Not every shot is created equal

Adjusting NHL Corsi statistics for quality of opponent

DP Rooney

October 22, 2017

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 on goal, other measures of individual merit have arisen. One popular quantity is the "Corsi-for percentage", or CF%, defined as follows:

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

Here, a "shot attempt" is the sum of shots on goal, shots missed, and shots blocked. The CF% 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 roughly 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 on-ice 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 from 2016 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' in the 2015-2016 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 this 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, where the goalie is pulled, will be considered power play situations.

Scraping and Wrangling

The data used is from the 2015-2016 season. It was scraped from three types of game reports from the NHL: game rosters, event summaries, and play-by-play reports. Unfortunately, the HTML pages have been re-named, and re-formatted, so that the scraper will no longer work. But the data was scraped before this happened.

The defunct URL's for the first game are:

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 (roster, event, play), 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 were 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. For comparison, the new game report can be seen here:

https://www.nhl.com/gamecenter/tor-vs-ott/2016/10/12/2016020001#game=2016020001,game_state=final
(https://www.nhl.com/gamecenter/tor-vs-ott/2016/10/12/2016020001#game=2016020001,game_state=final)

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: Qt5Agg
```

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 [2]:
```
df = pd.read_csv('data/Directory.csv')
df.head()
```

Out[2]:

	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

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 [3]:
```
df = pd.read_csv('data/Summary.csv')
df.head()
```

Out[3]:

	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

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):

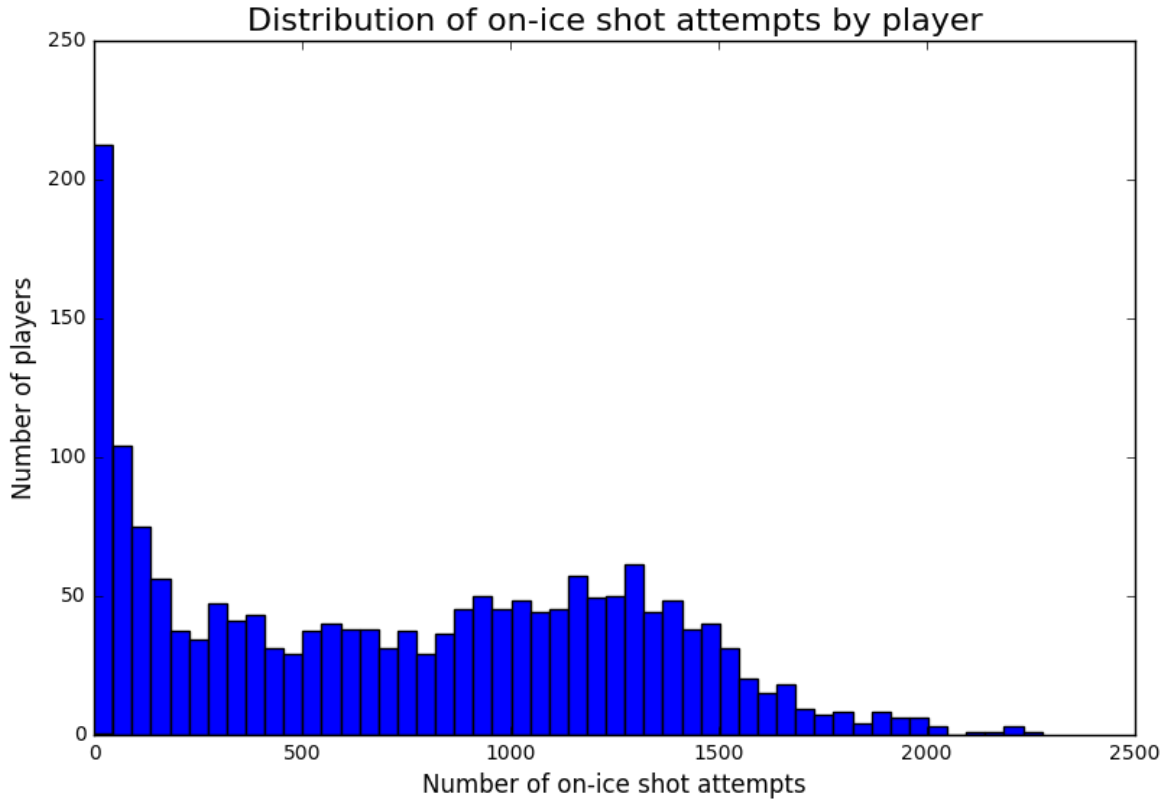
In [4]:
```
import wrangling.attempt_manager as am

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
Goals: 6565
Shots blocked: 34845
Shots saved: 66601
Shot missed: 28519
Total shot attempts: 136530
```

We can ask how many data points we have for each player. It turns out the average is over 700. However, the spread is quite large (over 70 precent of the mean), and 18 percent of players have fewer than 100:

In [5]:
```
pl_atts = list()
for k in range(0,1800):
    pl_atts.append(AM.att_matrix.getcol(k).nnz)

print('Mean of player on-ice shot attempts: ' +
str(pd.Series(pl_atts).mean()))
print('Median: ' + str(pd.Series(pl_atts).median()))
print('Standard deviation: ' + str(pd.Series(pl_atts).std()))
print('Fraction of players with 100 or fewer: '+ str(pd.Series(pl_att
s)[pd.Series(pl_atts) < 101].count()/1800))

fig = plt.figure()
ax = fig.add_axes([.1,.1,1.2,1.2])
plt.xlabel('Number of on-ice shot attempts', fontsize = 12)
plt.ylabel('Number of players', fontsize = 12)
plt.title('Distribution of on-ice shot attempts by player', fontsize
= 16)
ax.hist(pl_atts,bins=50)
plt.show()
```

```
Mean of player on-ice shot attempts: 740.4744444444444
Median: 737.5
Standard deviation: 547.8564539886445
Fraction of players with 100 or fewer: 0.180555555556
```



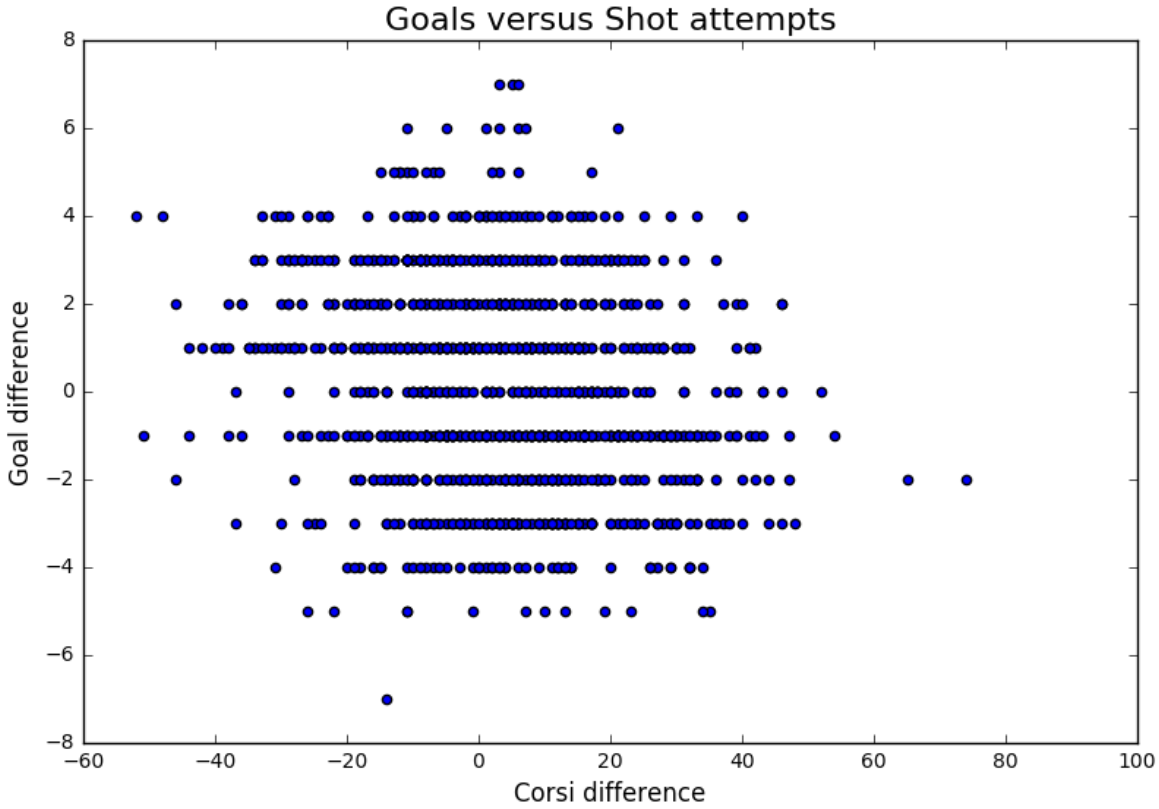
Note that in the foregoing, we are considering player X at home to be different than the same player away from home. This is because there is a clear home-ice advantage, in terms of goals and shot attempts, as seen below. So for the extent of this report, we will keep all 1,800 features, rather than attempt to combine into 900.

In [6]:
```
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 [7]:
```
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()
```



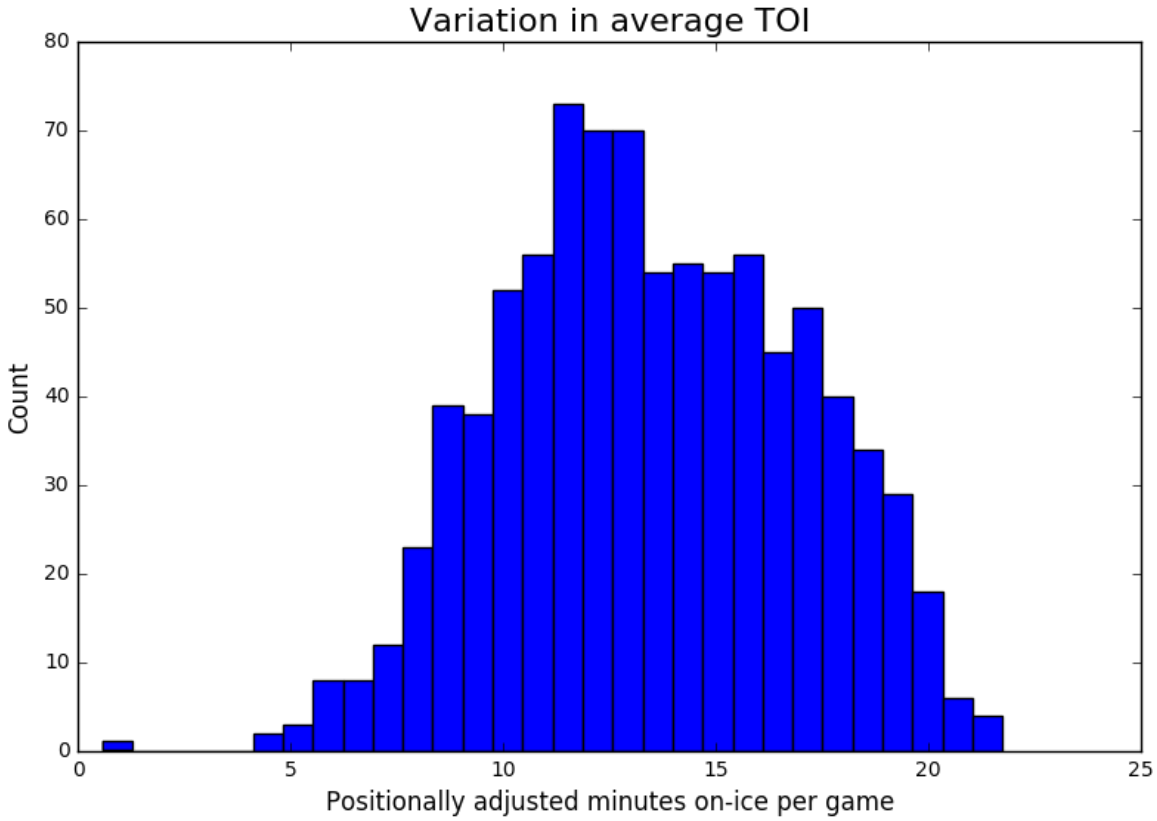
In [8]:
```
LR = LinearRegression()
LR.fit( Corsi_diff.reshape(-1,1),goal_diff.reshape(-1,1))
print('Regression coefficient: {}'.format(LR.coef_[0,0]))
```

```
Regression coefficient: -0.026606973456951434
```

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.

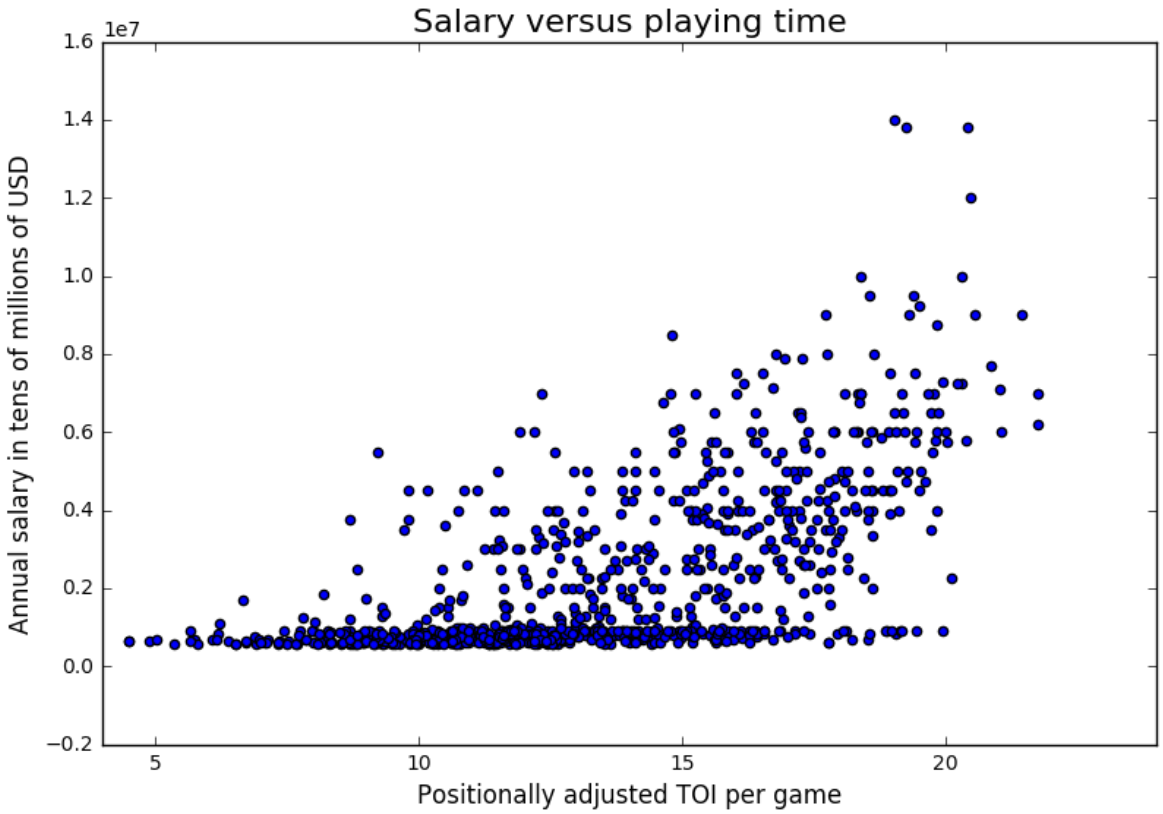
We can use two proxy statistics to assess 'quality of opponent' (or at least the perception of quality): player salary and average playing time. Salary is a little problematic, because younger players are certainly underpaid, and the production of older players can fall off after getting a fat contract. 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 [9]:
```
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 [10]:
```
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()
```

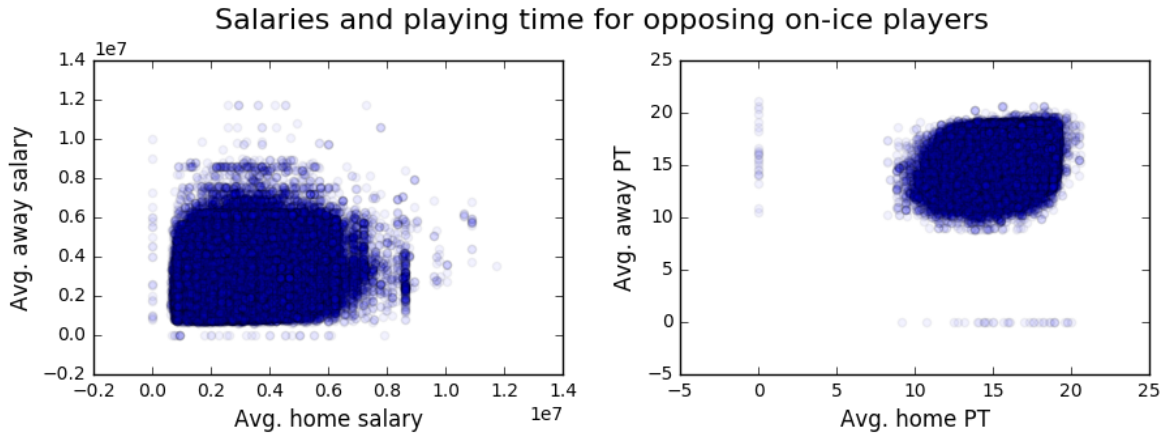


We see a clear correlation, although we can see a subgroup that is rather flat at the bottom. As it happens, entry-level contracts are capped at 925,000 USD, so that subgroup represents the underpaid young players.

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 [11]:
```
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 [12]:
```
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
8400255
```

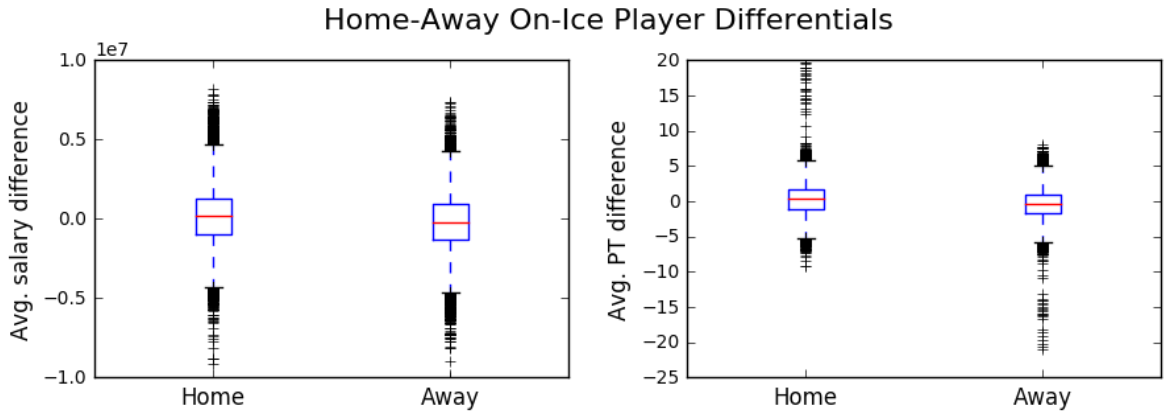
And when we do the same to the playing time data, we get a correlation coefficient of 0.30. This is around what I would expect: a definite correlation, but enough 'cross-talk' between various lines that we get good players on the ice with not-so-good players at a reasonable clip.

In [13]:
```
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: 0.29567157178338255
```

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

In [14]:
```
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 [15]:
```
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:
426308.743591288
The z-score is: 44.959248570399794
The probability that this is just a fluke is: 0.0
```

In [16]:
```
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: 0.6652949318987651
The z-score is: 57.9543230745336
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:

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

In [17]:
```
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: 285534.7842569862
The z-score is: 34.94615612878633
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.

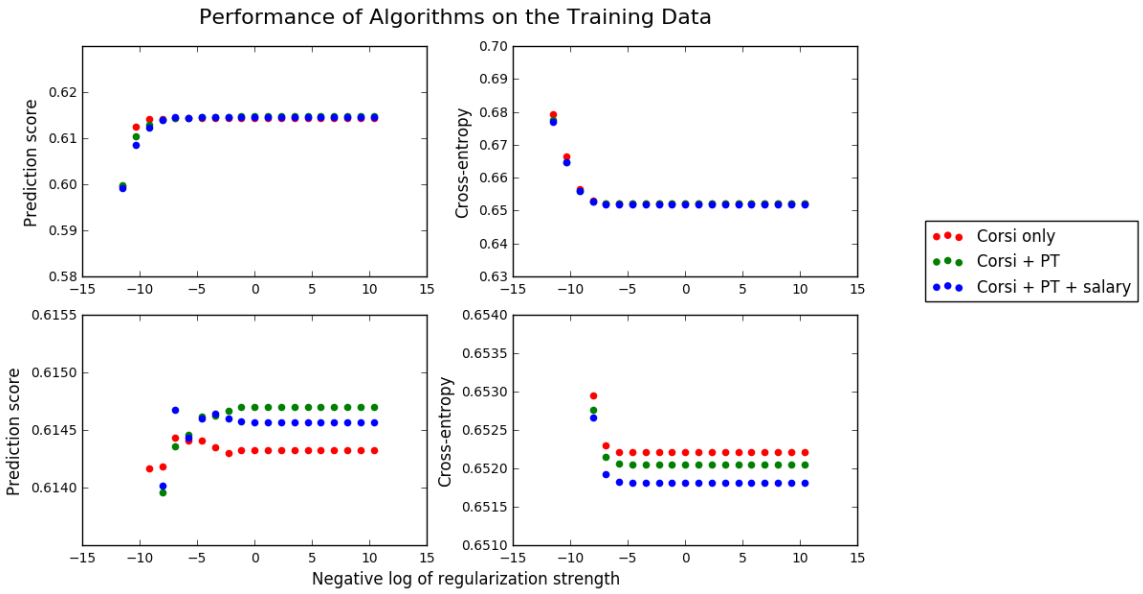
One practical issue that should be noted: since we are using log-likelihoods, any CF% that is 0 or 100 will cause an error. This is not an issue for most players, but for players that have little data available, we must use a fudge factor ϵ, and insist that every player is on the ice for at least ϵ shot 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 [18]:
```
import warnings
warnings.filterwarnings('ignore')

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()
```

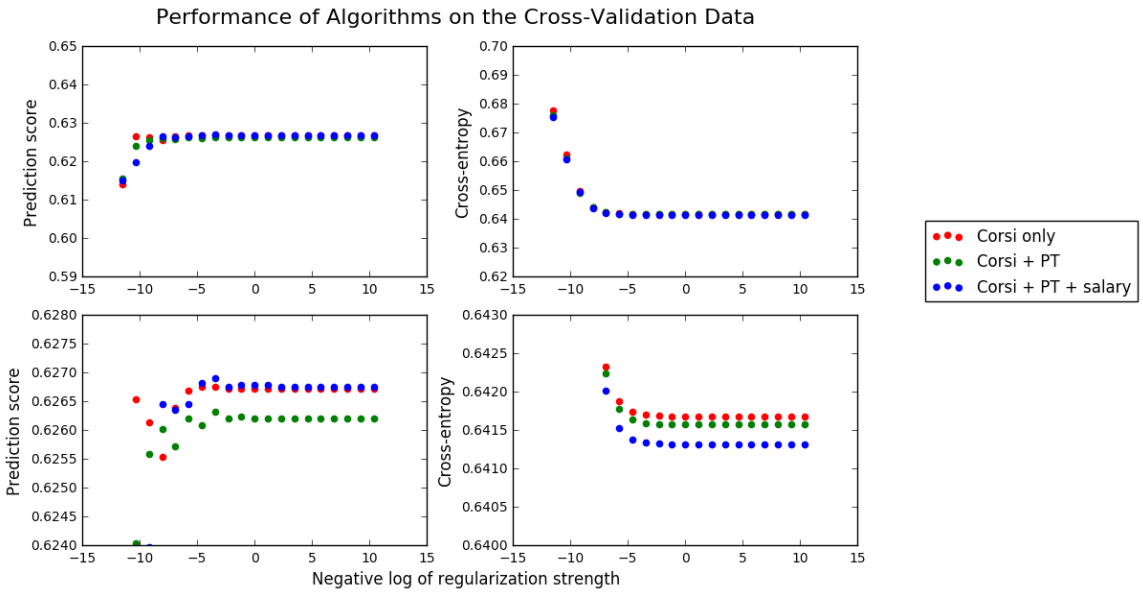
Now that we have trained the algorithm, we can display the learning curves. The four graphs below show the prediction score (left) and cross-entropy (right) as C varies, for all three methods. The lower graphs are identical to upper graphs, but with the vertical scale magnified, so that we can see the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

In [19]:
```
import plotting.small_log_plot as slp
slp.train_plot(SLS)
```



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

In [20]:
```
slp.cv_plot(SLS)
```



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

In [21]:
```
best_C = SLS.eval_matrix['all','CV','Cross entropy'].argmin()
SLS.eval_matrix.loc[best_C,'all']
```

Out[21]:
```
split     metric
Training  Scores           0.614566
          Cross entropy     0.65181
CV        Scores           0.626749
          Cross entropy    0.641316
Name: 31622.7766017, 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.

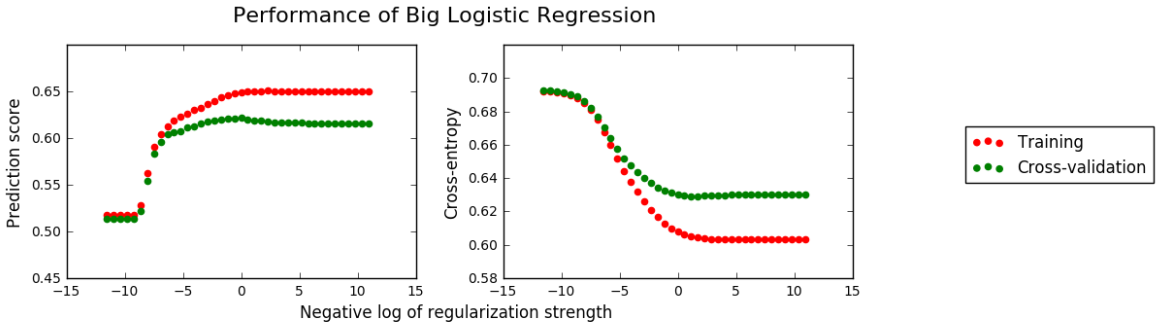
Logistic regression with 1800 features

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 [22]:
```
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()
```

In [23]:
```
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 [24]:
```
print('Optimal cross-validation accuracy using 6 features: {}'.\
    format(SLS.eval_matrix['all','CV','Scores'].max()))
print('Optimal cross-validation accuracy using 1800 features: {}'.\
    format(BLS.eval_matrix['CV','Scores'].max()))
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 accuracy using 6 features: 0.626895187870797
7
Optimal cross-validation accuracy using 1800 features: 0.621768109572
9877
Optimal cross-validation cross-entropy using 6 features: 0.6413161525
631554
Optimal cross-validation cross-entropy using 1800 features: 0.6292116
907923556
```

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

Random Forests

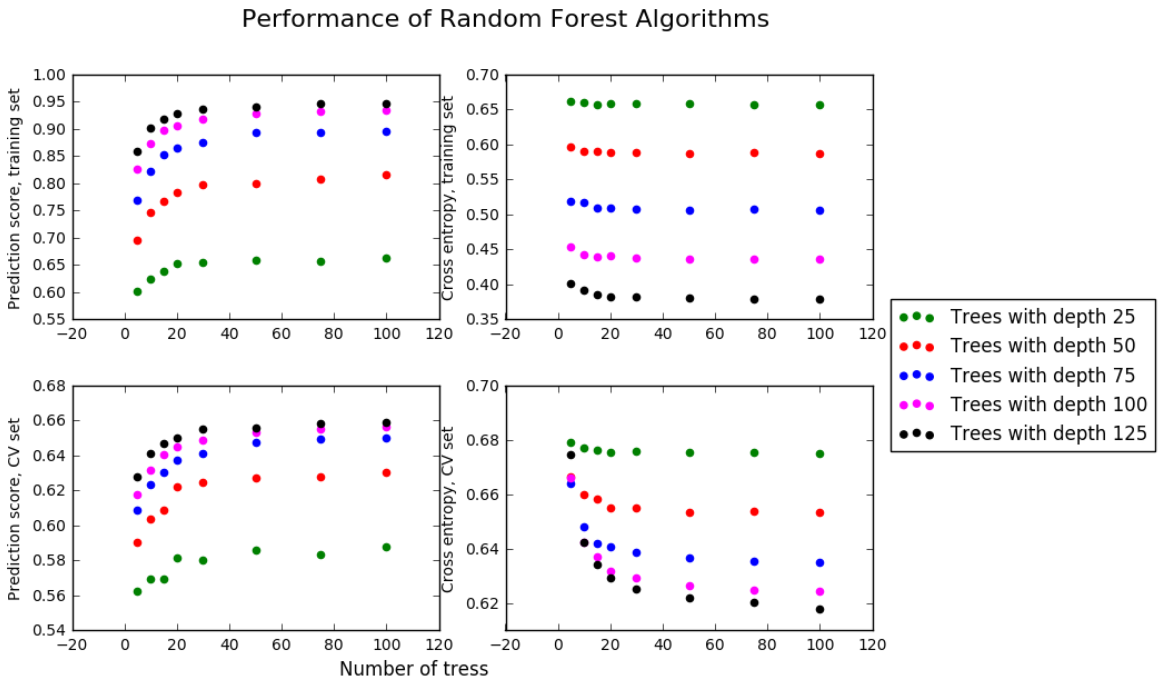
We would like an algorithm that has more significant improvement than the logarithmic regression used above. Neural networks and support vector machines are a possibility, but perhaps

In [25]:
```
import forests.rafo as rf

depths = [25,50,75,100,125]
n_tr = [5,10,15,20,30, 50,75, 100]

RFs = rf.rf_coll(depths,n_tr)
```

In [26]:
```
RFs.training()
```

In [27]:
```
RFs.evaluate()
```

In [28]:
```
import plotting.rf_plot as rfp

rfp.rf_plot(RFs)
```



In [31]:
```
best_ps=RFs.eval_matrix.loc[:,('CV', 'Scores')].idxmax()
best_cv=RFs.eval_matrix.loc[:,('CV', 'Cross entropy')].idxmin()

print('Best depth and tree number with respect to accuracy: ' + str(b
est_ps))
print('Best depth and tree number with respect to cross entropy: ' +
str(best_cv))
```

```
Best depth and tree number with respect to accuracy: (125, 100)
Best depth and tree number with respect to cross entropy: (125, 100)
```

In [32]:
```
print('Best prediction accuracy: ' + str(RFs.eval_matrix.loc[best_cv,
('CV','Scores')]))
print('Best cross entropy: '+ str(RFs.eval_matrix.loc[best_cv,
('CV','Cross entropy')]))
```

```
Best prediction accuracy: 0.658866183256
Best cross entropy: 0.617726700542
```

We have improved the accuracy to 65.9 percent, and the cross-entropy to 61.8 percent.

Interpretation and Recommendations

Given that a logistic regression model was clearly inferior to our random forest classifier, we shall use the latter to draw conclusions. To evaluate players, we can ask our model to predict shot attempts for each player, when only that player is on the ice. Clearly an artificial scenario, but since random forest algorithms split based on one feature at a time, this is the best way to analyze our model. An alternative would be to ask the model what happens for all possible player combinations, but the computation time for this task would be very high.

Additionally, the `sklearn` package provides a `feature_importances_` method, which allows us to look at how influential a player is within the model. This way, we can weed out spurious findings: if a player's Corsi number differs significantly from the model prediction, and

In [33]:
```
from sklearn.ensemble import RandomForestClassifier as RFC
import pickle

with open('data/Forest_100_125', 'rb') as input:
    unpickler = pickle.Unpickler(input)
    myrfc = unpickler.load()

pp = pd.DataFrame(index=AM.NGs, data=np.zeros([900,6]), columns=['Hom
e Corsi', 'Away Corsi',
             'Home RF prediction', 'Away RF prediction', 'Home import
ance', 'Away importance'])

for p in range(0,900):
    hoi = np.zeros(1800)
    aoi = np.zeros(1800)
    hoi[p] = True
    aoi[p+900] = True
    pp.iloc[p,0] = AM.compute_Corsi(AM.NGs[p], 'home')
    pp.iloc[p,1] = AM.compute_Corsi(AM.NGs[p], 'away')
    pp.iloc[p,2] = myrfc.predict_proba(hoi)[:,1]
    pp.iloc[p,3] = myrfc.predict_proba(aoi)[:,0]
    pp.iloc[p,4] = myrfc.feature_importances_[p]
    pp.iloc[p,5] = myrfc.feature_importances_[p+900]
```

First, it is interesting that our model is more cautious than the naive Corsi statistic:

In [34]:
```
print('Spread of Home Corsi stats: ' + str(pp.loc[:,'Home
Corsi'].std()))
print('Spread of Home model prediction: ' + str(pp.loc[:,'Home RF pre
diction'].std()))

print('Spread of Away Corsi stats: ' + str(pp.loc[:,'Away
Corsi'].std()))
print('Spread of Away model prediction: ' + str(pp.loc[:,'Away RF pre
diction'].std()))
```

```
Spread of Home Corsi stats: 0.08621862299310906
Spread of Home model prediction: 0.034498206900554794
Spread of Away Corsi stats: 0.08436779026498906
Spread of Away model prediction: 0.037975787323674605
```

However, if we restrict ourselves to the "important" players, this effect is smaller (but still non-negligible). The model is more confident for more important players, which is to be expected. But the spread is smaller for the raw stat, presumably because we are removing anomalous players.

In [35]:
```
fi_thr = min(pp.loc[:,'Home importance'].median(), pp.loc[:,'Away imp
ortance'].median())

imp_pp = pp[(pp.loc[:,'Home importance']>fi_thr) & (pp.loc[:,'Away im
portance']>fi_thr)]
print('Spread of Home Corsi stats: ' + str(imp_pp.loc[:,'Home
Corsi'].std()))
print('Spread of Home model prediction: ' + str(imp_pp.loc[:,'Home RF
 prediction'].std()))

print('Spread of Away Corsi stats: ' + str(imp_pp.loc[:,'Away
Corsi'].std()))
print('Spread of Away model prediction: ' + str(imp_pp.loc[:,'Away RF
 prediction'].std()))
```

```
Spread of Home Corsi stats: 0.06497455257267885
Spread of Home model prediction: 0.04477117529484894
Spread of Away Corsi stats: 0.06453695878441502
Spread of Away model prediction: 0.05042022331721116
```

We should not directly compare a player's raw Corsi number to his model prediction statistic, since these two quantities have different means and variances. We should first normalize these numbers by subtracting means and dividing by standard deviations.

In [36]:
```
pp_norm = pp.copy()

mean_hc = pp.loc[:,'Home Corsi'].mean()
mean_ac = pp.loc[:,'Away Corsi'].mean()
mean_hrf = pp.loc[:,'Home RF prediction'].mean()
mean_arf = pp.loc[:,'Away RF prediction'].mean()
std_hc = pp.loc[:,'Home Corsi'].std()
std_ac = pp.loc[:,'Away Corsi'].std()
std_hrf = pp.loc[:,'Home RF prediction'].std()
std_arf = pp.loc[:,'Away RF prediction'].std()

pp_norm.loc[:,'Home Corsi'] = (pp.loc[:,'Home Corsi']-mean_hc)/std_hc

pp_norm.loc[:,'Away Corsi'] = (pp.loc[:,'Away Corsi']-mean_ac)/std_ac

pp_norm.loc[:,'Home RF prediction'] = (pp.loc[:,'Home RF
prediction']-mean_hrf)/std_hrf
pp_norm.loc[:,'Away RF prediction'] = (pp.loc[:,'Away RF
prediction']-mean_arf)/std_arf
```

Now we come to the main objective of this report. We want to identify players who are better (or worse) than their raw Corsi number indicates. To accomplish this, we will extract players whose model prediction numbers differ the most from the corresponding Corsi data. Furthermore, we will only look at the "important" players to avoid spurious findings.

In [37]:
```
imp_pp_norm = pp_norm[(pp_norm.loc[:,'Home importance']>fi_thr) & (pp
_norm.loc[:,'Away importance']>fi_thr)]

home_dev = imp_pp_norm.loc[:, 'Home RF prediction'] -
imp_pp_norm.loc[:, 'Home Corsi']
away_dev = imp_pp_norm.loc[:, 'Away RF prediction'] -
imp_pp_norm.loc[:, 'Away Corsi']
```

First, let us look for good players. We want players who are under-rated both home and away, so we demand that their Corsi-model difference is at least one standard deviation from the norm. This narrows the field down to five players:

In [38]:
```
good_find = (away_dev > away_dev.std()) & (home_dev > home_dev.std())
imp_pp[good_find]
```

Out[38]:

	Home Corsi	Away Corsi	Home RF prediction	Away RF prediction	Home importance	Away importance
ALEX OVECHKIN	0.629032	0.602395	0.665825	0.632223	0.001140	0.001046
PAVEL DATSYUK	0.650000	0.622626	0.612959	0.594240	0.000854	0.000966
TOMAS HERTL	0.611201	0.556886	0.632630	0.550497	0.000890	0.000835
TOMAS TATAR	0.655615	0.605114	0.677919	0.574092	0.000758	0.000857
WAYNE SIMMONDS	0.638102	0.588738	0.618472	0.567817	0.001003	0.000992

Interestingly, all five players have already good Corsi numbers. Our model is not identifying players with mediocre Corsis that are better than those numbers appear. Rather, it is identifying players whose Corsi numbers are robust to a more nuanced model. It is telling us that we can be very confident in the raw Corsi stat.

We do the same for bad players, and we get a set that is a little larger:

In [39]:
```
bad_find = (away_dev < -away_dev.std()) & (home_dev < -
home_dev.std())
imp_pp[bad_find]
```

Out[39]:

	Home Corsi	Away Corsi	Home RF prediction	Away RF prediction	Home importance	Away importance
ANDY GREENE	0.391840	0.353275	0.413426	0.376806	0.000954	0.000870
CHRIS VANDEVELDE	0.410420	0.390743	0.447263	0.394437	0.000749	0.000708
GREGORY CAMPBELL	0.389034	0.310049	0.407542	0.103733	0.000673	0.001409
JAMIE BENN	0.579625	0.554815	0.488081	0.443369	0.000944	0.001008
JARRET STOLL	0.322464	0.308750	0.242315	0.167522	0.001217	0.000909
JOE PAVELSKI	0.646598	0.571617	0.495487	0.451434	0.000934	0.000747
LUKE GLENDENING	0.370331	0.339138	0.344779	0.367095	0.000949	0.000990
NICK SCHULTZ	0.412831	0.394309	0.375358	0.392586	0.001194	0.001005
PAUL GAUSTAD	0.345018	0.320312	0.273890	0.245152	0.000679	0.000759

As in the previous case, most of the players identified had bad raw numbers to begin with. So the model is selecting players whose badness is robust. However, two of the players had good Corsi numbers but relatively poor model numbers: Joe Pavelski and Jamie Benn.

As a point of interest, let us compare the players mentioned in the introduction, Toews and Virtanen:

In [42]:
```
print(pp_norm.loc['JONATHAN TOEWS'])
```

```
Home Corsi            0.608661
Away Corsi            0.498807
Home RF prediction    0.352635
Away RF prediction    0.028452
Home importance       0.001081
Away importance       0.001021
Name: JONATHAN TOEWS, dtype: float64
```

In [43]:
```
print(pp_norm.loc['JAKE VIRTANEN'])
```

```
Home Corsi            0.367090
Away Corsi            0.600977
Home RF prediction   -0.002721
Away RF prediction    0.344945
Home importance       0.000486
Away importance       0.000457
Name: JAKE VIRTANEN, dtype: float64
```

As expected, the model's estimation of Virtanen is lower than the Corsi indicates. On the other hand, the same applies to Toews! It appears the model is telling us that Jonathan Toews is not as good as his reputation would suggest.

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/`

```
/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

/forests/

    rafo.py

/plotting/

    plotting.py
    log_plot.py
    big_log_plot.py
    rf_plot.py
```

Additionally, there were a number of data files generated (some .csv, some .npz, one of them a pickle):

```
/data/
    Big_Roster.csv
    Directory.csv
    Summary.csv
    Attempts.npz
    Training.npz
    CV.npz
    TrainCorsis.csv
    CVCorsis.csv
    Forest_100_125
```

In [ ]: