In [1]:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import geocoder
import sklearn
from IPython.core.display import HTML
css = open('style-table.css').read() + open('style-notebook.css').read()
HTML('<style>{}</style>'.format(css))
```

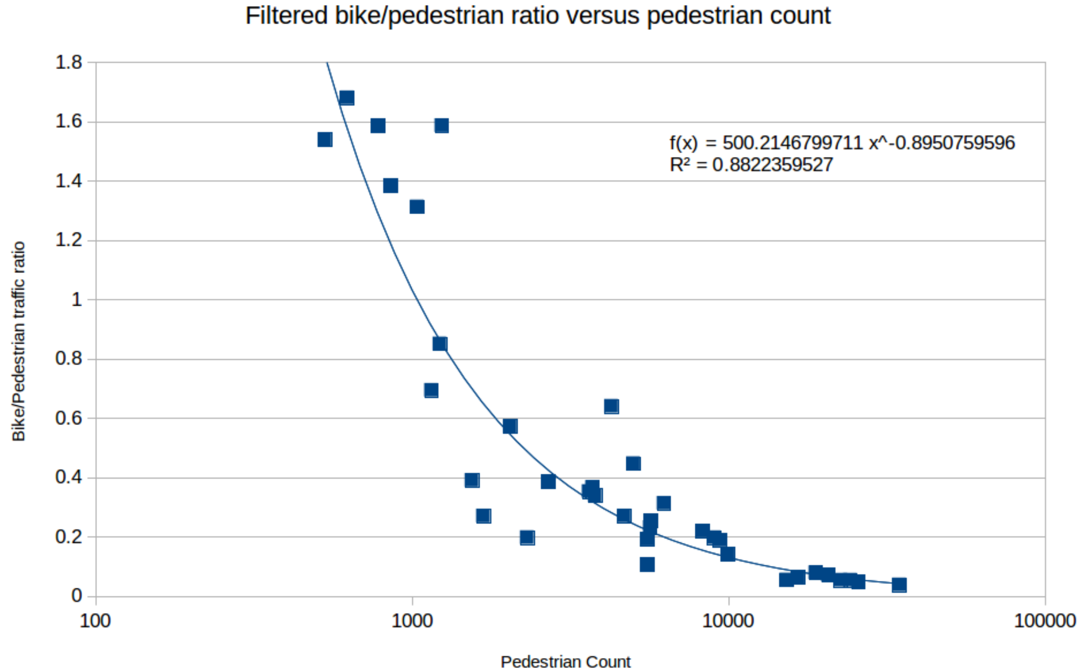
Out[1]:

We want to build a machine learning model that looks at various intersections in Toronto and predicts the collision rates expected from cyclists at each point. Using the accident history from 1986-2010, along with information such as the intersection's Bike traffic, control type, and classification (local, major arterial etc.) we should be able to use a supervised learning approach, training our algorithm on a subset of the collisions available, and the validating on other subsets. Once we are satisfied with the performance (and at some point we'll need to define where our cut-off for prediction should be), we will compare the output by applying it to accident data scrapped from a twitterbot.

Note that we do not include whether or not an intersection has a bike lane in our list of features. While one can download a dataset which includes all locations of cycle tracks and bike lanes by street sections, many are fairly new. With the exception of certain cycle tracks which were built after 2014, there doesn't seem to be any easy way to date various bike lane additions. As such it wouldn't make sense to label our intersections with this information, as most bike lanes and cycle tracks are fairly new, and were not around during the period of time our dataset is built from.

Since we don't have access to cycle app or bikeshare data to get accurate bike counts for a variety of locations, we'll need to make due with a limited dataset of city manual and automatic bike counts. These span only a few points in Toronto, and more importantly, are focused on Downtown.

From the bike counts above, we compared to vehicle and pedestrian counts from a sepearate data set. These are far more wide ranging, and if a relationship could be found, one could make predictions on bike traffic in all the missing points.



While there was unfortunately no obvious direct relationship between traffic numbers (something like a constant ratio, or a linearly increasing one), we do get an acceptable power fit when comparing the bike-to-pedestrian traffic ratio, once we filter out a few areas. The University of Toronto campus has a much higher B/P ratio than any other avaialble bike count locations. This isn't suprising, as a student there I can confirm it's the most popular place to see cyclists in the city. The other area is the waterfront right by financial and entertainment districts. These points had a far lower traffic ratio than given by this fit. So we'll treat them as seperate special cases, mapping the available bike counts directly to any nearby collisions, as long as they fit within a 'box' that we'll use to define those areas.

Note, done in openoffice spreadsheet. Should transfer over into Python for a prettier graph. Will perhaps change the fit equation a bit

Next we needed to clean and filter our dataset. Cleaning mostly invovled searching for NaNs or mixed up entries, as it prevents us from easily manipulating our dataframe. Once we have things roughly cleaned up, we want to filter our collisions. The bike count to pedestrian count relationship we had was based on points that were overwhelmingly found in central Toronto; downtown and the surrounding neighborhoods. So we need some way to do this

In [2]:
```
col_df = pd.DataFrame.from_csv('collisions_clean.csv', index_col='ID')
col_df=col_df.sort_index()
col_df.head()
```

Out[2]:

	INJURY	SAFETY EQUIP.	ROAD CLASS	CYCLIST CRASH TYPE	AGE OF CYCLIST	STNAME1	STET 1 TYPE	STNAME 2	STET 2 TYPE	LON(
ID										
1	Minimal	Unrecorded	Minor Arterial	Unrecorded	34.0	BIRCHMOUNT	RD	HIGHVIEW	AV	-79.2
2	Minimal	Unrecorded	Major Arterial	Unrecorded	54.0	LAKE SHORE	BLVD	THIRTY-FIFTH	ST	-79.5
3	Minor	Unrecorded	Major Arterial	Unrecorded	19.0	LAWRENCE	AV	FORTUNE	GATE	-79.2
4	Minimal	Unrecorded	Local	Unrecorded	34.0	EUCLID	AV	ULSTER	ST	-79.4
5	Minimal	Unrecorded	Major Arterial	Unrecorded	34.0	AVENUE	AV	DRAYTON	AV	-79.3

5 rows × 21 columns

The above collisions were all geocoded based on the crosstreets listed, rather than the GPS coordinates, since there were many coordinates that just were defaulted to the center of Toronto. Nonetheless, there were still a few "impossible" intersections, that were then geocded in a second pass using their coordinates.

We also use a list of ALL the intersections in the city, and geocode those, using the GPS coordinates. These also required a second pass using a different geocoder on about 10 or so entries. Later we'll cross reference with the collisions database and drop all the duplicates. This way we can keep track of the "zeroes". That is, intersections which never had a collision in those 25 years.

Below is an example of how this was done, for the entire list of intersections in the city. We don't recommened running this, as it'll take a few hours.

Skip cell below!

In [ ]:

```
#our datasetfrom the Toronto website of all intersections
all_real_df = pd.DataFrame.from_csv('all_intersections_real.csv', index_col='i
#Get the coordinates out, put them as a list, rather than iterating over the d
coords2 = all_real_df[['latitude','longitude']]
coords2 = coords2.replace(np.nan,0)
coordinate_list = coords2.values.tolist()
import geocoder
import csv
geo = []
idx = 0
for pair in coordinate_list:
    g = geocoder.arcgis(pair, method='reverse')
    geo.append(g.address)
    print(idx, '', geo[idx])
    idx += 1

myfile = open('centerline_coords_int.csv', 'w')
wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
wr.writerow(geo)

all_real_df['arcgis_int'] = geo
```

In [3]:

```
import geocoder
#Example of what geocoder output looks like, for two different locations. The
latlng = [43.6532,-79.3828]
g = geocoder.arcgis('Bay St & Bloor St, Toronto', method='geocode')
f= geocoder.arcgis(latlng, method='reverse')
print(g.address)
print(f.address)
```

```
Bay St & Bloor St W, Toronto, Ontario, M5R
Bay St & Albert St, Toronto, Ontario, M5G
```

Once this is done, we then add this "intersection" column to our dataframe, and selected all collisions that had a postal code beginning with one of those associated with the areas roughly east of the Junction, West of carlaw, and south of Dupont, with the neigbourhoods near Exhibition place also thrown out. This isn't a perfect system by any means, but we have no reason to believe that our nice power-fit relationship will hold outside the areas we used. In fact, outside central Toronto, it seems likely that it wouldn't hold, as most cyclists are typically found closer to the city center.

The SQl query to do this is:

SELECT *

FROM kit_farfan.centreline_reversegeocoded

WHERE "arcgis_int" LIKE '%M7A%' OR "arcgis_int" LIKE '%M6G%' OR "arcgis_int" LIKE '%M6H%' OR

"arcgis_int" LIKE '%M6P%' OR

"arcgis_int" LIKE '%M5A%' OR "arcgis_int" LIKE '%M5B%' OR "arcgis_int" LIKE '%M5C%' OR

"arcgis_int" LIKE '%M5E%' OR "arcgis_int" LIKE '%M5G%' OR "arcgis_int" LIKE '%M5H%' OR

"arcgis_int" LIKE '%M5J%' OR "arcgis_int" LIKE '%M5K%' OR "arcgis_int" LIKE '%M5L%' OR

"arcgis_int" LIKE '%M5S%' OR "arcgis_int" LIKE '%M5T%' OR

"arcgis_int" LIKE '%M5V%' OR "arcgis_int" LIKE '%M5W%' OR "arcgis_int" LIKE '%M5X%' OR

"arcgis_int" LIKE '%M4Y%' OR "arcgis_int" LIKE '%M4X%' OR "arcgis_int" LIKE '%M4W%' OR

"arcgis_int" LIKE '%M4X%' OR "arcgis_int" LIKE '%M4W%'

Did it in SQL originally since exploratory data analysis was done using Mode Analytic's platform.

In [4]:
```
central_realintersections_df = pd.DataFrame.from_csv('central_real_intersectio
central_realintersections_df= central_realintersections_df.rename(columns = {'

col_df = pd.DataFrame.from_csv('toronto_cycling_central_mode.csv', index_col='
col_df = col_df.sort_index()

central_col_df = col_df
from numpy import random
from scipy.spatial import distance
import matplotlib.path as mplPath
import numpy as np

#We want to follow a standard intersection convention to make our life easier.
#Unfortunately the free geocoder with unlimited request numbers per day doesn'
#So we will get the intersections, strip out the two streets, and order them a
st2 = col_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post = col_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 = col_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1] + ', ' +str(pair2[2]))
    intersection_list.append(intersection)

col_df['intersection'] = intersection_list

st2 = central_realintersections_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post =  central_realintersections_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 =  central_realintersections_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1] + ', ' +str(pair2[2]))
    intersection_list.append(intersection)
```

Once we have our naming convention sorted, we can start by trying to predict the Collision rates naively, without any machine learning. What do the raw numbers tell us?

We proceed by mapping the pedestrian data to our collisions and intersections, so we have a Bike traffic estimate. Rememebering that we have a few areas that we're treating apart, we have our boxes so we can check if points fall inside those areas.

In [5]:

```
ped_counts_df = pd.DataFrame.from_csv('Vehicle and Pedestrian Counts/TrafficPe

#Using the Power fit for the Bike/Pedestrian ratio, we get a function that pre
#intersection.

ped_counts_df['bike_prediction'] = (500.2146799711*ped_counts_df['8HrPedVol']*
ped_coords = ped_counts_df[['Latitude','Longitude']]
ped_coords = ped_coords.replace(np.nan,0)
ped_coordinate_list = ped_coords.values.tolist()
ped_counts_df['coordinates'] = ped_counts_df[['Latitude','Longitude']].apply(t
ped_counts_df.head()
```

Out[5]:

	Main	Midblock Route	Side 1 Route	Side 2 Route	Activation Date	Latitude	Longitude	Count Date	8HrPedVol	8H
PX2										
2	JARVIS ST	NaN	FRONT ST E	NaN	1948/11/15	43.649450	-79.371410	2011/09/08	17008	193
3	KING ST E	NaN	JARVIS ST	NaN	1950/08/23	43.650461	-79.371924	2011/09/07	37719	176
4	JARVIS ST	NaN	ADELAIDE ST E	NaN	1958/09/12	43.651534	-79.372360	2008/06/16	1991	197
5	JARVIS ST	NaN	RICHMOND ST E	NaN	1962/04/21	43.652718	-79.372824	2009/07/30	2696	248
6	JARVIS ST	NaN	QUEEN ST E	NaN	1928/08/24	43.653704	-79.373238	2011/05/18	3622	197

In [6]:

```
ped_dict = ped_counts_df.set_index('coordinates').to_dict()['bike_prediction']
col_df['coordinates'] = col_df[['lat','long']].apply(tuple, axis=1)
col_df.head()
central_realintersections_df['coordinates'] = central_realintersections_df[['l
equiv = {'Laneway':3,'Minor':2,'Major':1}
central_realintersections_df['num_class'] = central_realintersections_df['elev
central_realintersections_df= central_realintersections_df.sort_values(by=['in
#take the lowest numerical class, drop the other duplicates.
central_realintersections_df = central_realintersections_df.drop_duplicates(su
```

In [7]:
```
waterfront_Path = mplPath.Path(np.array([[43.635497, -79.398156],
[43.639000, -79.400725],
[43.640822, -79.401427],
[43.646984, -79.376977],
[43.649889, -79.370343],
[43.651614, -79.362725],
[43.648090, -79.361191],
[43.646451, -79.361937],
[43.641209, -79.376739],
[43.639969, -79.379965],
[43.637698, -79.391847],
[43.635666, -79.398368],
[43.636489, -79.399603]]))

campus_Path = mplPath.Path(np.array([[43.659838, -79.399772],
[43.661388, -79.401006],
[43.665592, -79.402705],
[43.666768, -79.401354],
[43.668213, -79.393958],
[43.663141, -79.392719],
[43.659264, -79.394100],
[43.658329, -79.398204]]
))

castleFrank_Path = mplPath.Path(np.array([[43.672105, -79.376696],
[43.671562, -79.370962],
[43.674418, -79.366821],
[43.676086, -79.358731],
[43.677056, -79.354021],
[43.677040, -79.355126],
[43.677622, -79.358516],
[43.676194, -79.359503],
[43.675170, -79.364760],
[43.674580, -79.367539],
[43.672019, -79.371112],
[43.672710, -79.376927]]
))

campus = [[43.657946,-79.39993],
          [43.663502,-79.40005],
          [43.663051,-79.402196],
          [43.665429,-79.398975]
         ]

campus_dict = {(43.657946,-79.39993):4495.8,
               (43.663502,-79.400050):2653,
               (43.663051,-79.402196):3574,
               (43.665429,-79.398975):2304
              }

waterfront = [[43.648208,-79.370923],
              [43.642711,-79.394043],
              [43.639944,-79.387032],
              [43.640625,-79.3932],
              [43.640093,-79.380152]
             ]

waterfront_dict = {(43.648208,-79.370923):330,
              (43.642711,-79.394043):745,
              (43.639944,-79.387032):128,
              (43.640625,-79.3932):154,
              (43.640093,-79.380152):235
             }
```

Here we have defined our boxes, and our pedestrian traffic dictionary. Now we go through the data sets, check if the point belongs in one of our boxes. If it does, we'll map the closest traffic count in the box. Otherwise, we find the closest intersection in the Pedestrian dataframe, and use the predicted bike traffic number from our fit

In [8]:
```
import csv
closest_traffic_point = []
bike_traffic = []
i = 0
for i in range(0,len(central_col_df)):
    point = central_col_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(ped_dict[tuple(closest)])

myfile3 = open('closest_intersection.csv', 'w')
wr = csv.writer(myfile3)
wr.writerow(closest_traffic_point)
myfile3.close()

myfile4 = open('closest_int_bike_predictions.csv', 'w')
wr = csv.writer(myfile4)
wr.writerow(bike_traffic)
myfile4.close()
```

In [9]:
```
central_col_df['closest_traffic'] = tuple(closest_traffic_point)
central_col_df['traffic_count'] = bike_traffic

central_col_df.rename(columns={'closest_traffic': 'closest_ped_count', 'traffi
central_col_df.head()
```

Out[9]:

	injury	safety_equip	road_class	cyclist_crash_type	age_of_cyclist	stname1	stet_1_type	stname_2
id								
4.0	Minimal	Unrecorded	Local	Unrecorded	34.0	EUCLID	AV	ULSTER
7.0	Minimal	Unrecorded	Major Arterial	Unrecorded	34.0	SALEM	AV	FERNBAN
8.0	Minor	Unrecorded	Major Arterial	Unrecorded	24.0	DUNDAS	ST	BEVERLEY
9.0	Minimal	Unrecorded	Major Arterial	Unrecorded	24.0	BLOOR	ST	MARGUEF
10.0	Minor	Unrecorded	Minor Arterial	Unrecorded	54.0	AVENUE	RD	SHANLY

5 rows × 25 columns

In [10]:
```
import csv
closest_traffic_point = []
bike_traffic = []
i = 0
for i in range(0,len(central_realintersections_df)):
    point = central_realintersections_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(ped_dict[tuple(closest)])

myfile5 = open('closest_intersection_centreline.csv', 'w')
wr = csv.writer(myfile5)
wr.writerow(closest_traffic_point)
myfile5.close()

myfile6 = open('closest_int_bike_predictions_centreline.csv', 'w')
wr = csv.writer(myfile6)
wr.writerow(bike_traffic)
myfile6.close()

central_realintersections_df['closest_traffic'] = tuple(closest_traffic_point)
central_realintersections_df['traffic_count'] = bike_traffic

central_realintersections_df.rename(columns={'closest_traffic': 'closest_ped_c
central_realintersections_df.sort_values(by='traffic_count').tail()
```

Out[10]:

	intersec5	elev_id	classifi6	classifi7	num_elev	elevatio9	elevatio10	elev_level	elevation	eleva
int_id										
13462813.0	Castle Frank Rd / Bloor St E	33572.0	MJRSL	Major-Single Level	1.0	501200.0	Major	0.0	0.0	NaN
14021473.0	Glen Morris St / Ln E Spadina N Harbord / Ln E...	16644.0	LSRSL	Lesser-Single Level	1.0	501700.0	Laneway	0.0	0.0	NaN
13465137.0	Galbraith Rd / St George St	5198.0	MNRSL	Minor-Single Level	1.0	501300.0	Minor	0.0	0.0	NaN

So now we have traffic numbers for our entire intersection dataframe, as well as our dataframe of collisions. Now let's count up our collisions (including the zeroes), and order them.

To make things cleaner, we'll strip away the columns that aren't helpful to us at this time. Once we group the collision dataframe by intersection, we can concat that with the dataframe of all intersections (now call zeroes_df).

Once it's sorted by our normalized accident rate, we can drop the non-zero intersections from our merged dataframe by just dropping the duplicated indexes. Then we should be left with our collisions, properly grouped, and the "zero" intersections.

In [11]:
```
central_realint_df = central_realintersections_df[['intersection','coordinates
central_realint_df.sort_values(by='traffic_count')

result = pd.merge(central_col_df,central_realint_df, how ='outer',on='intersec
result = result.sort_values(by='intersection')
traff2 =result['traffic_count_y']
result['traffic_count_x'] =result['traffic_count_x'].fillna(value = traff2 )
```

In [12]:
```
result.to_csv('merged_data.csv')
intersection_df = central_col_df.groupby('intersection').agg({'traffic_count':
intersection_df.columns = intersection_df.columns.set_levels(['traffic_count']
zeroes_df = central_realint_df.groupby('intersection').agg({'traffic_count':[n
zeroes_df['traffic_count','size'] = 0
#Let's use 360 as the days in the year. It's a nice round number. And since we
zeroes_df['traffic_count','normalized_accident_rate'] = zeroes_df['traffic_cou
zeroes_df = zeroes_df.sort_values(by=[('traffic_count','normalized_accident_ra
intersection_df['traffic_count','normalized_accident_rate'] = intersection_df[
intersection_df = intersection_df.sort_values(by=[('traffic_count','normalized
intersection_df.head(20)

totals_df = pd.concat([intersection_df,zeroes_df])
df_test = totals_df['traffic_count','normalized_accident_rate']
df = df_test.to_frame(name='normalized yearly accident rate')
df['total collisions'] = totals_df['traffic_count','size']
df['traffic estimate'] = totals_df['traffic_count','mean']
#df = df.dropna()
#Here we make the index a column, drop the duplicates from this intersection c
df = df.reset_index().drop_duplicates(subset='intersection', keep='first').set
len(df)
```

Out[12]: 2078

Now we can take a look at the dataframe we've created. The one we'll use for our machine learning approach will be roughly similar, but instead we'll have carried over the information on the type of intersection, and the type of control (stop sign, yield, no control, traffic lights etc). These will be part of the feature set we use.

For now though, we'll just proceed via Emperical Baysian analysis. We want to account for the fact that collisions are very rare events. Without a very long period of time, it would be unwise to assume that our sample distribution properly matches the actual probability of collisions.

In fact we can see that there is a strong "break" away from roughly linear spread when we plot the accident yearly rate against the total collisions.

Should I be concerned about this linear trend? I half expected a more uniform spread, which I believe is what one sees in the case of automobile collisions. Obviously, the power-fit that I used to estimate traffic numbers does mean that I'm imposing a shape on this distribution. So that makes sense. Accepting that, we can obviously still spot the outliers.

In [13]:
```
pd.set_option('display.float_format', '{:.4g}'.format)
df.sort_values(by='total_collisions', ascending = False).head()
```

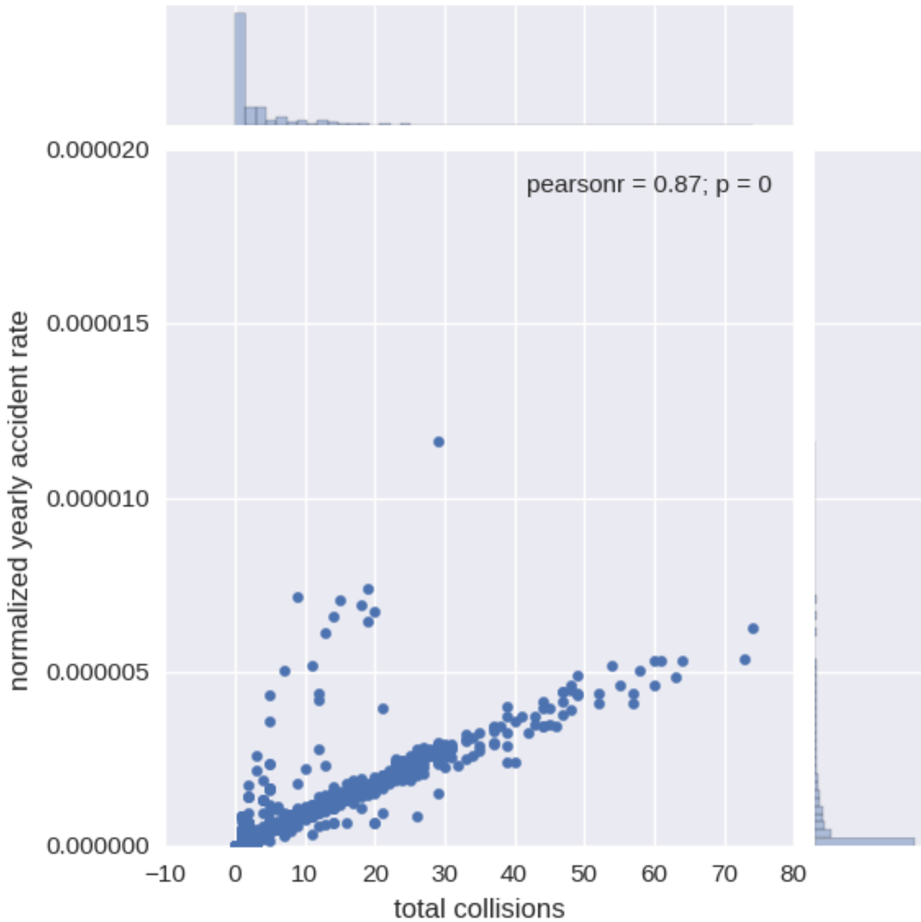
Out[13]:

	normalized yearly accident rate	total collisions	traffic estimate
intersection			
Queen St W & Spadina Ave, Toronto, Ontario, M5V	6.256e-06	74	1314
Bay St & Dundas St W, Toronto, Ontario, M5G	5.393e-06	73	1504
Queen St W & University Ave, Toronto, Ontario, M5H	5.315e-06	64	1338
Bloor St E & Yonge St, Toronto, Ontario, M4W	4.861e-06	63	1440
Bathurst St & Queen St W, Toronto, Ontario, M5T	5.334e-06	61	1271

In [14]:
```
df.loc['Jarvis St & King St E, Toronto, Ontario, M5A']
```

Out[14]:
```
normalized yearly accident rate   7.352e-08
total collisions                          1
traffic estimate                       1511
Name: Jarvis St & King St E, Toronto, Ontario, M5A, dtype: float64
```

In [15]:
```
sns.set_context("notebook", font_scale=1.1)
scatter = sns.jointplot(x='total collisions', y = 'normalized yearly accident
```



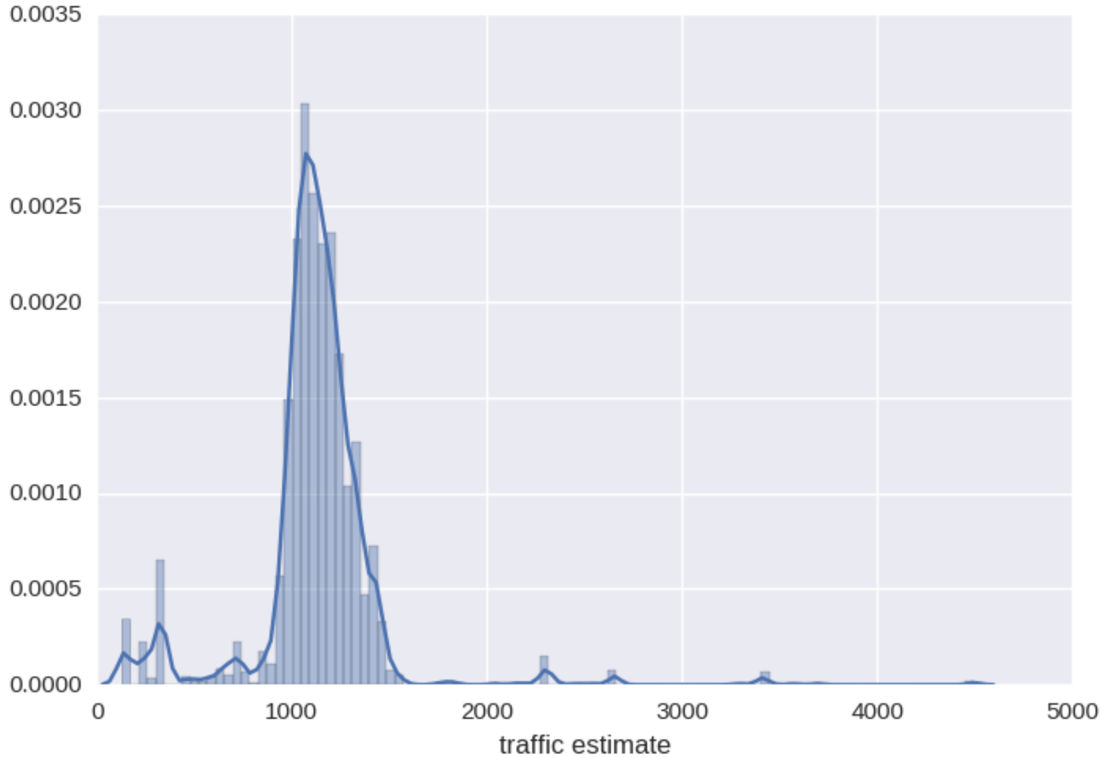
So we'll want to create a prior estimate on what the distribution actually is, using the samples that we have available. In a sense, we're using the data twice. First to estimate our prior. Then we'll use it along with this prior to create a posterior distribution.

There is a lot of discussion revolving around Full Bayesian analysis, versus Empirical Bayesian Analysis. In some sense, if you know enough about the system you are studying, you should be able to develop a good prior beforehand, rather than using your limited dataset to do so. On the otherhand, imposing beliefs about your system beforehand can be equally controversial to depending on the data twice.

In [16]:
```
df.to_csv('totals_test.csv')
import scipy.stats as stats
sns.distplot(df["traffic estimate"], bins = 100)
```

```
/home/christian/anaconda3/lib/python3.5/site-packages/statsmodels/nonparametr
ic/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number inst
ead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

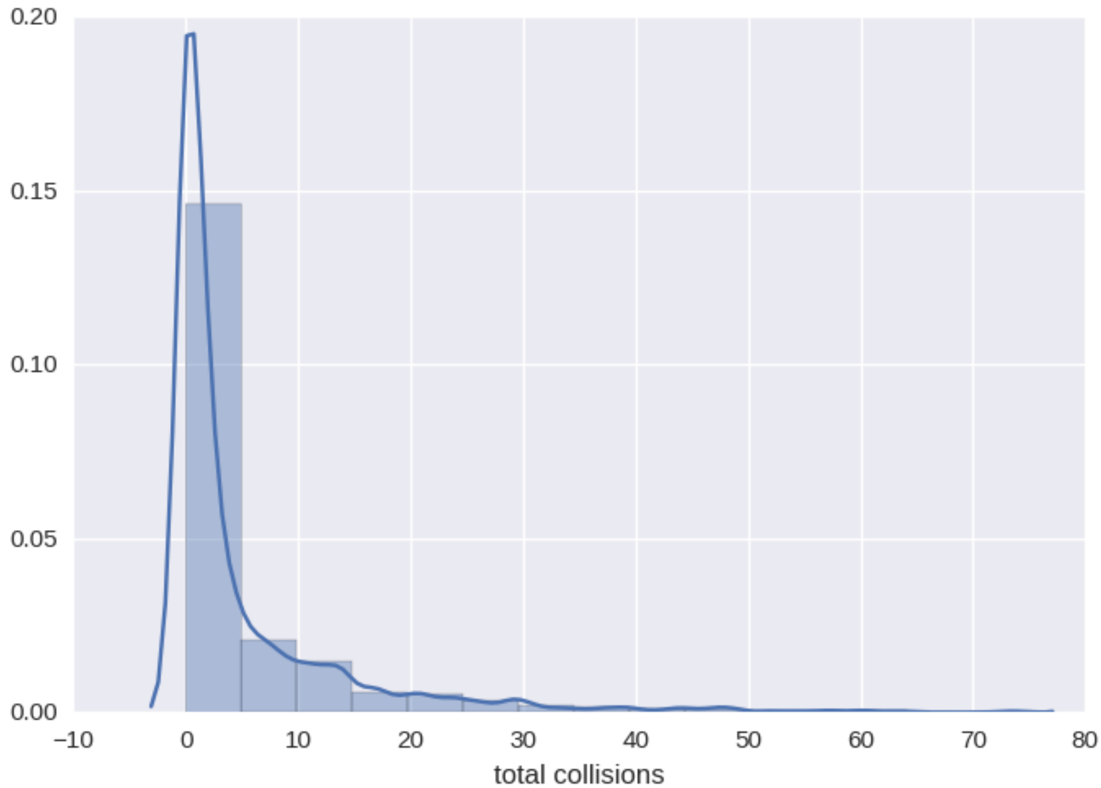
Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f802a0e5940>
```



In [17]:
```
sns.distplot(df["total collisions"], bins = 15)
```

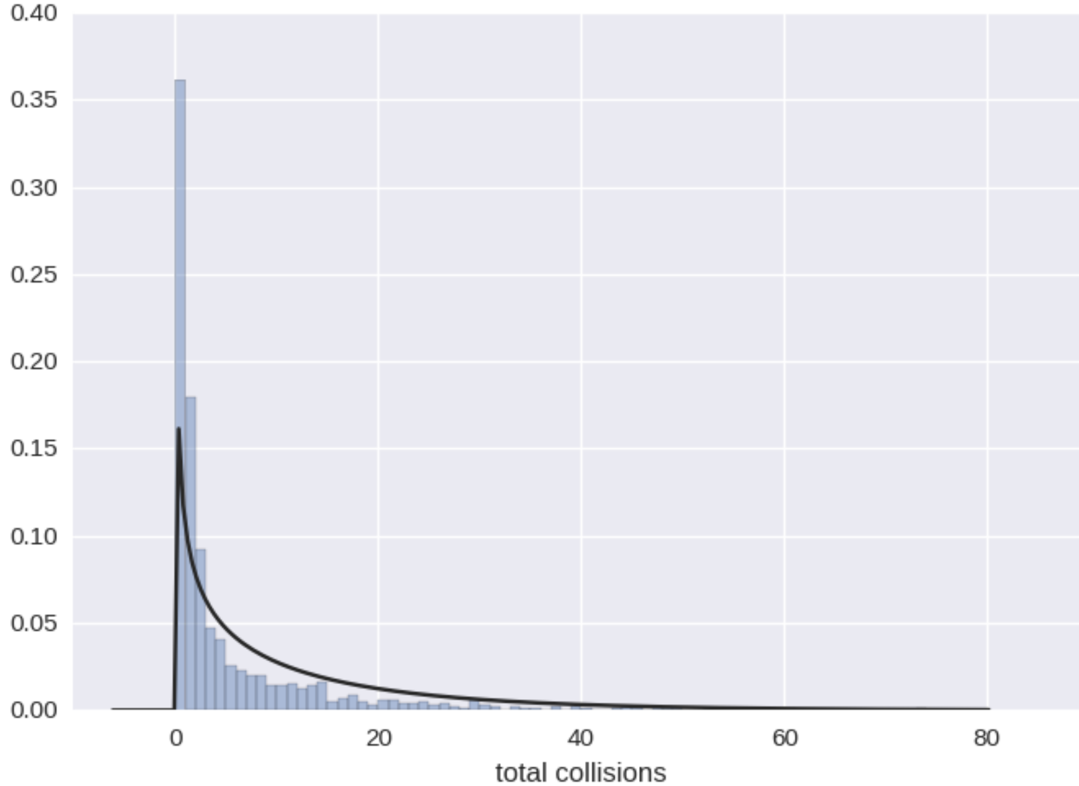
```
/home/christian/anaconda3/lib/python3.5/site-packages/statsmodels/nonparametr
ic/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number inst
ead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x7f802a0ab860>



In [18]:
```
sns.distplot(df["total collisions"], fit=stats.gamma,  bins=range(0, 75, 1), k
```

Out[18]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f8029e93da0>`



A gamma distribution looks like a good choice to fit our prior. We know that we are dealing with sums of poission distributions where each intersection is basically a poisson process, with the mean accidents for a year being the parameter. So the above histogram is really summing up the number of accidents each year, over these poisson processes. While we could try and fit each intersection individually, this is a whole lot of work. And it's not clear how one would use these estimated parameters to compute the prior distribution for the ensemble.

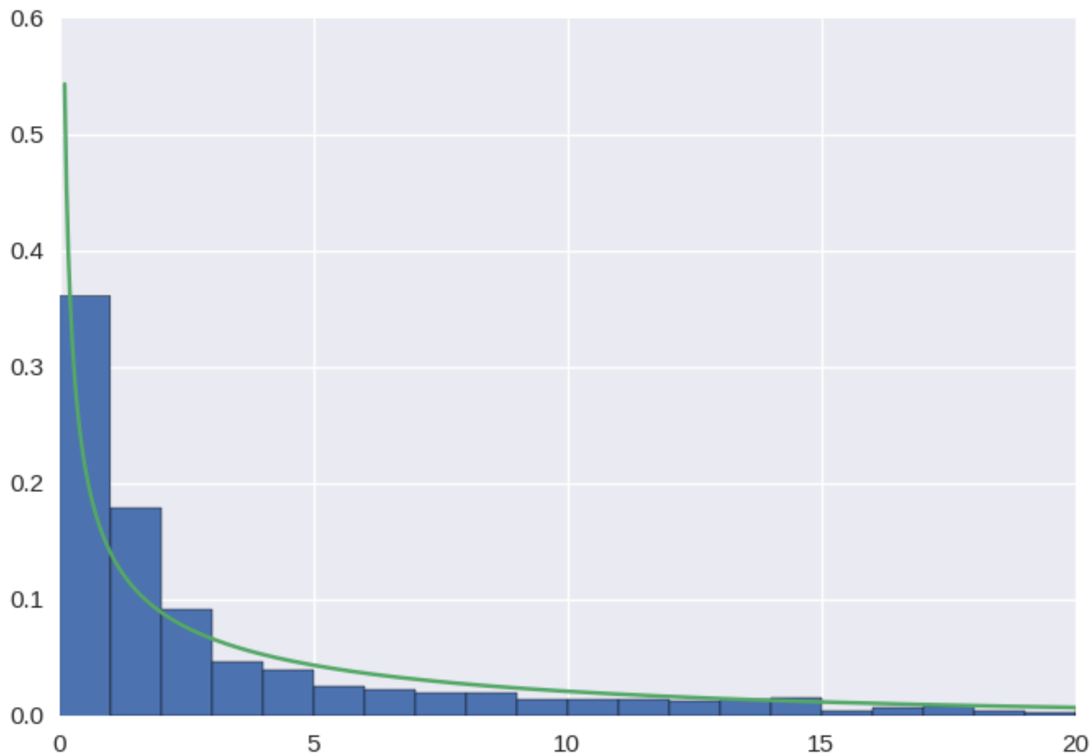
A gamma distribution looks nice, and intuitively makes sense since the prior for a poisson distribution is a gamma. Trying to read up on the intuition for how a ensemble of Poisson processes, each with their own paramater (not merely the same), has lead me to some very involved work on Bayesian inference for traffic problems, without directly using machine learning. We won't go that deep in this case though. Suffice to say, there are extensions to the above Poisson model, which attempt to account for the heterogenity of various intersections. The same features one would choose to use in a machine learning approach would show up as parameters in the distribution one tries to fit, and the poisson parameters for each intersection is assumed to be a random variablle, drawn from a chosen distribution

An overview of 3 models which extend our basic method is found in (Miranda-Mereno et al 2005): http://www.civil.uwaterloo.ca/itss/papers%5C2005-2%20(Alternative%20risk%20models%20for%20ranking%20sites).pdf (http://www.civil.uwaterloo.ca/itss/papers%5C2005-2%20(Alternative%20risk%20models%20for%20ranking%20sites).pdf)

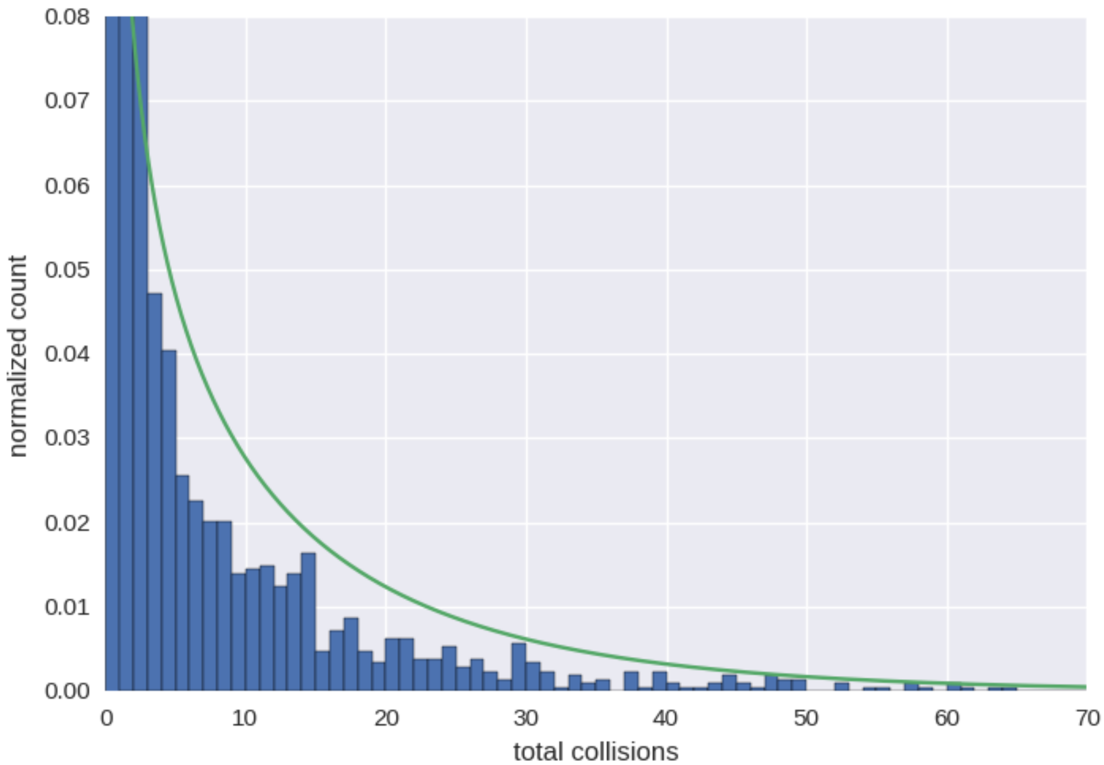
In [19]:
```
stats.gamma.fit(df['total collisions'], loc =0)
```

Out[19]: (0.43930468201111406, -7.7697069386935107e-32, 15.102836399931292)

In [23]:
```
import matplotlib.pylab as plt

#plot normed histogram
plt.hist(df['total collisions'], normed=True,  bins=range(0, 75, 1))

# find minimum and maximum of xticks, so we know
# where we should compute theoretical distribution
xt = plt.xticks()[0]
xmin, xmax = 0.1, 75
lnspc = np.linspace(xmin, xmax, len(df))

# exactly same as above
ag,bg,cg = stats.gamma.fit(df['total collisions'], loc=0)
pdf_gamma = stats.gamma.pdf(lnspc,ag, bg,cg)
plt.plot(lnspc, pdf_gamma, label="Gamma")

plt.axis([0,20,0,0.6])
plt.show()
```



In [25]:
```
#plot normed histogram
plt.hist(df['total collisions'], normed=True,  bins=range(0, 75, 1))

# find minimum and maximum of xticks, so we know
# where we should compute theoretical distribution
xt = plt.xticks()[0]
xmin, xmax = 0.1, 75
lnspc = np.linspace(xmin, xmax, len(df))

# exactly same as above
ag,bg,cg = stats.gamma.fit(df['total collisions'])
pdf_gamma = stats.gamma.pdf(lnspc,ag, bg,cg)
plt.plot(lnspc, pdf_gamma, label="Gamma")
plt.xlabel('total collisions')
plt.ylabel('normalized count')
plt.axis([0,70,0,0.08])
plt.show()
```



That's a pretty good looking fit. The histogram shows the normalized counts of integer collision totals for our merged dataframe. We can now use the fit paramters of our model, and use this to adjust our estimate of the accident rate.

The mathematics of why the calculation below works to update our estiamte is straight forward to derive from Bayesian probablity theory, but intuitively it just looks likes a weighted average of the prior and sample estimates (when expaneded), where more weight is given to the sample data when the time period is larger, or when the traffic numbers are greater. This makes sense, as in those cases we are MORE confident in our data providing us with a good idea of probability distibution for that intersection.

In [26]:
```
beta = 1/cg
df['posterior mean'] = (ag + df['total collisions'])/(beta+25*360*df['traffic
df['posterior STD'] = np.sqrt((ag + df['total collisions'])/((beta+25*360*df['
pd.set_option('display.float_format', '{:.5g}'.format)
df.sort_values(by='posterior mean',ascending = False).head(10)
```

Out[26]:

intersection	normalized yearly accident rate	total collisions	traffic estimate	posterior mean	posterior STD
Front St W & York St, Toronto, Ontario, M5J	1.1649e-05	29	276.61	1.19e-05	2.1863e-06
Queens Quay W & Rees St, Toronto, Ontario, M5J	7.1485e-06	9	139.89	7.6438e-06	2.464e-06
Bay St & Queens Quay W, Toronto, Ontario, M5J	7.3792e-06	19	286.09	7.6214e-06	1.7205e-06
Station St & York St, Toronto, Ontario, M5J	7.0922e-06	15	235	7.387e-06	1.8689e-06
Queens Quay W & York St, Toronto, Ontario, M5J	6.9128e-06	18	289.32	7.1523e-06	1.6574e-06
The Esplanade & Yonge St, Toronto, Ontario, M5E	6.734e-06	20	330	6.944e-06	1.5291e-06
Lake Shore Blvd W & York St, Toronto, Ontario, M5J	6.6194e-06	14	235	6.9142e-06	1.8081e-06
Front St W & John St, Toronto, Ontario, M5V	6.469e-06	19	326.34	6.6813e-06	1.5083e-06
Lake Shore Blvd E & Yonge St, Toronto, Ontario, M5E	6.1466e-06	13	235	6.4414e-06	1.7452e-06
Queen St W & Spadina Ave, Toronto, Ontario, M5V	6.2559e-06	74	1314.3	6.3087e-06	7.303e-07

In [27]:
```
df = df.sort_values(by='total collisions', ascending=False)
```

In [28]:
```
df.head()
```

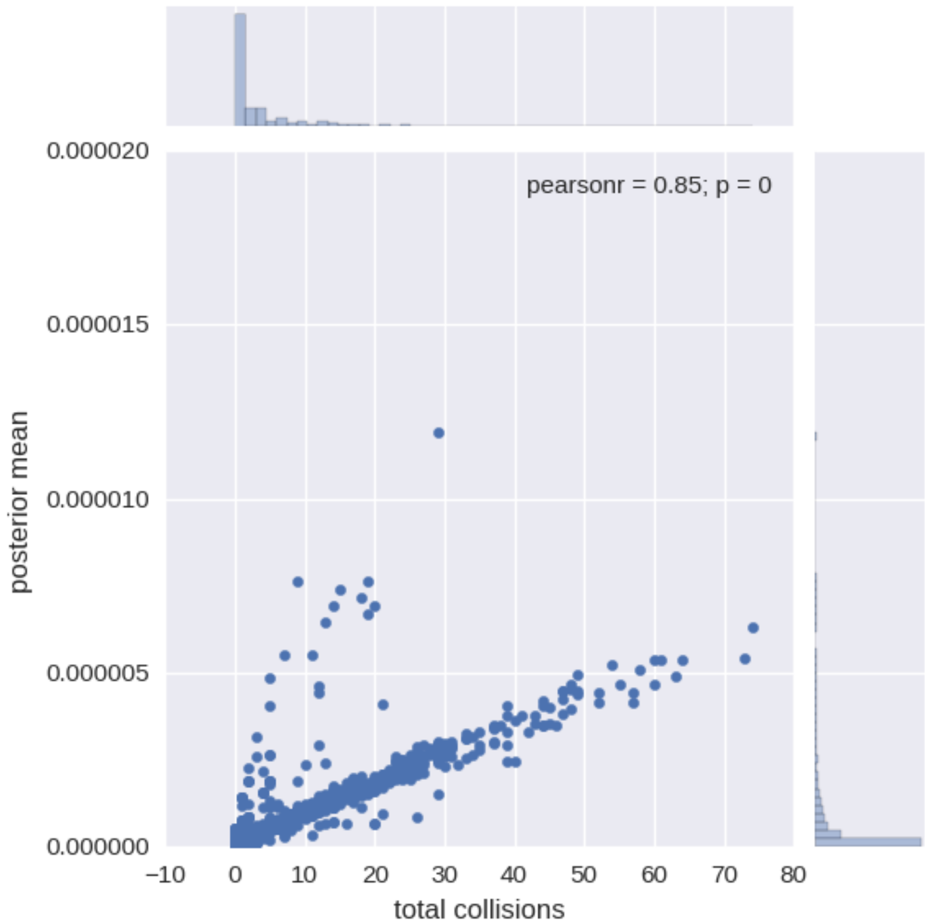
Out[28]:

intersection	normalized yearly accident rate	total collisions	traffic estimate	posterior mean	posterior STD
Queen St W & Spadina Ave, Toronto, Ontario, M5V	6.2559e-06	74	1314.3	6.3087e-06	7.303e-07
Bay St & Dundas St W, Toronto, Ontario, M5G	5.393e-06	73	1504	5.4391e-06	6.339e-07
Queen St W & University Ave, Toronto, Ontario, M5H	5.3153e-06	64	1337.9	5.3671e-06	6.6764e-07
Bloor St E & Yonge St, Toronto, Ontario, M4W	4.8614e-06	63	1439.9	4.9095e-06	6.1551e-07
Bathurst St & Queen St W, Toronto, Ontario, M5T	5.3338e-06	61	1270.7	5.3883e-06	6.864e-07

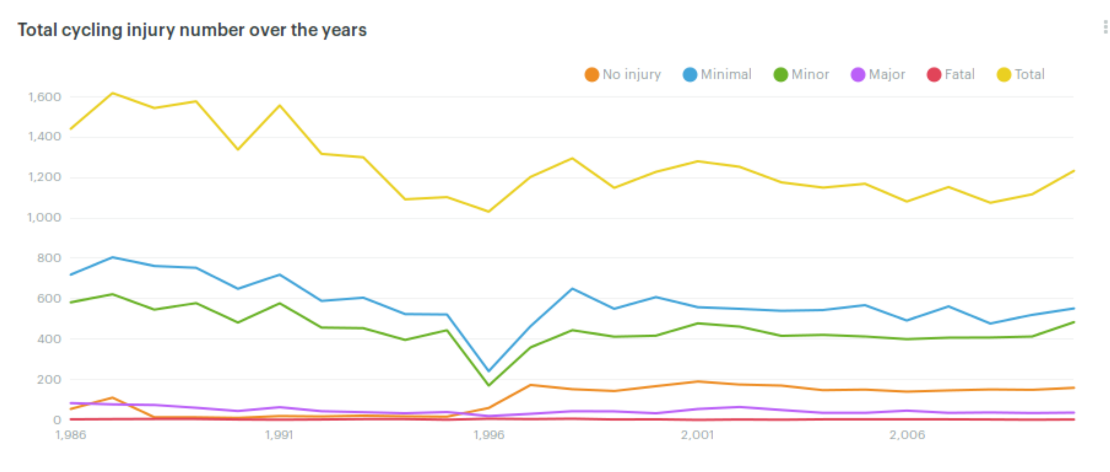
In [29]:
```
df.to_csv('eba_normalizedcollisions.csv')
```

So now we have our dataframe with posterior yearly accident rate, as well as a posterior standard deviation. If one looks deeper throughout the data set we see that the order changes slightly, but nothing too crazy. Since we're using so many years of data, the "prior" dominates. The biggest changes are for those intersections that have a very low traffic count, comparatively. Those with high traffic counts have smaller changes.

In [30]:
```
scatter = sns.jointplot(x='total_collisions', y = 'posterior_mean', data = df.
```



It is at this point that we may start wondering whether using ALL 25 years of data makes sense. If the predictions are to be for the past year, then surely data closer to this time frame is more applicable. We saw in our exploratory data analysis that there was a total decline in accidents year by year up until 1996. There was a sharp decline in minimal and minor injuries that year, with the slack partly picked up by more "No injuries". Whether this was due to a change in how accidents were classified, or due to the legislation mandating minors to use helmets is hard to say. Likely it could be both!



After this year however, cycling accidents pick up again. and start looking fairly consistent. Expecially as compared to the first dozen years.

It may be that predictive power would improve if we limited the data set to 1998 onwards. This would roughly half our number of starting collisions, but those remaining may be more indicative of current cycling trends. It also avoids that obvious decreasing trend and sudden jump in 96-97.

Moving forward, we can compare the predicted collision numbers to those observed over the past year, from the Twitterbot. We will have a trained model to also compare prediction numbers. One hopes that the increased sophistication of the machine learning approach, using the intersection features which are available, will get closer to the observed data.

In [31]:
```
full_df = pd.DataFrame.from_csv('collisions_clean.csv', index_col='ID')
```

In [32]:
```
fullgeocoded_df = pd.DataFrame.from_csv('collisions_geocode.csv', index_col ='
```

In [33]:
```
fullgeocoded_df.head()
```

Out[33]:

	INJURY	SAFETY EQUIP.	ROAD CLASS	CYCLIST CRASH TYPE	AGE OF CYCLIST	STNAME1	STET 1 TYPE	STNAME 2	STET 2 TYPE	LONG
ID										
3558	Minimal	No equipment available	Local	Unrecorded	11	ELEVENTH	ST	LAKE SHORE	BLVD	-79.50
499	Minimal	Unrecorded	Major Arterial	Unrecorded	19	ELEVENTH	ST	LAKE SHORE	BLVD	-79.51
1529	Minor	Unrecorded	Major Arterial	Unrecorded	19	LAKE SHORE	BLVD	ELEVENTH	ST	-79.50
7880	Minor	No equipment available	Major Arterial	Unrecorded	42	LAKE SHORE	BLVD	ELEVENTH	ST	-79.50
10049	Minor	Equipment not used but available	Local	Unrecorded	18	MORRISON	ST	ELEVENTH	ST	-79.50

5 rows × 22 columns

In [34]:
```
fullgeocoded_df= fullgeocoded_df.rename(columns = {'INTERSECTION':'intersectio

col_df = fullgeocoded_df.sort_index()
all_realintersections_df = pd.DataFrame.from_csv('real_reversegeocoded.csv')
all_realintersections_df= all_realintersections_df.rename(columns = {'arcgis_i

from numpy import random
from scipy.spatial import distance
import matplotlib.path as mplPath
import numpy as np

#We want to follow a standard intersection convention to make our life easier.
#Unfortunately the free geocoder with unlimited request numbers per day doesn'
#So we will get the intersections, strip out the two streets, and order them a
st2 = col_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post = col_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 = col_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str) and isinstance(pair[1],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1]) + ', ' +str(pair2[2])
    intersection_list.append(intersection)

col_df['intersection'] = intersection_list

st2 = all_realintersections_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post =  all_realintersections_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 =  all_realintersections_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1] + ', ' +str(pair2[2]))
    intersection_list.append(intersection)

all_realintersections_df['intersection'] = intersection_list
```

In [35]:
```
central_realintersections_df = pd.DataFrame.from_csv('central_real_intersectio
central_realintersections_df= all_realintersections_df.rename(columns = {'arcg
central_col_df = pd.DataFrame.from_csv('toronto_cycling_central_mode.csv', ind
from numpy import random
from scipy.spatial import distance
import matplotlib.path as mplPath
import numpy as np

#We want to follow a standard intersection convention to make our life easier.
#Unfortunately the free geocoder with unlimited request numbers per day doesn'
#So we will get the intersections, strip out the two streets, and order them a
st2 = central_col_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post = central_col_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 = central_col_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str) and isinstance(pair[1],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1]) + ', ' +str(pair2[2])
    intersection_list.append(intersection)

central_col_df['intersection'] = intersection_list

st2 = central_realintersections_df['intersection'].str.split('&').str.get(1)
st2 =st2.str.split(', Toronto').str[0].str.strip()

post =  central_realintersections_df['intersection'].str.split('&').str.get(1)
post = post.str.partition(', ')[2].str.strip()

st1 =  central_realintersections_df['intersection'].str.split('&').str[0]
st1 = st1.str.strip()

intersection_list = []
streets = pd.concat([st1,st2,post],axis=1)
streets = streets.values.tolist()
streets
for pair in streets:
    if isinstance(pair[0],str):
        if pair[1] <= pair[0]:
            temp = pair[0]
            pair[0] = pair[1]
            pair[1] = temp
for pair2 in streets:
    intersection = str(pair2[0]) + ' & ' + str(pair2[1] + ', ' +str(pair2[2]))
    intersection_list.append(intersection)

central_realintersections_df['intersection'] = intersection_list
central_realintersections_df.head()
```

Out[35]:

In [36]:
```
ped_counts_df = pd.DataFrame.from_csv('Vehicle and Pedestrian Counts/TrafficPe

#Using the Power fit for the Bike/Pedestrian ratio, we get a function that pre

ped_counts_df['bike_prediction'] = (500.2146799711*ped_counts_df['8HrPedVol']*
ped_coords = ped_counts_df[['Latitude','Longitude']]
ped_coords = ped_coords.replace(np.nan,0)
ped_coordinate_list = ped_coords.values.tolist()
ped_counts_df['coordinates'] = ped_counts_df[['Latitude','Longitude']].apply(t
ped_counts_df.head()
```

Out[36]:

	Main	Midblock Route	Side 1 Route	Side 2 Route	Activation Date	Latitude	Longitude	Count Date	8HrPedVol	8HrVeh
PX2										
2	JARVIS ST	NaN	FRONT ST E	NaN	1948/11/15	43.649	-79.371	2011/09/08	17008	19335
3	KING ST E	NaN	JARVIS ST	NaN	1950/08/23	43.65	-79.372	2011/09/07	37719	17665
4	JARVIS ST	NaN	ADELAIDE ST E	NaN	1958/09/12	43.652	-79.372	2008/06/16	1991	19726
5	JARVIS ST	NaN	RICHMOND ST E	NaN	1962/04/21	43.653	-79.373	2009/07/30	2696	24842
6	JARVIS ST	NaN	QUEEN ST E	NaN	1928/08/24	43.654	-79.373	2011/05/18	3622	19772

In [37]:
```
bike_dict = ped_counts_df.set_index('coordinates').to_dict()['bike_prediction'
ped_dict = ped_counts_df.set_index('coordinates').to_dict()['8HrPedVol']
veh_dict = ped_counts_df.set_index('coordinates').to_dict()['8HrVehVol']
col_df['coordinates'] = col_df[['LAT','LONG']].apply(tuple, axis=1)
col_df.head()
all_realintersections_df['coordinates'] = all_realintersections_df[['latitude'
equiv = {'Laneway':3,'Minor':2,'Major':1}
all_realintersections_df['num_class'] = all_realintersections_df['elevatio10']
all_realintersections_df= all_realintersections_df.sort_values(by=['intersecti
#take the lowest numerical class, drop the other duplicates.
all_realintersections_df = all_realintersections_df.drop_duplicates(subset='in

central_col_df['coordinates'] = central_col_df[['lat','long']].apply(tuple, ax
central_col_df.head()
central_realintersections_df['coordinates'] = central_realintersections_df[['l
equiv = {'Laneway':3,'Minor':2,'Major':1}
central_realintersections_df['num_class'] = central_realintersections_df['elev
central_realintersections_df= central_realintersections_df.sort_values(by=['in
#take the lowest numerical class, drop the other duplicates.
central_realintersections_df = central_realintersections_df.drop_duplicates(su
```

In [38]:
```
import csv
closest_traffic_point = []
bike_traffic = []
veh_traffic = []
ped_traffic = []
i = 0
for i in range(0,len(col_df)):
    point = col_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(bike_dict[tuple(closest)])
    closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_list)
    veh_traffic.append(veh_dict[tuple(closest)])
    ped_traffic.append(ped_dict[tuple(closest)])

myfile3 = open('closest_intersection.csv', 'w')
wr = csv.writer(myfile3)
wr.writerow(closest_traffic_point)
myfile3.close()

myfile4 = open('closest_int_bike_predictions.csv', 'w')
wr = csv.writer(myfile4)
wr.writerow(bike_traffic)
myfile4.close()

col_df['closest_traffic'] = tuple(closest_traffic_point)
col_df['traffic_count'] = bike_traffic
col_df['vehicle_traffic'] = veh_traffic
col_df['pedestrian_traffic'] = ped_traffic


col_df= col_df.rename(columns={'closest_traffic': 'closest_ped_count', 'traffi
```

In [39]:
```
import csv
closest_traffic_point = []
bike_traffic = []
veh_traffic = []
ped_traffic = []
i = 0
for i in range(0,len(central_col_df)):
    point = central_col_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(bike_dict[tuple(closest)])
    closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_list)
    veh_traffic.append(veh_dict[tuple(closest)])
    ped_traffic.append(ped_dict[tuple(closest)])
myfile3 = open('central_closest_intersection.csv', 'w')
wr = csv.writer(myfile3)
wr.writerow(closest_traffic_point)
myfile3.close()

myfile4 = open('central_closest_int_bike_predictions.csv', 'w')
wr = csv.writer(myfile4)
wr.writerow(bike_traffic)
myfile4.close()

central_col_df['closest_traffic'] = tuple(closest_traffic_point)
central_col_df['traffic_count'] = bike_traffic
central_col_df['vehicle_traffic'] = veh_traffic
central_col_df['pedestrian_traffic'] = ped_traffic


central_col_df= central_col_df.rename(columns={'closest_traffic': 'closest_ped
```

In [36]:
```
central_col_df.columns.values
```

Out[36]:
```
array(['injury', 'safety_equip', 'road_class', 'cyclist_crash_type',
       'age_of_cyclist', 'stname1', 'stet_1_type', 'stname_2',
       'stet_2_type', 'long', 'lat', 'acc_time', 'acc_time2',
       'traffic_control', 'road_surface2', 'date', 'date2',
       'driver_action', 'driver_condition', 'year', 'intersection',
       'coords', 'coordinates', 'closest_ped_count',
       'predicted_bike_count', 'vehicle_traffic', 'pedestrian_traffic'], dtyp
e=object)
```

In [37]:
```
import csv
closest_traffic_point = []
bike_traffic = []
veh_traffic = []
ped_traffic = []
i = 0
for i in range(0,len(all_realintersections_df)):
    point = all_realintersections_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(bike_dict[tuple(closest)])
    closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_list)
    veh_traffic.append(veh_dict[tuple(closest)])
    ped_traffic.append(ped_dict[tuple(closest)])
myfile5 = open('closest_intersection_centreline.csv', 'w')
wr = csv.writer(myfile5)
wr.writerow(closest_traffic_point)
myfile5.close()

myfile6 = open('closest_int_bike_predictions_centreline.csv', 'w')
wr = csv.writer(myfile6)
wr.writerow(bike_traffic)
myfile6.close()

all_realintersections_df['closest_traffic'] = tuple(closest_traffic_point)
all_realintersections_df['traffic_count'] = bike_traffic
all_realintersections_df['vehicle_traffic'] = veh_traffic
all_realintersections_df['pedestrian_traffic'] = ped_traffic

all_realintersections_df = all_realintersections_df.rename(columns={'closest_t
```

Adding traffic values for our list of intersections above. Cell below does the same for only those which have "central" Postal Code values.

In [38]:
```
import csv
closest_traffic_point = []
bike_traffic = []
veh_traffic = []
ped_traffic = []
i = 0
for i in range(0,len(central_realintersections_df)):
    point = central_realintersections_df['coordinates'].iloc[i]
    if waterfront_Path.contains_point(point):
        closest = waterfront[distance.cdist([point], waterfront).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(waterfront_dict[tuple(closest)])
    elif campus_Path.contains_point(point):
        closest = campus[distance.cdist([point], campus).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(campus_dict[tuple(closest)])
    elif castleFrank_Path.contains_point(point):
        closest = castleFrank[distance.cdist([point], castleFrank).argmin()]
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(castleFrank_dict[tuple(closest)])
    else:
        closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_l
        closest_traffic_point.append(tuple(closest))
        bike_traffic.append(bike_dict[tuple(closest)])
    closest = ped_coordinate_list[distance.cdist([point], ped_coordinate_list)
    veh_traffic.append(veh_dict[tuple(closest)])
    ped_traffic.append(ped_dict[tuple(closest)])
myfile5 = open('central_closest_intersection_centreline.csv', 'w')
wr = csv.writer(myfile5)
wr.writerow(closest_traffic_point)
myfile5.close()

myfile6 = open('central_closest_int_bike_predictions_centreline.csv', 'w')
wr = csv.writer(myfile6)
wr.writerow(bike_traffic)
myfile6.close()

central_realintersections_df['closest_traffic'] = tuple(closest_traffic_point)
central_realintersections_df['traffic_count'] = bike_traffic
central_realintersections_df['vehicle_traffic'] = veh_traffic
central_realintersections_df['pedestrian_traffic'] = ped_traffic

central_realintersections_df = central_realintersections_df.rename(columns={'c
```

In [39]:
```
all_realint_df = all_realintersections_df[['intersection','coordinates','close
#all_realint_df.sort_values(by='predicted_bike_count')

result = pd.merge(col_df,all_realint_df, how ='outer',on='intersection')
result = result.sort_values(by='intersection')
traff2 =result['predicted_bike_count_y']
traff2.fillna(1,inplace=True)
result['predicted_bike_count_x'] =result['predicted_bike_count_x'].fillna(valu
```

In [40]:
```
central_realint_df = central_realintersections_df[['intersection','coordinates
#all_realint_df.sort_values(by='predicted_bike_count')

result_central = pd.merge(central_col_df,central_realint_df, how ='outer',on='
result_central = result_central.sort_values(by='intersection')
traff_c =result_central['predicted_bike_count_y']
result_central['predicted_bike_count_x'] =result_central['predicted_bike_count
```

Now let's take a look at the total types of injuries after 1997

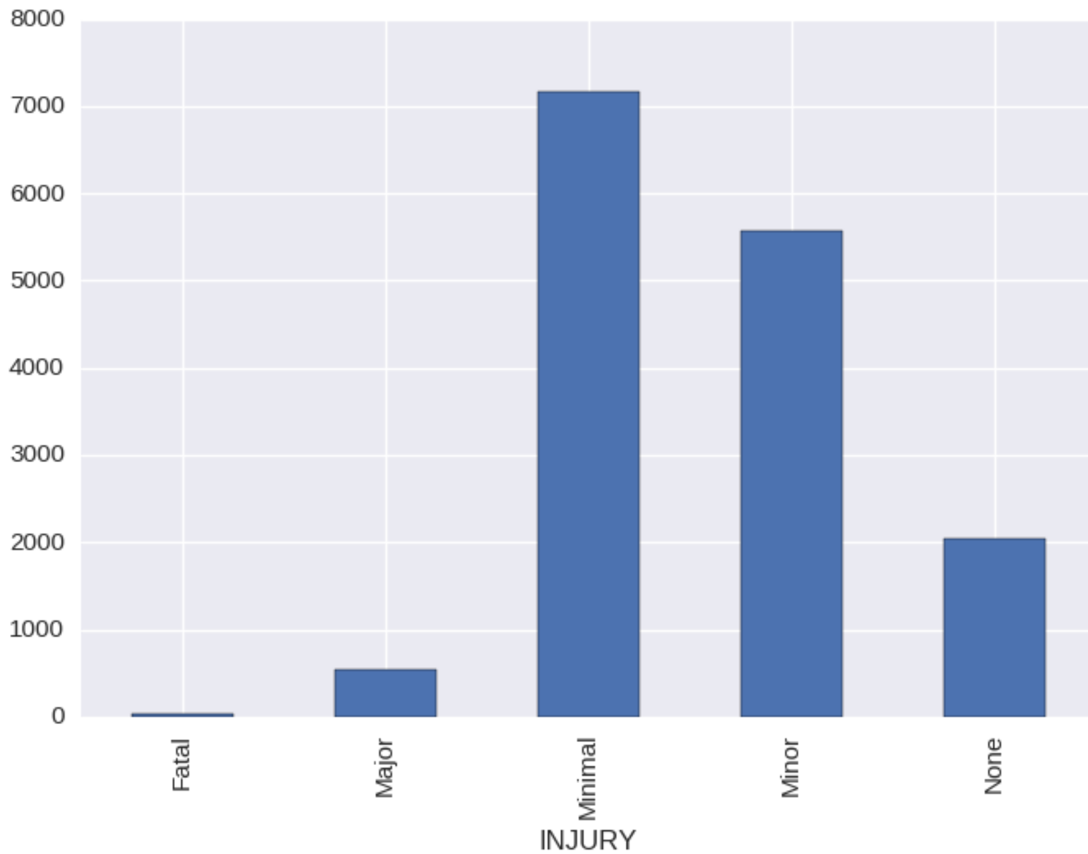
In [41]:
```
injury_df = full_df[full_df['YEAR'] >1997].groupby('INJURY').size()
total = injury_df.sum()
d = {'injury': pd.Series([total/31,total/543,total/5575,total/(7171+2039)], in
inverse_df = pd.DataFrame(d)
inverse_df = inverse_df/inverse_df.loc['Minimal']
inverse_df
```

Out[41]:

	injury
Fatal	297.1
Major	16.961
Minor	1.652
Minimal	1

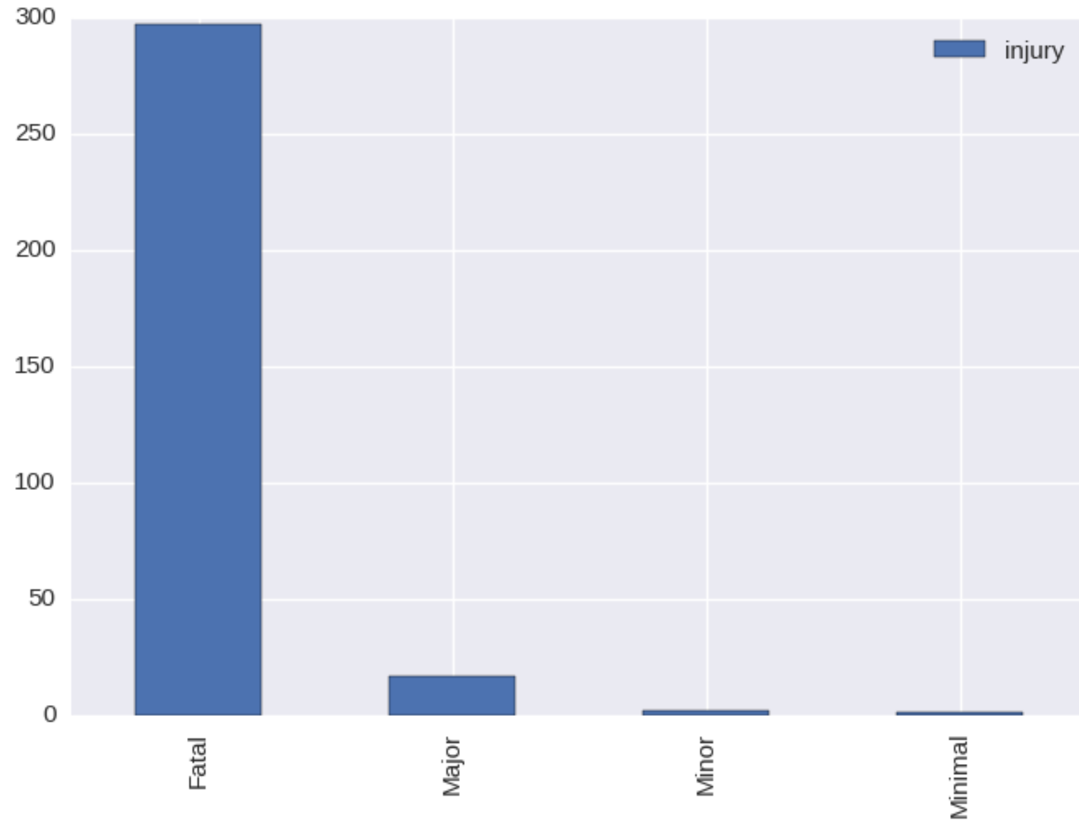
In [42]:
```
injury_df.plot(kind='bar')
```

Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff8979bbcf8>



In [43]:
```
inverse_df.plot(kind='bar')
```

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff8978e4908>



Filtering past 1997 got ridd of the "unrecorded" entries, so that's a helpful. Now combined 'minimal' and 'none' into one category. This is because we are going to estimate a "risk factor" for each intersection, where major accidents are worth more than a minor accident, which are worth more than a minor etc. In treating a 'none' it seems simplest to compine this with 'minimal' accidents, so we can merely just invert our data to get our weighting factors.

It can be difficult and subjective to make this sort of call. How is the loss of a human life (for fatal accidents), numerically compared to that in which a minor accident occurs? Obviously more serious accidents should be percieved as more dangerous. We are letting the data provide this estimate for us. We will merely compare the relative sizes of each.

This may not seem neccessary, as working off the raw collision numbers and normalizing those may feel more intuitive to some. However, I believe that when we run our algorithm against new data (Nov 2015 to Nov 2016), an algorithm which weighs these different accident types will make better predictions. Unfortunately it will not be an apples to apples comparison. To do this, one would like to know the severity of accidents in our test data as well. The twitter bot does not make this distinction clear. But it is merely retweeting collisions reported by the Toronto police twitter app, which is not the same as having access to the filed police reports of the Toronto Police database. One would then believe that our new data is made up of more serious collisions, as the police were immediately informed.

In [44]:
```
injury_dict = inverse_df.to_dict()
```

In [45]:
```
injury_dict['None']=1.0
injury_dict = injury_dict
col_df['injury_num'] = col_df['INJURY'].map(injury_dict)
col_df['injury_num'] = col_df['injury_num'].fillna(value=0)
central_col_df['injury_num'] = central_col_df['injury'].map(injury_dict)
central_col_df['injury_num'] = central_col_df['injury_num'].fillna(value=0)
```

Here we produce two resulting dataframes of intersections, featuring the total numbers of collisions, along with the features of that intersection.

In [46]:
```
col_df.columns.values
```

Out[46]:
```
array(['INJURY', 'SAFETY EQUIP.', 'ROAD CLASS', 'CYCLIST CRASH TYPE',
       'AGE OF CYCLIST', 'STNAME1', 'STET 1 TYPE', 'STNAME 2',
       'STET 2 TYPE', 'LONG', 'LAT', 'ACC TIME', 'ACC TIME2',
       'TRAFFIC CONTROL', 'ROAD SURFACE2', 'DATE', 'DATE2',
       'DRIVER ACTION', 'DRIVER CONDITION', 'YEAR', 'intersection',
       'coords', 'coordinates', 'closest_ped_count',
       'predicted_bike_count', 'vehicle_traffic', 'pedestrian_traffic',
       'injury_num'], dtype=object)
```

In [47]:
```
#result.to_csv('merged_data.csv')
new_col = col_df[col_df['YEAR']>1997]

result = pd.merge(new_col,all_realint_df, how ='outer',on='intersection')
result = result.sort_values(by='intersection')

severity_df = pd.get_dummies(new_col.INJURY, dummy_na=False)
new_col = pd.concat([new_col,severity_df],axis=1)
new_col['Minimal/None'] = new_col['Minimal'] + new_col['None']
intersection_df = new_col.groupby('intersection').agg({'predicted_bike_count':
```

In [48]:
```
#result.to_csv('merged_data.csv')
new_col = col_df[col_df['YEAR']>1997]

result = pd.merge(new_col,all_realint_df, how ='outer',on='intersection')
result = result.sort_values(by='intersection')

severity_df = pd.get_dummies(new_col.INJURY, dummy_na=False)
new_col = pd.concat([new_col,severity_df],axis=1)
new_col['Minimal/None'] = new_col['Minimal'] + new_col['None']
intersection_df = new_col.groupby('intersection').agg({'predicted_bike_count':
#intersection_df.columns = intersection_df.columns.set_levels(['minor','predic
zeroes_df = all_realint_df.groupby('intersection').agg({'predicted_bike_count'
zeroes_df['predicted_bike_count','size'] = 0
zeroes_df['injury_num','sum'] = 0
#Let's use 360 as the days in the year. It's a nice round number. And since we
zeroes_df['Rates','normalized_accident_rate'] = zeroes_df['predicted_bike_coun
zeroes_df = zeroes_df.sort_values(by=[('Rates','normalized_accident_rate')],as
intersection_df['Rates','normalized_accident_rate'] = intersection_df['predict
intersection_df = intersection_df.sort_values(by=[('Rates','normalized_acciden

zeroes_df['Rates','normalized_risk_factor'] = zeroes_df['injury_num','sum']/(1
zeroes_df = zeroes_df.sort_values(by=[('Rates','normalized_risk_factor')],asce
intersection_df['Rates','normalized_risk_factor'] = intersection_df['injury_nu
intersection_df = intersection_df.sort_values(by=[('Rates','normalized_risk_fa

totals_df = pd.concat([intersection_df,zeroes_df])
df_test = totals_df['Rates','normalized_accident_rate']
df_data2 = df_test.to_frame(name='normalized yearly accident rate')
df_data2['total collisions'] = totals_df['predicted_bike_count','size']
df_data2['bike traffic estimate'] = totals_df['predicted_bike_count','mean']
df_data2['scaled collisions'] = totals_df['injury_num','sum']
df_data2['normalized_risk_factor'] = totals_df['Rates','normalized_risk_factor
df_data2['vehicle traffic'] = totals_df['vehicle_traffic']
df_data2['pedestrian traffic'] = totals_df['pedestrian_traffic']
df_data2['fatal'] = totals_df['Fatal']
df_data2['major'] = totals_df.Major
df_data2['minor'] = totals_df.Minor
df_data2['minimal/none'] = totals_df['Minimal/None']
df_data2['normalized yearly accident rate'].fillna(0,inplace=True)
df_data2['normalized_risk_factor'].fillna(0,inplace=True)
df_data2.fillna(0,inplace=True)
df_data2.dropna(inplace=True)
#Here we make the index a column, drop the duplicates from this intersection c
df_data2 = df_data2.reset_index().drop_duplicates(subset='intersection', keep=
df_data2 = df_data2.sort_values(by='total collisions',ascending=False)
len(df_data2)
```

Out[48]: 20371

In [49]:
```
df_data2.head()
```

Out[49]:

	normalized yearly accident rate	total collisions	bike traffic estimate	scaled collisions	normalized_risk_factor	vehicle traffic	pedestrian traffic	fatal	majo
intersection									
Queen St W & Spadina Ave, Toronto, Ontario, M5V	7.0407e-06	43	1305	4	6.5495e-07	22206	9791.5	0	2
Bathurst St & Bloor St W, Toronto, Ontario, M5R	6.5778e-06	41	1331.9	7	1.123e-06	22976	11309	0	1
Bathurst St & Queen St W, Toronto, Ontario, M5T	6.7557e-06	40	1265.2	6	1.0134e-06	17330	7338.1	0	0
Bay St & Dundas St W, Toronto, Ontario, M5G	5.264e-06	37	1501.9	4	5.6909e-07	20558	37431	0	0
Queen St W & University Ave, Toronto, Ontario, M5H	5.8533e-06	37	1350.7	6	9.4918e-07	29023	14508	0	1

In [50]:
```
#result.to_csv('merged_data.csv')
new_col_central = central_col_df[central_col_df['year']>1997]

result_central = pd.merge(new_col_central,central_realint_df, how ='outer',on=
result_central = result_central.sort_values(by='intersection')

severity_df = pd.get_dummies(new_col_central.injury, dummy_na=False)
new_col_central = pd.concat([new_col_central,severity_df],axis=1)
new_col_central['Minimal/None'] = new_col_central['Minimal'] + new_col_central
intersection_df = new_col_central.groupby('intersection').agg({'predicted_bike
#intersection_df.columns = intersection_df.columns.set_levels(['minor','predic
zeroes_df = central_realint_df.groupby('intersection').agg({'predicted_bike_co
zeroes_df['predicted_bike_count','size'] = 0
zeroes_df['injury_num','sum'] = 0
#Let's use 360 as the days in the year. It's a nice round number. And since we
zeroes_df['Rates','normalized_accident_rate'] = zeroes_df['predicted_bike_coun
zeroes_df = zeroes_df.sort_values(by=[('Rates','normalized_accident_rate')],as
intersection_df['Rates','normalized_accident_rate'] = intersection_df['predict
intersection_df = intersection_df.sort_values(by=[('Rates','normalized_acciden

zeroes_df['Rates','normalized_risk_factor'] = zeroes_df['injury_num','sum']/(1
zeroes_df = zeroes_df.sort_values(by=[('Rates','normalized_risk_factor')],asce
intersection_df['Rates','normalized_risk_factor'] = intersection_df['injury_nu
intersection_df = intersection_df.sort_values(by=[('Rates','normalized_risk_fa

totals_df = pd.concat([intersection_df,zeroes_df])
df_test = totals_df['Rates','normalized_accident_rate']
df_data_central = df_test.to_frame(name='normalized yearly accident rate')
df_data_central['total collisions'] = totals_df['predicted_bike_count','size']
df_data_central['bike traffic estimate'] = totals_df['predicted_bike_count','m
df_data_central['scaled collisions'] = totals_df['injury_num','sum']
df_data_central['normalized_risk_factor'] = totals_df['Rates','normalized_risk
df_data_central['vehicle traffic'] = totals_df['vehicle_traffic']
df_data_central['pedestrian traffic'] = totals_df['pedestrian_traffic']
df_data_central['fatal'] = totals_df['Fatal']
df_data_central['major'] = totals_df.Major
df_data_central['minor'] = totals_df.Minor
df_data_central['minimal/none'] = totals_df['Minimal/None']
df_data_central['normalized yearly accident rate'].fillna(0,inplace=True)
df_data_central['normalized_risk_factor'].fillna(0,inplace=True)
df_data_central.fillna(0,inplace=True)
df_data_central.dropna(inplace=True)
#Here we make the index a column, drop the duplicates from this intersection c
df_data_central = df_data_central.reset_index().drop_duplicates(subset='inters
df_data_central = df_data_central.sort_values(by='total collisions',ascending=
len(df_data_central)
```

Out[50]: 18693

In [51]:
```
df_data2.to_csv('all_totals.csv')
df_data_central.to_csv('central_totals.csv')
```

Now we have our two main datasets. A collection of intersections in the central areas of the city with their associated accident numbers for cycling collisions from 1998 onwards, and one for the entire city of Toronto. To gain any predicted power however, we'll need to throw in some features of the intersections. We'll use the road class types from the collisions data set, using the "busiest" class of the two streets to define the intersection. We'll also use the intersection control type as well. The cells merely add these columns to our two dataframes.

In [52]:
```
temp = result
temp = temp.replace(np.nan,-1)
roadclass_df =temp.groupby(['intersection']+['ROAD CLASS']) \
            .size().to_frame('count') \
            .reset_index().sort_values('intersection',ascending=False)

#roadclass_df['road_class'] =roadclass_df['road_class'].fillna(value =elevatio
roadclass_df = roadclass_df.sort_values('count', ascending=False)
roadclass_df = roadclass_df.replace(np.nan,-1)
roadclass_dict = {'Collector':1, 'Expressway':1,'Expressway Ramp':1,'Major Art
roadclass_df['roadclass_num'] = roadclass_df['ROAD CLASS'].map(roadclass_dict)
roadclass_df = roadclass_df.sort_values(by = ['intersection','roadclass_num'],
roadclass_df = roadclass_df.drop_duplicates(subset='intersection', keep='first
roadclass_df = roadclass_df.sort_values(by='intersection', ascending= False)

temp = result
temp['elevatio10'].replace(np.nan,'unrecorded', inplace =True)
elevatio_df =temp.groupby(['intersection']+['elevatio10']) \
            .size().to_frame('count') \
            .reset_index().sort_values('intersection',ascending=False)

elevatio10_dict = {'Major':'Major Arterial', 'Minor':'Minor Arterial','Laneway
elevatio_df['intersection_type']= elevatio_df['elevatio10'].map(elevatio10_dic

elevatio_df = elevatio_df.sort_values(by = ['intersection','intersection_type'
elevatio_df = elevatio_df.drop_duplicates(subset='intersection', keep='first')
elevatio_df = elevatio_df.sort_values(by='intersection', ascending= False)
elevatio_df.reset_index(inplace = True)
elevatio10 = elevatio_df['intersection_type']

roadclass_df['ROAD CLASS'] =roadclass_df['ROAD CLASS'].replace(to_replace = -1
roadclass_df.reset_index(inplace=True)
roadclass_df['ROAD CLASS'] =roadclass_df['ROAD CLASS'].fillna(value = elevatio

roadclass_df['roadclass_num'] = roadclass_df['ROAD CLASS'].map(roadclass_dict)
roadclass_df = roadclass_df.sort_values(by = ['intersection','roadclass_num'],
roadclass_df = roadclass_df.drop_duplicates(subset='intersection', keep='first
roadclass_df = roadclass_df.sort_values(by='intersection', ascending= False)
```

In [53]:
```
roadclass_df.groupby(['ROAD CLASS']).size()
```

Out[53]:
```
ROAD CLASS
Collector               791
Expressway               23
Expressway Ramp          49
Hydro Line                1
Laneway                  10
Local                  1104
Major Arterial         2818
Major Arterial Ramp       3
Minor Arterial        15408
Minor Arterial Ramp       1
Other                     4
Private Property        158
River                     1
dtype: int64
```

In [54]:
```
result['TRAFFIC CONTROL'].replace(np.nan,'unrecorded', inplace=True)
result['ROAD CLASS'].replace(['Expressway Ramp', 'Hydro Line','Laneway','Major
temp = result
trafficcontrol_df =temp.groupby(['intersection']+['TRAFFIC CONTROL']) \
            .size().to_frame('count') \
            .reset_index().sort_values('count',ascending=False)

trafficcontrol_dict = {'traffic signal':1, 'traffic gate':1,'traffic controlle
trafficcontrol_df['trafficcontrol_num'] = trafficcontrol_df['TRAFFIC CONTROL']
trafficcontrol_df = trafficcontrol_df.sort_values(by = ['intersection','traffi
trafficcontrol_df = trafficcontrol_df.drop_duplicates(subset='intersection', k
trafficcontrol_df = trafficcontrol_df.sort_values(by='intersection', ascending
```

In [55]:
```
trafficcontrol_df.to_csv('trafficcontrol_df.csv')
```

In [56]:
```
reverse_dict = {1:'traffic signal/gate',2:'stop/yield sign', 3:'pedestrian cro
trafficcontrol_df['control category'] = trafficcontrol_df['trafficcontrol_num'

trafficcontrol_df.head()
```

Out[56]:

	intersection	TRAFFIC CONTROL	count	trafficcontrol_num	control category
22146	bpNichol Ln & bpNichol Ln, Toronto, Ontario, M5S	no control	1	5	no control
22145	Zachary Ct & Zachary Ct, Toronto, Ontario, M6A	unrecorded	1	6	other/unrecorded
22144	York Ridge Rd & York Ridge Rd, Toronto, Ontari...	unrecorded	1	6	other/unrecorded
22143	York Mills Rd & York Ridge Rd, Toronto, Ontari...	unrecorded	1	6	other/unrecorded
22142	York Mills Rd & York Mills Rd, Toronto, Ontari...	unrecorded	1	6	other/unrecorded

In [57]:
```
trafficcontrol_df.groupby(['control category']).size()
```

Out[57]:
```
control category
no control              2172
other/unrecorded       14693
pedestrian crossover      93
stop/yield sign         1702
traffic signal/gate     1711
dtype: int64
```

In [58]:
```
reverseROAD_dict = {1:'Collector/Expressway',2:'Major Arterial', 3:'Minor Arte
roadclass_df['road_class_cat'] = roadclass_df['roadclass_num'].map(reverseROAD

roadclass_df.head()
```

Out[58]:

	index	intersection	ROAD CLASS	count	roadclass_num	road_class_cat
0	21209	bpNichol Ln & bpNichol Ln, Toronto, Ontario, M5S	Major Arterial	1	2	Major Arterial
1	21208	Zachary Ct & Zachary Ct, Toronto, Ontario, M6A	Local	1	4	Local/Laneway
2	21207	York Ridge Rd & York Ridge Rd, Toronto, Ontari...	Minor Arterial	1	3	Minor Arterial
3	21206	York Mills Rd & York Ridge Rd, Toronto, Ontari...	Minor Arterial	1	3	Minor Arterial
4	21205	York Mills Rd & York Mills Rd, Toronto, Ontari...	Minor Arterial	1	3	Minor Arterial

In [59]:
```
roadclass_df.groupby(['road_class_cat']).size()
```

Out[59]:
```
road_class_cat
Collector/Expressway      863
Local/Laneway            1272
Major Arterial           2818
Minor Arterial          15408
dtype: int64
```

In [60]:
```
df_data2 = df_data2.sort_index(ascending = False)
df_data2['road_class'] = roadclass_df['road_class_cat'].values
df_data2['control type'] = trafficcontrol_df['control category'].values
df_data2[['fatal','major','minor','minimal/none']] = df_data2[['fatal','major'

traffdummy_df = pd.get_dummies(df_data2['control type'], dummy_na=False)
df_data2 = pd.concat([df_data2, traffdummy_df ],axis=1)

rdclass_dummy = pd.get_dummies(df_data2['road_class'])
df_data2 = pd.concat([df_data2, rdclass_dummy],axis=1)
```

We had to do a bit of work to map certain obvious groups together (such as minor arterial ramp grouping with arterial ramp). The same process was performed for traffic control types. Part of the thinking in the grouping was trying to use common sense, traffic gates would likely be the most restrictive intersections, but due to their very small number, they are combined with the next restrictive in a "traffic signal/gate" groups.

We note that the greatest number of of control categories are other or unrecorded. This is an interesting characterisitc of our data, as any bias in the REASON for an intersection's traffic control being identified as unrecorded can skew the predictive power of our algorithms.

In [61]:
```
df_data2['bike traffic estimate'].replace(0,1,inplace=True)
df_data2['pedestrian traffic'].replace(0,1,inplace=True)
```

In order to deal with duplicate entries, we'll assign a numerical value to the roat types, so as to order them by how large the expected traffic flow is. We can then keep the most repeated, highest traffic flow option when we ask Pandas to drop duplicate entries in the "intersection" index.

We're also mapping the intersection types found in the intersection list dataframe to correspond to those in the cycling collisions one.

In [62]:
```
temp = result_central
temp = temp.replace(np.nan,-1)
roadclass_df =temp.groupby(['intersection']+['road_class']) \
            .size().to_frame('count') \
            .reset_index().sort_values('intersection',ascending=False)

#roadclass_df['road_class'] =roadclass_df['road_class'].fillna(value =elevatio
roadclass_df = roadclass_df.sort_values('count', ascending=False)
roadclass_df = roadclass_df.replace(np.nan,-1)
roadclass_dict = {'Collector':1, 'Expressway':1,'Expressway Ramp':1,'Major Art
roadclass_df['roadclass_num'] = roadclass_df['road_class'].map(roadclass_dict)
roadclass_df = roadclass_df.sort_values(by = ['intersection','roadclass_num'],
roadclass_df = roadclass_df.drop_duplicates(subset='intersection', keep='first
roadclass_df = roadclass_df.sort_values(by='intersection', ascending= False)

temp = result_central
temp['elevatio10'].replace(np.nan,'unrecorded', inplace =True)
elevatio_df =temp.groupby(['intersection']+['elevatio10']) \
            .size().to_frame('count') \
            .reset_index().sort_values('intersection',ascending=False)

elevatio10_dict = {'Major':'Major Arterial', 'Minor':'Minor Arterial','Laneway
elevatio_df['intersection_type']= elevatio_df['elevatio10'].map(elevatio10_dic

elevatio_df = elevatio_df.sort_values(by = ['intersection','intersection_type'
elevatio_df = elevatio_df.drop_duplicates(subset='intersection', keep='first')
elevatio_df = elevatio_df.sort_values(by='intersection', ascending= False)
elevatio_df.reset_index(inplace = True)
elevatio10 = elevatio_df['intersection_type']

roadclass_df['ROAD CLASS'] =roadclass_df['road_class'].replace(to_replace = -1
roadclass_df.reset_index(inplace=True)
roadclass_df['ROAD CLASS'] =roadclass_df['ROAD CLASS'].fillna(value = elevatio

roadclass_df['roadclass_num'] = roadclass_df['ROAD CLASS'].map(roadclass_dict)
roadclass_df = roadclass_df.sort_values(by = ['intersection','roadclass_num'],
roadclass_df = roadclass_df.drop_duplicates(subset='intersection', keep='first
roadclass_df = roadclass_df.sort_values(by='intersection', ascending= False)

roadclass_df.groupby(['ROAD CLASS']).size()
```

Out[62]:
```
ROAD CLASS
Collector            169
Expressway             6
Expressway Ramp        5
Laneway                3
Local                469
Major Arterial       761
Minor Arterial     17275
Private Property       5
dtype: int64
```

In [63]:
```
result_central['traffic_control'].replace(np.nan,'unrecorded', inplace=True)
result_central['road_class'].replace(['Expressway Ramp', 'Hydro Line','Laneway
temp = result_central
trafficcontrol_df =temp.groupby(['intersection']+['traffic_control']) \
            .size().to_frame('count') \
            .reset_index().sort_values('count',ascending=False)

trafficcontrol_dict = {'traffic signal':1, 'traffic gate':1,'traffic controlle
trafficcontrol_df['trafficcontrol_num'] = trafficcontrol_df['traffic_control']
trafficcontrol_df = trafficcontrol_df.sort_values(by = ['intersection','traffi
trafficcontrol_df = trafficcontrol_df.drop_duplicates(subset='intersection', k
trafficcontrol_df = trafficcontrol_df.sort_values(by='intersection', ascending
```

In [64]:
```
reverse_dict = {1:'traffic signal/gate',2:'stop/yield sign', 3:'pedestrian cro
trafficcontrol_df['control category'] = trafficcontrol_df['trafficcontrol_num'

reverseROAD_dict = {1:'Collector/Expressway',2:'Major Arterial', 3:'Minor Arte
roadclass_df['road_class_cat'] = roadclass_df['roadclass_num'].map(reverseROAD

df_data_central = df_data_central.sort_index(ascending = False)
df_data_central['road_class'] = roadclass_df['road_class_cat'].values
df_data_central['control type'] = trafficcontrol_df['control category'].values
df_data_central[['fatal','major','minor','minimal/none']] = df_data_central[['

traffdummy_df = pd.get_dummies(df_data_central['control type'], dummy_na=False
df_data_central = pd.concat([df_data_central, traffdummy_df ],axis=1)

rdclass_dummy = pd.get_dummies(df_data_central['road_class'])
df_data_central = pd.concat([df_data_central, rdclass_dummy],axis=1)
```

We've now added our features, as well as corresponding dummy variables for each class. These are neccessary to perform our linear regressions in a little bit.

We're going to fit posterior means via empirical bayesian estimation as before. These won't be used for prediction, but it will be interesting to see if any of our predictive approaches perform better than the estimates from the posterior rates when we take a look at data from this past year scrapped from twitter.

In [65]:
```
ag,bg,cg = stats.gamma.fit(df_data2['total collisions'])
beta = 1/cg
df_data2['posterior mean rate'] = (ag + df_data2['total collisions'])/(beta+13
df_data2['posterior STD'] = np.sqrt((ag + df_data2['total collisions'])/((beta
#pd.set_option('display.float_format', '{:.5g}'.format)
df_data2.sort_values(by='posterior mean rate',ascending = False).head()
```

Out[65]:

	normalized yearly accident rate	total collisions	bike traffic estimate	scaled collisions	normalized_risk_factor	vehicle traffic	pedestrian traffic	fatal	ma
intersection									
MacDonald-Cartier Fwy & Victoria Park Ave, Toronto, Ontario, M1R	0	9	1	1	0	35267	1	0	0
Reesor Rd & Steeles Ave E, Scarborough, Ontario, M1X, Scarborough, Ontario, M1X	0	3	1	0	0	1081	1	0	0
HWY-401, Mississauga, Ontario, L4W & Renforth Dr, Mississauga, Ontario, L4W	0	2	1	0	0	24049	1	0	2
Highway of Heroes & Victoria Park Ave, Toronto, Ontario, M1R	0	2	1	0	0	35267	1	0	0
Blackhurst Ct & Morningview Trl, Scarborough, Ontario, M1B, Scarborough, Ontario, M1B	0	1	1	0	0	1081	1	0	1

5 rows × 24 columns

In [66]:
```
ag,bg,cg = stats.gamma.fit(df_data_central['total collisions'])
beta = 1/cg
df_data_central['posterior mean rate'] = (ag + df_data_central['total collisio
df_data_central['posterior STD'] = np.sqrt((ag + df_data_central['total collis
#pd.set_option('display.float_format', '{:.5g}'.format)
df_data_central.sort_values(by='posterior mean rate',ascending = False).head()
```

Out[66]:

	normalized yearly accident rate	total collisions	bike traffic estimate	scaled collisions	normalized_risk_factor	vehicle traffic	pedestrian traffic	fatal	majo
intersection									
Farm Greenway & Victoria Park Ave, Toronto, Ontario, M1R	0	0	0	0	0	35267	0	0	0
Bingley Rd & Valley Centre Dr, Toronto, Ontario, M1X	0	0	0	0	0	1081	0	0	0
Lake Shore Blvd W & Lakeshore British Columbia Rd, Toronto, Ontario, M6K	0	0	0	0	0	27298	0	0	0
Pharmacy Ave & Seabury Gate, Toronto, Ontario, M1T	0	0	0	0	0	35267	0	0	0
Foxcote Cres & Rangoon Rd, Toronto, Ontario, M9C	0	0	0	0	0	16435	0	0	0

5 rows × 24 columns

In [67]:
```
df_data2.to_csv('preinput_data2.csv')
df_data_central.to_csv('preinput_data_central.csv')
```

In [93]:
```
df_data = pd.DataFrame.from_csv('preinput_data2.csv')
df_data_central = pd.DataFrame.from_csv('preinput_data_central.csv')
```

In [94]:
```
#df_data['injury'] = df_data.fatal + df_data.major + df_data.minor + df_data['
df_data['fatal/major'] = df_data.fatal + df_data.major

#df_data_central['injury'] = df_data_central.fatal + df_data_central.major + d
df_data_central['fatal/major'] = df_data_central.fatal + df_data_central.major
```

In [95]:
```
df_data.columns.values
```

Out[95]:
```
array(['normalized yearly accident rate', 'total collisions',
       'bike traffic estimate', 'scaled collisions',
       'normalized_risk_factor', 'vehicle traffic', 'pedestrian traffic',
       'fatal', 'major', 'minor', 'minimal/none', 'road_class',
       'control type', 'no control', 'other/unrecorded',
       'pedestrian crossover', 'stop/yield sign', 'traffic signal/gate',
       'Collector/Expressway', 'Local/Laneway', 'Major Arterial',
       'Minor Arterial', 'posterior mean rate', 'posterior STD',
       'fatal/major'], dtype=object)
```

Training and testing linear models.

First we break set up our features and target data for both the full, and central-only data sets.

The barious linear regression algorithms available in sci-kit learn expect features which are gaussian. Otherwise they may perform very badly. We use the preproceessing library to scale the various traffic counts.

In [96]:
```
X= df_data.drop(['normalized yearly accident rate','posterior mean rate','post
Y_priorTotals = df_data['total collisions']
Y_priorRate = df_data['total collisions']/13
Y_postRate = df_data['posterior mean rate']
Y_priorRiskRate = df_data['normalized_risk_factor']

Xc= df_data_central.drop(['normalized yearly accident rate','posterior mean ra
Yc_priorTotals = df_data_central['total collisions']
Yc_priorRate = df_data_central['total collisions']/13
Yc_postRate = df_data_central['posterior mean rate']
Yc_priorRiskRate = df_data_central['normalized_risk_factor']
```

In [97]:
```
from sklearn.linear_model import LinearRegression
from sklearn import preprocessing
raw_bike = X['bike traffic estimate'].reshape(-1,1)
raw_veh = X['vehicle traffic'].reshape(-1,1)
raw_ped = X['pedestrian traffic'].reshape(-1,1)
bike_scaler = preprocessing.StandardScaler().fit(raw_bike)
veh_scaler = preprocessing.StandardScaler().fit(raw_veh)
ped_scaler = preprocessing.StandardScaler().fit(raw_ped)
X_scaled_bike = bike_scaler.transform(raw_bike)
X_scaled_veh = veh_scaler.transform(raw_veh)
X_scaled_ped = ped_scaler.transform(raw_ped)
X['bike traffic estimate'] = X_scaled_bike
X['vehicle traffic'] = X_scaled_veh
X['ped scaler'] = X_scaled_ped
```

In [98]:
```
raw_bike = Xc['bike traffic estimate'].reshape(-1,1)
raw_veh = Xc['vehicle traffic'].reshape(-1,1)
raw_ped = Xc['pedestrian traffic'].reshape(-1,1)
bike_scaler_c = preprocessing.StandardScaler().fit(raw_bike)
veh_scaler_c = preprocessing.StandardScaler().fit(raw_veh)
ped_scaler_c = preprocessing.StandardScaler().fit(raw_ped)
Xc_scaled_bike = bike_scaler_c.transform(raw_bike)
Xc_scaled_veh = veh_scaler_c.transform(raw_veh)
Xc_scaled_ped = ped_scaler_c.transform(raw_ped)
Xc['bike traffic estimate'] = Xc_scaled_bike
Xc['vehicle traffic'] = Xc_scaled_veh
Xc['ped scaler'] = Xc scaled ped
```

First we train a basic linear regression. Rather than splitting the data manually, or coding up a cross validation method ourselves (as was required in the linear regression exercise for the course), we'll use the included cross_val_score and _predict methods.

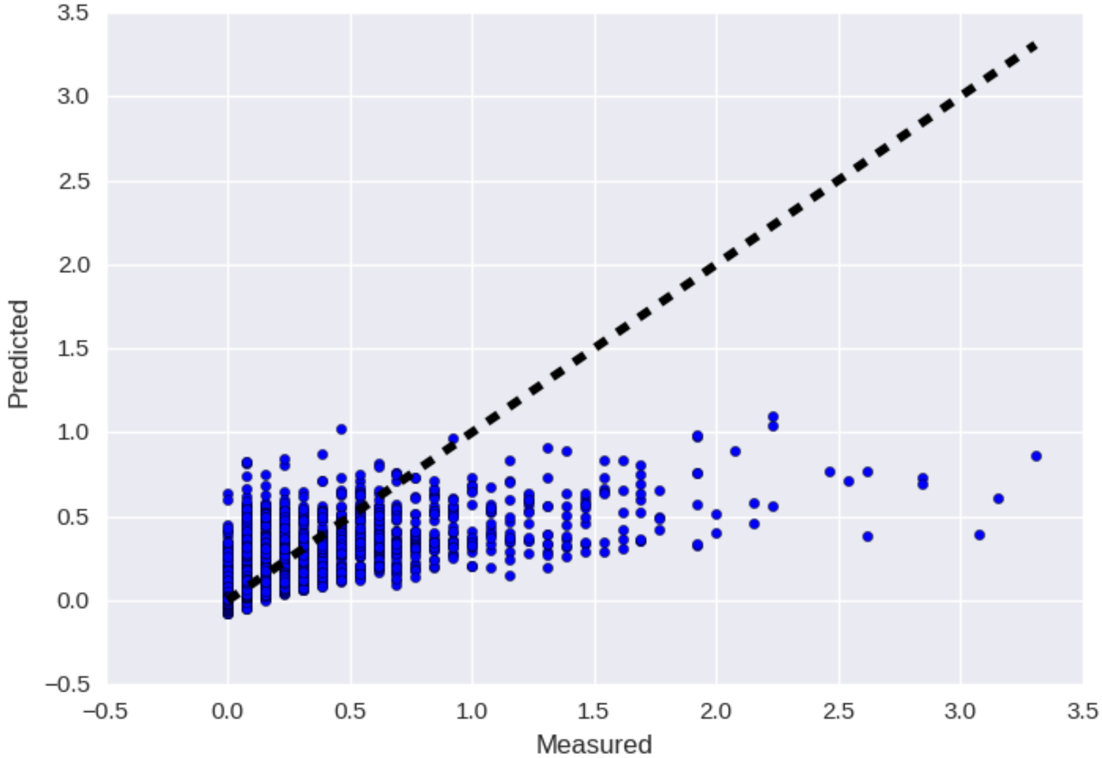
In [142]:
```
from sklearn import linear_model as lm
cycling_linear = lm.LinearRegression()
cycling_lasso = lm.Lasso()
cycling ridge = lm.Ridge(alpha =10)
```

In [143]:
```
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
predicted = cross_val_predict(cycling_linear,X,Y_priorRate,cv=5)
linear_score = cross_val_score(cycling_linear,X,Y_priorRate,cv=5)
print("linear score: ", np.mean(linear_score))
print("predicted: ", predicted)
```

```
linear score:  0.508928284915
predicted:  [ 0.17652133 -0.01308693 -0.02239764 ...,  0.12101306  0.01118716
  0.01118716]
```

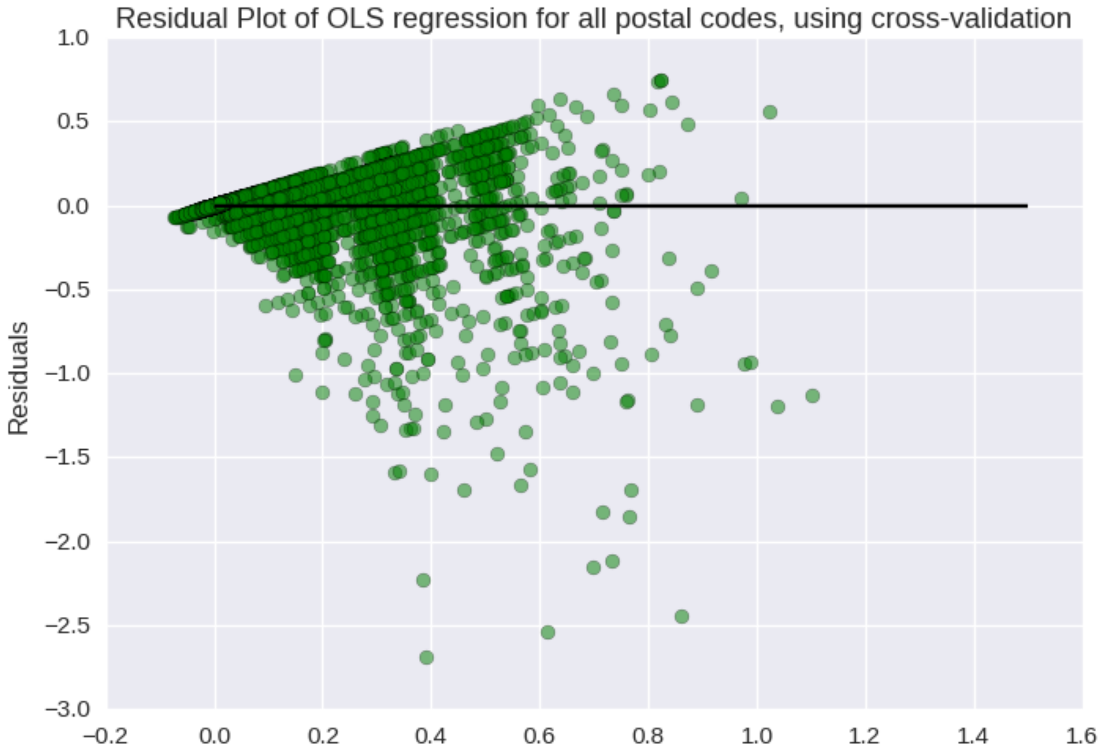
We first notice that the score isn't terribly good. There is a lot of room for improvement here. How do these predictions look visually when plotted against the actual data? Let's take a look

In [144]:
```
fig, ax = plt.subplots()
Y = Y_priorRate
r2 = linear_score
ax.scatter(Y,predicted)
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--',label ="r^2="+str(r2), l
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```



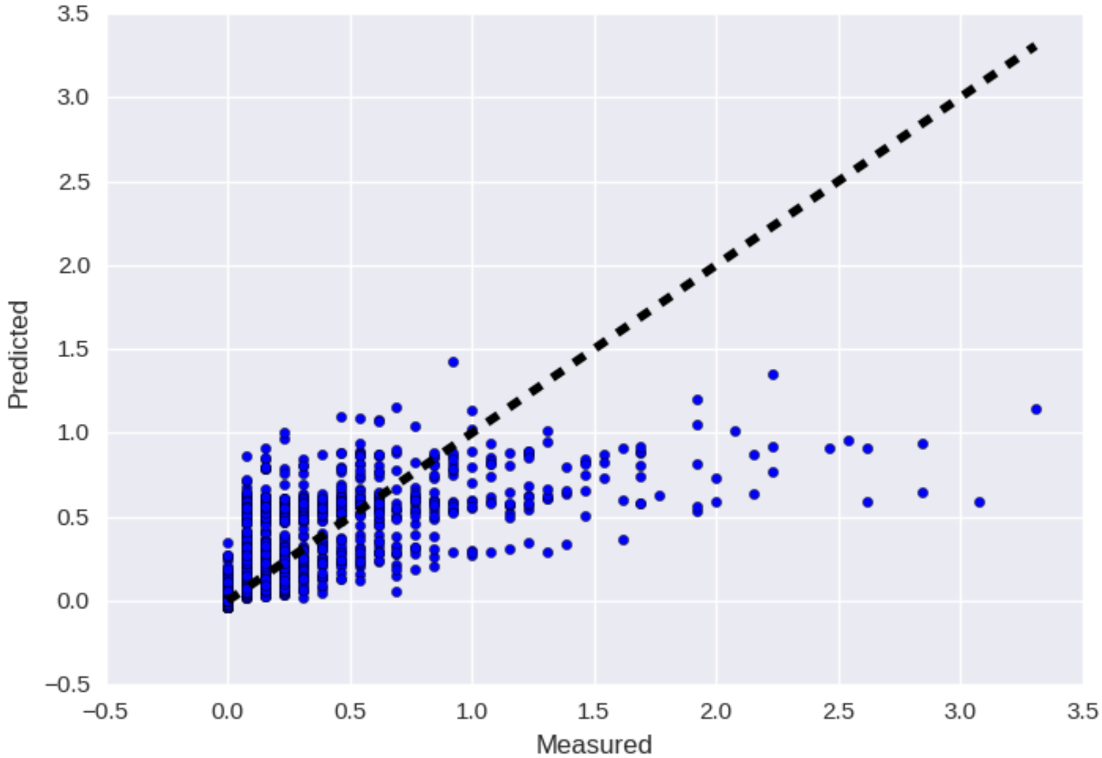
In [145]:
```
plt.scatter(predicted, predicted- Y, c='g', s=40, alpha=0.5)
plt.hlines(y = 0, xmin=0, xmax = 1.5)
plt.title('Residual Plot of OLS regression for all postal codes, using cross-v
plt.ylabel('Residuals')
```

Out[145]: <matplotlib.text.Text at 0x7f8002032d68>



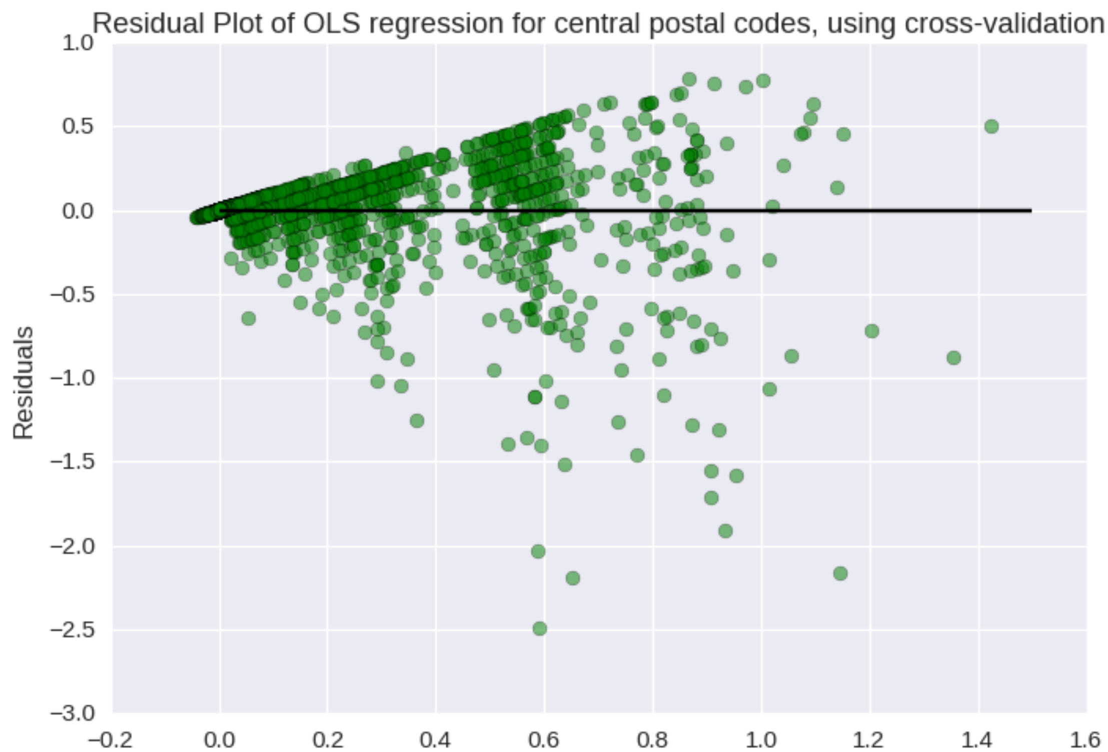
In [146]:
```
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score

predicted = cross_val_predict(cycling_linear,Xc,Yc_priorRate,cv=5)
linear_score = cross_val_score(cycling_linear,Xc,Yc_priorRate,cv=5)
print("linear score: ", np.mean(linear_score))
print("predicted: ", predicted)

fig, ax = plt.subplots()
Y = Yc_priorRate
r2 = linear_score
ax.scatter(Y,predicted)
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--',label ="r^2="+str(r2), l
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

```
linear score:  0.60266120383
predicted:  [ 0.1582576  -0.02152201 -0.00971354 ...,  0.002264    0.002264
  0.002264  ]
```



In [147]:
```
plt.scatter(predicted, predicted- Y, c='g', s=40, alpha=0.5)
plt.hlines(y = 0, xmin=0, xmax = 1.5)
plt.title('Residual Plot of OLS regression for central postal codes, using cro
plt.ylabel('Residuals')
```

Out[147]:
```
<matplotlib.text.Text at 0x7f8001fffcf8>
```



So it's very obvious to us that there is a very large tendency to understimate collision numbers, particularily for mid, to high risk intersections. We see that restricting ourselves to intersections which are in central Toronto gives a pretty sizable imporvement in R^2. From 0.5 to about 0.6. The residuals show an improvement in the underestimation as well.

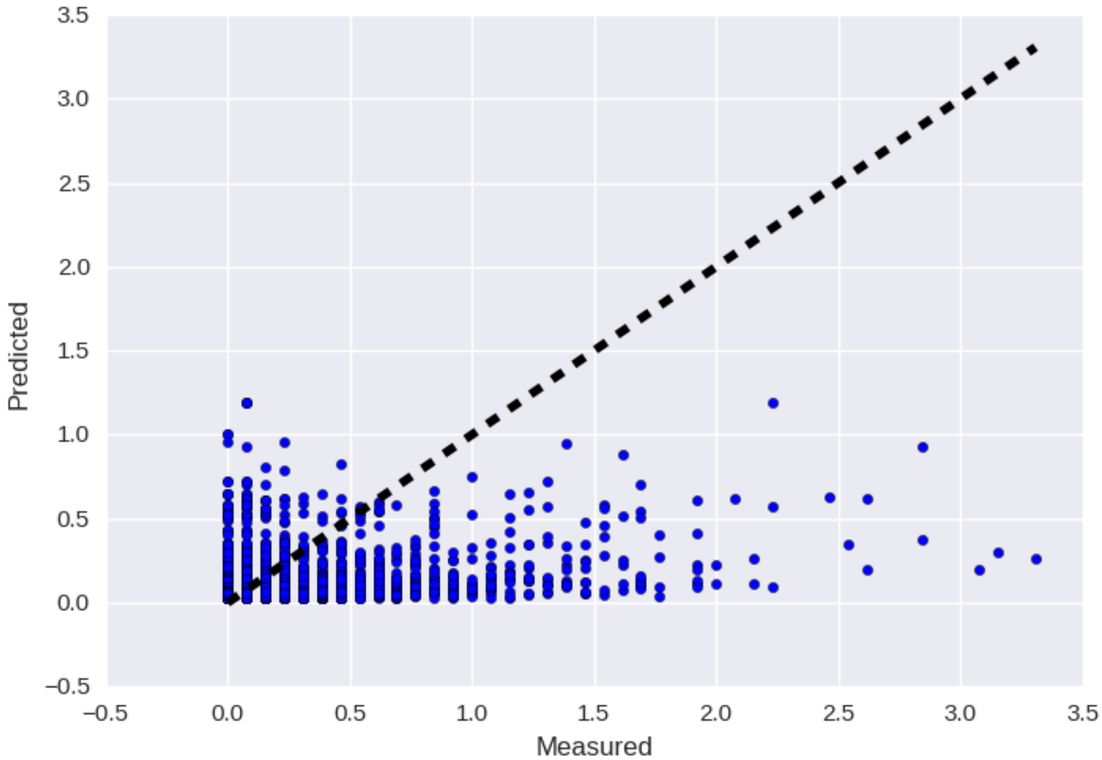
Let's try a more sophisticated model. Lasso regression yields a sparse model. It is useful for reducing the dimensionality of a problem. In this case, we probably are going to get pretty bad results, since we actually have very few features to work with already.

We'll let the use the LassoCV method to fit an alpha, and then make our prediction from there.

In [105]:
```
lassomodel = lm.LassoCV(cv=5).fit(X,Y_priorRate)
predicted = lassomodel.predict(X)
lasso_score = lassomodel.score(X,Y_priorRate)

print("chosen alpha: ",lassomodel.alpha_)
print("lasso score: ", np.mean(lasso_score))
print("predicted: ", predicted)
```

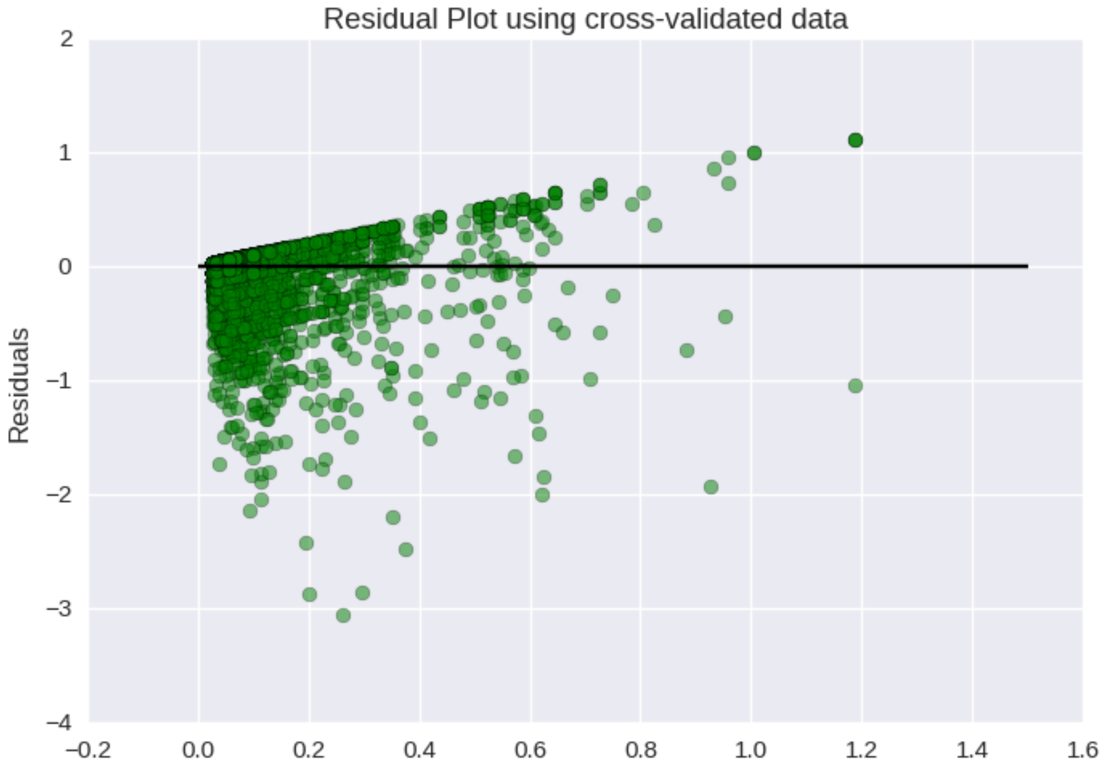
```
chosen alpha:  0.159143550513
lasso score:  0.133455369252
predicted:  [ 0.13981183  0.07347667  0.02680618 ...,  0.05670033  0.05670033
  0.05670033]
```

In [106]:
```
fig, ax = plt.subplots()
Y = Y_priorRate
r2= linear_score
ax.scatter(Y,predicted)
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--',label ="r^2="+str(r2), l
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```



In [107]:
```
plt.scatter(predicted, predicted- Y, c='g', s=40, alpha=0.5)
plt.hlines(y = 0, xmin=0, xmax = 1.5)
plt.title('Residual Plot using cross-validated data')
plt.ylabel('Residuals')
```

Out[107]: <matplotlib.text.Text at 0x7f8002cd5cf8>

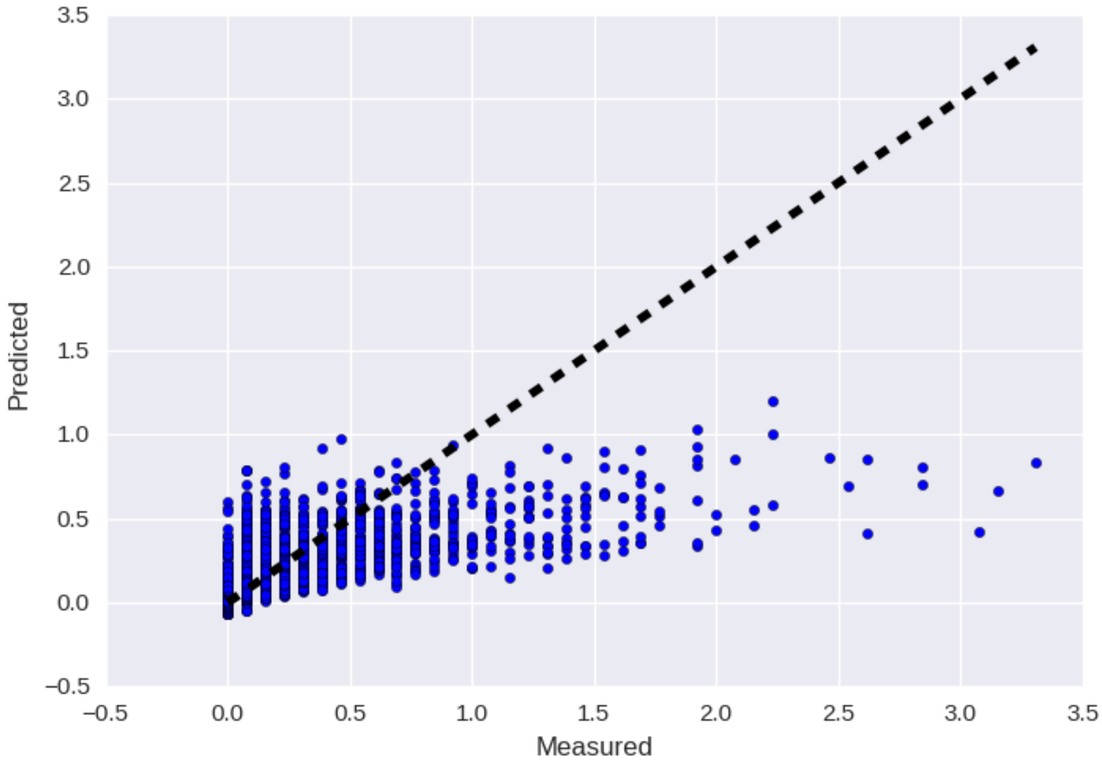


So a rather poor performance overall. Worse than regular least squares regression, just as we predicted. It doesn't make sense to continue with this model by checking the performance with the central-only dataframe, as this isn't the kidn of problem it's designed for. Now what about ridge regression?

In [108]:
```
ridgemodel = lm.RidgeCV(cv=5).fit(X,Y_priorRate)
predicted = ridgemodel.predict(X)
ridge_score = ridgemodel.score(X,Y_priorRate)

print("chosen alpha: ",ridgemodel.alpha_)
print("ridge score: ", np.mean(ridge_score))
print("predicted: ", predicted)
```

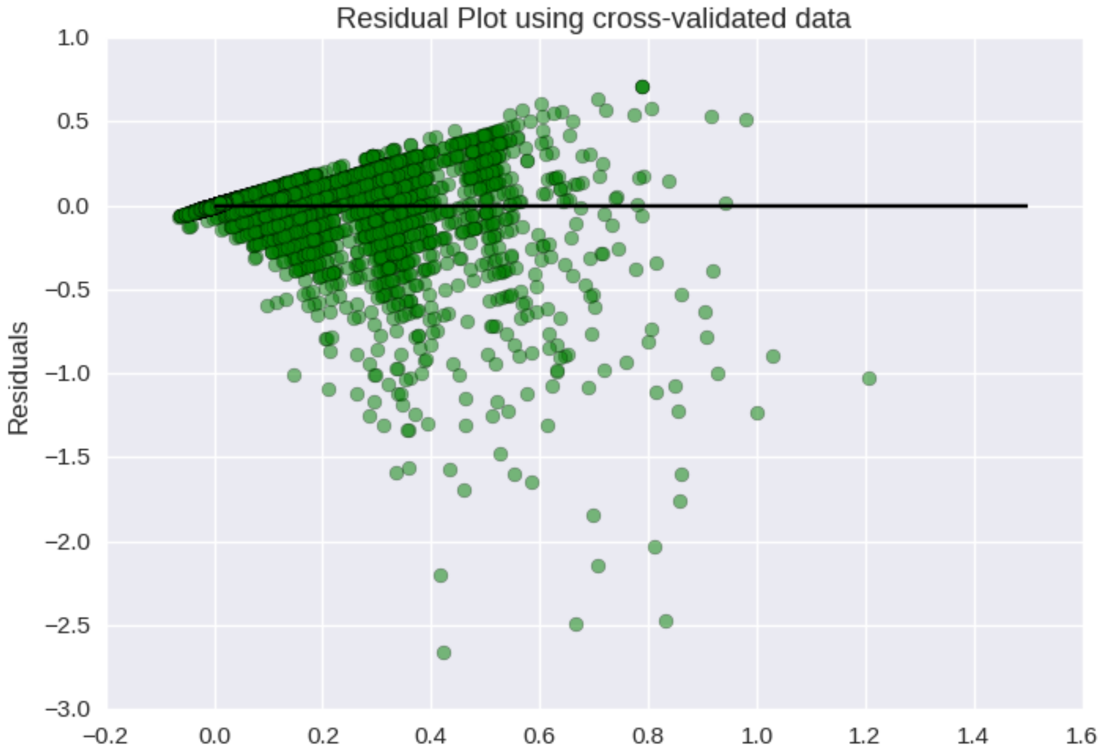
```
chosen alpha:  10.0
ridge score:  0.519202385901
predicted:  [ 0.17457528 -0.00871101 -0.01980315 ...,  0.12200059  0.01135503
  0.01135503]
```

In [109]:
```
fig, ax = plt.subplots()
Y = Y_priorRate
r2 = linear_score
ax.scatter(Y,predicted)
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--', label ="r^2="+str(r2),l
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```



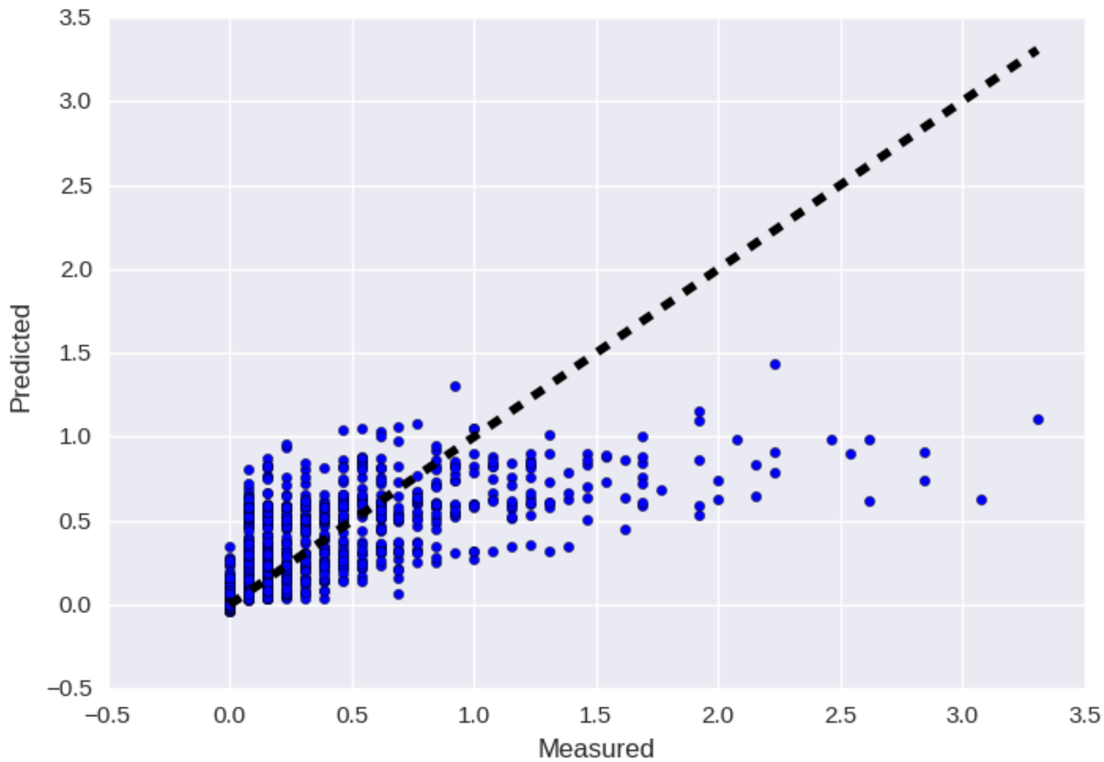
In [110]:
```
plt.scatter(predicted, predicted- Y, c='g', s=40, alpha=0.5)
plt.hlines(y = 0, xmin=0, xmax = 1.5)
plt.title('Residual Plot using cross-validated data')
plt.ylabel('Residuals')
```

Out[110]: <matplotlib.text.Text at 0x7f8002b82390>



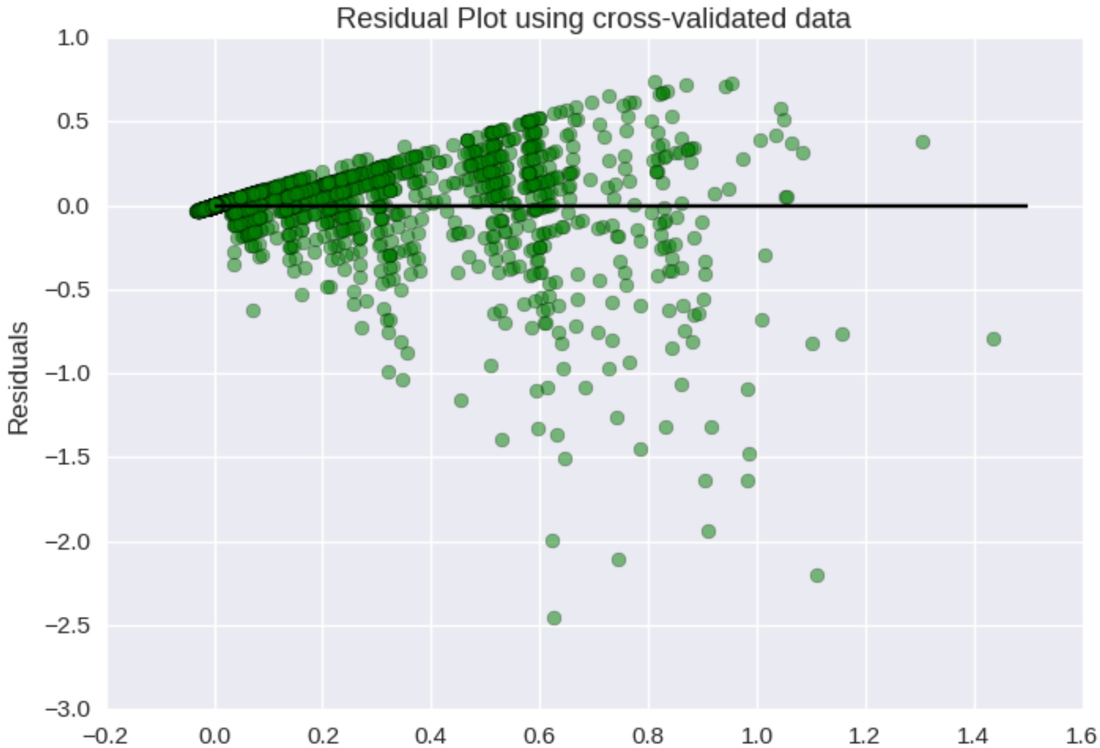
In [125]:
```
ridgemodel = lm.RidgeCV(cv=5).fit(Xc,Yc_priorRate)
predicted = ridgemodel.predict(Xc)
ridge_score = ridgemodel.score(Xc,Yc_priorRate)

print("chosen alpha: ",ridgemodel.alpha_)
print("ridge score: ", np.mean(ridge_score))
print("predicted: ", predicted)

fig, ax = plt.subplots()
Y = Yc_priorRate
r2 = linear_score
ax.scatter(Y,predicted)
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--',label ="r^2="+str(r2), l
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

```
chosen alpha:  10.0
ridge score:  0.620701229935
predicted:  [ 0.159844   -0.01657434 -0.00851878 ...,  0.00251754  0.00251754
  0.00251754]
```



In [112]:
```
plt.scatter(predicted, predicted- Y, c='g', s=40, alpha=0.5)
plt.hlines(y = 0, xmin=0, xmax = 1.5)
plt.title('Residual Plot using cross-validated data')
plt.ylabel('Residuals')
```

Out[112]: <matplotlib.text.Text at 0x7f8002af3860>



We have a slight improvement over our least squares model. Unfortunately still not very good. And we still underestimate systematically the high risk intersections. What can we accomplish by trying an ensemble method? And perhaps including another feature, like postal codes?

Not every postal code will be working to add information. But a few will be ranked highly when we look at the feature importance output for a random forrest.

In [113]:
```
Xrf = df_data[['bike traffic estimate','vehicle traffic','pedestrian traffic',
Yrf_priorTotals = df_data['total collisions']
Yrf_priorRate = df_data['total collisions']/13
Yrf_postRate = df_data['posterior mean rate']
Yrf_RiskRate = df_data['scaled collisions']/13

Xrf_c = df_data_central[['bike traffic estimate','vehicle traffic','pedestrian
Yrf_priorTotals_c = df_data_central['total collisions']
Yrf_priorRate_c = df_data_central['total collisions']/13
Yrf_postRate_c = df_data_central['posterior mean rate']
Yrf_RiskRate_c = df_data_central['scaled collisions']/13
```

In [114]:
```
Xrf.fillna('Minor Arterial',inplace= True)
```

```
/home/christian/anaconda3/lib/python3.5/site-packages/pandas/core/frame.py:27
56: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  downcast=downcast, **kwargs)
```

In [115]:
```
control_dict = {'traffic signal/gate':1,'stop/yield sign':2, 'pedestrian cross
Xrf['control_type_num'] = Xrf['control type'].map(control_dict)
Xrf_c['control_type_num'] = Xrf_c['control type'].map(control_dict)
```

```
/home/christian/anaconda3/lib/python3.5/site-packages/ipykernel/__main__.py:2
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  from ipykernel import kernelapp as app
/home/christian/anaconda3/lib/python3.5/site-packages/ipykernel/__main__.py:3
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  app.launch_new_instance()
```

In [116]:
```
roadclass_dict = {'Collector/Expressway':4,'Major Arterial':3,'Minor Arterial'
Xrf['road_class_num'] = Xrf['road_class'].map(roadclass_dict)

Xrf_c.loc[:,'road_class_num'] = Xrf_c.loc[:,'road_class'].map(roadclass_dict)
```

```
/home/christian/anaconda3/lib/python3.5/site-packages/ipykernel/__main__.py:2
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  from ipykernel import kernelapp as app
/home/christian/anaconda3/lib/python3.5/site-packages/pandas/core/indexing.py
:284: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  self.obj[key] = _infer_fill_value(value)
/home/christian/anaconda3/lib/python3.5/site-packages/pandas/core/indexing.py
:461: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/st
able/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas
-docs/stable/indexing.html#indexing-view-versus-copy)
  self.obj[item] = s
```

In [117]:
```
Xrf = Xrf.drop(['road_class','control type'] , axis=1)
Xrf_c = Xrf_c.drop(['road_class','control type'],axis =1)
```

In [65]:
```
from sklearn.model_selection import train_test_split
Xc_train, Xc_test, Yc_train, Yc_test = train_test_split(Xrf_c, Yrf_priorRate_c
print (Xc_train.shape)
print (Xc_test.shape)
print (Yc_train.shape)
print (Yc_test.shape)

X_train, X_test, Y_train, Y_test = train_test_split(Xrf, Yrf_priorRate, test_s
print (X_train.shape)
print (X_test.shape)
print (Y_train.shape)
print (Y_test.shape)
```

```
(14019, 5)
(4674, 5)
(14019,)
(4674,)
(15278, 5)
(5093, 5)
(15278,)
(5093,)
```

In [140]:
```
rf_c= RandomForestRegressor(n_estimators = 200)
prediction_c = cross_val_predict(rf_c,Xrf_c,Yrf_priorRate_c, cv=6)
r2_c = np.mean(cross_val_score(rf_c,Xrf_c,Yrf_priorRate_c,cv=6))
mse = np.mean((Yrf_priorRate_c-prediction_c)**2)
print('MSE : ',mse)

print("R^2 : ", r2_c)

plt.scatter(Yrf_priorRate_c,prediction_c)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2_c), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

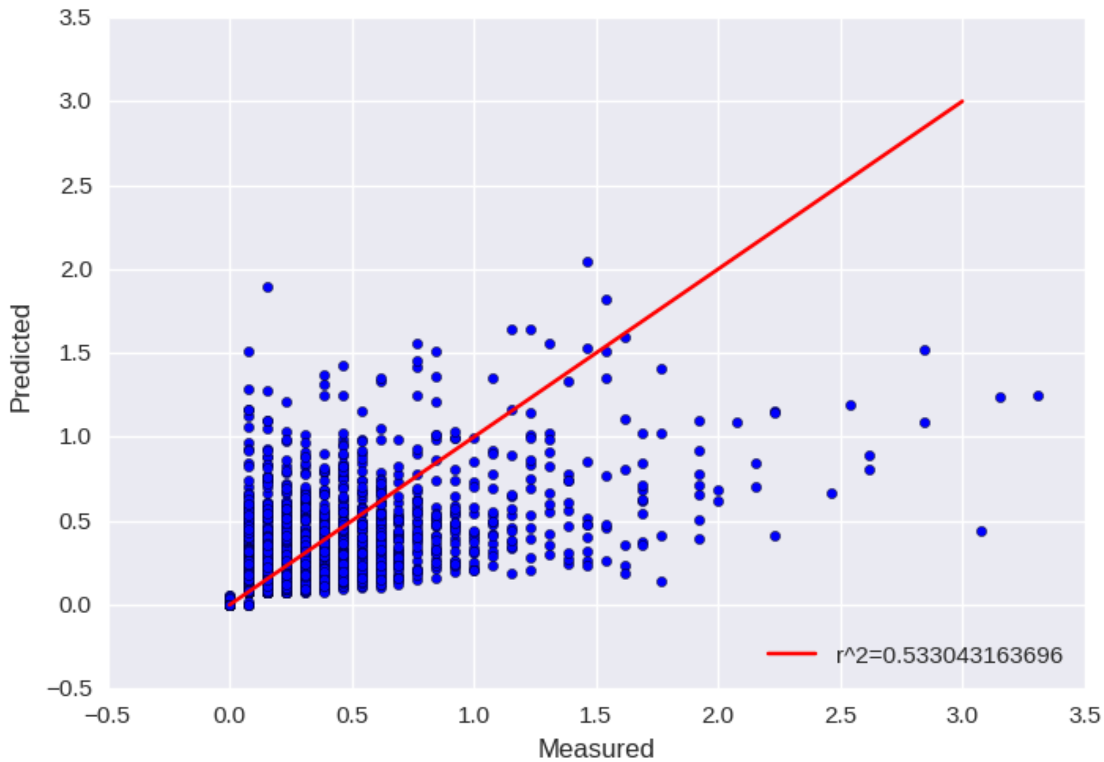
```
MSE :  0.007104826716178006
R^2 :  0.613469493597
```

Out[140]: <matplotlib.text.Text at 0x7f8002250f28>

In [135]:
```
rf= RandomForestRegressor(n_estimators = 200)
prediction = cross_val_predict(rf,Xrf,Yrf_priorRate, cv=5)
r2 = np.mean(cross_val_score(rf,Xrf,Yrf_priorRate,cv=5))
mse = np.mean((Yrf_priorRate-prediction)**2)
print('MSE : ',mse)

print("R^2 : ", r2)

plt.scatter(Yrf_priorRate,prediction)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.013283379627525 27
R^2 :  0.533043163696
```

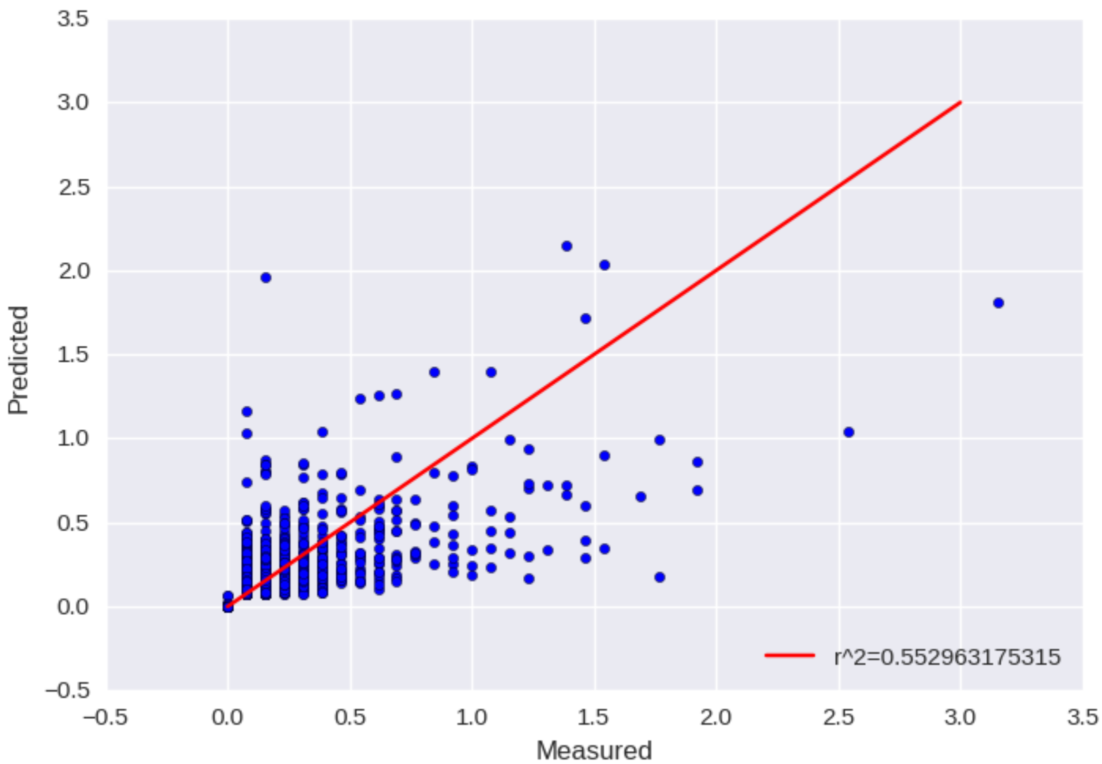
Out[135]: <matplotlib.text.Text at 0x7f80023e0400>



In [66]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
rf= RandomForestRegressor(n_estimators = 200)
rf.fit(X_train,Y_train)
r2 = r2_score(Y_test, rf.predict(X_test))
mse = np.mean((Y_test-rf.predict(X_test))**2)
print('MSE : ',mse)

print("R^2 : ", r2)

plt.scatter(Y_test,rf.predict(X_test))
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.012309838111176129
R^2 :  0.552963175315
```

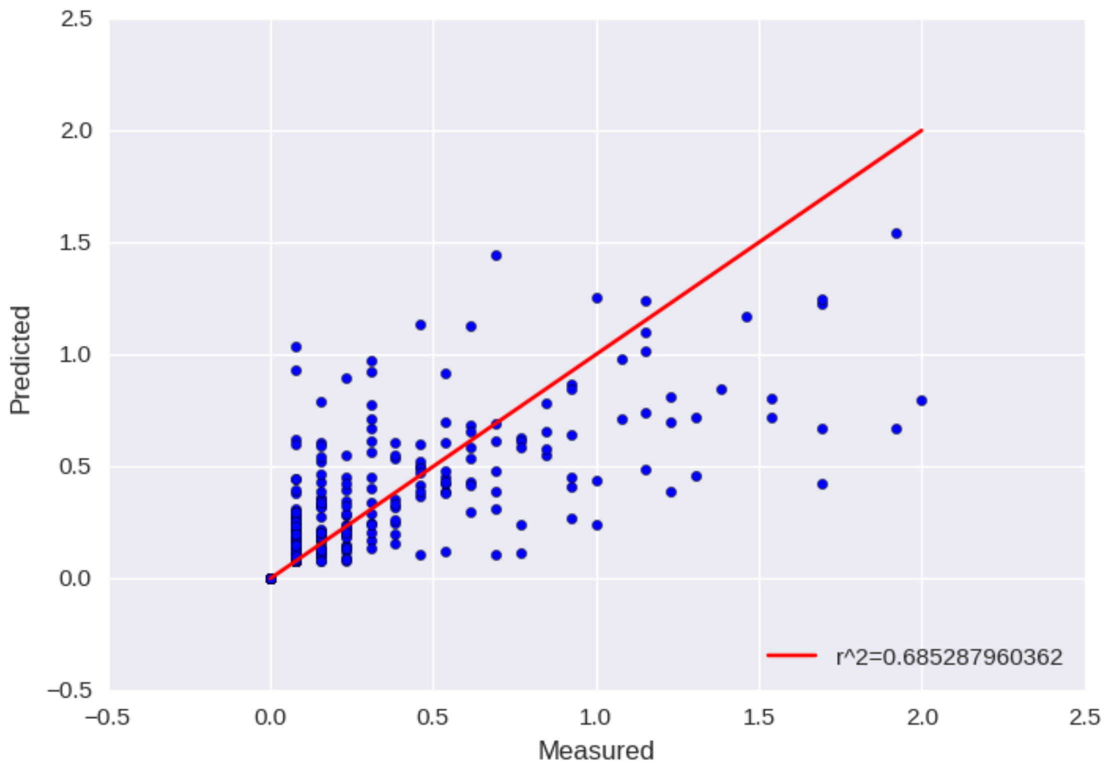
Out[66]: <matplotlib.text.Text at 0x7f802a26f208>



In [67]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
rf_c= RandomForestRegressor(n_estimators = 200)
rf_c.fit(Xc_train,Yc_train)
r2_c = r2_score(Yc_test, rf_c.predict(Xc_test))
mse = np.mean((Yc_test-rf_c.predict(Xc_test))**2)
print('MSE : ',mse)

print("R^2 : ", r2_c)

plt.scatter(Yc_test,rf_c.predict(Xc_test))
plt.plot(np.arange(0,3),np.arange(0,3), label ="r^2="+str(r2_c), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.0052371722752427245
R^2 :  0.685287960362
```

Out[67]: <matplotlib.text.Text at 0x7f801c5ec5c0>



In [68]:
```
names = list(Xrf.columns.values)
print("Features sorted by their score:")
print(sorted(zip(map(lambda x: round(x, 4), rf.feature_importances_), names),
             reverse=True))
```

```
Features sorted by their score:
[(0.43190000000000001, 'control_type_num'), (0.23899999999999999, 'pedestrian
 traffic'), (0.15890000000000001, 'vehicle traffic'), (0.14030000000000001, '
bike traffic estimate'), (0.029999999999999999, 'road_class_num')]
```

Let's drop our bike traffic estimates and see how performance changes

In [69]:
```
X2 = Xrf.drop(['bike traffic estimate'],axis=1)
Xc2 = Xrf_c.drop(['bike traffic estimate'],axis=1)
```

In [70]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
rf= RandomForestRegressor(n_estimators = 200)
prediction = cross_val_predict(rf,X2,Yrf_priorRate)
r2 = np.mean(cross_val_score(rf,X2,Yrf_priorRate))
mse = np.mean((Yrf_priorRate-prediction)**2)
print('MSE : ',mse)

print("R^2 : ", r2)

plt.scatter(Yrf_priorRate,prediction)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.013841523328337628
R^2 :  0.510496830828
```

Out[70]: <matplotlib.text.Text at 0x7f801c5bed30>

In [71]:
```
from sklearn.ensemble import RandomForestRegressor
rf= RandomForestRegressor(n_estimators = 200)
prediction = cross_val_predict(rf,Xc2,Yrf_priorRate_c)
r2 = np.mean(cross_val_score(rf,Xc2,Yrf_priorRate_c))
mse = np.mean((Yrf_priorRate_c-prediction)**2)
print('MSE : ',mse)

print("R^2 : ", r2)

plt.scatter(Yrf_priorRate_c,prediction)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.00777256328566422l
R^2 :  0.577596318089
```

Out[71]: <matplotlib.text.Text at 0x7f8003310048>

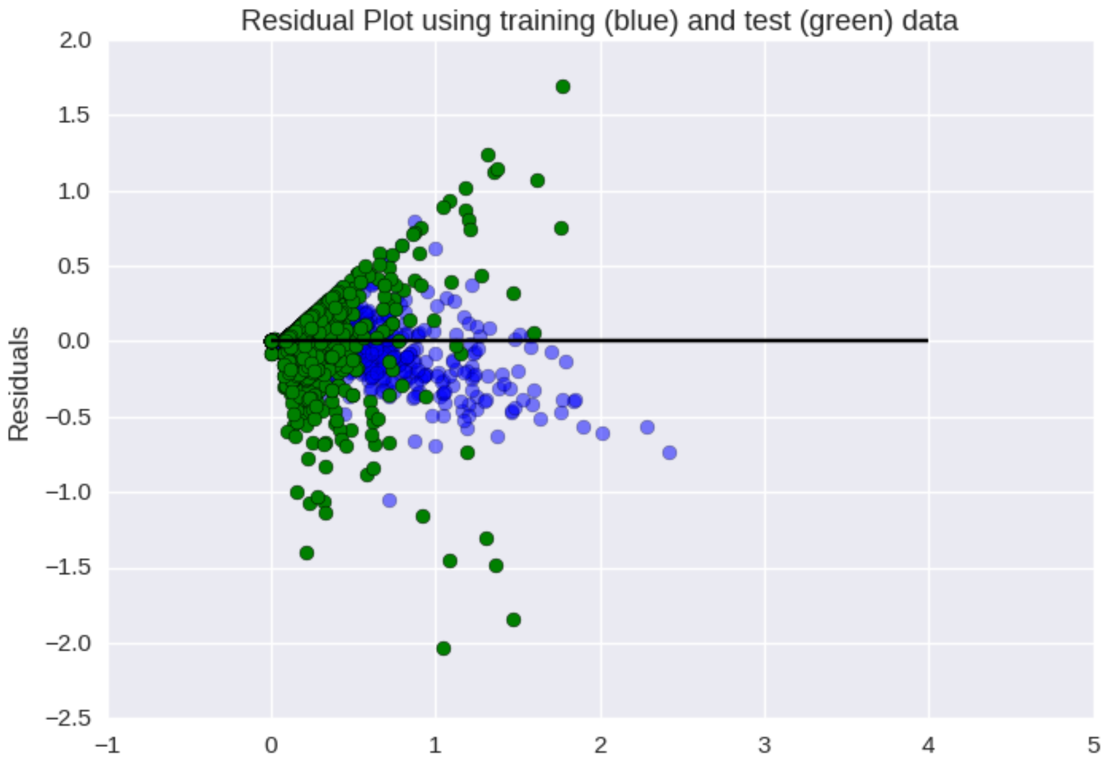
Testing performance is worse without the bike estimates.

Compared to our singular linear regression approaches, this ensemble of decision trees gives a significant improvement in predictive power.

In [106]:

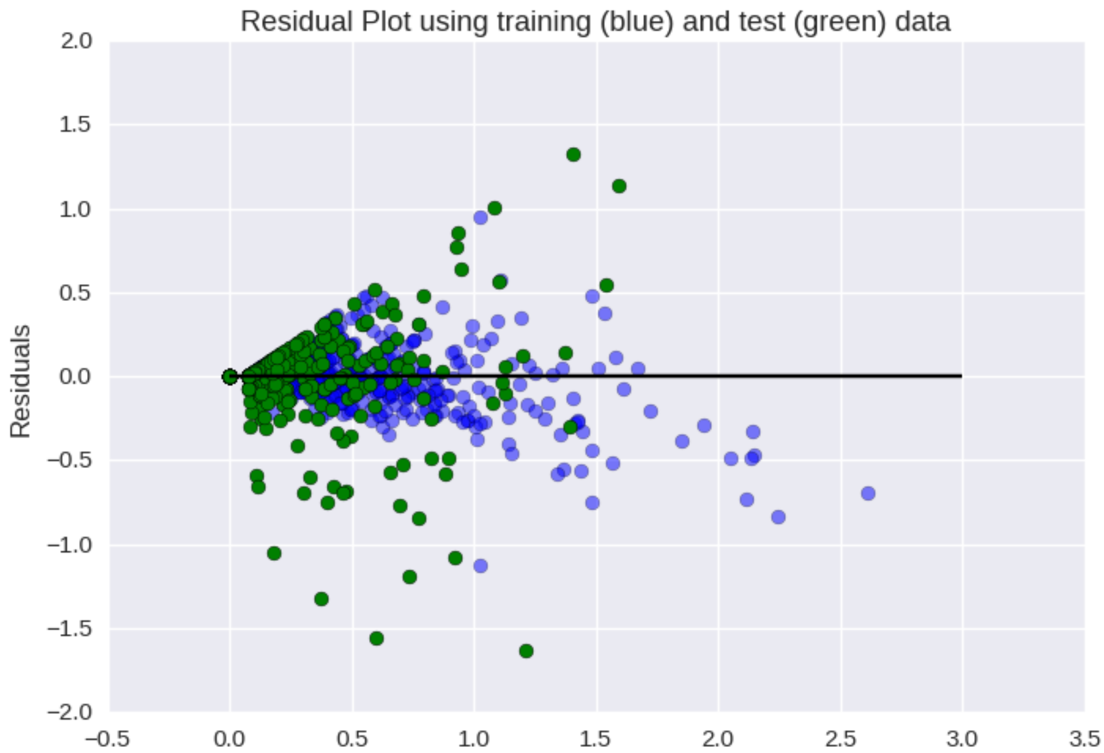
```
plt.scatter(rf.predict(X_train), rf.predict(X_train) - Y_train, c='b', s=40, a
plt.scatter(rf.predict(X_test), rf.predict(X_test) - Y_test, c='g', s=40)
plt.hlines(y = 0, xmin=0, xmax = 4)
plt.title('Residual Plot using training (blue) and test (green) data')
plt.ylabel('Residuals')
```

Out[106]: <matplotlib.text.Text at 0x7faf5d03d9e8>



In [107]:
```
plt.scatter(rf_c.predict(X2c_train), rf_c.predict(X2c_train) - Yc_train, c='b'
plt.scatter(rf_c.predict(X2c_test), rf_c.predict(X2c_test) - Yc_test, c='g', s
plt.hlines(y = 0, xmin=0, xmax = 3)
plt.title('Residual Plot using training (blue) and test (green) data')
plt.ylabel('Residuals')
```

Out[107]: <matplotlib.text.Text at 0x7faf5cfd5630>



Defintely an improvement there.

In [72]:
```
X4 = Xrf.copy()
Xc4 = Xrf_c.copy()
```

In [73]:
```
X4.reset_index(inplace=True)
st1 = X4['intersection'].str.split('Ontario, ').str.get(1)

Xc4.reset_index(inplace=True)
st1_c = Xc4['intersection'].str.split('Ontario, ').str.get(1)

st2 = st1.str.split(' ').str.get(0)
st2.to_csv('postal_codes.csv')
st2_c = st1_c.str.split(' ').str.get(0)
st2_c.to_csv('postal_codes_c.csv')
```

In [74]:
```
st2 = pd.DataFrame.from_csv('postal_codes2.csv',header=None)

st2_c = pd.DataFrame.from_csv('postal_codes_c2.csv',header=None)
```

In [75]:
```
X4.head()
```

Out[75]:

	intersection	bike traffic estimate	vehicle traffic	pedestrian traffic	control_type_num	road_class_num
0	bpNichol Ln & bpNichol Ln, Toronto, Ontario, M5S	1216.4	11213	4767	5	3
1	Zachary Ct & Zachary Ct, Toronto, Ontario, M6A	1111.1	12798	2011	4	1
2	York Ridge Rd & York Ridge Rd, Toronto, Ontari...	783.49	13638	72	4	2
3	York Mills Rd & York Ridge Rd, Toronto, Ontari...	783.49	13638	72	4	2
4	York Mills Rd & York Mills Rd, Toronto, Ontari...	1106.1	25718	1926	4	2

In [76]:
```
X4.set_index('intersection', inplace= True)
Xc4.set_index('intersection', inplace= True)
#X4.drop('index',inplace = True, axis=1)
```

In [77]:
```
X4['postal'] = st2[1].values
Xc4['postal'] = st2_c[1].values
```

In [78]:
```
postal_df = pd.get_dummies(X4.postal, dummy_na = False)
X4 = pd.concat([X4,postal_df],axis=1)

postal_df = pd.get_dummies(Xc4.postal,dummy_na=False)
Xc4 = pd.concat([Xc4,postal_df],axis=1)
```

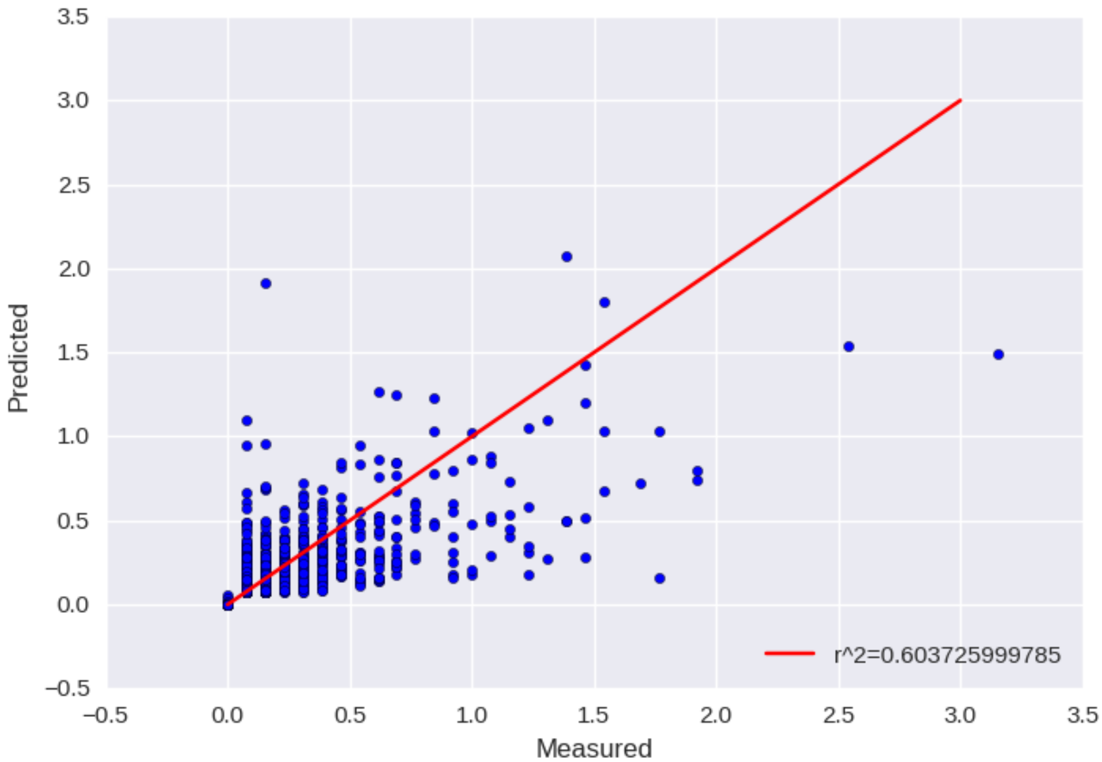
In [79]:
```
X4.drop('postal',inplace = True, axis = 1)
Xc4.drop('postal',inplace = True, axis = 1)
```

In [80]:
```
Xc_train, Xc_test, Yc_train, Yc_test = train_test_split(Xc4, Yrf_priorRate_c,
print (Xc_train.shape)
print (Xc_test.shape)
print (Yc_train.shape)
print (Yc_test.shape)

X_train, X_test, Y_train, Y_test = train_test_split(X4, Yrf_priorRate, test_si
print (X_train.shape)
print (X_test.shape)
print (Y_train.shape)
print (Y_test.shape)

rf= RandomForestRegressor(n_estimators = 200)

rf.fit(X_train,Y_train)
r2 = r2_score(Y_test, rf.predict(X_test))
mse = np.mean((Y_test-rf.predict(X_test))**2)
print('MSE for all data: ', mse)

rf_c= RandomForestRegressor(n_estimators = 200)
rf_c.fit(Xc_train,Yc_train)
r2_c = r2_score(Yc_test, rf_c.predict(Xc_test))
mse = np.mean((Yc_test-rf_c.predict(Xc_test))**2)
print("MSE for central data: ", mse)

names = list(X4.columns.values)
print("Features sorted by their score:")
print(sorted(zip(map(lambda x: round(x, 4), rf.feature_importances_), names),
             reverse=True))

names2 = list(Xc4.columns.values)
print("Features sorted by their score:")
print(sorted(zip(map(lambda x: round(x, 4), rf_c.feature_importances_), names2
             reverse=True))
```

```
(14019, 115)
(4674, 115)
(14019,)
(4674,)
(15278, 119)
(5093, 119)
(15278,)
(5093,)
MSE for all data:  0.010912006619940696
MSE for central data:  0.004897617639714547
Features sorted by their score:
[(0.43369999999999997, 'control_type_num'), (0.1956, 'pedestrian traffic'), (
0.1048, 'bike traffic estimate'), (0.1013, 'vehicle traffic'), (0.02280000000
0000001, 'road_class_num'), (0.012500000000000001, 'M6G'), (0.0109, 'M5T'), (
0.010699999999999999, 'M6H'), (0.0085000000000000006, 'M5A'), (0.008300000000
0000001, 'M6J'), (0.0064000000000000003, 'M4M'), (0.0061999999999999998, 'M5V
'), (0.0061000000000000004, 'M5B'), (0.0045999999999999999, 'M4L'), (0.003599
9999999999999, 'M5S'), (0.0035000000000000001, 'M4W'), (0.0033, 'M5R'), (0.00
25000000000000001, 'M6K'), (0.0025000000000000001, 'M4K'), (0.0023, 'M6P'), (
0.0022000000000000001, 'N0A'), (0.0022000000000000001, 'M5H'), (0.002, 'M2R')
, (0.0019, 'M5G'), (0.0019, 'M4J'), (0.0019, 'M1K'), (0.0018, 'M4Y'), (0.0016
999999999999999, 'M5E'), (0.0015, 'M1S'), (0.0014, 'M5C'), (0.0014, 'M4V'), (
0.0012999999999999999, 'M5J'), (0.0012999999999999999, 'M3H'), (0.00119999999
99999999, 'M6S'), (0.0011999999999999999, 'M4C'), (0.0011000000000000001, 'M9
A'), (0.0011000000000000001, 'M1V'), (0.0011000000000000001, 'M1E'), (0.00089
999999999999998, 'M2N'), (0.00080000000000000004, 'M3J'), (0.0008000000000000
0004, 'M3C'), (0.00080000000000000004, 'M1R'), (0.00069999999999999999, 'M4X'
), (0.00069999999999999999, 'M1W'), (0.00059999999999999995, 'M9N'), (0.00059
```

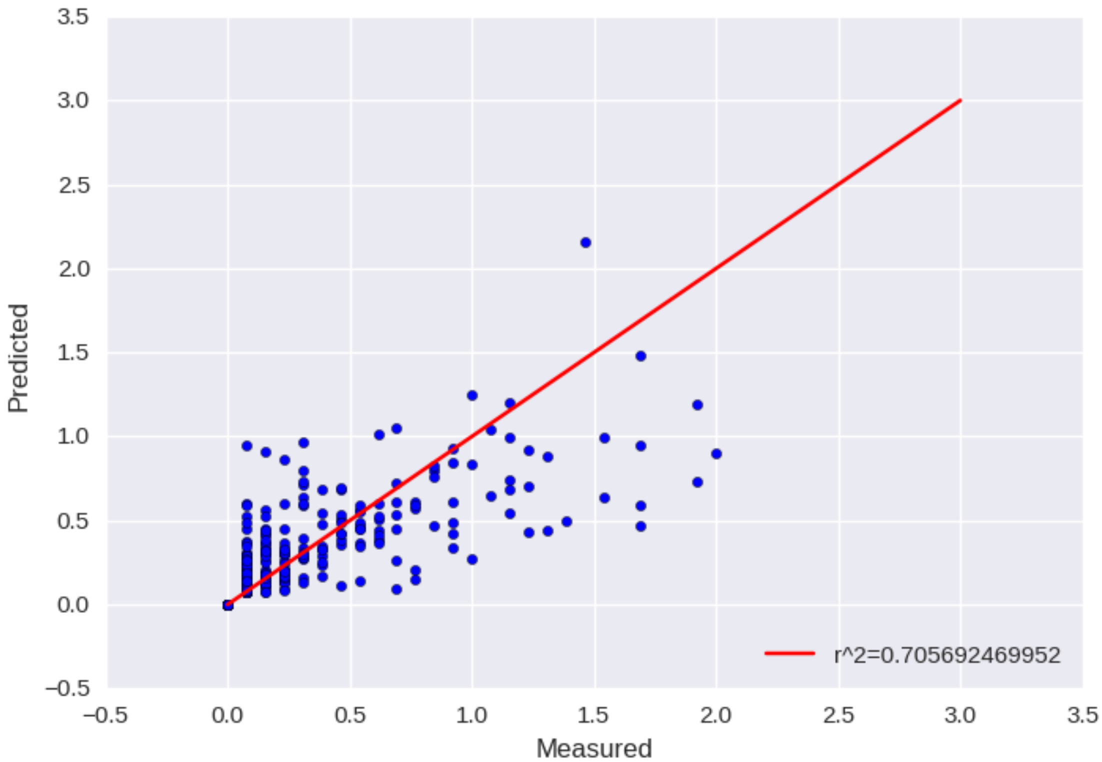
In [81]:
```
plt.scatter(Y_test,rf.predict(X_test))
plt.plot(np.arange(0,3.5),np.arange(0,3.5), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

Out[81]: <matplotlib.text.Text at 0x7f8003288ba8>



In [82]:
```
plt.scatter(Yc_test,rf_c.predict(Xc_test))
plt.plot(np.arange(0,3.5),np.arange(0,3.5), label ="r^2="+str(r2_c), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

Out[82]: <matplotlib.text.Text at 0x7f80031e0cf8>



The fit from our random forest is starting to look pretty good, particularily when we restrict ourselves to the central intersections only, which still contain the bulk of our data. Looking at the output in terms of feature importance, we see that for BOTH datasets, most of the postal codes don't add any information. We could use the SelectFromModel meta-transformer to specify a threshold (manually or as some multiple of the mean), to remove features which are irrelevant. sci-kit lear ndoes this using mean decrease impurity. This is how much the variance is decreased by each feature on average.

Another feature selection method is to measure how much each feature impacts the accuracy of our model. By permuating the values of each feature, we can check how much it affets our prediction. For important variables, there should be a significant decrease. The cell below calculates the mean decrease accuracty for our model

In [83]:

```python
from sklearn.model_selection import ShuffleSplit
from sklearn.metrics import r2_score
from collections import defaultdict

X = X4.values
Y = Yrf_priorRate

rf_alt = RandomForestRegressor()
scores = defaultdict(list)
forest_split = ShuffleSplit(n_splits=10,test_size = .25,random_state=1)
#crossvalidate the scores on a number of different random splits of the data
for train_idx, test_idx in forest_split.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    Y_train, Y_test = Y[train_idx], Y[test_idx]
    r = rf_alt.fit(X_train, Y_train)
    acc = r2_score(Y_test, rf.predict(X_test))
    for i in range(X.shape[1]):
        X_t = X_test.copy()
        np.random.shuffle(X_t[:, i])
        shuff_acc = r2_score(Y_test, rf.predict(X_t))
        scores[names[i]].append((acc-shuff_acc)/acc)
print ("Features sorted by their score:")
print (sorted([(round(np.mean(score), 4), feat) for
              feat, score in scores.items()], reverse=True))
```

```
Features sorted by their score:
[(1.2686999999999999, 'control_type_num'), (0.32779999999999998, 'pedestrian
traffic'), (0.18279999999999999, 'vehicle traffic'), (0.1676, 'bike traffic e
stimate'), (0.055100000000000003, 'road_class_num'), (0.0258, 'M6G'), (0.0212
, 'M6J'), (0.021100000000000001, 'M5A'), (0.020899999999999998, 'M5T'), (0.01
9199999999999998, 'M6H'), (0.014200000000000001, 'M4M'), (0.01020000000000000
1, 'M5B'), (0.0086999999999999994, 'M4L'), (0.0074999999999999997, 'M5V'), (0
.0071999999999999998, 'M6P'), (0.0060000000000000001, 'M5R'), (0.005000000000
0000001, 'M1K'), (0.0045999999999999999, 'M4J'), (0.0043, 'M5S'), (0.0038, 'N
0A'), (0.0035999999999999999, 'M4W'), (0.0035000000000000001, 'M6K'), (0.0033
, 'M4K'), (0.0030000000000000001, 'M1S'), (0.0028, 'M2R'), (0.002200000000000
0001, 'M4V'), (0.0020999999999999999, 'M4C'), (0.0019, 'M1E'), (0.0018, 'M1W'
), (0.0018, 'M1V'), (0.0016999999999999999, 'M4Y'), (0.0016000000000000001, '
M5H'), (0.0016000000000000001, 'M3H'), (0.0015, 'M2N'), (0.0014, 'M5E'), (0.0
012999999999999999, 'M6S'), (0.0012999999999999999, 'M5G'), (0.00129999999999
99999, 'M3J'), (0.0011000000000000001, 'M9A'), (0.00089999999999999998, 'M8V'
), (0.00089999999999999998, 'M3C'), (0.00089999999999999998, 'M1L'), (0.00080
000000000000004, 'M6R'), (0.00080000000000000004, 'M6N'), (0.0008000000000000
0004, 'M5J'), (0.00069999999999999999, 'M8Z'), (0.00069999999999999999, 'M4X'
), (0.00069999999999999999, 'M1R'), (0.00059999999999999995, 'M9M'), (0.00059
999999999999995, 'M9C'), (0.00050000000000000001, 'M9N'), (0.0005000000000000
0001, 'M9B'), (0.00050000000000000001, 'M4N'), (0.00050000000000000001, 'M3L'
), (0.00040000000000000002, 'M9R'), (0.00040000000000000002, 'M6M'), (0.00040
000000000000002, 'M5C'), (0.00040000000000000002, 'M4R'), (0.0004000000000000
0002, 'M4E'), (0.00040000000000000002, 'M3B'), (0.00040000000000000002, 'M1T'
), (0.00040000000000000002, 'M1H'), (0.00029999999999999997, 'M9W'), (0.00029
999999999999997, 'M6E'), (0.00029999999999999997, 'M6B'), (0.0002999999999999
9997, 'M5M'), (0.00029999999999999997, 'M4P'), (0.00029999999999999997, 'M2K'
), (0.00029999999999999997, 'M2J'), (0.00020000000000000001, 'M9V'), (0.00020
000000000000001, 'M9P'), (0.00020000000000000001, 'M8Y'), (0.0002000000000000
0001, 'M8W'), (0.00020000000000000001, 'M6A'), (0.00020000000000000001, 'M4S'
), (0.00020000000000000001, 'M4B'), (0.00020000000000000001, 'M3N'), (0.00020
000000000000001, 'M3A'), (0.00020000000000000001, 'M2H'), (0.0002000000000000
0001, 'M1P'), (0.00020000000000000001, 'M1J'), (0.00020000000000000001, 'M1B'
), (0.0001, 'M8X'), (0.0001, 'M7A'), (0.0001, 'M6L'), (0.0001, 'M6C'), (0.000
1, 'M5P'), (0.0001, 'M5N'), (0.0001, 'M4T'), (0.0001, 'M4H'), (0.0001, 'M4G')
, (0.0001, 'M4A'), (0.0001, 'M3M'), (0.0001, 'M2L'), (0.0001, 'M1N'), (0.0001
, 'M1G'), (0.0001, 'M1C'), (0.0, 'M9L'), (0.0, 'M5X'), (0.0, 'M5L'), (0.0, 'M
3K'), (0.0, 'M2P'), ( 0.0, 'M2M'), (0.0, 'M1X'), (0.0, 'M1M'), (0.0, 'L6T'),
```

This gives similar results as before. Let's reduce our postal code features, leaving only those which are scoring above half the mean by the decrease in mean variance.

In [84]:
```
from sklearn.feature_selection import SelectFromModel

rf= RandomForestRegressor(n_estimators = 20)


rf_c= RandomForestRegressor(n_estimators = 20)

model = SelectFromModel(rf,threshold = '0.25*mean')
model.fit(X4,Yrf_priorRate)

X5 = model.transform(X4)

model_c = SelectFromModel(rf_c,threshold = '0.25*mean')
model_c.fit(Xc4,Yrf_priorRate_c)
Xc5 = model_c.transform(Xc4)
```

In [85]:
```
X5.shape
```

Out[85]: (20371, 23)

In [86]:
```
Xc5.shape
```

Out[86]: (18693, 16)

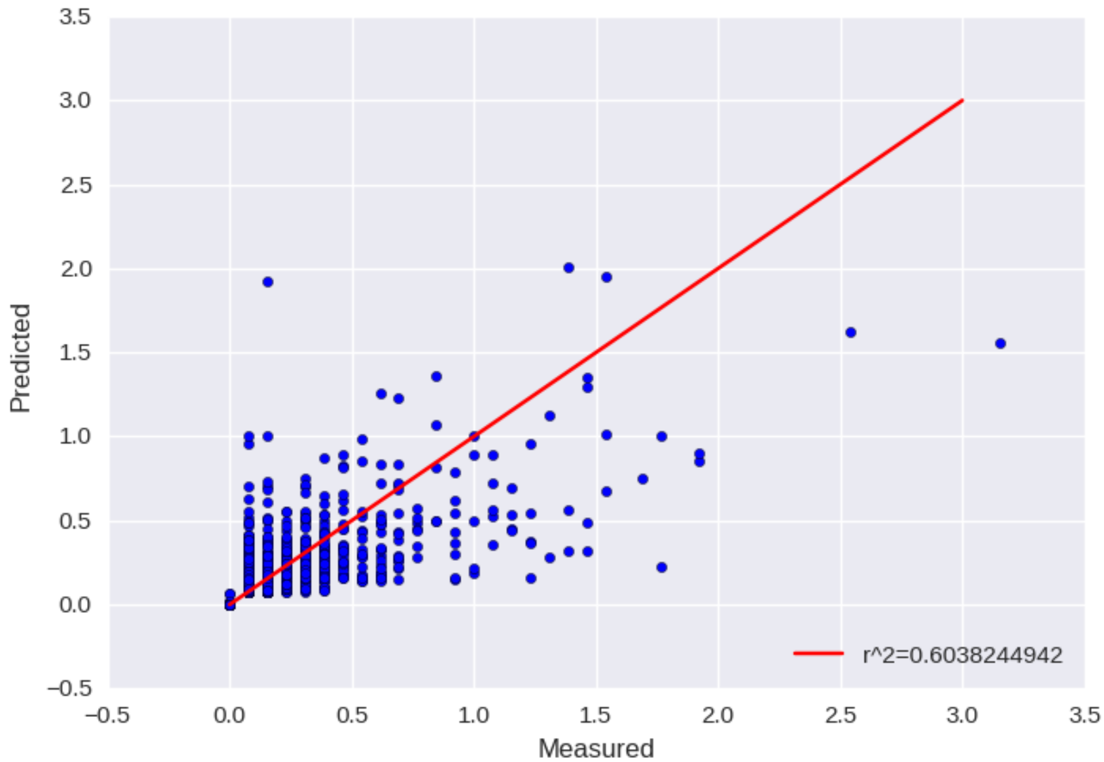
So we are taking the first 23 features for our full data set, and the first 17 for our central only data set. This corresponds to 18 postal codes and 12 postal codes, respectively. Looking at the individual postal codes, there is strong overlap between the two lists.

In [87]:
```
Xc_train, Xc_test, Yc_train, Yc_test = train_test_split(Xc5, Yrf_priorRate_c,
print (Xc_train.shape)
print (Xc_test.shape)
print (Yc_train.shape)
print (Yc_test.shape)

X_train, X_test, Y_train, Y_test = train_test_split(X5, Yrf_priorRate, test_si
print (X_train.shape)
print (X_test.shape)
print (Y_train.shape)
print (Y_test.shape)

rf= RandomForestRegressor(n_estimators = 200)

rf.fit(X_train,Y_train)
r2 = r2_score(Y_test, rf.predict(X_test))
mse = np.mean((Y_test-rf.predict(X_test))**2)
print('MSE for all data: ', mse)

rf_c= RandomForestRegressor(n_estimators = 200)
rf_c.fit(Xc_train,Yc_train)
r2_c = r2_score(Yc_test, rf_c.predict(Xc_test))
mse = np.mean((Yc_test-rf_c.predict(Xc_test))**2)
print("MSE for central data: ", mse)
```

```
(14019, 16)
(4674, 16)
(14019,)
(4674,)
(15278, 23)
(5093, 23)
(15278,)
(5093,)
MSE for all data:  0.010909294426584823
MSE for central data:  0.0048862256129217855
```

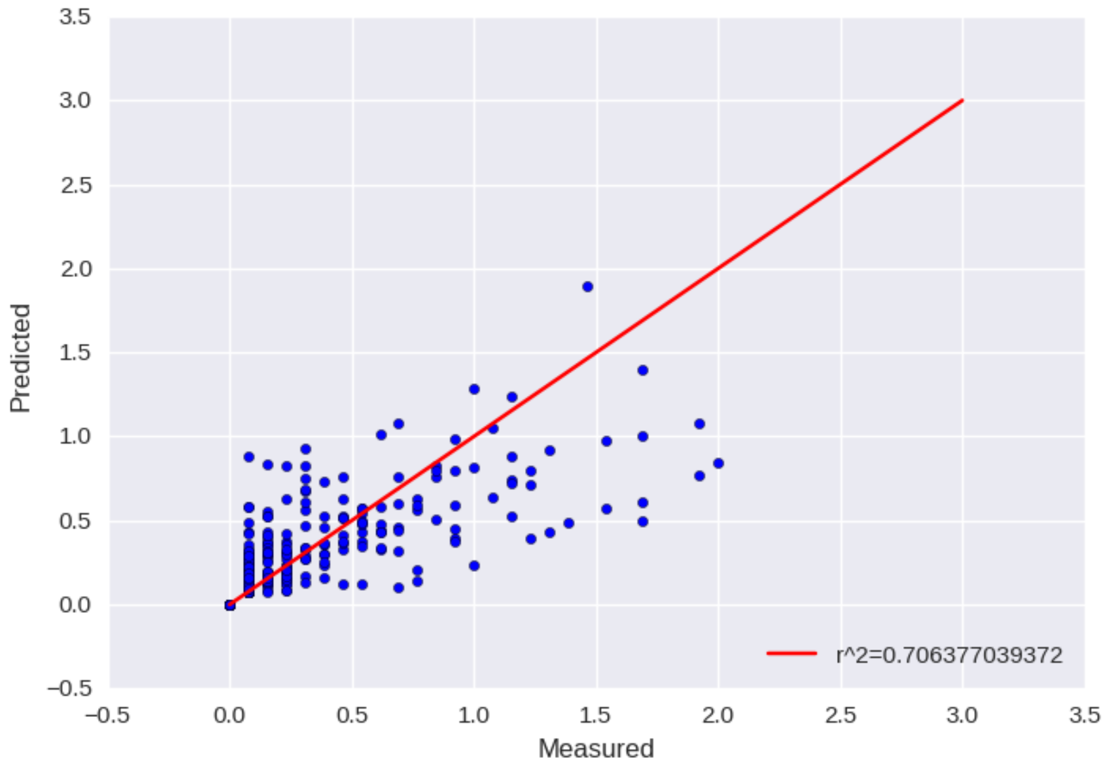
In [88]:
```
plt.scatter(Y_test,rf.predict(X_test))
plt.plot(np.arange(0,3.5),np.arange(0,3.5), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

Out[88]: <matplotlib.text.Text at 0x7f800319e828>



In [89]:
```
plt.scatter(Yc_test,rf_c.predict(Xc_test))
plt.plot(np.arange(0,3.5),np.arange(0,3.5), label ="r^2="+str(r2_c), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

Out[89]: <matplotlib.text.Text at 0x7f8003058278>



Our predictive power actually decreased in the full dataset, from about $R^2 = 0.61$ to $R^2=0.6$. However there was a slight improvement for the central only data set. This indicates that we should not use the same theshold in both cases, even if it's scaled by the mean score and not a constant value.

Ultimately, we have managed to build a predictive random forest, which using only traffic numbers (and an exponentially fit bike traffic estimate), along with the road type of the busiest road, and the traffic control type, performs a reasonable estimate of the yearly collision rates (accidents per year), at intersections for cyclists in the City of Toronto.

The scores are patricularily good when we restrict ourselves to the central post codes. This is partly attributed to the fact that the bike traffic estimates were fit with a few dozen points, all in the central neighbourhoods of Toronto. Had we better bike counts throughout the whole city, the predictive power would no doubt improve.

To output a final result, we'd lke to see how our random forest performs on the entire dataset, by cross validating so that every single point is contained in the test set at some point. Up to this point we were using the random state of our training splits to compare changes to the same test sets each time. While a random forest should not be prone to overfitting, thus not requiring cross_validation in the vast majority of cases, in order to gain a final answer on how well our forest performs, we'd like to generalize it's testing as much as possible.

In [90]:

```
rf= RandomForestRegressor(n_estimators = 200)
prediction = cross_val_predict(rf,X5,Yrf_priorRate)
r2 = np.mean(cross_val_score(rf,X5,Yrf_priorRate))
mse = np.mean((Yrf_priorRate-prediction)**2)
print('MSE : ',mse)

print("R^2 : ", r2)

plt.scatter(Yrf_priorRate,prediction)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.012094328875085904
R^2 :  0.571129046125
```

Out[90]: <matplotlib.text.Text at 0x7f80030c6898>

In [91]:

```
rf_c= RandomForestRegressor(n_estimators = 200)
prediction_c = cross_val_predict(rf_c,Xc5,Yrf_priorRate_c)
r2_c = np.mean(cross_val_score(rf_c,Xc5,Yrf_priorRate_c))
mse = np.mean((Yrf_priorRate_c-prediction_c)**2)
print('MSE : ',mse)

print("R^2 : ", r2_c)

plt.scatter(Yrf_priorRate_c,prediction_c)
plt.plot(np.arange(0,4),np.arange(0,4), label ="r^2="+str(r2_c), c="r")
plt.legend(loc="lower right")
plt.xlabel('Measured')
plt.ylabel('Predicted')
```

```
MSE :  0.006807085857277491
R^2 :  0.626891405509
```

Out[91]: <matplotlib.text.Text at 0x7f801c4ad160>

We see here that our choice of random state was fortuitous, and it so happened to give us a training/test split that lent itself to high scores.

While our score is lowered significantly, we still perform quite will, being able to explain over 63 percent of the variance in our data set with our very barebones model. Further improvements could be made by including distances to bike posts/parking lots, and whether or not a bike lane is present at the intersection. While this would make the project more involved, combining these extra features with more data from the years since 2010 should result in a model that is able to closely predict real life collision rates.

In [ ]:

In [ ]:

```

```

Bonus:

If we were to really try and ring out the best performance possible out of our forrest, the correct approach would be to run a randomized search over the various possible parameters, so as to select the forest type which performs best. One could even then re-do the search, wich narrower parameter ranges, to fine tune things furter.

The code below performs this search for the central only database. It will run for a very long time however, and the results will be updated into the notebook at a later date, since for our relatively sparsely featured and small data set, reducing the max features, max depth, or other parametes will not improve our score, or change the computational cost much. Nonetheless, it's a good exercise to carry out.

In [ ]:
```
import numpy as np

from time import time
from scipy.stats import randint as sp_randint

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier

# build our random forest
clf = RandomForestRegressor(n_estimators=200)


# Utility function to report best scores
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (std: {1:.3f})".format(
                  results['mean_test_score'][candidate],
                  results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")


# specify parameters and distributions to sample from
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1,115),
              "min_samples_split": sp_randint(1, 20),
              "min_samples_leaf": sp_randint(1, 20),
              "bootstrap": [True, False],
              "criterion": ["mse", "mae"]} ##mean squared error, and mean abso

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search)

start = time()
random_search.fit(Xc5,Yrf_priorRate_c)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)
```