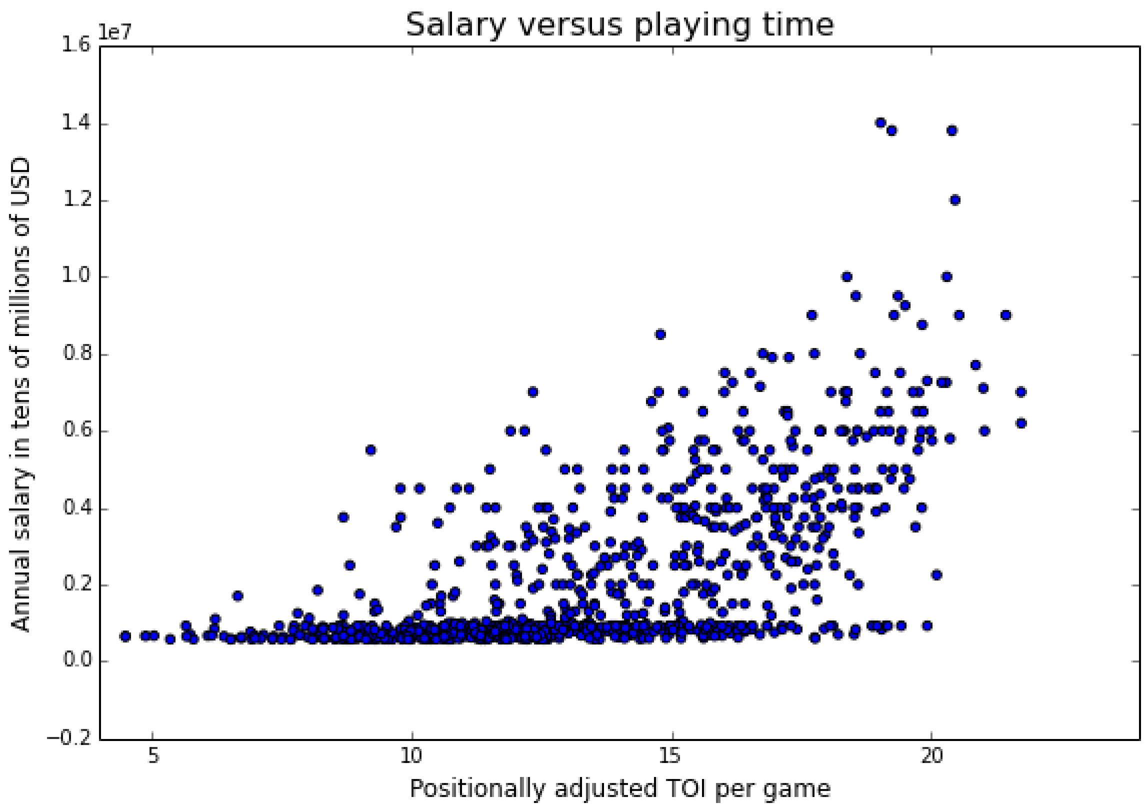
We can use two proxy statistics to assess 'quality of opponent' (or at least the perception of quality): player salary and average playing time. Salary is a little problematic, because younger players are certainly underpaid, and the production of older players can fall off after getting a fat contract. So salary and quality do not correlate as much as we would like. The latter statistic is probably better, because a coach can respond quickly to changes in playing quality, and because there is a decent spread in player time on ice (TOI):

In [11]:
```
players = pd.read_csv('data/Directory.csv')
players=players[ ~players.TOI.isnull()]
fig = plt.figure()
ax = fig.add_axes([.1,.1,1.2,1.2])

plt.hist( list(players['paTOI/G']),bins=30)
plt.xlabel('Positionally adjusted minutes on-ice per game', fontsize
=12)
plt.ylabel('Count', fontsize =12)
plt.title('Variation in average TOI', fontsize = 16)
plt.show()
```



If playing time and salary are both good measures of player quality, we should expect them to correlate:

In [12]:
```
fig = plt.figure()
ax = fig.add_axes([.1,.1,1.2,1.2])
plt.scatter(players['paTOI/G'],players['Salary'])
plt.xlim(4,24)
plt.xlabel('Positionally adjusted TOI per game', fontsize =12)
plt.ylabel('Annual salary in tens of millions of USD', fontsize =12)
plt.title('Salary versus playing time', fontsize = 16)
plt.show()
```

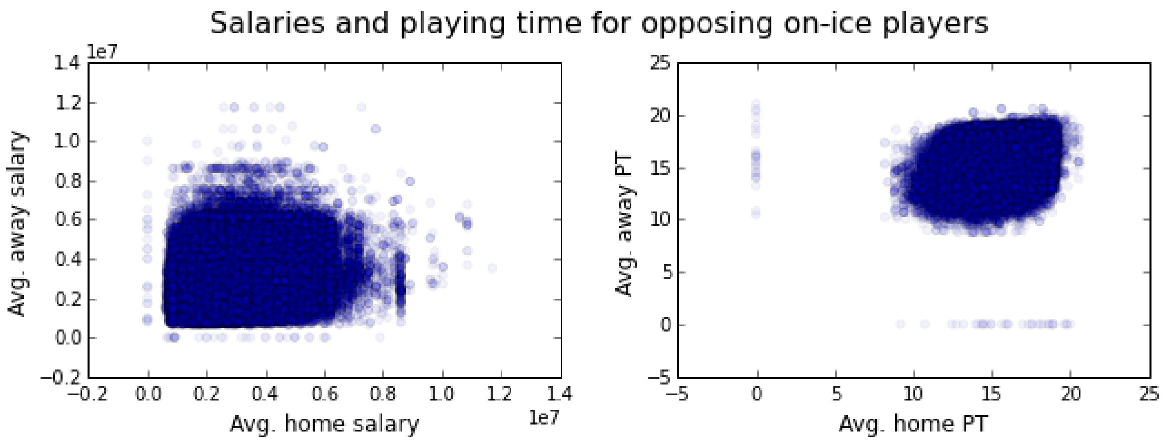


We see a clear correlation, although we can see a subgroup that is rather flat at the bottom. As it happens, entry-level contracts are capped at 925,000 USD, so that subgroup represents the underpaid young players.

To support our claim that better players tend to share the ice with other better players, we consider the average salary and playing time for home players versus away players. `attempt_manager` has class variables recording this data.

Visually it's difficult to tell whether there's a good correlation, especially with respect to salary:

In [13]:
```
myplot.sal_PT_plot()
```



If we run a linear regression however, we can see that, after proper scaling, there is a moderate salary correlation:

In [15]:
```
import wrangling.attempt_manager as am

AM = am.attempt_manager()
AM.Load()

h_sal = pd.Series(AM.home_OI_sal)
a_sal = pd.Series(AM.away_OI_sal)
h_PT = pd.Series(AM.home_OI_PT)
a_PT = pd.Series(AM.away_OI_PT)

h_sal_norm = (h_sal-h_sal.mean())/h_sal.std()
a_sal_norm = (a_sal-a_sal.mean())/a_sal.std()

LR = LinearRegression()
LR.fit(h_sal_norm.reshape(-1,1),a_sal_norm.reshape(-1,1))
print 'Correlation of home on-ice salary to away on-ice salary: {}'.f
ormat(LR.coef_[0,0])
```

```
Correlation of home on-ice salary to away on-ice salary: 0.1421601394
84
```

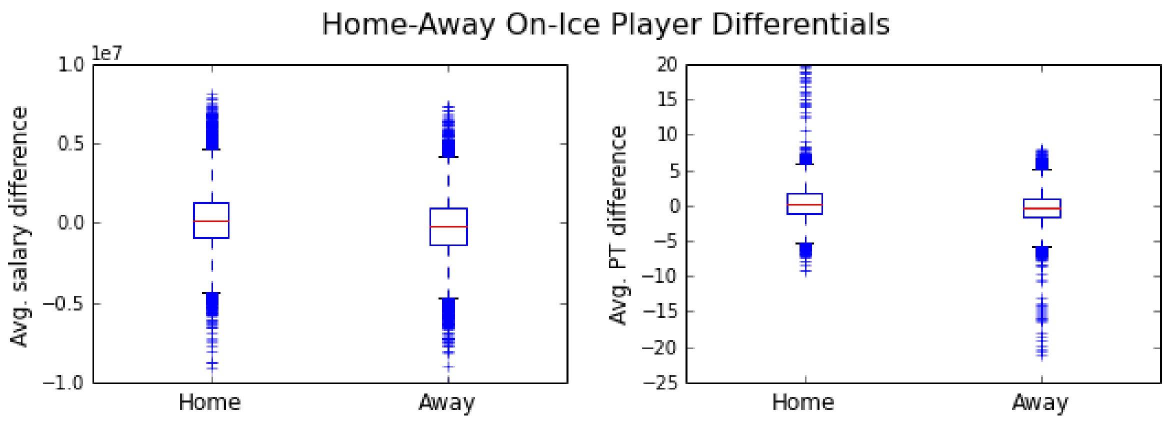
And when we do the same to the playing time data, we get a correlation coefficient of 0.30. This is around what I would expect: a definite correlation, but enough 'cross-talk' between various lines that we get good players on the ice with not-so-good players at a reasonable clip.

In [16]:
```
h_PT_norm = (h_PT-h_PT.mean())/h_PT.std()
a_PT_norm = (a_PT-a_PT.mean())/a_PT.std()

LR1 = LinearRegression()
LR1.fit(h_PT_norm.reshape(-1,1),a_PT_norm.reshape(-1,1))
print 'Correlation of home on-ice PT to away on-ice PT: {}'.format(LR
1.coef_[0,0])
```

```
Correlation of home on-ice PT to away on-ice PT: 0.295671571783
```

We can now ask whether these two metrics affect shot attempts:

In [17]:
```
myplot.OI_diff_plot()
```



It is clear that effect size is actually quite small, and in the case of salary, there may not be any effect at all. Despite the very significant overlap in both cases, our sample size is very large, so we may get very large z-statistics anyway.

In [19]:
```
AM = am.attempt_manager()
AM.Load()
away_indices = list(set(range(0,len(AM.attempt_type))) - set(AM.home_
indices))
away_indices.sort()
sal_diff = pd.DataFrame()
sal_diff_home = list(pd.Series(AM.home_OI_sal)[AM.home_indices]- \
                     pd.Series(AM.away_OI_sal)[AM.home_indices])
sal_diff_away = list(pd.Series(AM.home_OI_sal)[away_indices] - \
                     pd.Series(AM.away_OI_sal)[away_indices])

sal_diff_mean_diff = np.mean(np.array(sal_diff_home)) - np.mean(np.ar
ray(sal_diff_away))
sal_diff_var_home = np.var(np.array(sal_diff_home))
sal_diff_var_away = np.var(np.array(sal_diff_away))
nh = len(sal_diff_home)
na = len(sal_diff_away)
sal_diff_error = np.sqrt( 1/ float(nh)+1/float(na)) * np.sqrt( (nh*sa
l_diff_var_home + nh*sal_diff_var_away)/ float(nh+na-2))

z_statistic = sal_diff_mean_diff / sal_diff_error
print 'Difference in salary discrepancy between home and away shot at
tempts: {}'.format(sal_diff_mean_diff)
print 'The z-score is: {}'.format(z_statistic)
print 'The probability that this is just a fluke is: {}'.format(1-
sps.norm.cdf(z_statistic))
```

```
Difference in salary discrepancy between home and away shot attempts:
426308.743591
The z-score is: 44.9592485704
The probability that this is just a fluke is: 0.0
```

In [21]:
```
PT_diff_home = pd.Series(AM.home_OI_PT)[AM.home_indices] - \
                     pd.Series(AM.away_OI_PT)[AM.home_indices]
PT_diff_away = pd.Series(AM.home_OI_PT)[away_indices] - \
                     pd.Series(AM.away_OI_PT)[away_indices]
PT_diff_mean_diff = np.mean(np.array(PT_diff_home)) - np.mean(np.arra
y(PT_diff_away))
PT_diff_var_home = np.var(np.array(PT_diff_home))
PT_diff_var_away = np.var(np.array(PT_diff_away))
nh = len(PT_diff_home)
na = len(PT_diff_away)
PT_diff_error = np.sqrt( 1/ float(nh)+1/float(na)) * np.sqrt( (nh*PT_
diff_var_home + nh*PT_diff_var_away)/ float(nh+na-2))

z_statistic = PT_diff_mean_diff / PT_diff_error
print 'Difference in playing time discrepancy between home and away s
hot attempts: {}'.format(PT_diff_mean_diff)
print 'The z-score is: {}'.format(z_statistic)
print 'The probability that this is just a fluke is: {}'.format(1-
sps.norm.cdf(z_statistic))
```

```
Difference in playing time discrepancy between home and away shot att
empts: 0.665294931899
The z-score is: 57.9543230745
The probability that this is just a fluke is: 0.0
```

Both of the effect sizes (average salary bump of $436,000 and playing time bump of 36 seconds per 60 minutes) are rather small but definitely not trivial. Additionally, the large sample sizes allow us to conclude that these effects are very significant. If salary and playing time are good indicators of player ability, then the ability of players on the ice affects frequency of shot attempts.

We have already mentioned that playing time is probably a better indicator of player ability than salary. Moreover, both variables are correlated, so is it necessary to consider both? More variables lead to overfitting and we don't want two features when only one is needed.

Now, one can imagine reasons why salary could still be useful. Some older players may be quite skilled but with lower stamina. Therefore they may be well compensated, but given less playing time. A more important effect is that players on bad teams get more playing time than they deserve, because their competition is limited. Presumably their salary would not reflect this lower competition. Conversely, players on loaded teams may be well-paid to reflect their ability, but may see less ice-team because their teammates are good. This effect is probably mitigated considerably by the salary cap, but let us examine the data.

Let's create a third feature by projecting away PT from the salary data. Define adjusted salary to be:

$$adjusted\ salary = salary\text{ - }(\ PT\text{-}salary\ correlation)\ *\ (playing\ time)$$

```
In [23]: numerator1 = np.matrix(AM.home_OI_sal).dot(np.matrix(AM.home_OI_PT).T
          )[0,0]
         denom1 = np.matrix(AM.home_OI_PT).dot(np.matrix(AM.home_OI_PT).T )
         [0,0]
         adj_sal_home = np.matrix(AM.home_OI_sal) - numerator1/denom1 * np.mat
         rix(AM.home_OI_PT)

         numerator2 = np.matrix(AM.away_OI_sal).dot(np.matrix(AM.away_OI_PT).T
          )[0,0]
         denom2 = np.matrix(AM.away_OI_PT).dot(np.matrix(AM.away_OI_PT).T )
         [0,0]
         adj_sal_away = np.matrix(AM.away_OI_sal) - numerator2/denom2 * np.mat
         rix(AM.away_OI_PT)

         adj_sal_diff_home = np.array(adj_sal_home)[0][AM.home_indices] - np.a
         rray(adj_sal_away)[0][AM.home_indices]
         adj_sal_diff_away = np.array(adj_sal_home)[0][away_indices] - np.arra
         y(adj_sal_away)[0][away_indices]

         adj_sal_diff_mean_diff = np.mean(adj_sal_diff_home) - np.mean(adj_sal
         _diff_away)
         adj_sal_diff_var_home = np.var(adj_sal_diff_home)
         adj_sal_diff_var_away = np.var(adj_sal_diff_away)
         nh = len(adj_sal_diff_home)
         na = len(adj_sal_diff_away)
         adj_sal_diff_error = np.sqrt( 1/ float(nh)+1/float(na)) * np.sqrt( (n
         h*adj_sal_diff_var_home + nh*adj_sal_diff_var_away)/ float(nh+na-2))

         z_statistic = adj_sal_diff_mean_diff / adj_sal_diff_error
         print 'Difference in adjusted salary discrepancy between home and awa
         y shot attempts: {}'.format( adj_sal_diff_mean_diff)
         print 'The z-score is: {}'.format(z_statistic)
         print 'The probability that this is just a fluke is: {}'.format(1-
         sps.norm.cdf(z_statistic))
```

```
Difference in adjusted salary discrepancy between home and away shot
 attempts: 285534.784257
The z-score is: 34.9461561288
The probability that this is just a fluke is: 0.0
```

So, the adjustment decreases the effect size, by slightly less than half. Yet the confidence in a significant result is still extremely high. The large sample size has lot to do with this, but as of now, the data indicates we should keep both variables.

Basic models

The first step is to use a logistic regression. We will look at three simple models:

1. A logistic regression that just uses the Corsi numbers of the home and away players. The correct features are not the CF% numbers themselves, but the logarithms thereof. Additionally, we are not going to use the *averages*, but the sum. The reason is that power play situations mean that the number of players on-ice are not constant, and this certainly affects the shot attempts. Therefore, our first two variables are $x_1 = \sum_{i, X_i \text{ on-ice}} \log(CF\%(X_i))$ and $x_2 = \sum_{i, Y_i \text{ on-ice}} \log(CF\%(Y_i))$, where X_i and Y_i represent home and away players, respectively.
2. A logistic regression that adds two more variables: x_3 and x_4, the average playing-time of the home on-ice players and the away on-ice players.
3. A logistic regression that adds another two variables: x_5 and x_6, the average salary of the home on-ice players and the away on-ice players.

First, to properly evaluate our models, we need to make a training / cross-validation split. We have some code in the file `data_split.py` that does this. Sixty percent of our data set is sent to the file `Training.npz` to be use to train our algorithm. The split is done randomly (and seeded properly). The remaining data are split evenly and sent to `CPV.npz` and `Test.npz`.

After we obtain our training set, we must reduce our data set to the six desired features. We have code in the file `feature_assemble.py` that accomplishes this. It contains a class `FeatureAssemble`, which includes methods `Corsis()` and `Assemble()`. The former computes the CF% stats, but only using the shot attempts in the training set, and the latter constructs a table containing x_1 through x_6.

One practical issue that should be noted: since we are using log-likelihoods, any CF% that is 0 or 100 will cause an error. This is not an issue for most players, but for players that have little data available, we must use a fudge factor ϵ, and insist that every player is on the ice for at least ϵ shot attempts in either direction. In our code we use $\epsilon = 0.1$, so that a player that is on the ice for only one shot attempt has a $CF\%$ of either 9.09% or 90.9%. This also means a player who sees zero shot attempts has a CF% of exactly 50%.

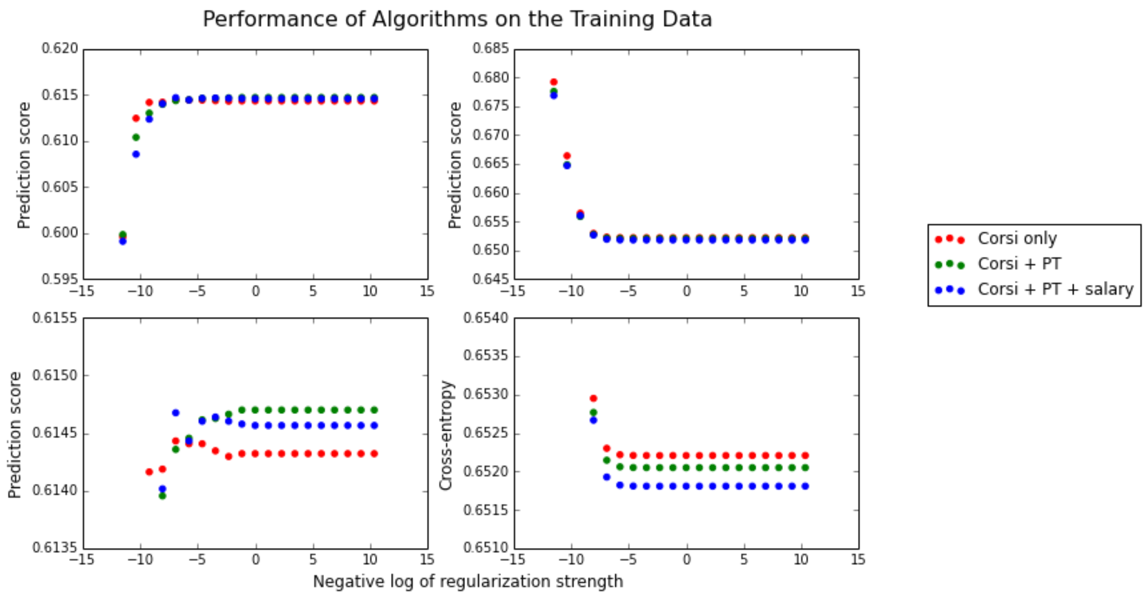
Now that we have constructed our data, we code a logistic classifier object `small_logistic`, contained in the file `small_logistic.py`. This object first scales the data (this is important, as salary is much larger in scale than playing time), and then fits the data three times. First using only x_1 and x_2, second adding x_3 and x_4, and third adding x_5 and x_6. It can also compute the prediction score and cross-entropy of the trained classifier on both the training and cross-validation data. We use another object called `SL_sweeper` that varies the regularization strength (actually the parameter is C, the inverse of the regularization strength):

In [8]:
```
import small_logistic.small_logistic as slr

C = [ pow(10, c/2.) for c in range(-10,10)]
SLS = slr.SL_sweeper(C)
SLS.training()
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/utils/optimize.py:193:
UserWarning: Line Search failed
  warnings.warn('Line Search failed')
```

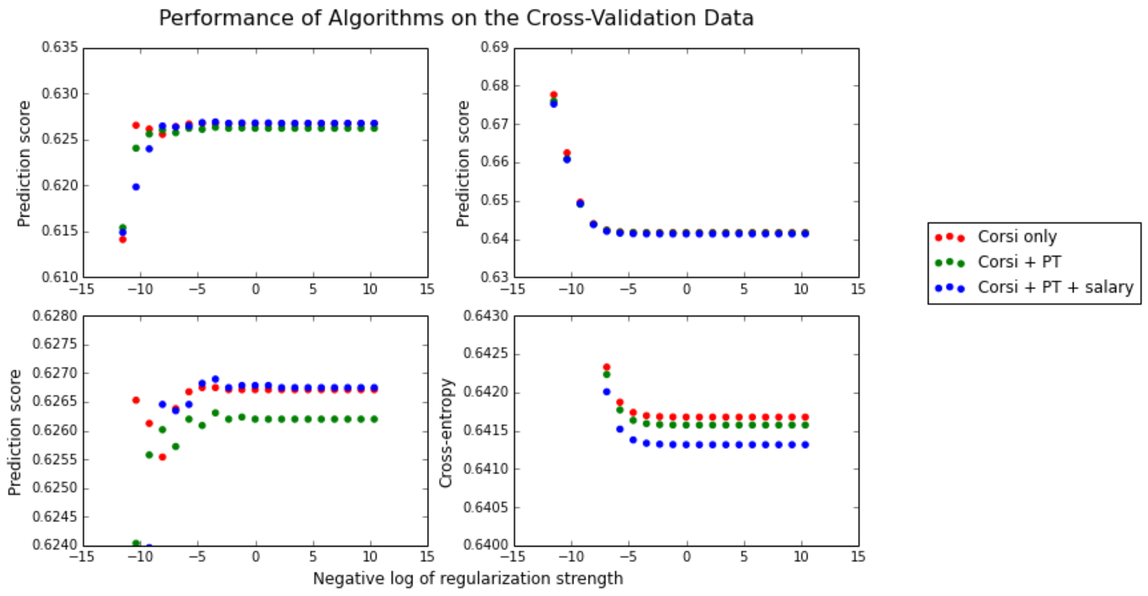
Now that we have trained the algorithm, we can display the learning curves. The four graphs below show the prediction score (left) and cross-entropy (right) as C varies, for all three methods. The lower graphs are identical to upper graphs, but with the vertical scale magnified, so that we can see the separation of the three algorithms:

```
In [9]: import plotting.small_log_plot as slp
        slp.train_plot(SLS)
```



Here is how the algorithms fare on the cross-validation data:

```
In [6]: slp.cv_plot(SLS)
```



If we pick the algorithm that performs best, and choose the best C, these are our performance metrics:

```
In [ ]: best_C = SLS.eval_matrix['all','CV','Cross entropy'].argmin()
        SLS.eval_matrix.loc[best_C,'all']
```

```
Out[ ]: split     metric
        Training  Scores           0.6145658
                  Cross entropy    0.6518095
        CV        Scores           0.6267487
                  Cross entropy    0.6413162
        Name: 3162.27766017, dtype: object
```

Some observations:

1. We must admit that the improvement we get by adding salary and PT is very small (so small that we needed to magnify). Ultimately, this has to be somewhat disappointing.
2. The prediction score using just the Corsi numbers however is not that bad. This is an inherently noisy process, so we do not expect do get anywhere near to even 90% prediction score. Getting above 60% is rather satisfying. Clearly, we have some predictive power.
3. Judging by cross-entropy on the training and cross-validation sets, adding PT improves our model, and adding salary improves it even more. Interestingly, these improvements do not necessarily carry over to the prediction score. Adding salary hurts the prediction score on the training set, while on the cross-validation set, the Corsi + PT model performs the worst! Ultimately, we have to judge our model on cross-entropy rather than prediction score.
4. Strangely, our metrics are slightly better on the cross-validation data! Presumably, this is because of a smaller data set (although of course we are using average cross-entropy).
5. The cross-validation learning curves do not really a show an optimum: the performance does not suffer by eliminating regularization.

We can conclude from 4. and 5. that we are definitely not over-fitting our algorithms. Combining that with the disappointing improvement indicates we need to implement more powerful models.

Logistic regression with 1800 features

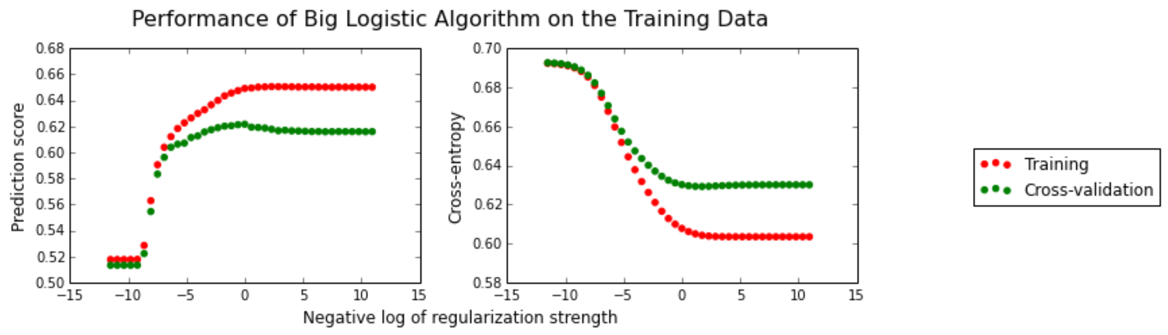
The models described so far are quite basic, and more sophistication is required. The obvious next step is to implement a logistic classifier that uses all 1800 features. Presumably the much larger feature space will allow us to build a better classifier.

We use an object `big_logistic` in the file `big_logistic/big_logistic.py` which is similar to `small_logistic`. Actually, it is simpler, since we have one rather than three models. Additionally, we don't need to scale the training set, since all of our feature variables are binary. We also have an object `BL_sweeper` which sweeps across a list of regularization constants.

```
In [3]: import big_logistic.big_logistic as blr

        C = [ pow(10, c/4.) for c in range(-20,20)]
        BLS = blr.BL_sweeper(C)
        BLS.training()
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/linear_model/sag.py:26
7: ConvergenceWarning: The max_iter was reached which means the coef_
did not converge
  "the coef_ did not converge", ConvergenceWarning)
```

```
In [4]: import plotting.big_log_plot as blp
        blp.comp_plot(BLS)
```



What we can observe:

1. Performance on the training set is noticeably improved. Peak prediction score has increased from around 61.5 to 65 percent. Cross-entropy has decreased from around 0.65 to 0.60.
2. We begin to see the hallmark characteristics of learning curves: there is considerable separation between training and cross-validation as regularization is removed. Additionally, we can detect an optimum in the cross-validation curve (it is somewhat harder to see on the right, but there is a minimum).
3. The bottom-line is that the cross-entropy of the algorithm on the cross-validation set has improved. It is certainly a moderate improvement:

```
In [16]: print 'Optimal cross-validation cross-entropy using 6 features: {}'.\
             format(SLS.eval_matrix['all','CV','Cross entropy'].min())
         print 'Optimal cross-validation cross-entropy using 1800 features:
         {}'.\
             format(BLS.eval_matrix['CV','Cross entropy'].min())
```

```
Optimal cross-validation cross-entropy using 6 features: 0.6413161526
98
Optimal cross-validation cross-entropy using 1800 features: 0.6292129
29993
```

Nevertheless, we can conclude that our logistic classifier performs better than the previous basic models.

Force-equilibrium model

We now consider a third model that is roughly equivalent in complexity to the previous 1800-feature model. It is inspired by the principle of detailed balance that Einstein used to explain the quantum emission and absorption of photons. The basic idea is that our naive $CF\%$ statistic needs adjusting, and we should adjust according to the data available. We write the adjustment as a derivative: if the derivative is positive, this mean we should adjust upward, and if negative, downward. If the derivatives of all our numbers are zero, we require no adjustment. In other words, our data exert "forces" according to our estimate of the "true Corsi" and when we reach equilibrium, we can stop adjusting.

Assume that player j has an adjusted Corsi likelihood p_j, and let S_{jk} be the number of shot attempts for player k 's team while players j and k are on the ice against each other. We should make an adjustment upward that that increases with S_{kj} and an adjustment downward that increases with S_{jk}. The negative adjustment should be proportional to p_j to penalize over-estimation, and the positive adjustment should be proportional to p_k to reward shot attempts against good players. We get a set of linear differential equations:

$$\frac{dr_j}{dt} = \sum_{k \neq j} \left(S_{kj}^2 r_k - S_{jk}^2 r_j \right)$$

We then find the equilibrium solution by setting the left-hand-side to zero and solving the linear equation.

We use the squares of the shot attempts so that it works out in the two-player case. If there are only two players, the solution (up to a multiplicative constant) is $r_1 = \frac{S_{12}}{S_{21}}$ and $r_2 = \frac{S_{21}}{S_{12}}$. This is clearly the most parsimonious approach: the corresponding probabilities would be $\frac{S_{12}}{S_{12}+S_{21}}$ and $\frac{S_{21}}{S_{12}+S_{21}}$, which of course are the usual $CF\%$ numbers.

Now let us add a third player. Suppose players 1 and 2 play each other and there are five shot attempts for each team. Then players 1 and 3 play each other and there are nine shot attempts for player 1's team and one for player 3's team. The CF percentages end up being 70\%, 50\% and 10\%. This is unfair to player 2, as the shot attempts indicate he is even with player 1 and suffers from not playing the inferior player 3. If we feed these numbers into the differential equations above and solve, we get $r_1 = r_2 = 1.491$ and $r_3 = 0.0137$. This is leads to adjusted $CF\%$ of 59.8\%, 59.8\% and 13.6\%, which appear to be more sensible.

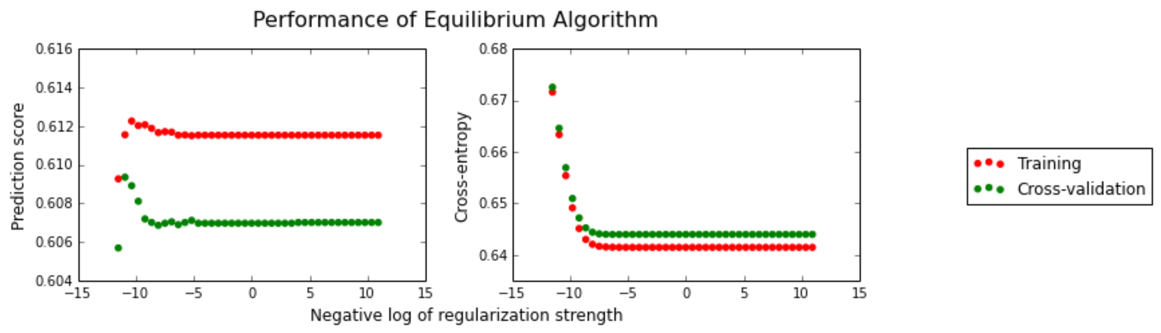
If we do this for all shot attempts over the course of the season, we can construct a 900x900 matrix out of the numbers S_{jk}.

There are some wrinkles with this approach:

- The matrix S will not be invertible, because the solutions
- This approach does not distinguish between player-at-home and player-away. If we used the 1800 feature approach, this would introduce a degeneracy in the matrix preventing solution. We could get around this by adding another constraint that

In [ ]:
```
import equilibrium.eq_logistic as el
C = [ pow(10, c/4.) for c in range(-20,20)]

EqLS = el.EL_sweeper(C)
EqLS.training()
```

In [3]:
```
import plotting.log_plot as plp
plp.comp_plot(EqLS, 'Performance of Equilibrium Algorithm')
```



As we can see, this algorithm does not perform very well at all, and in fact does not even out-perform the basic models. So we can disregard this model.

The Next Steps

Going forward there are two directions we wish to go:

1. Support Vector Machines. Here we will attempt the divide the feature space into two regions, wherein different lineups are placed depending on. This is attractive from an interpretational point of view: to evaluate a player, we can look at how many of his lineups fall in the "good" region, and how far these lineups are from the boundary.
2. Neural Nets. We do not intend to use Deep Learning here, as the data set is likely not large enough. However, we will attempt to construct a network that is four or five layers deep. Since we have 1800 features, one possibility is the following: 1800 input nodes, then a layer of 180, then a layer of 18, then the output layer of two nodes. Presumably this approach will give us more predictive power. However, how to interpret the resulting model to evaluate a given player is not yet clear.

Appendix: summary of Python and data files

The amount of Python code I used is too large to dump into this report, so I will list the files here with short descriptions. All code is available at the github repo:

http://www.github.com/darraghrooney/Springboard_Capstone/
(http://www.github.com/darraghrooney/Springboard_Capstone/)

```
/scraping/
    roster_scrape.py
    directory_build.py
    es_scrape.py
    salary_fill.py
    report_downloader.py
    attempt_scrape.py

/wrangling/
    attempt_manager.py
    summary_manager.py
    data_split.py

/small_logistic/
    feature_assemble.py
    small_logistic.py

/big_logistic/
    big_logistic.py

/equilibrium/
    equilibrium.py
```

Additionally, there were quite a few data files generated (some .csv, some .npz):

```
/data/
    Big_Roster.csv
    Attempts.npz
```