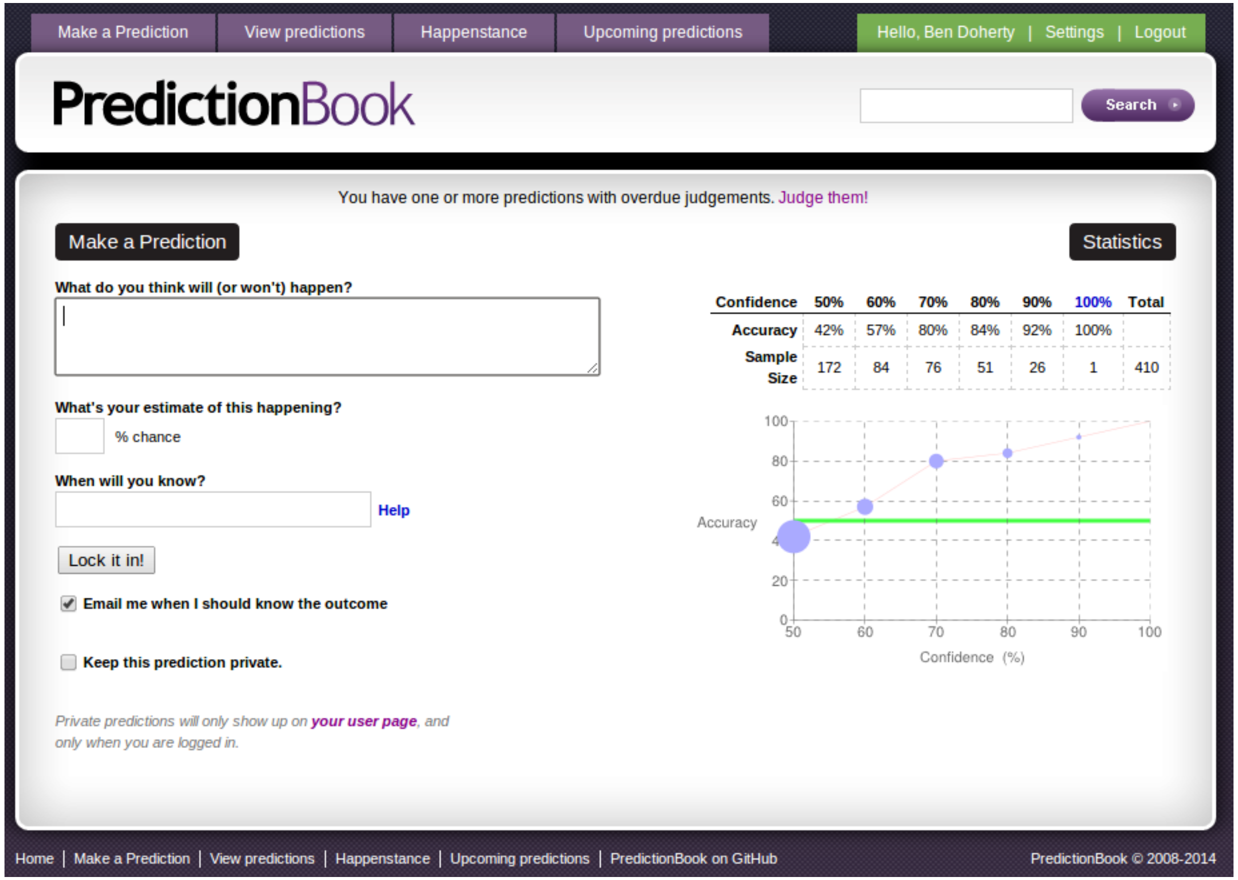
PredictionBook Analysis



Prediction book is a service that allows people to record their predictions along with their confidence in these predictions. Explicitly stating, and recording, confidence in a prediction started with meteorologists in Australia in 1906 , but has been explicitly been done by anyone who was making or accepting a bet. If the person was certain that the event would occur then they should be willing to stake their entire worth on it, so if someone isn't willing to do so they are indicating (imperfectly: risk aversion, discounting etc.) their degree of confidence in an event's likelihood to occur.

Don't worry about the `DeprecationWarning: height has been deprecated.` warning. It is caused by something that they did in Pandas .13 that is fixed in .14 which will be deployed soon

It is helpful to have an understanding of one's calibration with regard to prediction confidence. If a well calibrated person has 70% confidence in an event, then over 100 events that they are this confident of happening there should be about 70 of them that happen and 30 that don't. Or in other words, the confidence can be taken to be the likelihood of that event occurring. Problem Their current feedback on prediction calibration is quite limited. It aggregates predictions across the entire time that a user has been with the site. One of the site's key jobs is to train people to be more accurate predictors, so if they improve then their output won't fully reflect this (e.g. if you start off overconfident, then your output will stay overconfident until you become underconfident for a while).

What can the data tell us that we didn't already know? How can the data be presented in a way that will make people better calibrated? Are there any lessons about general decision making that we can discover from this set?

Data

Prediction book gave me a sanitised version of their data (no names, email addresses, prediction text or anything that could be used to make the data personal) as an sql database (7.7mb)

Hypothesis

[I'm still a bit hazy about the hypothesis] These data can reveal some trends that will be useful in teaching people to make better decisions.

Methods

I'd like to do some clustering to see if people fall into different user patterns: prediction outlook (time between now and judgement), time of day prediction made, confidence. Also some general correlations between these factors: are people over confident in the morning?

Until I get more into the data I'm not sure what I can find from it.

Business applications

This might open up more features for the site itself. The insight into how people think might have academic value (I wrote a thesis about this some time ago at university).

defining functions - boring alert

In [1]:
```
import MySQLdb
import pandas as pd
from __future__ import division
from sklearn import preprocessing

from pylab import plot,show
from numpy import vstack,array
from numpy.random import rand
from scipy.cluster.vq import kmeans,vq,whiten
```

In [2]:
```
def queryAsTable(query, maxrows=5, how=2):
    """This queries a db and returns the response as a pandas dataframe"""
    # Open database connection
    db = MySQLdb.connect(user = 'root',
                         passwd = 'password',
                         db = 'mysql',
                         host = 'localhost')
    db.query(query)
    result = db.store_result()
    rdict = list(result.fetch_row(maxrows=maxrows, how=how)) #max=0 means all, how=0 means tuple, how=1-dict, how=2 dict with fully qualified names
    df = pd.DataFrame(rdict)
    # disconnect from server
    db.close()
    return df
```

In [3]:
```
def binConfidence(predictions):
    confBins = {}
    for b in range(0,110,10):
        confBins[b] = 0   #produces: {0: 0, 100: 0, 70: 0, 40: 0, 10: 0, 80: 0, 50: 0, 20: 0, 90: 0, 60: 0, 30: 0} (dicts are unordered)

    for p in predictions.values:
        count = p[0]
        conf  = p[1]
        binNum = int(round(conf/10.0)*10)
        confBins[binNum] += count
    return confBins
```

In [4]:
```
def makeLables(mid):
```

```
    if mid == 0:
        return "{}-{}".format(mid, mid+5)
    elif mid == 100:
        return "{}-{}".format(mid-5, mid)
    else:
        return "{}-{}".format(mid-5, mid+5)
```

In [5]:
```
def sqError(target, actual):
    if actual == 0 or target == 0:
        return 0
    return (target-actual) ** 2
```

In [6]:
```
def signedSqError(target, actual):
    dif = target-actual
    sign = 1 if dif<0 else -1
    return sqError(target, actual) * sign
```

In [7]:
```
def pc_col(col1, col2):
        if col2==0:
            return 0
        else:
            return col1 / col2
```

In [8]:
```
def binTimes(delta):
    """
produces a dict with the bin number and the name
0: postHoc
1: simultanious
2: day
3: week
4: month
5: year
6: fiveYear
7: tenYear
8: FiftyYear
9: overFiftyYear
"""
    if type(delta) != datetime.timedelta:
        print delta, "not a datetime.timedelta"
    outlookClassification = {'bin':0, 'bin_name':''}

    if delta < datetime.timedelta(days=0):
        outlookClassification['bin'] = 0
        outlookClassification['bin_name'] = "    postHoc"

    elif delta == datetime.timedelta(days=0):
        outlookClassification['bin'] = 1
        outlookClassification['bin_name'] = "simultanious"

    elif delta < datetime.timedelta(days=1):
        outlookClassification['bin'] = 2
        outlookClassification['bin_name'] = "        day"

    elif delta < datetime.timedelta(days=7):
        outlookClassification['bin'] = 3
        outlookClassification['bin_name'] = "       week"
```

```
    elif delta < datetime.timedelta(days=30):
        outlookClassification['bin'] = 4
        outlookClassification['bin_name'] = "      month"

    elif delta < datetime.timedelta(days=365):
        outlookClassification['bin'] = 5
        outlookClassification['bin_name'] = "       year"

    elif delta < datetime.timedelta(days=365*5):
        outlookClassification['bin'] = 6
        outlookClassification['bin_name'] = "   fiveYear"

    elif delta < datetime.timedelta(days=365*10):
        outlookClassification['bin'] = 7
        outlookClassification['bin_name'] = "    tenYear"

    elif delta < datetime.timedelta(days=365*50):
        outlookClassification['bin'] = 8
        outlookClassification['bin_name'] = "  FiftyYear"

    else:
        outlookClassification['bin'] = 9
        outlookClassification['bin_name'] = "overFiftyYear"

    return outlookClassification
```

In [9]:

```
def times_are_legit(thing_that_should_be_a_time):
    """This checks the type. It seems that very long predictions aren't pandas dates,
       they default to python dates, so you need to check for both."""
    isPD = type(thing_that_should_be_a_time) == pd.tslib.Timestamp
    isDS = type(thing_that_should_be_a_time) == datetime.datetime
    return isPD or isDS
```

In [10]:

```
def timeDelta(created, deadline):
    if times_are_legit(created) and times_are_legit(deadline): #check that the inputs are well formed
        return  binTimes(deadline - created)
```

In [11]:

```
def make_prediction_count_profile(this_user = 500):
    userQ = "WHERE predictions.creator_id = '{}'".format(this_user)
    query = """select * from  mysql.predictions {}""".format(userQ)
    predictions = queryAsTable(query, maxrows=0)
    #this all feels very ugly, especially the part where I make a coumn, split it and then delete it :(
    pair = zip(predictions["predictions.created_at"], predictions["predictions.deadline"])

    predictions["outlook"]          = [timeDelta(x[0],x[1]) for x in pair]
    predictions["outlook_bin"]      = [x["bin"] for x in predictions["outlook"]]
    predictions["outlook_bin_name"] = [x["bin_name"] for x in predictions["outlook"]]
    firstPredDate = predictions["predictions.created_at"].min()
    predictions["time_since_start"] = [x-firstPredDate for x in predictions["predictions.created_at"]]
    predictions["seconds_since_start"] = [x.total_seconds() for x in predictions["time_since_start"]]

    predictions = predictions.drop(["outlook","predictions.uuid"],1)

    desc = predictions["outlook_bin"].describe()

    prediction_count_profile = dict(predictions["outlook_bin_name"].value_counts().to_dict(), **desc.to_dict())
```

```
    pcp = {}
    for k in prediction_count_profile:
        pcp["prediction_count_profile_"+k.strip()] = prediction_count_profile[k]

    timeBins = "postHoc","simultanious","day","week","month","year","fiveYear","tenYear","FiftyYear","overFiftyYear","fakeBin"
    counts = []
    for tbin in timeBins:
        hopefulKey = 'prediction_count_profile_'+tbin
        if hopefulKey in pcp:
            counts.append( pcp[hopefulKey])

    total = np.sum(counts)

    for tbin in timeBins:
        hopefulKey = 'prediction_count_profile_'+tbin
        if hopefulKey in pcp:
            pcp[hopefulKey+"_pc"] = pcp[hopefulKey]/total


    return pcp
#make_prediction_count_profile()
```

In [12]:
```
def summarise_conf_profile(cp):
        cps = {}
        for b in range(11):
            numLabel = str(cp["Confidence"][b])
            cps[numLabel + "_truePC"] = cp["truePC"][b]
            cps[numLabel + "_falsePC"] = cp["falsePC"][b]

            cps[numLabel + "_sqErrorPC"] = cp["sqError"][b]
            cps[numLabel + "_signedSqErrorPC"] = cp["signedSqError"][b]
        return cps
```

In [13]:
```
def tidyDate(x):
    """
    basically, *fuck you* to whoever wrote these 3 date time objects in a way that isn't compatible.
    This takes datetime objects in any format and returns a numpy np.datetime64, or at least it is supposed to!
    """
    if type(x) == np.datetime64:
        return x#.astype(datetime)
    elif type(x) == pd.tslib.Timestamp:
        return x
    elif type(x) == datetime.datetime:
        return np.datetime64(x)
```

In [14]:
```
def quantifyUser(this_user = 500):

    columns = [ 'j.outcome',                                                          #from judgements
                #'p.withdrawn',                                                        #from predictions
                'r.confidence', 'r.created_at', 'r.id', 'r.prediction_id', 'r.user_id'] #from responses
    columns = ", ".join(columns) #make the array into a string

    query = """
        SELECT myp.confidence        AS conf,
```

```
            Count(myp.confidence) AS cnt
    FROM    (SELECT {0}
             FROM   mysql.responses r
                    LEFT OUTER JOIN mysql.judgements j
                                 ON r.prediction_id = j.prediction_id
             WHERE  j.outcome IS NOT NULL AND
                    r.confidence IS NOT NULL AND
                    r.user_id = '{1}' AND
                    j.outcome = '{2}'
    ) AS myp
    GROUP BY myp.confidence
    ORDER BY myp.confidence
"""

#Q is this really a good way to do this? Can sql do this for me better?
trueP  = queryAsTable(query.format(columns, this_user, 1), maxrows=0, how=2)
falseP = queryAsTable(query.format(columns, this_user, 0), maxrows=0, how=2)

trueConfBins  = binConfidence(trueP)
falseConfBins = binConfidence(falseP)

trueDF = pd.DataFrame(trueConfBins.items(), columns=['Confidence', 'True_Prediction_count']).sort(columns="Confidence")
falseDF = pd.DataFrame(falseConfBins.items(), columns=['Confidence', 'False_Prediction_count']).sort(columns="Confidence")

#merge the true anf false data frames on the confidence bins
conf_profile = pd.merge(trueDF, falseDF, on="Confidence")
#make a new column that gives the total number of predictions in that bracket
conf_profile["PredictionCountTotal"] = conf_profile["True_Prediction_count"] + conf_profile["False_Prediction_count"]
#make a new column that gives the % of predictions that came true at in that bracket
conf_profile["truePC"]  = [pc_col(x[0],x[1]) for x in zip(conf_profile["True_Prediction_count"],  conf_profile["PredictionCountTotal"])]
conf_profile["falsePC"] = [pc_col(x[0],x[1]) for x in zip(conf_profile["False_Prediction_count"],  conf_profile["PredictionCountTotal"])] # not needed, but used in sanity check

#make labels for the x axis
conf_profile["ConfidenceIntervals"] = conf_profile["Confidence"].map(lambda x: makeLables(x))
#sanity check
conf_profile["check"] = conf_profile["truePC"] + conf_profile["falsePC"]

pair = zip(conf_profile["Confidence"],conf_profile["truePC"])
conf_profile["sqError"]       = [      sqError(x[0]/100,x[1]) for x in pair]
conf_profile["signedSqError"] = [signedSqError(x[0]/100,x[1]) for x in pair]



"""
calibrationPlot = plt.figure(1)
plt.subplot(211)

#calibrationPlot = Figure()#(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None)
#Plot the scatter, x is confidences
plt.scatter( conf_profile["Confidence"].map(lambda x: x/10).tolist(),   #vector of y values (currently [0,1,2,3,4,5,6,7,8,9,10])
                  conf_profile.truePC,                                       #vector of x values
                  s=conf_profile.PredictionCountTotal*3)                     #vector of sizes
#draw a diagonal line across the graph
plt.plot([0, 10], [0, 1], 'k-', lw=1)
#set the limits of the graph
pylab.ylim([0,1])
pylab.xlim([0,10])
#add a background grid
plt.grid()
#make the ticks and labels
plt.xticks([x    for x in range(11)], [x           for x in conf_profile["ConfidenceIntervals"].tolist()]) #locations, labels
```

```
    plt.yticks([x/10 for x in range(11)], [str(x)+ "%" for x in conf_profile["Confidence"].tolist()])

    plt.subplot(212)

    plt.plot( conf_profile["Confidence"].map(lambda x: x/10).tolist(),   #vector of y values (currently [0,1,2,3,4,5,6,7,8,9,10])
              conf_profile["PredictionCountTotal"])                       #vector of x values
    """
    #sum of squared errors is an attempt to capture overall calibration
    sqe = sum(conf_profile["sqError"].tolist())
    #sum of signed squared errors is an attempt to capture the direction of calibration. If the result comes up negative then the predictor is generally overconfident.
    ssqe = sum(conf_profile["signedSqError"].tolist())

    #print sqe
    #print ssqe
    #print this_user
    #print sum(conf_profile["PredictionCountTotal"].tolist())
    #print conf_profile

    rtn = {}
    rtn["user"]                     = this_user
    #rtn["calibrationPlot"]          = calibrationPlot
    rtn["summedSquaredError"]       = sqe
    rtn["signedSummedSquaredError"] = ssqe
    rtn["totalPredictions"]         = sum(conf_profile["PredictionCountTotal"].tolist())
    rtn = dict(rtn, **summarise_conf_profile(conf_profile))
    rtn = dict(rtn, **make_prediction_count_profile(this_user))

    return rtn
```

Exploration

tables

This section goes through the tables to get an idea of what's going on inside them

In [15]:

```
queryAsTable("""
SELECT table_name,
       table_rows
FROM   information_schema.tables
WHERE  table_type = 'BASE TABLE'
       AND table_schema = 'mysql'
ORDER  BY table_rows DESC
""", maxrows=10)
```

Out[15]:

	tables.table_name	tables.table_rows
0	responses	45450
1	predictions	22542
2	users	21297
3	judgements	10991

4	help_relation	1047
5	help_topic	511
6	help_keyword	467
7	help_category	40
8	user	7
9	general_log	2

10 rows × 2 columns

So it looks like there's not really any point thinking about any tables other than responses, predictions, users and judgements.

The help tables seem to be mysql help, not predictionbook help!

responses

In [16]:
```
queryAsTable("""select * from  mysql.responses""", maxrows=10)
```

Out[16]:

	responses.confidence	responses.created_at	responses.id	responses.prediction_id	responses.updated_at	responses.user_id
0	80	2008-06-20 03:46:16	1	1	2008-08-01 09:18:02	1
1	60	2008-06-20 04:37:13	2	2	2008-08-01 09:18:02	2
2	80	2008-06-20 04:38:35	3	3	2008-08-01 09:18:02	3
3	80	2008-06-20 04:40:06	4	4	2008-08-01 09:18:02	3
4	70	2008-06-20 05:08:31	5	5	2008-08-01 09:18:02	4
5	70	2008-06-20 05:24:24	6	6	2008-08-01 09:18:02	2
6	90	2008-06-20 06:46:23	7	7	2008-08-01 09:18:02	5
7	80	2008-06-20 11:43:07	8	8	2008-08-01 09:18:02	6
8	95	2008-06-20 11:47:54	9	9	2008-08-01 09:18:02	6
9	78	2008-06-20 12:12:28	10	10	2008-08-01 09:18:02	7

10 rows × 6 columns

predictions

In [17]:
```
queryAsTable("""select * from  mysql.predictions""", maxrows=10)
```

Out[17]:

	predictions.created_at	predictions.creator_id	predictions.deadline	predictions.id	predictions.private	predictions.updated_at	predictions.uuid	predictions.version	predictions.withdrawn
0	2008-06-20 03:46:16	1	2008-06-20 12:00:00	1	0	2008-08-14 02:28:40	d52eeaf6-cb20-456f-a375-bec2d1acc44d	1	0
1	2008-06-20 04:37:13	2	2008-06-23 12:00:00	2	0	2008-08-01 09:18:01	86530ca2-fbd5-47a8-b2d8-12110e05c3b4	1	0
2	2008-06-20 04:38:35	3	2008-06-30 12:00:00	3	0	2008-08-01 09:18:01	d5adb0ee-ecf2-48ce-88ce-7b40b5b2179f	1	0

3	2008-06-20 04:40:06	3	2008-06-19 12:00:00	4	0	2008-08-01 09:18:01	100f4ff6-4ee7-44e8-aa2d-5394dcaead7e	1	0
4	2008-06-20 05:08:31	4	2008-06-20 06:08:31	5	0	2008-08-01 09:18:01	c0314cd4-070a-4d4b-ae3c-ec03ddbdc621	1	0
5	2008-06-20 05:24:24	2	2008-06-26 12:00:00	6	0	2008-08-01 09:18:01	5c73114a-e22d-4a5a-9026-03f45acc0f19	1	0
6	2008-06-20 06:46:23	5	2008-12-30 12:00:00	7	0	2008-08-01 09:18:01	54df1626-bf7a-4ae9-b162-826d2a67fec1	1	0
7	2008-06-20 11:43:07	6	2008-08-01 12:00:00	8	0	2008-08-01 09:18:01	d5fbc343-9de5-41a2-ba89-b599bf60ae81	1	0
8	2008-06-20 11:47:54	6	2013-01-06 12:00:00	9	1	2013-01-26 01:15:36	ffa789d4-c40e-489e-af69-7000ce34cbf0	2	0
9	2008-06-20 12:12:28	7	2013-06-20 22:00:00	10	1	2013-11-01 18:38:07	19128b35-38ac-4748-9a44-42cbda3309fb	2	0

10 rows × 9 columns

Users

In [18]:
```
queryAsTable("""select * from  mysql.users""", maxrows=10)
```

Out[18]:

	users.created_at	users.id	users.private_default	users.timezone	users.updated_at
0	2008-08-01 09:18:00	1	0	None	2013-08-28 06:34:18
1	2008-08-01 09:18:00	2	0	None	2012-10-12 01:52:14
2	2008-08-01 09:18:00	3	0	None	2008-08-01 09:18:00
3	2008-08-01 09:18:00	4	0	London	2013-01-11 01:47:45
4	2008-08-01 09:18:00	5	0	None	2008-08-01 09:18:00
5	2008-08-01 09:18:00	6	0	Melbourne	2008-09-08 05:58:02
6	2008-08-01 09:18:00	7	0	None	2013-06-06 02:00:49
7	2008-08-01 09:18:00	8	0	Melbourne	2009-05-13 06:49:42
8	2008-08-01 09:18:00	9	0	Melbourne	2013-03-15 04:21:33
9	2008-08-01 09:18:00	10	0	None	2008-08-01 09:18:00

10 rows × 5 columns

In [19]:
```
queryAsTable("""select * from  mysql.judgements""", maxrows=10)
```

Out[19]:

	judgements.created_at	judgements.id	judgements.outcome	judgements.prediction_id	judgements.updated_at	judgements.user_id
0	2008-08-14 02:28:40	1	1	1	2008-08-14 02:28:40	None
1	2008-08-01 09:18:01	2	0	2	2008-08-01 09:18:01	None
2	2008-08-01 09:18:01	3	0	3	2008-08-01 09:18:01	None
3	2008-08-01 09:18:01	4	0	4	2008-08-01 09:18:01	None
4	2008-08-01 09:18:01	5	1	5	2008-08-01 09:18:01	None

5	2008-08-01 09:18:01	6	0	6	2008-08-01 09:18:01	None
6	2008-08-01 09:18:01	8	0	8	2008-08-01 09:18:01	None
7	2008-08-01 09:18:01	12	0	12	2008-08-01 09:18:01	None
8	2008-08-01 09:18:01	13	0	15	2008-08-01 09:18:01	None
9	2008-08-01 09:18:01	14	1	16	2008-08-01 09:18:01	None

10 rows × 6 columns

So the general structure of this seems to be that a **user** makes a **prediction**. The prediction is just the *what* part of the **prediction**, i.e. the text, e.g. '*Newtown jets will win the superbowl*'. At the same time the user makes a **response** to that **prediction**. The **response** is the part where they assign a *confidence* in it.

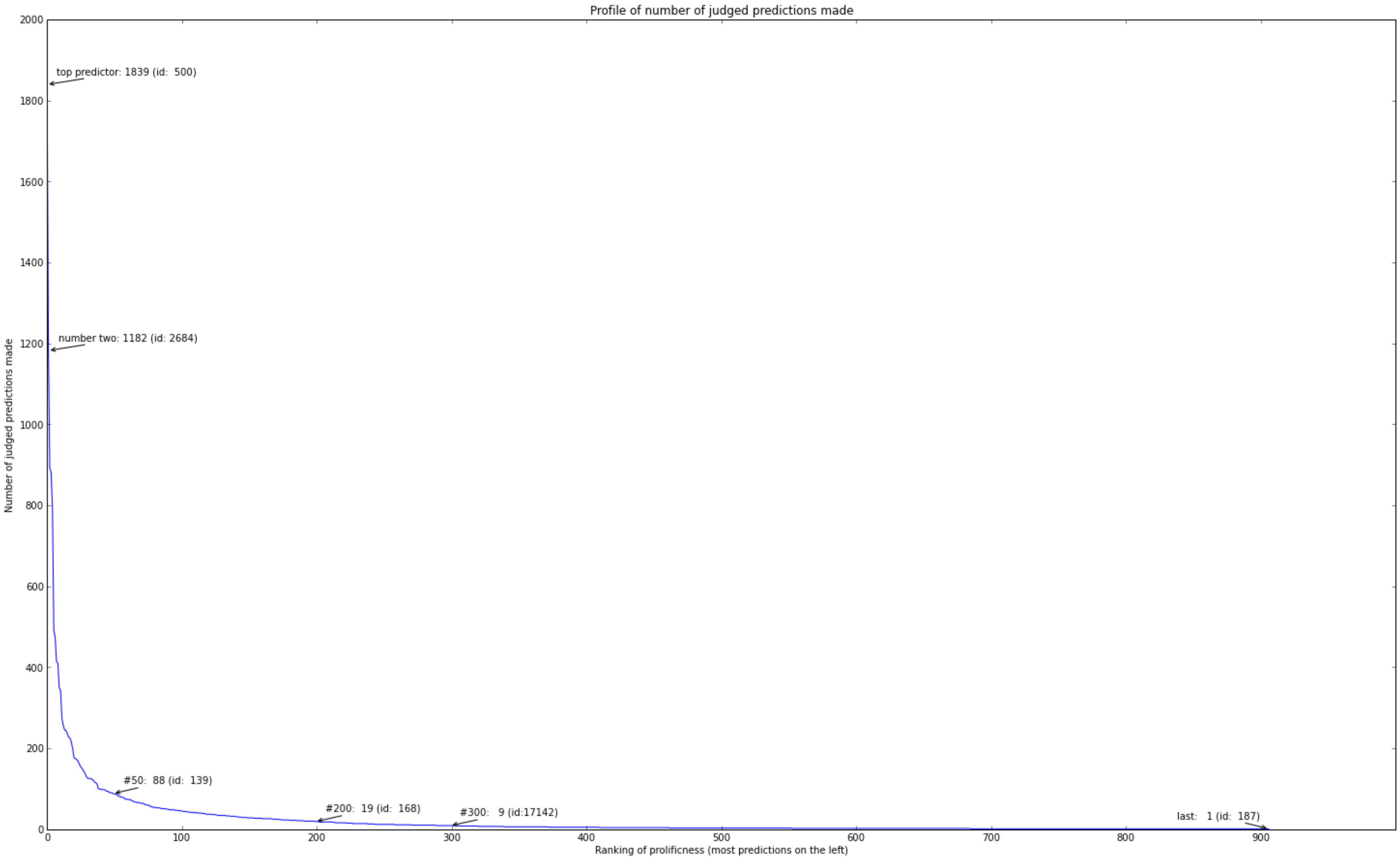
Each **prediction** can have many **response**s.

A **prediction** can later be judged, [TODO: check if each prediction only hasone or can hasmany judgements? The site records many judgements, so my guess is that it post processes them at render time; meaning that I'll need to search for the most recent one.]

Before we get into trying to merge these, lets see how many users there are, and how many predictions they've made:

In [20]:

```
#we know that user 500 is the most prolific, so we can use them to set some boundaries on the graph
u = quantifyUser(500)

#This query counds the number of judged predictions each user has made
countPredictions_query = """
SELECT r.user_id,
       Count(r.user_id)
FROM   mysql.responses r
       LEFT OUTER JOIN mysql.judgements j
                    ON r.prediction_id = j.prediction_id
WHERE  j.outcome {0} NULL
       AND r.confidence {1} NULL
GROUP  BY r.user_id
ORDER  BY Count(r.user_id) DESC
"""
count_judged_prediction_df = queryAsTable(countPredictions_query.format('IS NOT','IS NOT'), maxrows=0)
response_counts = count_judged_prediction_df.drop("r.user_id",1)

fig, ax = plt.subplots(figsize=(25,15), dpi=100)
ax.plot(response_counts)
yticks(range(0,u["totalPredictions"]+200,200))
xticks(range(0,1000,100))
arrow = dict(arrowstyle='->', shrinkA=0)
ax.annotate("top predictor: {:>4} (id:{:>5})".format( count_judged_prediction_df["Count(r.user_id)"][0],   count_judged_prediction_df.iloc[0]['r.user_id']  ), xy=(0,   count_judged_
ax.annotate("number two: {:>4} (id:{:>5})".format(    count_judged_prediction_df["Count(r.user_id)"][1],   count_judged_prediction_df.iloc[1]['r.user_id']  ), xy=(1,   count_judged_
ax.annotate("#50:{:>4} (id:{:>5})".format(            count_judged_prediction_df["Count(r.user_id)"][49],  count_judged_prediction_df.iloc[49]['r.user_id'] ), xy=(49,  count_judged_
ax.annotate("#200:{:>4} (id:{:>5})".format(           count_judged_prediction_df["Count(r.user_id)"][199], count_judged_prediction_df.iloc[199]['r.user_id']), xy=(199, count_judged_
ax.annotate("#300:{:>4} (id:{:>5})".format(           count_judged_prediction_df["Count(r.user_id)"][299], count_judged_prediction_df.iloc[299]['r.user_id']), xy=(299, count_judged_
last=len(count_judged_prediction_df['r.user_id'])-1
ax.annotate("last:{:>4} (id:{:>5})".format(           count_judged_prediction_df["Count(r.user_id)"][last], count_judged_prediction_df.iloc[last]['r.user_id']), xy=(last, count_judg

plt.xlabel('Ranking of prolificness (most predictions on the left)')
plt.ylabel('Number of judged predictions made')
plt.title('Profile of number of judged predictions made')

plt.show()
```



This graph shows how fast the number of people who have made predictions drops off.

Below is where we pluck people from that complete dataset to work with. The value of people who have made less than 20 predictions is going to be pretty low, so lets cut it off at the top 200 users.

In [21]:
```
set_size = 200
```

```
peopleToLookAt = []
for i in range(set_size):
    peopleToLookAt.append( count_judged_prediction_df.iloc[i]['r.user_id'])
print peopleToLookAt
```

```
[500, 2684, 535, 642, 4758, 2678, 579, 292, 9112, 2649, 496, 560, 140, 247, 591, 557, 9, 16489, 18514, 2504, 85, 205, 1, 2632, 2499, 94, 559, 16970, 15788, 1434, 667, 164, 946,
4611, 12401, 3954, 4600, 2939, 136, 7148, 905, 131, 3876, 16304, 554, 16898, 135, 527, 98, 139, 641, 137, 17348, 143, 6036, 1285, 1293, 149, 1074, 2982, 33, 21058, 2038, 2584,
6820, 16658, 903, 2, 93, 586, 88, 7259, 5319, 6826, 16175, 17549, 15515, 19238, 4707, 3553, 1521, 92, 188, 15135, 2576, 4797, 4, 668, 6, 216, 4713, 11118, 183, 197, 1333, 90, 210,
6608, 62, 18806, 369, 42, 147, 12343, 20547, 12367, 1083, 17955, 74, 585, 16545, 19596, 1468, 1177, 173, 598, 2663, 200, 181, 19, 6729, 142, 358, 6736, 7, 4752, 170, 2546, 227,
190, 119, 582, 1590, 590, 101, 9254, 16302, 154, 3965, 184, 542, 599, 174, 6977, 491, 4396, 18978, 163, 18651, 2178, 14302, 9214, 215, 18291, 982, 9666, 3879, 175, 7352, 478, 5286,
19500, 2761, 630, 595, 4598, 530, 607, 573, 13309, 2321, 290, 15279, 2124, 9852, 603, 6776, 644, 5219, 658, 391, 10399, 652, 16430, 3840, 656, 3908, 10122, 18477, 15016, 5537, 249,
9296, 605, 6688, 217, 17858, 393, 63, 168]
```

The `quantifyUser` function chews through the people in the set produced above, and tries to make a row for the Grand Dataset Of Everything out of them. Some fail for no good reason (that I've found), so they get ignored for the moment.

In [22]:
```
#u
```

In [23]:
```
#check the types
#for k in u:
#    print type(u[k]),k,u[k]
```

In [24]:
```
quantified_users=[]
counter=0
error_people = []
for id in peopleToLookAt:
    try:
        person = quantifyUser(id)
        quantified_users.append(person)
    except:
        error_people.append( id)
print  error_people,"caused errors"
```

```
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
/usr/local/lib/python2.7/dist-packages/pandas/compat/scipy.py:68: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  score = values[idx]
```

In [25]:
```
len(quantified_users) #this is how many people made it through unscathed
```

Out[25]:
```
178
```

At this point we have a massive wide dataset, with lots of rows. They get less useful as we go down, but it might be more useful than just having 45 fairly OK ones.

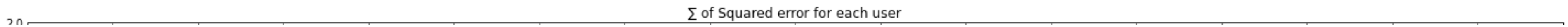
In [26]:
```
qudf = pd.DataFrame(quantified_users) #convert the list of dictionaries into a pandas dataframe
```

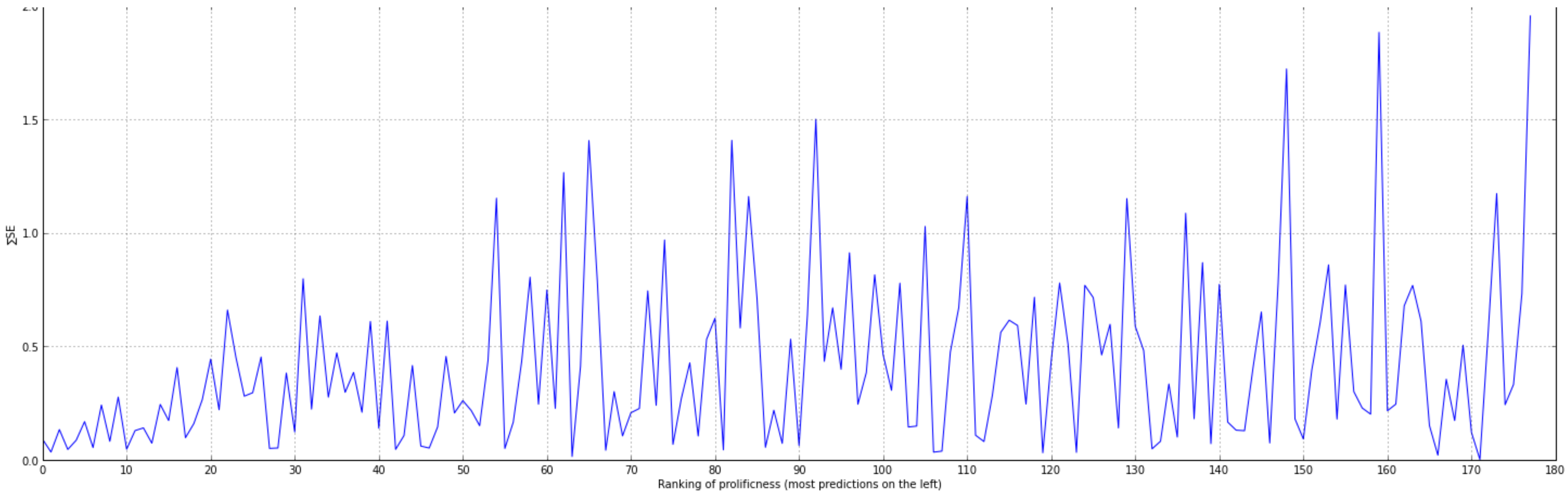
In [27]:
```
fig = plt.figure(figsize=(25,7.5), dpi=100)
plot(qudf[ 'signedSummedSquaredError'])
grid(True)
xticks(range(0,len(quantified_users)+10,10))
plt.xlabel('Ranking of prolificness (most predictions on the left)')
plt.ylabel(u'signed ∑SE')
plt.title(u'Signed ∑ of Squared error for each user')
show()
#it would be worth ploting these two with number of predictions on the x axis. it would probably bunch them up way too much, but maybe with a log transform it would be useful
```

There seems to be a bit of a funnel away from the top predictors. The most enthusiastic predictors are the best calibrated, although I don't know how to disentangle this from just the law of large numbers (or if this has any effect at all!)

In general it seems that the top 30 or so predictors are below the 0 line, whereas in general the rest of the pack is above it. I need to check this, but I think that below is generally underconfident, and above, generally overconfident.

In [28]:
```
fig = plt.figure(figsize=(25,7.5), dpi=100)
plot(qudf[ 'summedSquaredError'])
grid(True)
xticks(range(0,len(quantified_users)+10,10))
plt.xlabel('Ranking of prolificness (most predictions on the left)')
plt.ylabel(u'∑SE')
plt.title(u'∑ of Squared error for each user')
show()
```





The range of the SSE seems to be pretty constant through the middle section of the graph. I think it goes down again when we get to about 200 because people at that point have made so few prediction that there's not much to go on by then. (Remember predictor #100 had only made 46 predictions!)

To perform a k means clustering on this dataset it needs to be cleaned up a little bit. There are some missing values where there was no count (e.g. no predictions at 100%, or none with a >50 year outlook). The data also needs to be scaled so that big numbers like the prediction count don't swamp small ones like the prediction percentages.

In [29]:
```
qudf = qudf.fillna(0) #put a 0 in all the empty spots as they come about through there being nothing to count
qudf.to_csv("./users.csv")
```

In [30]:
```
#qudf["prediction_count_profile_FiftyYear"] #check that there are no nan values
```

In [31]:
```
qudf_scaled = preprocessing.scale(qudf)
numpy.savetxt("./qudf_scaled.csv", qudf_scaled, delimiter=",") #qudf_scaled.to_csv("./scaled_users.csv") #error: 'numpy.ndarray' object has no attribute 'to_csv'
type(qudf_scaled)
```

Out[31]:
```
numpy.ndarray
```

Computing K-Means with K = n (n clusters)

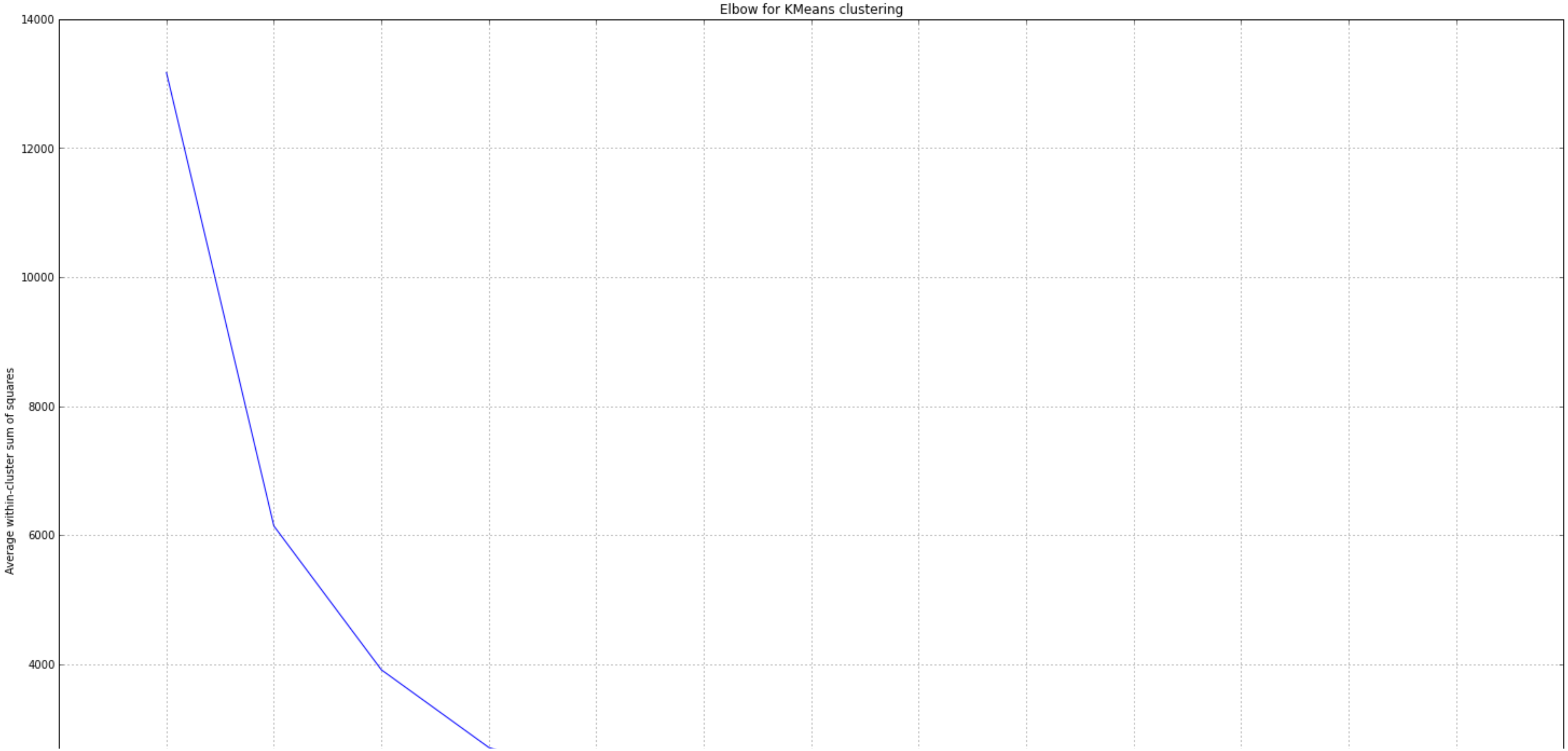
Trying it with `clusterRange` clusters, i.e. with 1 cluster, then 2... and then graphing the within cluster errors at the end to see where the value of adding more clusters drops off.

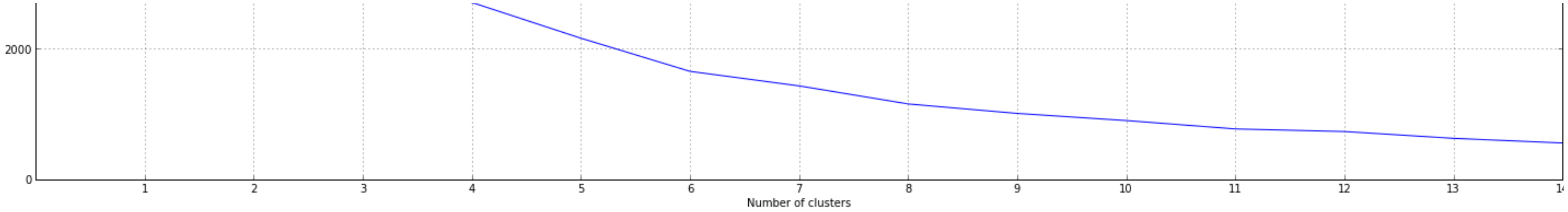
In [32]:
```
clusterRange = range(1,15)
msse4k = []
for k in clusterRange:
    centroids,junk = kmeans(qudf_scaled,k)
    idx,dist = vq(qudf_scaled,centroids)
    #idx #vector assigning people to a given cluster
    #dist #The distortion (distance) between the observation and its nearest code.
```

```
    clusterStat = [[] for x in range(k)] # make a list of empty lists e.g. [[], [], [], [], [], [], [], []]

    for distance,index in zip(dist,idx):
        clusterStat[index].append(distance**2)

    sums    = [sum(x) for x in clusterStat]
    average = np.mean(sums)
    msse4k.append(average)
```

In [33]:

```
plt.figure(figsize=(25,15), dpi=100)
plt.plot(clusterRange,msse4k)
plt.xticks(clusterRange)
plt.grid(True)
plt.xlabel('Number of clusters')
plt.ylabel('Average within-cluster sum of squares')
plt.title('Elbow for KMeans clustering')
plt.show()
```





It looks like most of the usefullness has gone out of the clustering by the time we get to 5 clusters.

Oddly the elbow isn't as pronounced as it is in the examples I've seen. I wonder if the high dimenntionality of the dataset is what is causing the rounded elbow?

So doing a clustering with 5 clusters we get a vector of cluster indexes. They aren't scaled so if we glue them onto the end of the scaled data it'll throw it out of whack, so I'm going to put them onto the end of the quantified user data frame (`qudf`) and then rescale it.

In [34]:
```
k=5
centroids,junk = kmeans(qudf_scaled,k)
idx,dist = vq(qudf_scaled,centroids)

idx
```

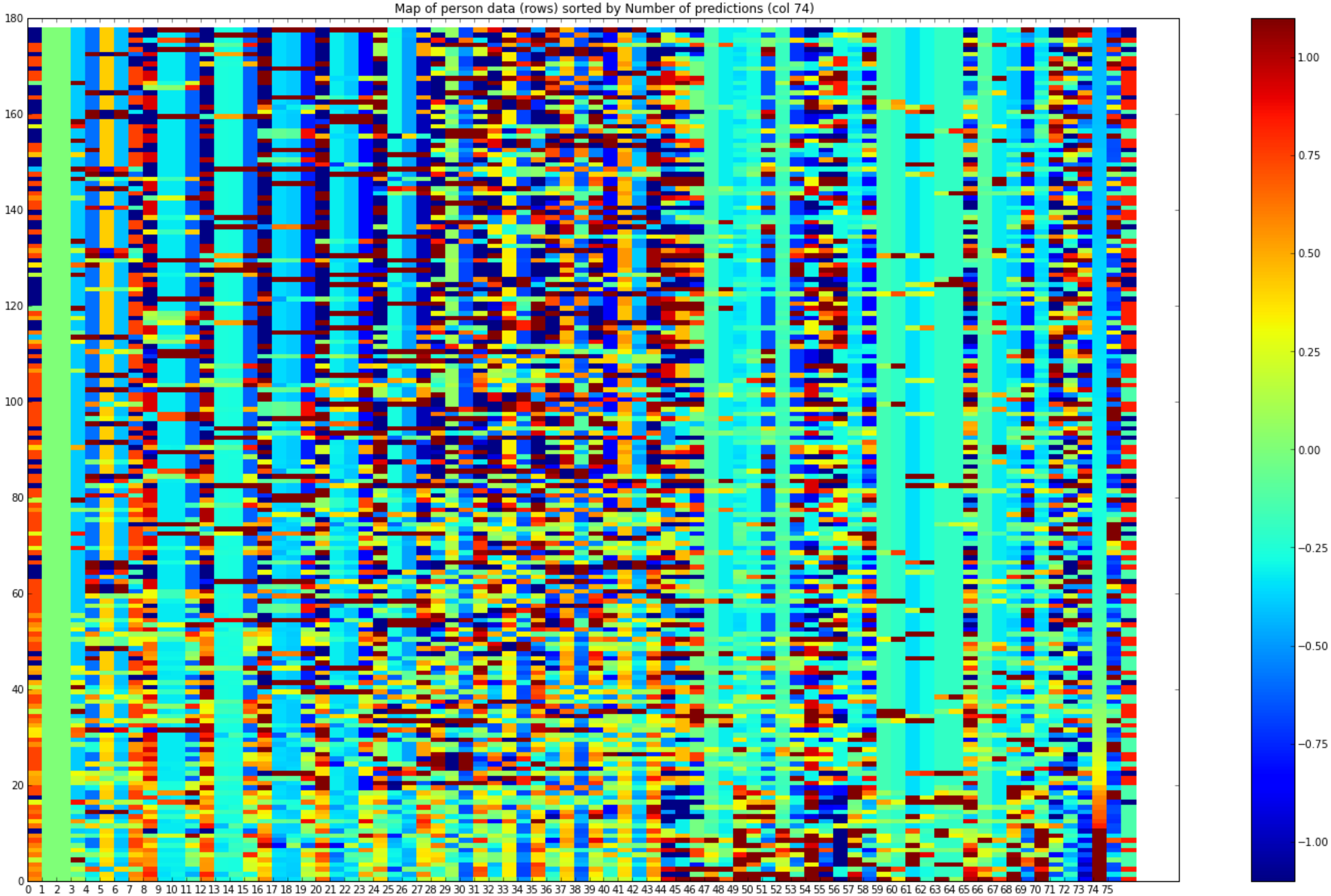
Out[34]:
```
array([4, 3, 4, 3, 3, 3, 4, 3, 4, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 4, 4, 2,
       4, 3, 4, 4, 3, 3, 3, 4, 3, 4, 3, 4, 4, 4, 3, 3, 4, 4, 2, 3, 4, 3, 3,
       3, 4, 3, 4, 4, 3, 3, 4, 1, 3, 3, 3, 4, 3, 4, 4, 1, 3, 3, 4, 3, 4, 3,
       4, 4, 3, 1, 3, 0, 4, 3, 4, 4, 2, 3, 4, 1, 4, 2, 3, 4, 4, 3, 4, 4, 3,
       1, 3, 1, 3, 2, 3, 4, 4, 3, 3, 3, 4, 3, 4, 4, 4, 3, 3, 0, 4, 4, 4, 2,
       3, 4, 4, 4, 4, 4, 0, 3, 4, 2, 2, 4, 1, 4, 0, 4, 4, 3, 4, 3, 3, 1, 3,
       4, 4, 3, 4, 4, 3, 3, 4, 4, 3, 1, 3, 4, 3, 2, 4, 4, 3, 3, 4, 4, 0, 3,
       4, 4, 2, 4, 4, 4, 4, 4, 2, 4, 4, 3, 0, 3, 4, 1, 0])
```

In [35]:
```
qudf_wID = qudf
qudf_wID["clusterID"]=idx

qudf.to_csv("./users.csv")

qudf_scaled_wID = preprocessing.scale(qudf_wID)
```

↞ predictors in this clustering, coloured by cluster, ordered by number of predictions.

This image, grabbed after doing some manual data fiddling in a spreadsheet seems to say that there is *some* correlation between the number of predictions a person has made, and their likelyhood of being in cluster 2 (seemingly for frequent predictors) or cluster 3 (seemingly for infrequent predictors). I think that it would be worth rerunning the clustering without the `totalPredictions` column to see if this tendency still exists.

In [36]:
```
colors = [(cm.jet(i)) for i in xrange(1,256)]
new_map = matplotlib.colors.LinearSegmentedColormap.from_list('new_map', colors, N=256)

fig = plt.figure(figsize=(25,15), dpi=100)
pcolor(qudf_scaled_wID, cmap=new_map, vmin=-1.1, vmax=1.1)
colorbar()
data_width = qudf_scaled_wID.shape[1]-1
xticks(range(0,data_width,1))
title('Map of person data (rows) sorted by Number of predictions (col 74)')
show()
```



0	'0_falsePC'	20	'40_falsePC'	40	'90_falsePC'	60	'prediction_count_profile_overFiftyYear_pc'
1	'0_signedSqErrorPC'	21	'40_signedSqErrorPC'	41	'90_signedSqErrorPC'	61	'prediction_count_profile_postHoc'
2	'0_sqErrorPC'	22	'40_sqErrorPC'	42	'90_sqErrorPC'	62	'prediction_count_profile_postHoc_pc'
3	'0_truePC'	23	'40_truePC'	43	'90_truePC'	63	'prediction_count_profile_simultanious'
4	'100_falsePC'	24	'50_falsePC'	44	'prediction_count_profile_25%'	64	'prediction_count_profile_simultanious_pc'
5	'100_signedSqErrorPC'	25	'50_signedSqErrorPC'	45	'prediction_count_profile_50%'	65	'prediction_count_profile_std'
6	'100_sqErrorPC'	26	'50_sqErrorPC'	46	'prediction_count_profile_75%'	66	'prediction_count_profile_tenYear'
7	'100_truePC'	27	'50_truePC'	47	'prediction_count_profile_FiftyYear'	67	'prediction_count_profile_tenYear_pc'

```
8  | '10_falsePC'          | 28 | '60_falsePC'          | 48 | 'prediction_count_profile_FiftyYear_pc'   | 68 | 'prediction_count_profile_week'         |
9  | '10_signedSqErrorPC'  | 29 | '60_signedSqErrorPC'  | 49 | 'prediction_count_profile_count'          | 69 | 'prediction_count_profile_week_pc'      |
10 | '10_sqErrorPC'        | 30 | '60_sqErrorPC'        | 50 | 'prediction_count_profile_day'            | 70 | 'prediction_count_profile_year'         |
11 | '10_truePC'           | 31 | '60_truePC'           | 51 | 'prediction_count_profile_day_pc'         | 71 | 'prediction_count_profile_year_pc'      |
12 | '20_falsePC'          | 32 | '70_falsePC'          | 52 | 'prediction_count_profile_fiveYear'       | 72 | 'signedSummedSquaredError'              |
13 | '20_signedSqErrorPC'  | 33 | '70_signedSqErrorPC'  | 53 | 'prediction_count_profile_fiveYear_pc'    | 73 | 'summedSquaredError'                    |
14 | '20_sqErrorPC'        | 34 | '70_sqErrorPC'        | 54 | 'prediction_count_profile_max'            | 74 | 'totalPredictions'                      |
15 | '20_truePC'           | 35 | '70_truePC'           | 55 | 'prediction_count_profile_mean'           | 75 | 'user'                                  |
16 | '30_falsePC'          | 36 | '80_falsePC'          | 56 | 'prediction_count_profile_min'            | 76 | 'clusterID'                             |
17 | '30_signedSqErrorPC'  | 37 | '80_signedSqErrorPC'  | 57 | 'prediction_count_profile_month'          |    |                                         |
18 | '30_sqErrorPC'        | 38 | '80_sqErrorPC'        | 58 | 'prediction_count_profile_month_pc'       |    |                                         |
19 | '30_truePC'           | 39 | '80_truePC'           | 59 | 'prediction_count_profile_overFiftyYear'  |    |                                         |
```

This colour map is sorted by virtue of the order that the data was created in; most prolific predictors to least. It seems to get more chaotic towards the bottom, but this might just because the data quality increases, wheras near the top there are a lot of missing values and zeroes.

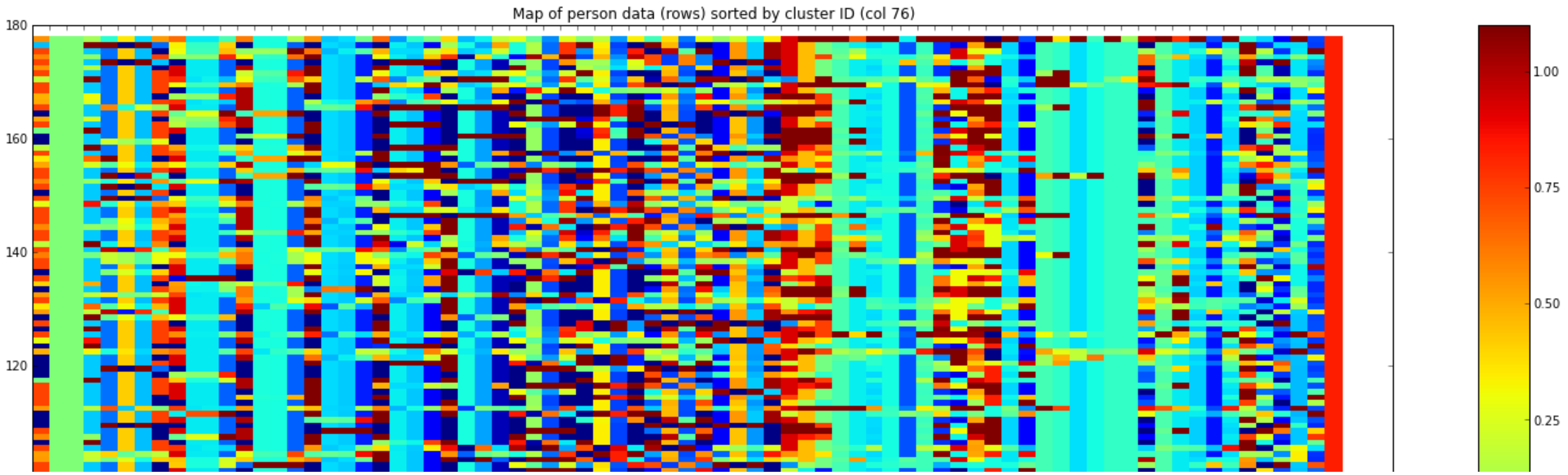
Column 64 is prediction count, it has a smooth gradient from a few reds, then quickly dropping off to a slow change from green to blue. This is the same elbow seen above in the predictions count graph.

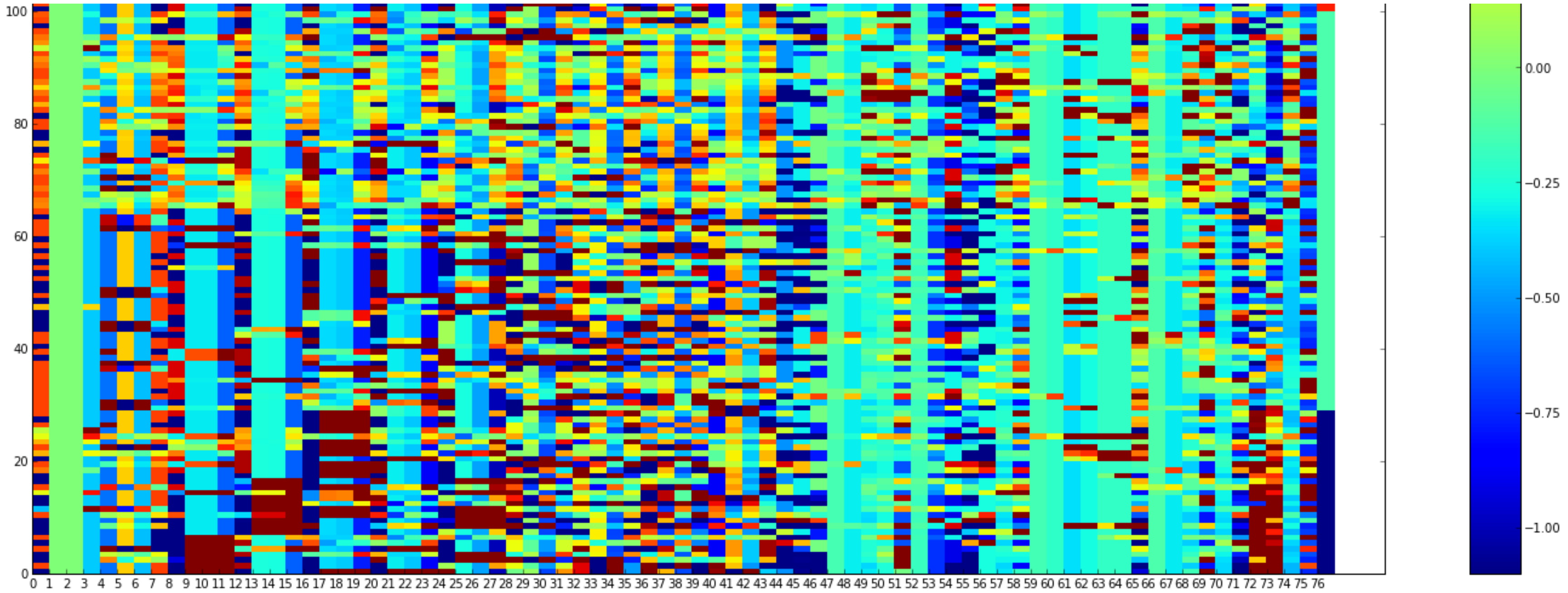
Column 66 is cluster ID, it is almost all red up to person #40 which is where the hockey stick's head starts to turn into it's shaft. There isn't that much grouping of colours in the rest of that bar, so it seems that predicton count is the most powerful factor in this clustering. Realistically this is probably completely legitimate as prolific predictors also seem to be well calibrated, but lets see what happens when we cluster without the prediction count column.

The colour map is drawn in the opposite order to the array, with the first element at the bottom.
`zip(range(0,len(qudf.columns)),qudf.columns)` produces

Here it is again sorted by cluster

In [37]:
```
data_width = qudf_scaled_wID.shape[1]-1
qudf_scaled_wID_sorted = qudf_scaled_wID[qudf_scaled_wID[:,data_width].argsort()]

colors = [(cm.jet(i)) for i in xrange(1,256)]
new_map = matplotlib.colors.LinearSegmentedColormap.from_list('new_map', colors, N=256)

fig = plt.figure(figsize=(25,15), dpi=100)
pcolor(qudf_scaled_wID_sorted, cmap=new_map, vmin=-1.1, vmax=1.1)
colorbar()
data_width = qudf_scaled_wID_sorted.shape[1]
xticks(range(0,data_width,1))
title('Map of person data (rows) sorted by cluster ID (col 76)')
show()
```

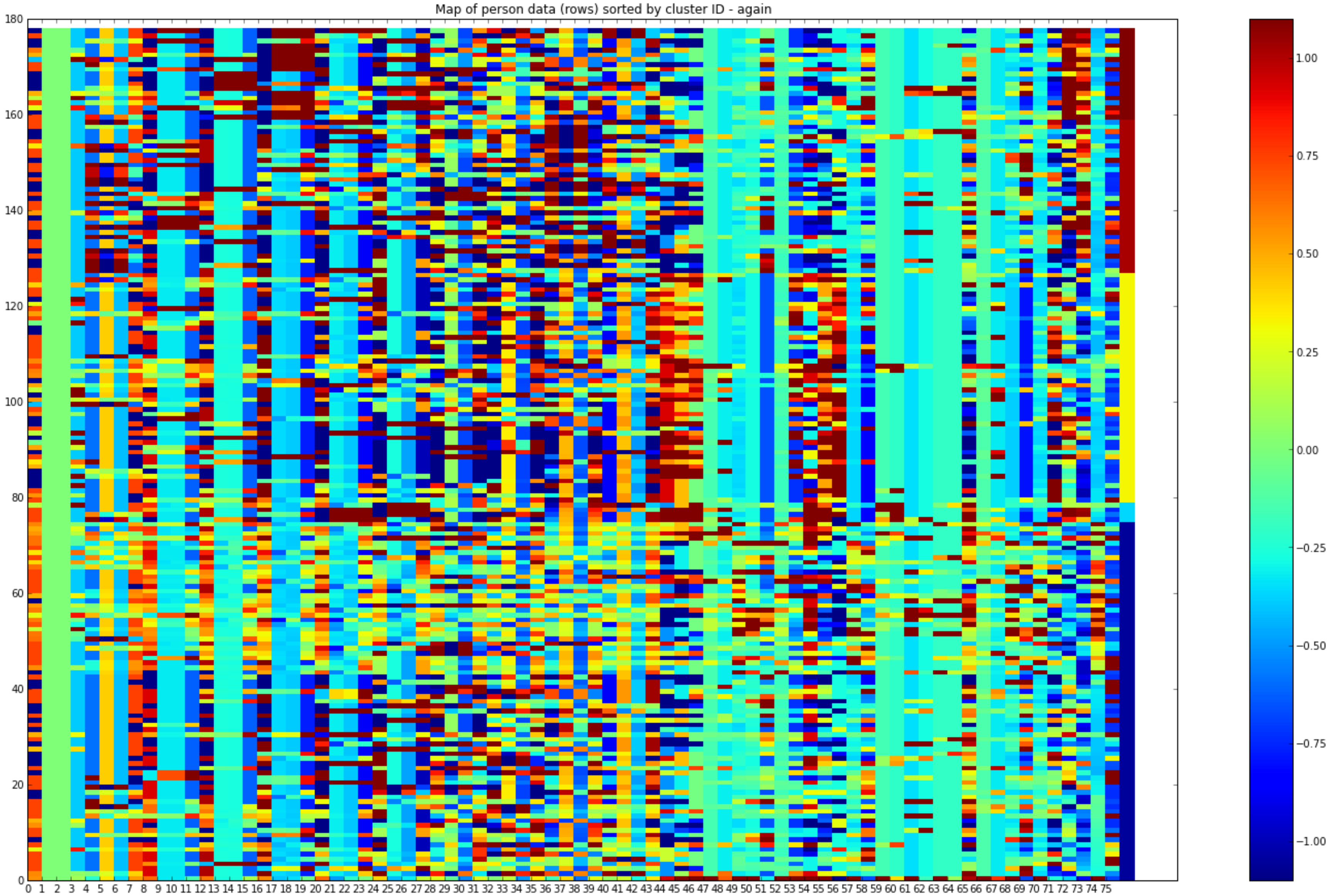




The last column is the cluster ID, and we can see from the blocking that it has sorted it properly. Looking for characteristics in the rest of the table there seems to be a band through the bottom of cluster 0 (the red one). This band seems to correspond to a block of high predictors in column 64. All of these people are in the same cluster.

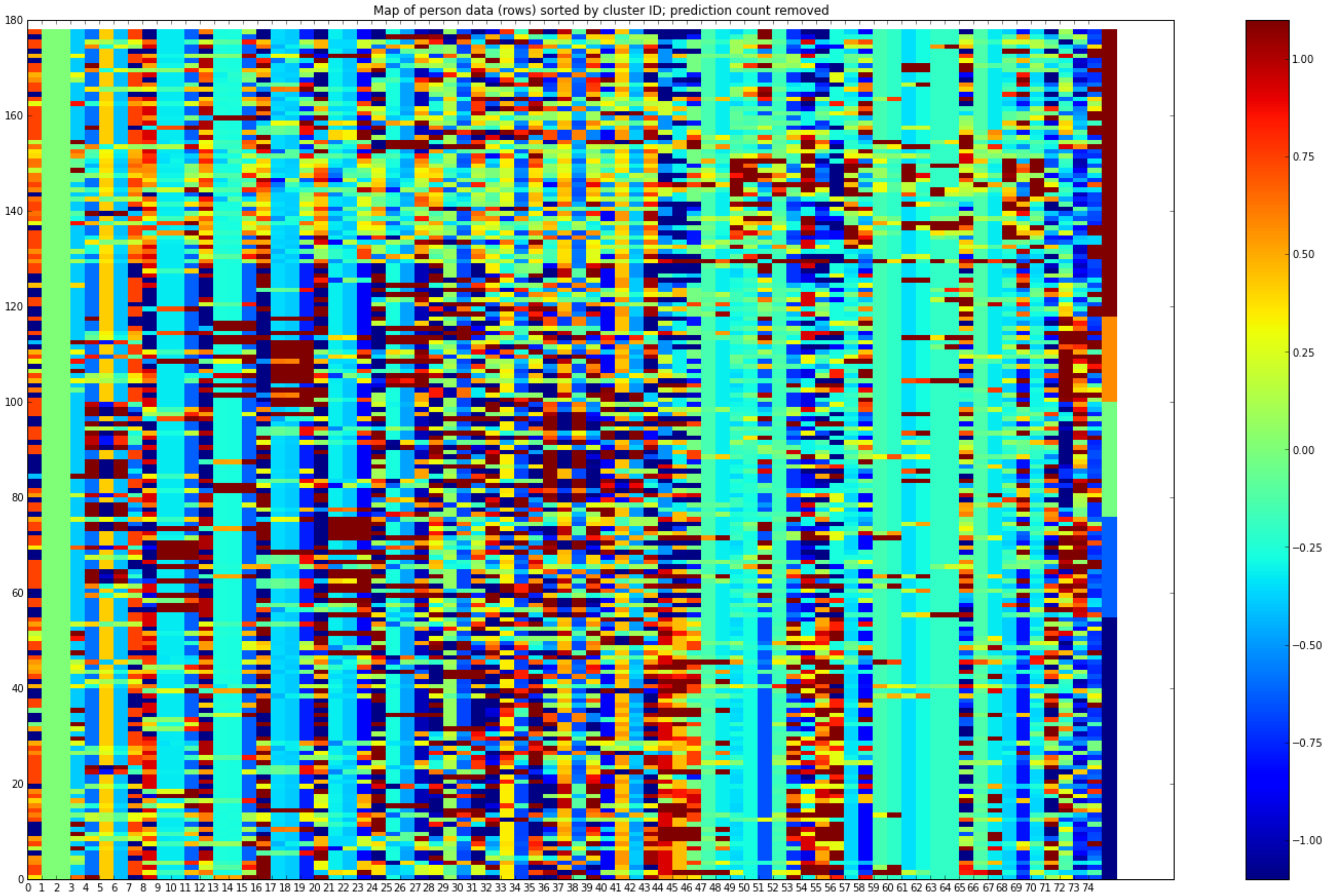
In [38]:
```
def cmap_of_clustering(data, k=5, titleString=""):
    #scale the data
    scaled_data = preprocessing.scale(data)

    #run the clustering
    centroids,junk = kmeans(scaled_data,k)
    idx,dist = vq(scaled_data,centroids)

    #append the cluster IDs to the data
    data["clusterID"] = idx
    #scale the data again not that it has cluster IDs (this feels ineficient)
    scaled_data = preprocessing.scale(data)

    #sort the data by cluster ID
    data_width = scaled_data.shape[1]-1
    sorted_scaled_data = scaled_data[scaled_data[:,data_width].argsort()]

    #make the plot
    colors = [(cm.jet(i)) for i in xrange(1,256)]
    new_map = matplotlib.colors.LinearSegmentedColormap.from_list('new_map', colors, N=256)

    fig = plt.figure(figsize=(25,15), dpi=100)
    pcolor(sorted_scaled_data, cmap=new_map, vmin=-1.1, vmax=1.1)
    colorbar()
    xticks(range(0,data_width,1))
    title(titleString)
    show()
```

In [39]:
```
cmap_of_clustering(qudf, k=5, titleString='Map of person data (rows) sorted by cluster ID - again')
```



Immediatly above is exactly the same data as above, so *should* come out roughly the same.

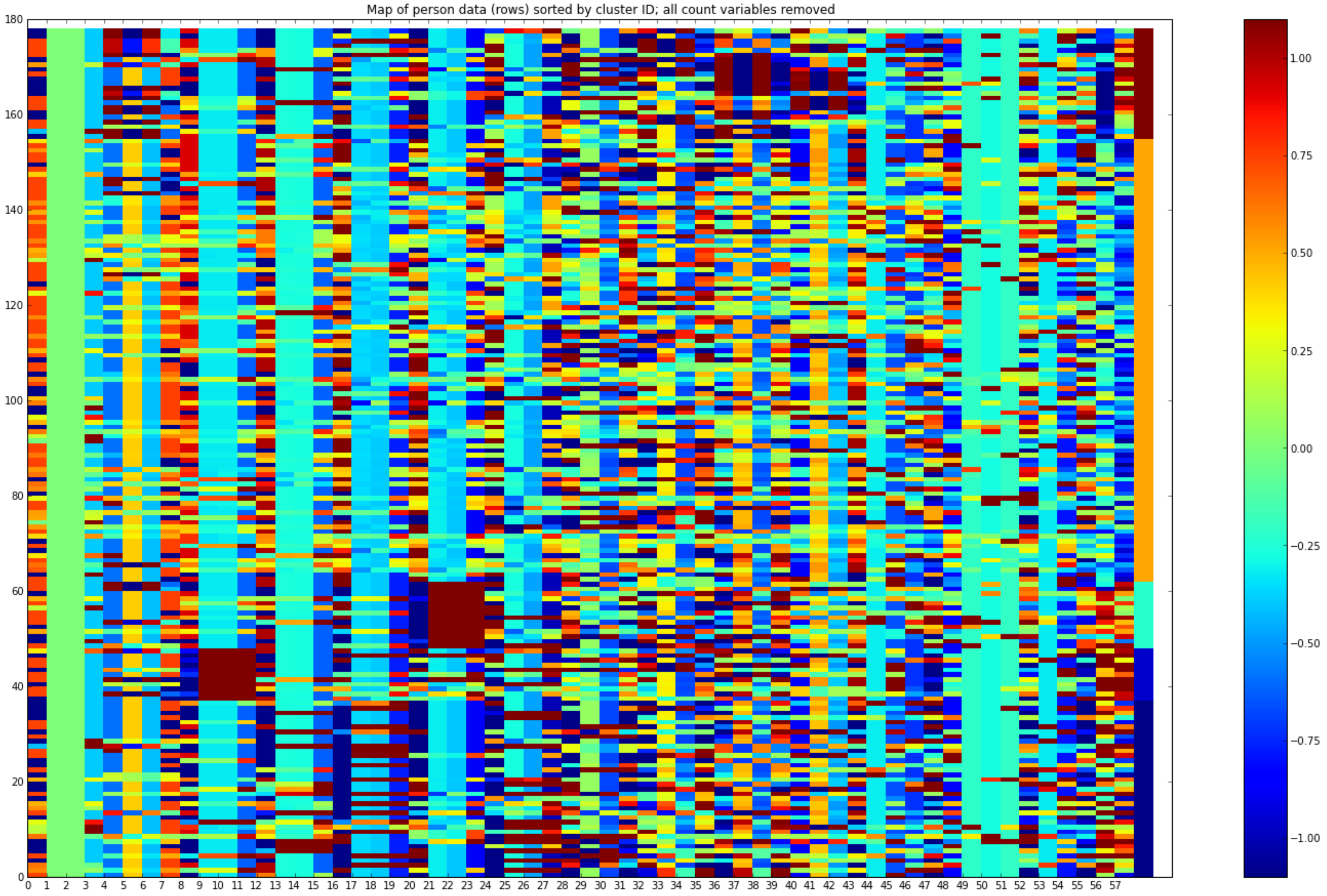
In [40]:
```
dataWithNoTotalCounts = qudf.drop(["totalPredictions"],1)
cmap_of_clustering(dataWithNoTotalCounts, k=5, titleString='Map of person data (rows) sorted by cluster ID; prediction count removed')
```



Without the total counts there still seems to be a strong band. Lets see if it's the count columns as these would order people by their prolificness. I've also take out user ID as this isn't related to anything at all, but if we can get away with it I'd like to leave it in so that we can see who's who

In [42]:
```
dataWithNoCountsAtAll = qudf.drop(["user",
                                   "totalPredictions","prediction_count_profile_min","prediction_count_profile_month",
                                   "prediction_count_profile_overFiftyYear","prediction_count_profile_postHoc",
```

```
                                              "prediction_count_profile_simultanious","prediction_count_profile_tenYear",
                                              "prediction_count_profile_week","prediction_count_profile_year","totalPredictions",
                                              "clusterID","prediction_count_profile_25%","prediction_count_profile_50%",
                                              "prediction_count_profile_75%","prediction_count_profile_FiftyYear","prediction_count_profile_count",
                                              "prediction_count_profile_day","prediction_count_profile_fiveYear","prediction_count_profile_max"],1)

cmap_of_clustering(dataWithNoCountsAtAll, k=5, titleString='Map of person data (rows) sorted by cluster ID; all count variables removed')
```



I can see two major blobs here if I screw up my eyes, one is reflected in a cluster (the largest cluster), but the other is probably just an accident.

The big, dark red blocks are proving to be a strong enough feature that they are makign clusters that only include them. They only really start to show up in this view though. The blocks are all associated with error values, but as they dont seem to have any other blocks that line up with them they are probably just anolylous groups of people who are hugely overconfident in a certain area of the calibration spectrum.

```
 | 0 | 0_falsePC           | 10 | 10_sqErrorPC       | 20 | 40_falsePC          | 30 | 60_sqErrorPC       | 40 | 90_falsePC                                | 50 |
prediction_count_profile_postHoc_pc      |
 | 1 | 0_signedSqErrorPC   | 11 | 10_truePC          | 21 | 40_signedSqErrorPC  | 31 | 60_truePC          | 41 | 90_signedSqErrorPC                        | 51 |
prediction_count_profile_simultanious_pc |
 | 2 | 0_sqErrorPC         | 12 | 20_falsePC         | 22 | 40_sqErrorPC        | 32 | 70_falsePC         | 42 | 90_sqErrorPC                              | 52 |
prediction_count_profile_std             |
 | 3 | 0_truePC            | 13 | 20_signedSqErrorPC | 23 | 40_truePC           | 33 | 70_signedSqErrorPC | 43 | 90_truePC                                 | 53 |
prediction_count_profile_tenYear_pc      |
 | 4 | 100_falsePC         | 14 | 20_sqErrorPC       | 24 | 50_falsePC          | 34 | 70_sqErrorPC       | 44 | prediction_count_profile_FiftyYear_pc     | 54 |
prediction_count_profile_week_pc         |
 | 5 | 100_signedSqErrorPC | 15 | 20_truePC          | 25 | 50_signedSqErrorPC  | 35 | 70_truePC          | 45 | prediction_count_profile_day_pc           | 55 |
prediction_count_profile_year_pc         |
 | 6 | 100_sqErrorPC       | 16 | 30_falsePC         | 26 | 50_sqErrorPC        | 36 | 80_falsePC         | 46 | prediction_count_profile_fiveYear_pc      | 56 |
signedSummedSquaredError                 |
 | 7 | 100_truePC          | 17 | 30_signedSqErrorPC | 27 | 50_truePC           | 37 | 80_signedSqErrorPC | 47 | prediction_count_profile_mean             | 57 | summedSquaredError
|
 | 8 | 10_falsePC          | 18 | 30_sqErrorPC       | 28 | 60_falsePC          | 38 | 80_sqErrorPC       | 48 | prediction_count_profile_month_pc         | 58 | clusterID
|
 | 9 | 10_signedSqErrorPC  | 19 | 30_truePC          | 29 | 60_signedSqErrorPC  | 39 | 80_truePC          | 49 | prediction_count_profile_overFiftyYear_pc |    |
|
```

todo next:

- look at the vectors in each cluster, see if there is any obvious commonality between them
- start taking columns out of the clustering to see if it makes any difference
- tidy up the notebook and graphs,
  - learn about subplots
  - put all graphs into their own variable

In [43]:
```
#for i in range(0,60,10):
#    print zip(range(0,len(dataWithNoCountsAtAll.columns)),dataWithNoCountsAtAll.columns)[i:i+10]
```

General site stats:

In [44]:
```
#query = """select * from  mysql.predictions {}""".format("WHERE predictions.creator_id = '2684'")
query = """select * from  mysql.predictions {}""".format("")
predictions = queryAsTable(query, maxrows=0)
#this all feels very ugly, especially the part where I make a coumn, split it and then delete it :(
pair = zip(predictions["predictions.created_at"], predictions["predictions.deadline"])

predictions["outlook"] = [timeDelta(x[0],x[1]) for x in pair]
predictions["outlook_bin"] = [x["bin"] for x in predictions["outlook"]]
predictions["outlook_bin_name"] = [x["bin_name"] for x in predictions["outlook"]]
firstPredDate = predictions["predictions.created_at"].min()
predictions["time_since_start"] = [x-firstPredDate for x in predictions["predictions.created_at"]]
predictions["seconds_since_start"] = [x.total_seconds() for x in predictions["time_since_start"]]

predictions = predictions.drop(["outlook","predictions.uuid"],1)
predictions[110:120]
```

Out[44]:

Out[44]:

	predictions.created_at	predictions.creator_id	predictions.deadline	predictions.id	predictions.private	predictions.updated_at	predictions.version	predictions.withdrawn	outlook_bin	outlook_bin_r
110	2008-08-01 00:31:39	1	2008-08-01 00:31:39	118	0	2008-08-01 09:18:02	1	0	1	simultanious
111	2008-08-01 01:49:57	9	2008-08-01 17:00:00	119	0	2008-08-01 09:18:02	1	0	2	day
112	2008-08-01 09:06:16	9	2009-11-11 01:20:03	120	0	2009-05-11 02:20:03	2	0	6	fiveYear
113	2008-08-01 09:20:26	4	2008-08-01 14:20:26	121	0	2008-08-01 15:17:14	1	0	2	day
114	2008-08-01 09:25:11	1	2008-08-01 09:25:11	122	0	2008-08-01 09:26:18	1	0	1	simultanious
115	2008-08-01 09:27:16	9	2008-08-06 12:00:00	123	0	2008-08-01 09:27:16	1	0	3	week
116	2008-08-02 04:36:37	9	2008-08-09 04:36:37	124	0	2008-08-14 09:03:33	1	0	4	month
117	2008-08-02 04:43:42	9	3008-08-02 04:43:42	125	0	2008-08-18 04:06:20	1	0	9	overFiftyYear
118	2008-08-03 02:43:27	9	2009-08-03 02:43:27	126	0	2008-08-03 02:43:27	1	0	6	fiveYear
119	2008-08-03 11:00:40	6	2008-12-31 12:00:00	127	0	2008-08-03 11:00:40	1	0	5	year

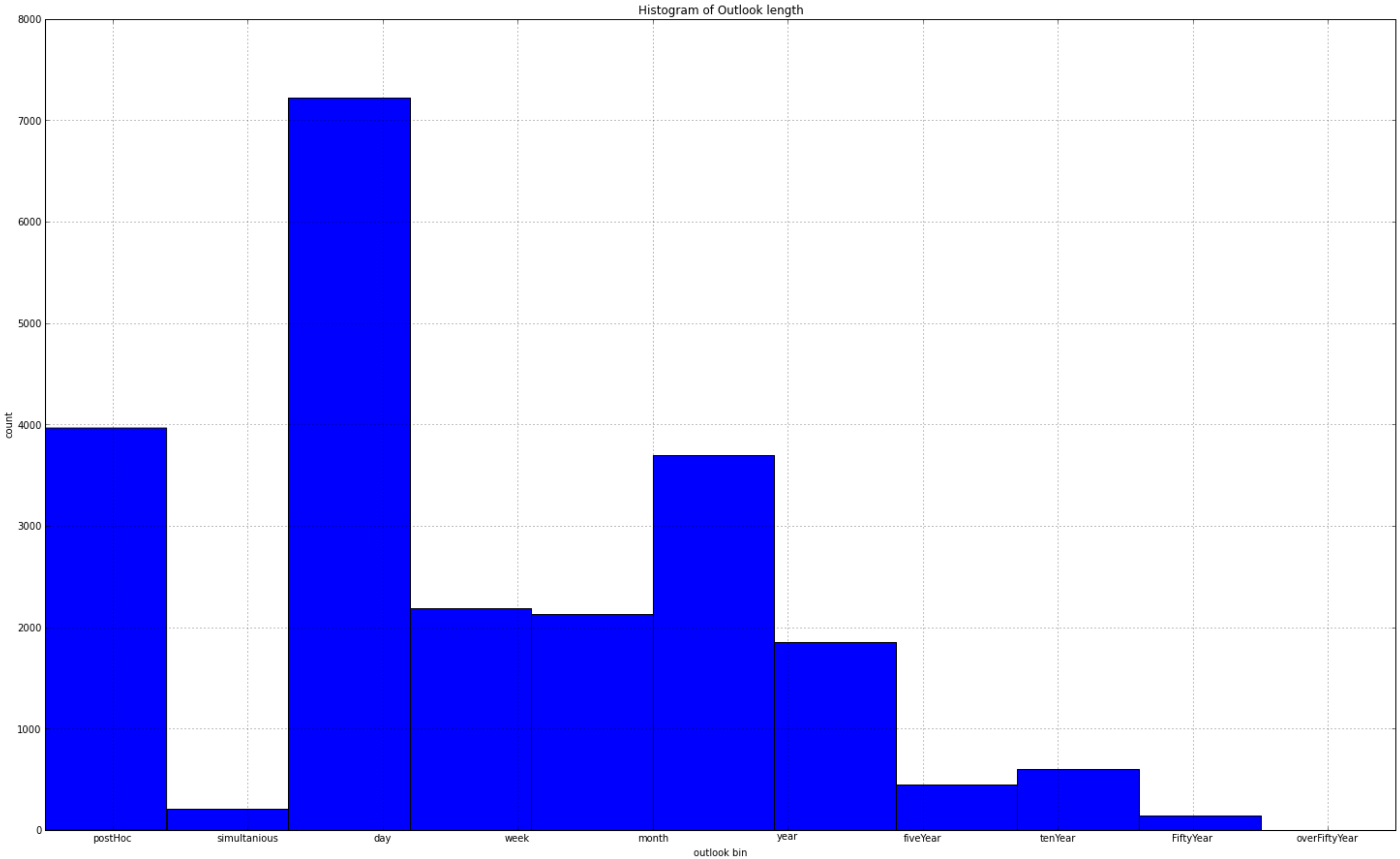
10 rows × 12 columns

In [45]:

```
figure(figsize=(25,15), dpi=100)
predictions["outlook_bin"].hist()

xlabel('outlook bin')
ylabel('count')
tickNames = ["postHoc","simultanious","day","week","month","year","fiveYear","tenYear","FiftyYear","overFiftyYear"]
xticks([x+0.5 for x in range(len(tickNames))], #locations,
       [x for x in tickNames],                 #labels

       fontsize=10)
title('Histogram of Outlook length')

desc = predictions["outlook_bin"].describe()
print predictions["outlook_bin_name"].value_counts()
print "mean",desc["mean"]
print "SD",desc["std"]
print "kurt",predictions["outlook_bin"].kurt()
```

```
         day      7226
     postHoc      3964
        year      3700
        week      2184
       month      2129
    fiveYear      1852
   FiftyYear       605
     tenYear       443
simultanious       209
overFiftyYear      143
dtype: int64
```

```
mean 3.05361834781
SD 2.15866002585
kurt -0.53837908226
```



My guess is that simultanous predictions are a mistake.

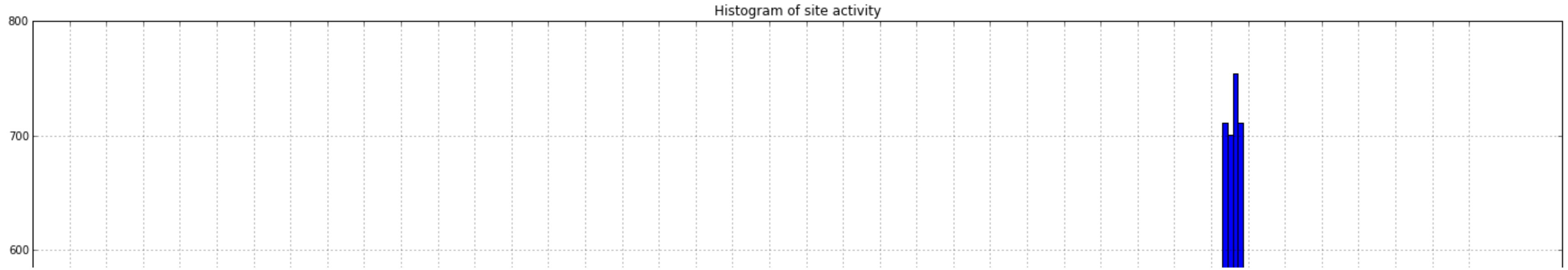
You might predict with a past date if you made predictions on paper and then entered them at a later date.

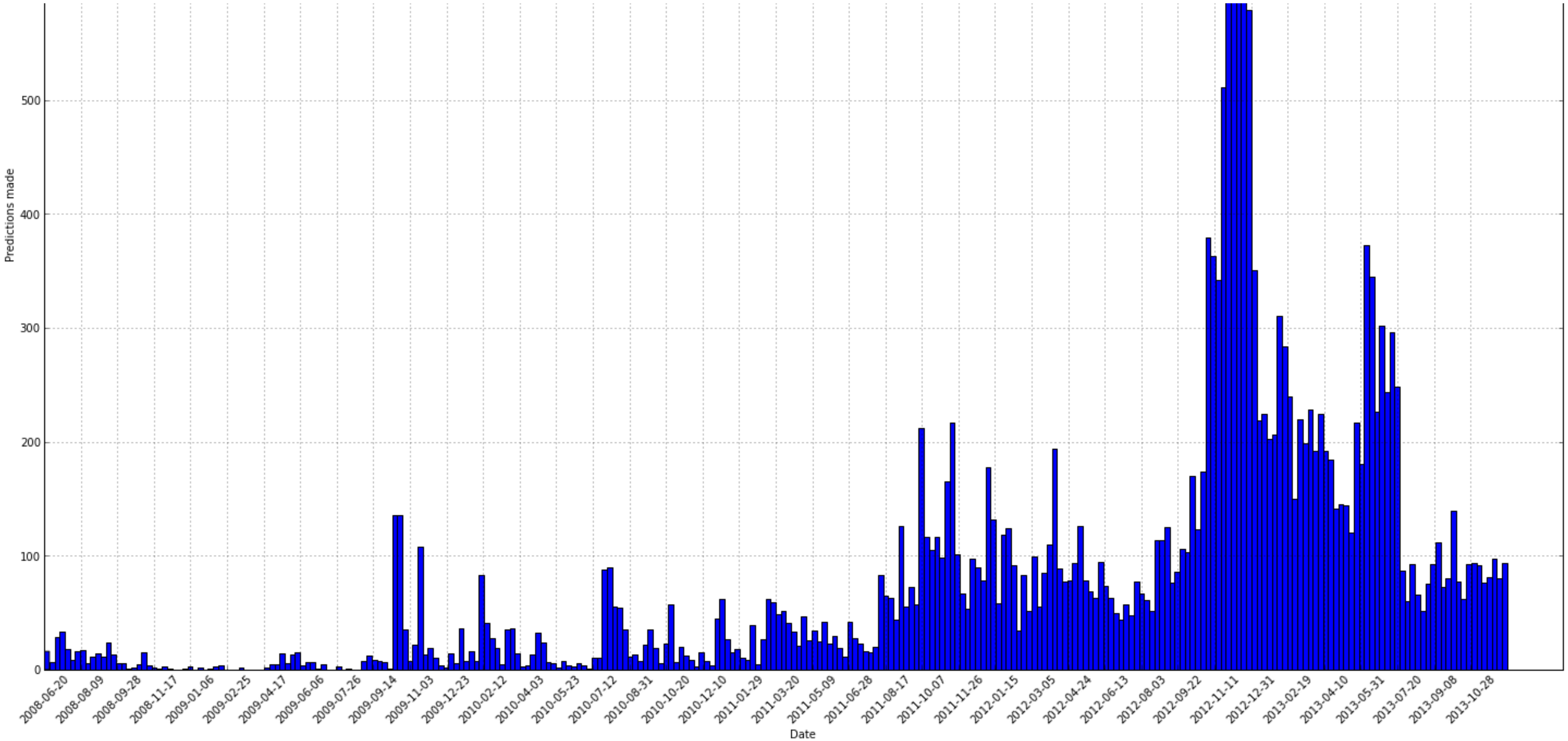
In [46]:
```
last = predictions["predictions_created_at"].max()
```

In [46]:
```
last  = predictions["predictions.created_at"].max()
first = predictions["predictions.created_at"].min()

days_in_e = (last - first).days
epoch = (last - first).total_seconds()
days_per_bin = 7
binNum = int(days_in_e / days_per_bin)
target_label_number = 40
labelPerNbins = binNum/target_label_number
spacedBinNum = int(binNum/labelPerNbins)

print "first", first
print "last",  last
print "date range", last - first
print "days_in_e",days_in_e
print "epoch", epoch, "seconds"
print "days_per_bin",days_per_bin
print "binNum", binNum
```

```
first 2008-06-20 03:46:16
last 2013-12-17 18:12:11
date range 2006 days, 14:25:55
days_in_e 2006
epoch 173370355.0 seconds
days_per_bin 7
binNum 286
```

In [47]:
```
def labelDate(first,epoch,binNum,step):
    secondsThisStep = (epoch/binNum)*step
    offset = datetime.timedelta(seconds=secondsThisStep)
    newDT = first + offset
    justDate=newDT.date()
    return justDate

plt.figure(figsize=(25,15), dpi=100)
predictions["seconds_since_start"].hist(bins=binNum)#, range=(0,30)
xlabel('Date')
ylabel('Predictions made')
tickNames = [labelDate(first,epoch,spacedBinNum,x) for x in range(spacedBinNum)]
plt.xticks([x*(epoch/spacedBinNum) for x in range(len(tickNames))], #locations,
           [x for x in tickNames],             #labels
           rotation=45,
           fontsize=10)
plt.title('Histogram of site activity')
plt.show()
```





I can't find a reason for the spike in 09-12 2012, there doesn't seem to be an obvious reason in a [time bracketed google search]

In [48]:
```
print pd.__version__
#when this reads 0.14.x it should fix the deprication errors
```

```
0.13.1
```

In [48]:
```

```

In [49]:
```
predictors = set(queryAsTable("""select DISTINCT p.creator_id from  mysql.predictions as p""", maxrows=0)["p.creator_id"].tolist())
#print predictors
print "Length: ",len(predictors)
```

```
Length:  886
```

In [50]:
```
responders = set(queryAsTable("""select DISTINCT r.user_id from  mysql.responses as r""", maxrows=0)["r.user_id"].tolist())
#print responders
print "Length: ",len(responders)
```

```
Length:  1198
```

In [51]:
```
allUsers = set(queryAsTable("""select DISTINCT u.id from  mysql.users as u""", maxrows=0)["u.id"].tolist())
#print allUsers # way too many to print!
print "Length: ",len(allUsers)

#queryAsTable("""select * from  mysql.users""", maxrows=10)
```

```
Length:  22578
```

In [52]:
```
#people who've made a prediction but not a response
p_no_s = predictors - responders

#people who've made a response but not a prediction
s_no_p = responders - predictors

#print p_no_s
print "{} people have made a prediction but not made a response".format(len(p_no_s))
print
#print s_no_p
print "{} people have made a response but not made a prediction".format(len(s_no_p))
print "{} as a difference".format(len(responders) - len(predictors))

#allUsers
```

```
0 people have made a prediction but not made a response

312 people have made a response but not made a prediction
312 as a difference
```

So this is an interesting finding, by making a prediction one is making a big mental investment, but just weighing in on someone else's is much easier. I suppose this is the *youTube commenter* problem manifesting itself again!

In [53]:
```
orstring = ""
for person in s_no_p:
    orstring += "r.user_id='{}' OR ".format(person)
orstring += "r.user_id='{}'".format("not a real user") ## this is because I'm too lazy to take the last or off
#print orstring

query = """select r.user_id, COUNT(r.user_id) from  mysql.responses as r WHERE {} GROUP BY r.user_id ORDER BY COUNT(r.user_id) DESC""".format(orstring)
lurkers = queryAsTable(query, maxrows=0)
lurkers[:5]
```

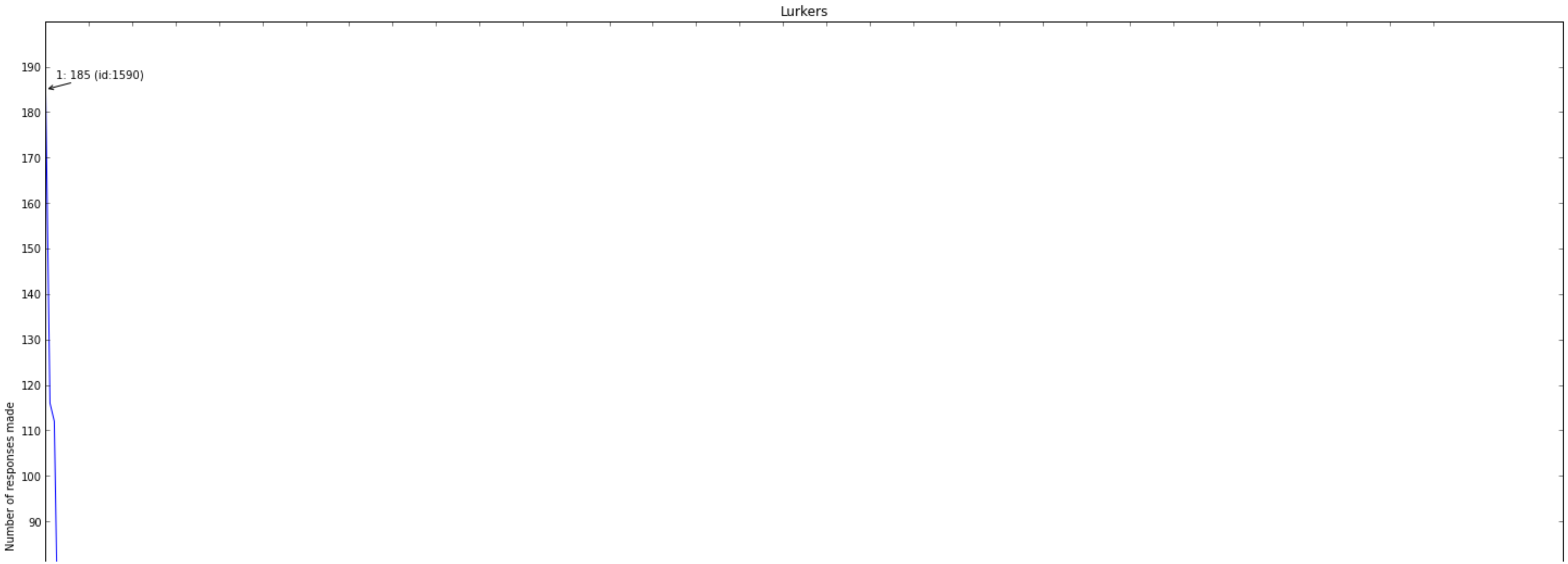
Out[53]:

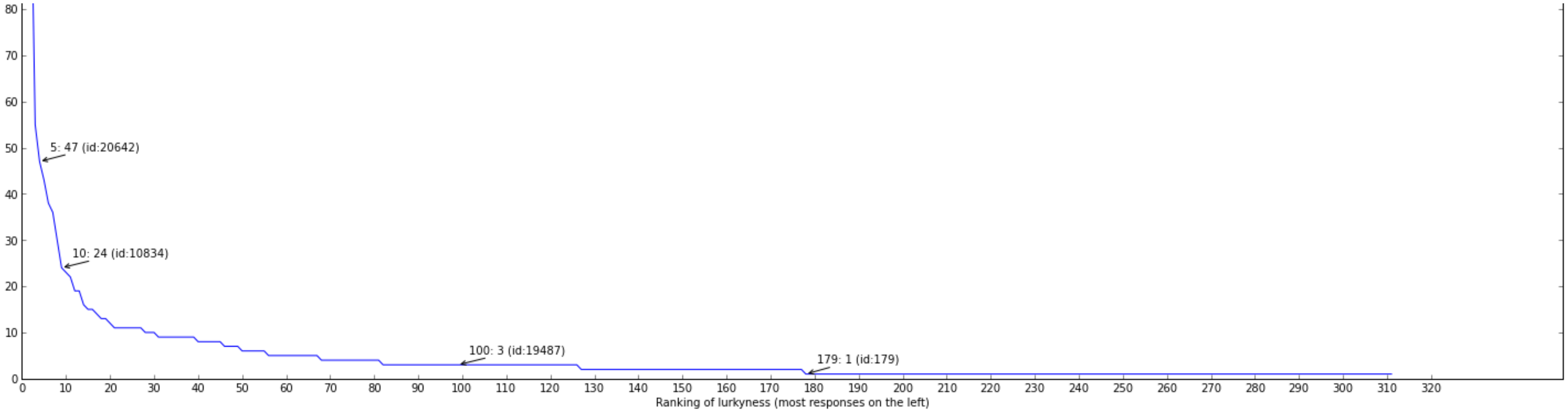
	COUNT(r.user_id)	r.user_id
0	185	1590
1	116	1285
2	112	946
3	55	16430
4	47	20642

5 rows × 2 columns

In [54]:
```
fig, ax = plt.subplots(figsize=(25,15), dpi=100)
ax.plot(lurkers["COUNT(r.user_id)"])
```

```
yticks(range(0,    lurkers["COUNT(r.user_id)"][0]+10,10))
xticks(range(0,len(lurkers["COUNT(r.user_id)"])  +10,10))

def annotateArrow(theObject, xval, yval, index, isTime=False):
    theID    = theObject.iloc[index][xval]
    theValue = theObject[yval][index]
    annotation = ""
    if isTime:
        timeValue = np.timedelta64(np.timedelta64(theValue, "ns"),'D')
        annotation = "{}: {} (id:{})".format(index+1, str(timeValue),  theID )
    else:
        annotation = "{}: {} (id:{})".format(index+1, str(theValue),  theID )
    pos = (index,  theValue)
    arrow = dict(arrowstyle='->', shrinkA=0)
    ax.annotate( annotation , xy=pos,  xytext=(10, 10), ha='left', textcoords='offset points', arrowprops=arrow)

annotateArrow(lurkers, 'r.user_id', "COUNT(r.user_id)", 0)
annotateArrow(lurkers, 'r.user_id', "COUNT(r.user_id)", 4)
annotateArrow(lurkers, 'r.user_id', "COUNT(r.user_id)", 9)
annotateArrow(lurkers, 'r.user_id', "COUNT(r.user_id)", 99)
last = lurkers["COUNT(r.user_id)"].tolist().index(1)
annotateArrow(lurkers, 'r.user_id', "COUNT(r.user_id)", last)

plt.xlabel('Ranking of lurkyness (most responses on the left)')
plt.ylabel('Number of responses made')
plt.title('Lurkers')

plt.show()
```





This graph shows the number of 'lurkers' and how active they are. There are 2 users who have over 100 responses but have *never* made a prediction!

In [55]:
```
earliest_time = queryAsTable("""select p.created_at from  mysql.predictions as p ORDER BY p.created_at LIMIT 1""", maxrows=0)["p.created_at"][0]
earliest_time
```

Out[55]:
```
Timestamp('2008-06-20 03:46:16', tz=None)
```

In [56]:
```
queryAsTable("""select * from  mysql.predictions as p ORDER BY p.created_at LIMIT 1""", maxrows=3)
```

Out[56]:

	p.created_at	p.creator_id	p.deadline	p.id	p.private	p.updated_at	p.uuid	p.version	p.withdrawn
0	2008-06-20 03:46:16	1	2008-06-20 12:00:00	1	0	2008-08-14 02:28:40	d52eeaf6-cb20-456f-a375-bec2d1acc44d	1	0

1 rows × 9 columns

In [57]:
```
predictions = queryAsTable("""select * from mysql.predictions as p""", maxrows=0)
print len(predictions)
predictions[:5]
```

```
22455
```

Out[57]:

	p.created_at	p.creator_id	p.deadline	p.id	p.private	p.updated_at	p.uuid	p.version	p.withdrawn
0	2008-06-20 03:46:16	1	2008-06-20 12:00:00	1	0	2008-08-14 02:28:40	d52eeaf6-cb20-456f-a375-bec2d1acc44d	1	0
1	2008-06-20 04:37:13	2	2008-06-23 12:00:00	2	0	2008-08-01 09:18:01	86530ca2-fbd5-47a8-b2d8-12110e05c3b4	1	0
2	2008-06-20 04:38:35	3	2008-06-30 12:00:00	3	0	2008-08-01 09:18:01	d5adb0ee-ecf2-48ce-88ce-7b40b5b2179f	1	0
3	2008-06-20 04:40:06	3	2008-06-19 12:00:00	4	0	2008-08-01 09:18:01	100f4ff6-4ee7-44e8-aa2d-5394dcaead7e	1	0
4	2008-06-20 05:08:31	4	2008-06-20 06:08:31	5	0	2008-08-01 09:18:01	c0314cd4-070a-4d4b-ae3c-ec03ddbdc621	1	0

5 rows × 9 columns

In [58]:
```
allUsers = set(predictions["p.creator_id"])
```

```
len(allUsers)
```

Out[58]: 886

In [59]:
```
firstEverPrediction = sort(predictions["p.created_at"])[0]
print "firstEverPrediction", firstEverPrediction, type(firstEverPrediction)
```

```
firstEverPrediction 2008-06-20T13:46:16.000000000+1000 <type 'numpy.datetime64'>
```

In [59]:

```
import random

startDate = [] #this is a byproduct for showing off the aquisition rate
individualsDFs = []

for person in allUsers:#random.sample(allUsers, 5):#for testing on a set
    thisUsersPredictions = predictions[person == predictions["p.creator_id"] ]
    tup = thisUsersPredictions


    dates = tup.sort(["p.created_at"])

    firstPredictionDate = tidyDate(dates.iloc[0]["p.created_at"])
    lastPredictionDate  = tidyDate(dates.iloc[len(dates)-1]["p.created_at"])

    startDate.append({"ID":person, "startDate":firstPredictionDate})

    delta         = firstPredictionDate - firstEverPrediction
    activityRange = lastPredictionDate  - firstPredictionDate
    keeper = activityRange > datetime.timedelta(days=30) #np.timedelta64(30, 'D')

    tup["keeper"] =  keeper
    tup["range"] =  activityRange
    tup["delta"] =  delta
    tup["p.updated_at_Z"] = tup["p.updated_at"] - delta
    tup["p.deadline_Z"]   = tup["p.deadline"]   - delta
    tup["p.created_at_Z"] = tup["p.created_at"] - delta
    tup = tup.drop(["p.created_at",  "p.deadline", "p.private", "p.updated_at", "p.version", "p.withdrawn", "p.uuid"], 1)

    individualsDFs.append(tup)
```

```
individualsDFs[0][:10]
#This is the data to summarise to describe
```

```
type(firstEverPrediction)
```

```
cutOffDate = firstEverPrediction + relativedelta(days=30)
print firstEverPrediction
print cutOffDate
print cutOffDate - firstEverPrediction

summaryPeople = []

for df in individualsDFs:
    tempDF = df[df["p.created_at_Z"] < cutOffDate]
```

```
    tempDF["p.deadline_Z"]   = tempDF["p.deadline_Z"].apply(tidyDate)
    tempDF["p.created_at_Z"] = tempDF["p.created_at_Z"].apply(tidyDate)
    tempDF["outlook"] = tempDF["p.deadline_Z"] - tempDF["p.created_at_Z"]
    outlookDescription =  tempDF["outlook"].describe()

    summaryPeople.append(dict({"keeper":df["keeper"].iloc[0],
                               "ID":    df["p.creator_id"].iloc[0],
                               "range": df["range"].iloc[0]
                              },
                              **outlookDescription))
```

```
summaryPeopleDF = pd.DataFrame.from_dict(summaryPeople).fillna(0)
summaryPeopleDF[:10]
```

In [ ]:
```
#TODO
#scale values in DF
#Look at different modeling techniques
#  we have 2 target values, so we can go for a supervised method
#  one is binary, so would be worth looking at KNN
#  one is continuous so I could look at a standard regression
#if there isn't any predictive power, add in the confidence (response) data
```

```
#this code is taken almost verbaitim from:
#http://blog.yhathq.com/posts/classification-using-knn-and-python.html

import pylab as pl
from sklearn.neighbors import KNeighborsClassifier

def knnLoop(df, trainCol="keeper", colsToRemove=["keeper","range"]):
    pl.subplots(figsize=(25,15), dpi=100)

    #I've added in multiple runs of multiple runs as it seems to have pretty varied results over time,
    #this results in the KNN being run 3600 times :(
    #But it does give some good insight into the results!
    for i in xrange(30):
        test_idx = np.random.uniform(0, 1, len(df)) <= 0.3
        train = df[test_idx==True]
        test  = df[test_idx==False]

        features = df.columns.tolist()#["25%", "50%", "75%", "ID", "count",  "max", "mean", "min",  "std"] # removing ["keeper","range"]
        for r in colsToRemove:
            features.remove(r)

        results = []
        for n in range(1, 30, 1):
            clf = KNeighborsClassifier(n_neighbors=n)
            clf.fit(train[features], train[trainCol])
            preds = clf.predict(test[features])
            accuracy = np.where(preds==test[trainCol], 1, 0).sum() / float(len(test))
            #print "Neighbors: %d, Accuracy: %3f" % (n, accuracy)

            results.append([n, accuracy])

        results = pd.DataFrame(results, columns=["n", "accuracy"])

        pl.plot(results.n, results.accuracy,alpha=0.5)
    pl.title("Accuracy with Increasing K")
    pl.xticks(range(0,31,1))
```

```
    pl.show()
```

```
knnLoop(summaryPeopleDF)
```

With the full dataset it looks like we go from roughly random to slightly better than random. There is a graph below that shows the distributuon of ranges, and it seems that there are about 250 out of 900 that are keepers, that's about 75% not keepers, so if the model predicted *not a keeper* for everything we'd be somewhere in the 75% range. Lets rerun this with a balanced dataset; sampling teh same number of keepers and not keepers. If that still doesn't work we'll need to add more variables and/or scale the values.

```
#grouped = summaryPeopleDF.groupby("keeper")
#print grouped.apply(len)

def stratifiedSample(df, key, n):

    def sampleN(df, n):
        #print type(df)
        #print "this DF is:", len(df)
        trues = [True  for x in range(n)]
        falses= [False for x in range(len(df)-n)]
        samplingVector = random.sample((trues+falses), len(df))
        return samplingVector

    true_group  = df[df[key] == True]
    false_group = df[df[key] == False]

    sampled_true_group  = true_group[sampleN(true_group,   n)]
    sampled_false_group = false_group[sampleN(false_group, n)]

    return pd.concat([sampled_true_group, sampled_false_group])

summaryPeopleDFbalanced = stratifiedSample(summaryPeopleDF, "keeper", 260)
summaryPeopleDFbalanced.describe()
```

```
knnLoop(summaryPeopleDFbalanced)
```

That isn't very optimistic! It looks like this confirms the supposition that there ins't any predictive power in the data that we've handed it. Given how disasterous this run is it would be worth going back over it and adding in aditional variables.

Jump down, over the outlook graphs, for this.

```
predictionsWdelta = pd.concat(individualsDFs)
predictionsWdelta[:15]
```

```
len(predictionsWdelta)
```

```
ranges = predictionsWdelta[["p.creator_id", "range"]]
ranges = ranges.drop_duplicates(cols="p.creator_id")
ranges = ranges.sort("range",ascending=False)
ranges = ranges.reset_index(drop=True)
print len(ranges)
ranges[:10]
```

```
fig, ax = plt.subplots(figsize=(25,15), dpi=100)
ax.plot(ranges["range"])


annotateArrow(ranges, "p.creator_id", "range", 0 , True)
annotateArrow(ranges, "p.creator_id", "range", 4 , True)
annotateArrow(ranges, "p.creator_id", "range", 99, True)
last = ranges["range"].tolist().index(0)
annotateArrow(ranges, "p.creator_id", "range", last)

f = lambda x : x.item()#.total_seconds()

longestRange = ranges["range"].apply(f).max()

step = int(longestRange/10)
steps = range(0,longestRange+step,step )

yticks( [y for y in steps], [np.timedelta64(np.timedelta64(x, "ns"),'D') for x in steps])
#xticks(range(0,len(ranges["range"])  +10,10))

plt.xlabel('Number of users')
plt.ylabel('date range active')
plt.title('Active Date Ranges')

plt.show()
```

```
starters = pd.DataFrame.from_dict(startDate).sort("startDate")

fig, ax = plt.subplots(figsize=(25,15), dpi=100)
ax.plot(starters["startDate"])


#yticks(range(0,    lurkers["COUNT(r.user_id)"][0]+10,10))
#xticks(range(0,len(lurkers["COUNT(r.user_id)"])  +10,10))

plt.xlabel('Number of users')
plt.ylabel('date Aquired')
plt.title('User aquisition')

plt.show()

#TODO this plot is super retarded, it should be time on the x axis. It sould be rotated 90 left and then mirrored
```

These are a couple of quick* graphs that come out as a side effect of trying to get an description of people's first month of activity.

The first is the distribution of active range. This is the ammount of time between their first prediction and their most recent one. This shows that there are a *lot* of people who have used the site for more than a month.

The second (somewhat munted) graph is the total number of active predicters on the site ever. The lurkers graph shows that most people who have done more than just sign up have made a prediction.

*not quick! working with `np.timedelta64`s is a real pain in the arse!

In an attempt to make a better prediction I'm pulling data from all useful tables: responses, predictions, users, judgements

```
predictions = queryAsTable("""SELECT * FROM mysql.predictions p """, maxrows=0)
responses   = queryAsTable("""SELECT * FROM mysql.responses   r """, maxrows=0)
users       = queryAsTable("""select * from mysql.users u""",        maxrows=0)
judgements  = queryAsTable("""select * from mysql.judgements j""",   maxrows=0)
```

```
print "predictions",predictions.columns
print "judgements", judgements.columns
print "responses",  responses.columns
print "users",      users.columns
```

```
allUsers = set(predictionsAndResponses["p.creator_id"])
len(allUsers)

individualsDFs = []
```

```
def processUserPredictions(thisUsersPredictions):
    tup = thisUsersPredictions
    dates = sort(tup["p.created_at"])
    firstPredictionDate = dates.iloc[0]
    lastPredictionDate  = dates.iloc[len(dates)-1]

    delta         = firstPredictionDate - firstEverPrediction
    activityRange = lastPredictionDate  - firstPredictionDate
    keeper = activityRange > datetime.timedelta(days=30) #np.timedelta64(30, 'D')

    tup["keeper"] =  keeper
    tup["range"]  =  activityRange
    tup["delta"]  =  delta
    tup["p.updated_at_Z"] = tup["p.updated_at"] - delta
    tup["p.deadline_Z"]   = tup["p.deadline"]   - delta
    tup["p.created_at_Z"] = tup["p.created_at"] - delta
    #["p.created_at", "p.creator_id", "p.deadline", "p.id", "p.private", "p.updated_at", "p.uuid",
    # "p.version", "p.withdrawn", "r.confidence", "r.created_at", "r.id", "r.prediction_id", "r.updated_at", "r.user_id"]
    tup = tup.drop(["p.created_at",  "p.deadline", "p.private", "p.updated_at", "p.version", "p.withdrawn", "p.uuid"], 1)

    #clip at 30
    tup = tup[tup["r.created_at_Z"] < (firstEverPrediction + datetime.timedelta(days=30))]

    return tup
```

```
def processUserResponses(thisUsersResponses):
    #responses Index([u'r.confidence', u'r.created_at', u'r.id', u'r.prediction_id', u'r.updated_at', u'r.user_id'], dtype=object)
    tur = thisUsersResponses

    dates = sort(tur["r.created_at"])
    firstPredictionDate = dates.iloc[0]
    lastPredictionDate  = dates.iloc[len(dates)-1]
    delta         = firstPredictionDate - firstEverPrediction
    activityRange = lastPredictionDate  - firstPredictionDate

    tur["range"]  =  activityRange
    tur["delta"]  =  delta

    tur["r.updated_at_Z"] = tur["r.updated_at"] - delta
    tur["r.created_at_Z"] = tur["r.created_at"] - delta

    tur = tur.drop([ u'r.created_at', u'r.id', u'r.prediction_id', u'r.updated_at', u'r.user_id'] ,1)

    #clip at 30
    tur = tur[tur["r.created_at_Z"] < (firstEverPrediction + datetime.timedelta(days=30))]

    return tur
```

```
def processUserDetails(thisUsersDetails):
    tud = thisUsersDetails
    #from datetime import datetime
    #tud["age"] = datetime.today() - tud["u.created_at"].iloc[0]
    return tud
```

```
def processUserJudgements(thisUsersJudgements, this_userID):
    #judgements Index([u'j.created_at', u'j.id', u'j.outcome', u'j.prediction_id', u'j.updated_at', u'j.user_id'], dtype=object)
    tuj = thisUsersJudgements

    tuj["selfJudge"] = tuj['j.user_id'] == this_userID

    dates = sort(tuj["j.created_at"])
    firstPredictionDate = dates.iloc[0]
    lastPredictionDate  = dates.iloc[len(dates)-1]
    delta         = firstPredictionDate - firstEverPrediction
    activityRange = lastPredictionDate  - firstPredictionDate

    tuj["j.created_at_Z"] = tuj["j.created_at"] - delta

    tuj = tuj.drop([u'j.created_at', u'j.id', u'j.prediction_id', u'j.updated_at', u'j.user_id'] ,1)


    #clip at 30
    tuj = tuj[tuj["j.created_at_Z"] < (firstEverPrediction + datetime.timedelta(days=30))]


    return tuj
```

```
for person in allUsers:#random.sample(allUsers, 5):#for testing on a set
    #subset the dataframe
    thisUsersPredictions = predictions[person == predictions["p.creator_id"]]
    thisUsersResponses   =   responses[person == responses[  "r.user_id"]   ]
    thisUsersDetails     =       users[person == users[      "u.id"]    ]
    thisUsersJudgements  =  judgements[person == judgements[ "j.user_id"]   ]
    #process the subsets
    up = processUserPredictions(thisUsersPredictions)
    ur = processUserResponses(thisUsersResponses)
    ud = processUserDetails(thisUsersDetails)
    uj = processUserJudgements(thisUsersJudgements, person)


    #from here down is the next loop, sumarise and combine each table's data.
    tempDF = df[df["p.created_at_Z"] < cutOffDate]

    tempDF["p.deadline_Z"]   = tempDF["p.deadline_Z"].apply(tidyDate)
    tempDF["p.created_at_Z"] = tempDF["p.created_at_Z"].apply(tidyDate)
    tempDF["outlook"] = tempDF["p.deadline_Z"] - tempDF["p.created_at_Z"]
    outlookDescription =  tempDF["outlook"].describe()

    ##responses
    #describe confidence
    #ratio of self userID to other userID

    ##user
    #u.timezone == none
```

```
    ##judgements
    #self ratio
    #describe times

    summaryPeople.append(dict({"keeper":df["keeper"].iloc[0],
                               "ID":    df["p.creator_id"].iloc[0],
                               "range": df["range"].iloc[0]
                              },
                              **outlookDescription))

summaryPeopleDF_PandR = pd.DataFrame.from_dict(summaryPeople).fillna(0)
summaryPeopleDF_PandR[:10]
```

In [ ]:

```
summaryPeopleDF_PandR_balanced = stratifiedSample(summaryPeopleDF_PandR, "keeper", 260)
```

```
knnLoop(summaryPeopleDF_PandR_balanced)
knnLoop(df, trainCol="keeper", colsToRemove=["keeper","range"]):
```