Part II - Predicting When an EV is Charging

September 24, 2016

1 Predicting when an EV Charging

The goal of this section is the following: Given that a household owns an EV, is it possible to predict at which times the EV is charging?

1.1 Introduction

The most appropriate tool for this kind of problem is the Markov Model. Markov Models are well-designed for time-series data with the following properties: - There exist one or more "observation" variables, which are known with absolute precision (for example, the power consumption at each timestep) - There exist one or more "hidden" variables, which have to be inferred given the observation variables (for example, whether a household is charging at each timestep) - The "hidden" variables can only take discrete values (there are other tools for continuous hidden variables, such as Kalman Filters or Auto-Regressions)

A first-order Markov Model basically works as follows: - Construct a model with hidden variables X_i and evidence variables E_i for timesteps $t = i$ - Calculate transition probabilities $P(X_i|E_i, X_{i-1})$ - Assign probabilities $P(X_0)$ at time $t = 0$ - Update each timestep using Bayes' Theorem: $P(X_i, E_i) = \sum_{x_{i-1} \in X_{i-1}} P(X_i|E_i, x_{i-1})P(x_{i-1})$

Afterward, the list $P(X_i, E_i)$ would give the probabilities that an EV is charging at each timestep $t = i$.

For this problem, the only hidden variable is whether a car is charging or not.
Let $C_i \in 0, 1$ be whether an EV is charging (0 = not charging, 1 = charging).
(The observation variables will be discussed shortly.)

The problem of constructing a transition model can be reduced to a classification problem. There are four possible transitions: - $0 \to 0$ (call 00) - $0 \to 1$ (call 01) - $1 \to 0$ (call 10) - $1 \to 1$ (call 11)

The evidence variables (eg, usage) can be grouped according to the transition which occured at the same timestep. For example, suppose there are 100,000 data points with transition 00 and 10,000 data points with transition 01. Also suppose that most of the 01 transitions had a usage spike greater than 2kWh, and most of the 00 didn't have a usage spike. The value of "delta usage" could then be used to classify 00 against 01 transitions. The same approach could be used for classifying 10 against 11.

The real data is of course much trickier. The ratio of number of 00 to 01 transitions is closer to 100:1 than 10:1. The sheer overabundance of 00 transitions makes it difficult to construct a model which "picks out" the rare 01 transitions. I played around with finite differences for a while, but they didn't seem to work too well.

I eventually settled on using splines to find the first few derivatives at each point (y, y', y'', y''), and then used these as a 4-dimensional dataset for classification.

This section is organized as follows: - Constructing transition models - Testing the accuracy of the transition models - Testing the accuracy of the Markov Models

1.2 Preprocessing

Some imports, reading data, etc. All the houses which don't own EVs are ignored throughout (the Markov Model will assume an EV is already present).

In [1]:
```
%matplotlib inline

import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as pl
import matplotlib.colors as colors
import numpy as np
import scipy
import scipy.stats as stats
import scipy.stats.mstats as mstats
import scipy.interpolate as interpolate
import sklearn.neighbors as neighbors
import sklearn
import random

# DataFrame where index = intervals, columns = house ids
usages = pd.read_csv('./data/EV_train.csv', index_col='House ID').transpose().dropna()
when_charging = pd.read_csv('./data/EV_train_labels.csv', index_col='House ID').transpose()[0:l

# Only use houses that own EVs
usages.index = range(len(usages.index))
when_charging.index = range(len(when_charging.index))
houses = usages.columns

original_usages = usages

houses = houses[when_charging.sum() > 0]
usages = usages[houses]
when_charging = when_charging[houses]
```

Since a Markov Model runs over the timeseries for an entire house at once, it makes sense to split into train/test data by house. The code below makes this split.

In [2]:
```
# showing this code for now, but will be put into util.py for reuse in later sections
def split_data(data, ratios=[0.8, 0.2]):
    ind_sets = []
    num_points = len(data)
    remaining_inds = range(num_points)
    for r in ratios:
        curr_num = int(num_points * r)
        new_inds = random.sample(remaining_inds, curr_num)
        ind_sets.append(new_inds)
        remaining_inds = list(set(remaining_inds) - set(new_inds))
    split_data = [data[inds] for inds in ind_sets]
    return split_data
```

In [3]:
```
random.seed(0)
train_markov, test_markov = split_data(houses, ratios=[0.8, 0.2])
```

The next section creates the data to be used as input to the classifer. The most important variables are `X` and `y`. `X` is a numpy array, where each element is a 4D array, representing the first four derivatives (including the value itself, the "zeroth" derivative). `y` is a numpy array, where each element is one of $\{0, 1, 2, 3\}$, representing the class of each transition at that point (0 is 00, 1 is 01, 2 is 10, 3 is 11). `X` and `y` are basically a "bag of values" with all of the house information removed.

The values are preprocessed in a couple ways. First, the consumption data for each house is subtracted by its average and divided by its standard deviation, so only the "shape" of the usage graph is left. Second,

each type derivative is subtracted by its global average and divided by its global standard deviation, so each derivative has the same weight in the classifier.

In [4]:
```
# normalize each house with itself (average = 0, std = 1)
mean = usages.apply(np.average)
std = usages.apply(np.std)
usages = (usages - mean) / std

# fit a spline for each house
raw_splines = {}
xs = np.linspace(0, len(usages.index)-1, len(usages.index)).astype(float)
for house in houses:
    raw_splines[house] = interpolate.UnivariateSpline(xs, usages[house], k=3, s=0)

# get the derivatives at each timestep
d0 = pd.DataFrame(index=usages.index)
d1 = pd.DataFrame(index=usages.index)
d2 = pd.DataFrame(index=usages.index)
d3 = pd.DataFrame(index=usages.index)
for house in houses:
    d0[house] = [raw_splines[house].derivatives(x)[0] for x in xs]
    d1[house] = [raw_splines[house].derivatives(x)[1] for x in xs]
    d2[house] = [raw_splines[house].derivatives(x)[2] for x in xs]
    d3[house] = [raw_splines[house].derivatives(x)[3] for x in xs]
derivatives = [d0, d1, d2, d3]

# since the derivatives act as a "4D" vector, normalize each dimension so they have the same we
for i,d in enumerate(derivatives):
    mean = np.average(d.values)
    std = np.std(d.values)
    d = (d - mean) / std
    derivatives[i] = d

# get transitions at each timestep
back = when_charging[:-1]
back.index = range(1, len(back.index)+1)
front = when_charging[1:]
#front.index = range(len(front.index))

n00 = (back == 0) & (front == 0)
n01 = (back == 0) & (front == 1)
n10 = (back == 1) & (front == 0)
n11 = (back == 1) & (front == 1)
transition_times = [n00, n01, n10, n11]

# put derivatives in terms of X (a 4D array at each timestep) and y (label at each timestep)
T00 = 0
T01 = 1
T10 = 2
T11 = 3
transition_labels = [T00, T01, T10, T11]
transition_names = ['00', '01', '10', '11']
```

This next part splits the classifier data again into training and test data. The split is used to test the accuracy of the classifier itself (as opposed to the split earlier, which split houses is used to the accuracy of

the resulting Markov Model.)

```
In [5]: train_classifier, test_classifier = split_data(train_markov, ratios=[0.8, 0.2])

        def remove_nan(x):
            return x[~np.isnan(x)]

        def get_all(x, index):
            return remove_nan(x[index].values.flatten())

        train_X = []
        train_y = []
        test_X = []
        test_y = []
        split_X = []
        split_y = []

        for (label, times) in zip(transition_labels, transition_times):
            curr_train_X = np.array([get_all(d, times[train_classifier]) for d in derivatives]).T
            curr_train_y = np.array([label] * len(curr_train_X))
            train_X.extend(curr_train_X)
            train_y.extend(curr_train_y)

            curr_test_X = np.array([get_all(d, times[test_classifier]) for d in derivatives]).T
            curr_test_y = np.array([label] * len(curr_test_X))
            test_X.extend(curr_test_X)
            test_y.extend(curr_test_y)

            split_X.append(curr_train_X)
            split_y.append(curr_train_y)

        train_X = np.array(train_X)
        train_y = np.array(train_y)
        test_X = np.array(test_X)
        test_y = np.array(test_y)
```

Before moving on, it's worth noting that to create a classifier for (derivatives -> transition), there's basically two classification problems, (00, 01) and (10, 11), since $p_{00} + p_{01} = 1$, and $p_{10} + p_{11} = 1$.

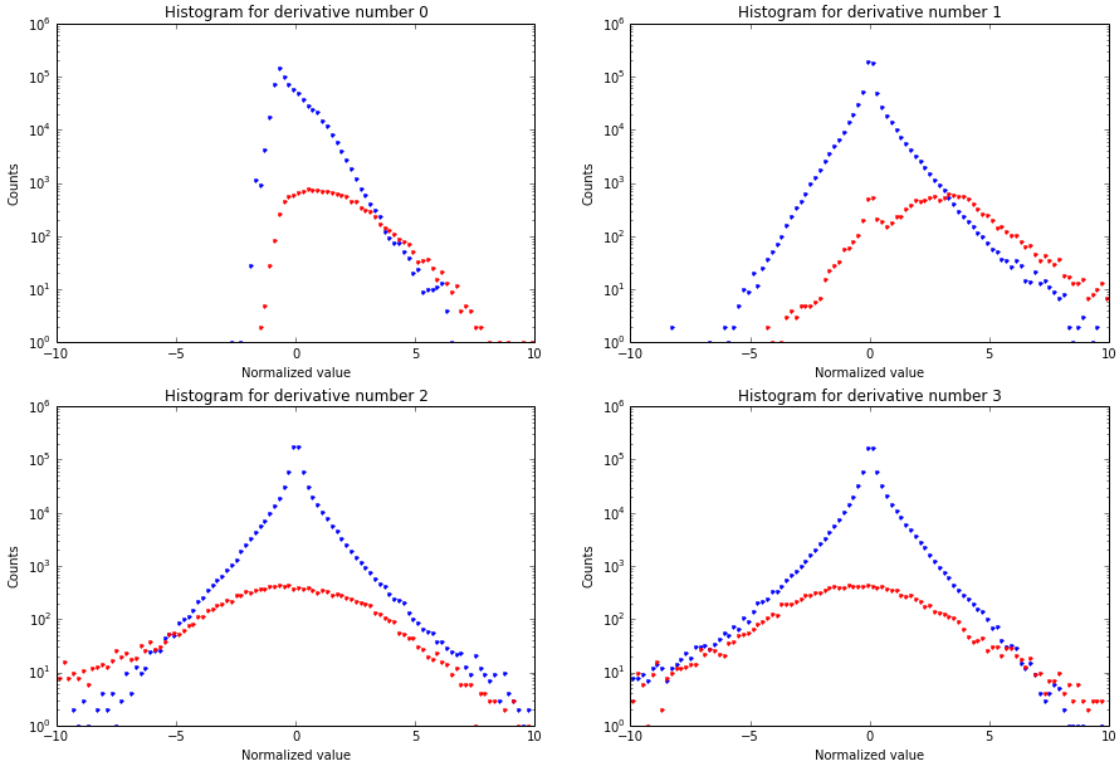
Below are some plots demonstrating the difficulty of classifying the 00 transition against the 01 transition. These are histograms of the values of each derivative, separated into 00 and 01.

```
In [6]: bins = np.linspace(-10, 10, 101)
        xs = (bins[:-1] + bins[1:])/2
        X00 = split_X[0]
        X01 = split_X[1]
        pl.figure(figsize=(15,10))
        print xs
        for i, (d00, d01) in enumerate(zip(X00.T, X01.T)):
            hist00 = np.histogram(d00, bins=bins)
            hist01 = np.histogram(d01, bins=bins)
            pl.subplot(2, 2, i+1)
            pl.semilogy(xs, hist00[0], linestyle='None', marker='.', color='blue')
            pl.semilogy(xs, hist01[0], linestyle='None', marker='.', color='red')
            pl.title('Histogram for derivative number ' + str(i))
            pl.xlabel('Normalized value')
```

```
        pl.ylabel('Counts')
    pl.show
```

```
[-9.9 -9.7 -9.5 -9.3 -9.1 -8.9 -8.7 -8.5 -8.3 -8.1 -7.9 -7.7 -7.5 -7.3 -7.1
 -6.9 -6.7 -6.5 -6.3 -6.1 -5.9 -5.7 -5.5 -5.3 -5.1 -4.9 -4.7 -4.5 -4.3 -4.1
 -3.9 -3.7 -3.5 -3.3 -3.1 -2.9 -2.7 -2.5 -2.3 -2.1 -1.9 -1.7 -1.5 -1.3 -1.1
 -0.9 -0.7 -0.5 -0.3 -0.1  0.1  0.3  0.5  0.7  0.9  1.1  1.3  1.5  1.7  1.9
  2.1  2.3  2.5  2.7  2.9  3.1  3.3  3.5  3.7  3.9  4.1  4.3  4.5  4.7  4.9
  5.1  5.3  5.5  5.7  5.9  6.1  6.3  6.5  6.7  6.9  7.1  7.3  7.5  7.7  7.9
  8.1  8.3  8.5  8.7  8.9  9.1  9.3  9.5  9.7  9.9]
```

Out[6]: <function matplotlib.pyplot.show>

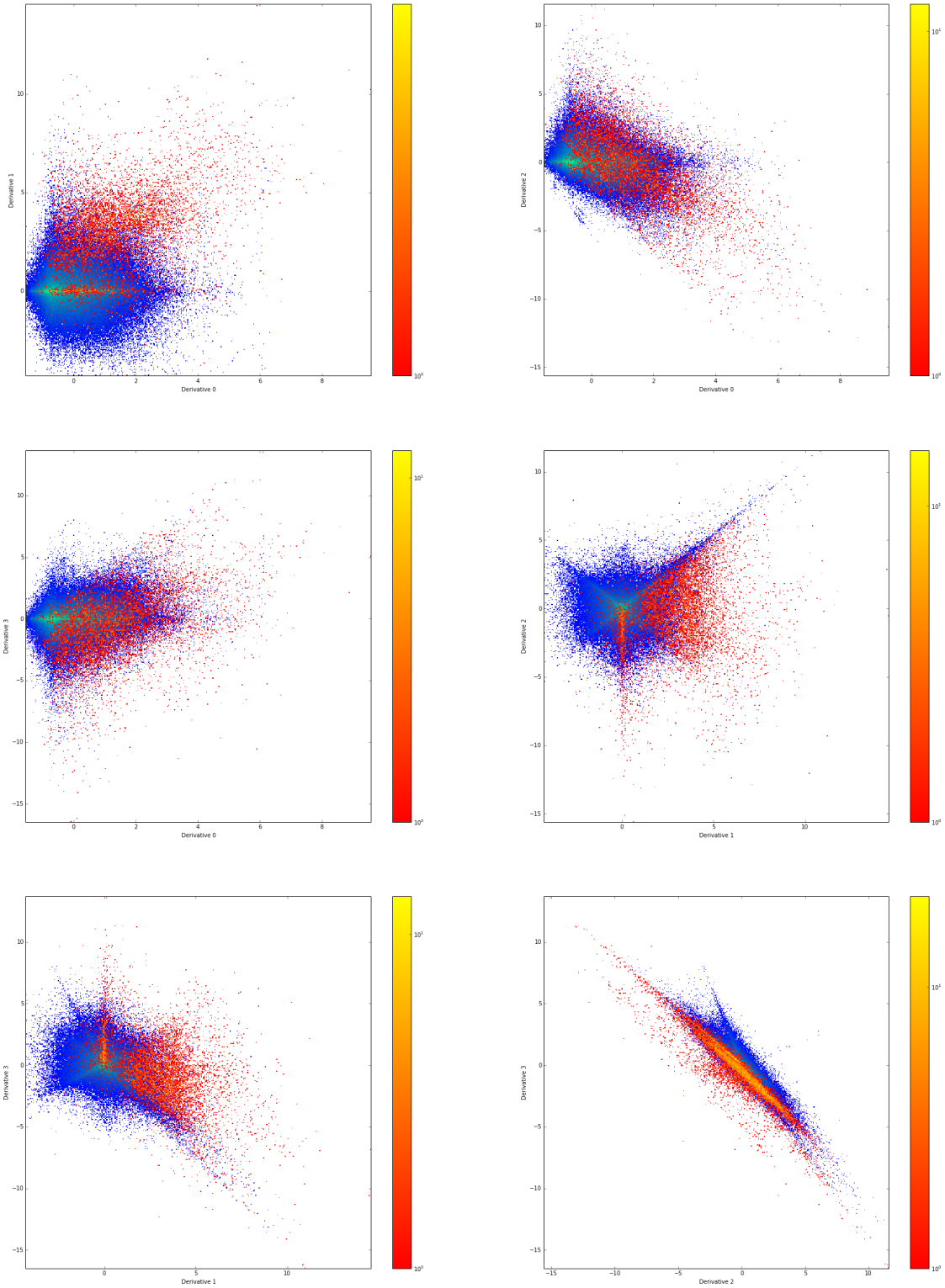


When looking at each derivative in isolation, the most useful for classification seems to be the 1st derivative. This matches intuition, since if an EV is plugged in, the usage should increase rapidly, hence a large value for the first derivative.

Next are some 2D histograms, where the x-axis and y-axis take on all 6 combinations of each pair of derivative. The blue/green values are for 00 transitions, while the red/orange values are for 01 transitions.

In [7]:
```
random.seed(0)

num00 = len(split_X[0])
num01 = len(split_X[1])

num_01_plot = 10000
num_00_plot = (num00 / num01) * num_01_plot

id00 = random.sample(range(num00), num_00_plot)
```

```
id01 = random.sample(range(num01), num_01_plot)

X00 = split_X[0]
X01 = split_X[1]

i=1
pl.figure(figsize=(30, 40))
num_derivs = len(derivatives)
for d1 in range(0, num_derivs-1):
    for d2 in range(d1+1, num_derivs):
        ax = pl.subplot(3, 2, i)
        i += 1
        pl.hist2d(X00.T[d1][id00], X00.T[d2][id00], bins=400, norm=colors.LogNorm(), cmap='winte
        pl.hist2d(X01.T[d1][id01], X01.T[d2][id01], bins=400, norm=colors.LogNorm(), cmap='autu
        #pl.plot(X00.T[d1][id00], X00.T[d2][id00], marker=',', linestyle='None', color='blue')
        #pl.plot(X01.T[d1][id01], X01.T[d2][id01], marker=',', linestyle='None', color='red')
        pl.xlabel('Derivative ' + str(d1))
        pl.ylabel('Derivative ' + str(d2))
        pl.colorbar()
pl.show()
```



There's some interesting patterns here, therefore it might be possible to build a robust classifier. Next part splits data into (0->n) and (1->n) transitions, since each will have its own classifier.

```
In [8]: random.seed(0)

        X0 = train_X[(train_y==0) | (train_y==1)]
        y0 = train_y[(train_y==0) | (train_y==1)]
        train_inds, test_inds = split_data(np.arange(len(y0)), [0.8, 0.2])
        train_X0, train_y0 = X0[train_inds], y0[train_inds]
        test_X0, test_y0 = X0[test_inds], y0[test_inds]

        X1 = train_X[(train_y==2) | (train_y==3)]
        y1 = train_y[(train_y==2) | (train_y==3)]
        train_inds, test_inds = split_data(np.arange(len(y1)), [0.8, 0.2])
        train_X1, train_y1 = X1[train_inds], y1[train_inds]
        test_X1, test_y1 = X1[test_inds], y1[test_inds]
```

Picking a good classifier in this next code block is the most crucial part of this entire section. After playing around with different classifiers for a bit, the Quadratic Discriminant Analysis classifier seemed to work pretty well (76% accuracy for predicting 01 transitions). I think Support Vector Machines could also work pretty well, but when I tried running one it took too long (a trick to making this faster is to compress the data first using "KNN clusters", but I wanted to make something that works before refining it too much).

```
In [9]: from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnaly
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn import svm

        #clf0 = KNeighborsClassifier()
        #clf1 = KNeighborsClassifier()

        clf0 = QuadraticDiscriminantAnalysis(store_covariances=True)
        clf1 = QuadraticDiscriminantAnalysis(store_covariances=True)

        #clf0 = LinearDiscriminantAnalysis()
        #clf1 = LinearDiscriminantAnalysis()

        clf0.fit(train_X0, train_y0)
        clf1.fit(train_X1, train_y1)

Out[9]: QuadraticDiscriminantAnalysis(priors=None, reg_param=0.0,
                        store_covariances=True, tol=0.0001)
```

This function basically gets all the true/false negatives, accuracies, with uncertainties. Note the uncertainties in the counts were assumed to be Poissonian (eg, take square root, since it's just a simple count). The other uncertainties were propagated with partial derivatives.

```
In [10]: # this function was placed in util.py for future reference
         def acc_info(pred, true):

             tn = float(sum((pred == true) & (pred == 0)))
             tp = float(sum((pred == true) & (pred == 1)))
             fn = float(sum((pred != true) & (pred == 0)))
             fp = float(sum((pred != true) & (pred == 1)))
             tn_unc = max(1, np.sqrt(tn))
             tp_unc = max(1, np.sqrt(tp))
             fn_unc = max(1, np.sqrt(fn))
             fp_unc = max(1, np.sqrt(fp))
```

```
    pos_acc = tp / (tp + fn)
    pos_acc_unc = np.sqrt( ((fn/(tp+fn)**2) * tp_unc)**2 + ((tp/(tp+fn)**2) * fn_unc)**2 )

    neg_acc = tn / (tn + fp)
    neg_acc_unc = np.sqrt( ((fp/(tn+fp)**2) * tn_unc)**2 + ((tn/(tn+fp)**2) * fp_unc)**2 )

    t = tp + tn
    t_unc = np.sqrt( tp_unc**2 + tn_unc**2 )

    f = fn + fp
    f_unc = np.sqrt( fn_unc**2 + fp_unc**2 )

    acc = t / (t + f)
    acc_unc = np.sqrt( ((f/(t+f)**2) * t_unc)**2 + ((t/(t+f)**2) * f_unc)**2 )

    info = {}
    info['tn'] = tn
    info['tp'] = tp
    info['fn'] = fn
    info['fp'] = fp
    info['tn_unc'] = tn_unc
    info['tp_unc'] = tp_unc
    info['fn_unc'] = fn_unc
    info['fp_unc'] = fp_unc
    info['pos_acc'] = pos_acc
    info['pos_acc_unc'] = pos_acc_unc
    info['neg_acc'] = neg_acc
    info['neg_acc_unc'] = neg_acc_unc
    info['t'] = t
    info['t_unc'] = t_unc
    info['f'] = f
    info['f_unc'] = f_unc
    info['acc'] = acc
    info['acc_unc'] = acc_unc
    return info
```

Accuracies for the 0->n transitions

```
In [11]: # testing accuracy of 0 -> n transition
         # n=0 is "negative", n=1 is "positive"
         pred_y0 = clf0.predict(test_X0)
         info = acc_info(pred_y0, test_y0)
         print "00 prediction accuracy: %.4f (%.4f)" % (info['pos_acc'], info['pos_acc_unc'])
         print "01 prediction accuracy: %.4f (%.4f)" % (info['neg_acc'], info['neg_acc_unc'])
         print "total accuracy: %.4f (%.4f)" % (info['acc'], info['acc_unc'])

00 prediction accuracy: 0.7441 (0.0086)
01 prediction accuracy: 0.9646 (0.0005)
total accuracy: 0.9606 (0.0005)
```

Accuracies for the 1->n transitions

```
In [12]: # testing accuracy of 1 -> n transition
         pred_y1 = clf1.predict(test_X1)-2
         info = acc_info(pred_y1, test_y1-2)
```

```
        print "01 prediction accuracy: %.4f (%.4f)" % (info['pos_acc'], info['pos_acc_unc'])
        print "11 prediction accuracy: %.4f (%.4f)" % (info['neg_acc'], info['neg_acc_unc'])
        print "total accuracy: %.4f (%.4f)" % (info['acc'], info['acc_unc'])
```

```
01 prediction accuracy: 0.8814 (0.0034)
11 prediction accuracy: 0.8459 (0.0073)
total accuracy: 0.8739 (0.0031)
```

This is the function the Markov Model will call in order to calculate transition probabilities. Note the use of `predict_proba` from the scikit-learn classifier (as opposed to just `predict`).

```
In [13]: def transition_prob(x, initial_y, final_y):
             x = [x]
             if (initial_y, final_y) == (0,0):
                 p = clf0.predict_proba(x)[0][0]
             elif (initial_y, final_y) == (0,1):
                 p = clf0.predict_proba(x)[0][1]
             elif (initial_y, final_y) == (1,0):
                 p = clf1.predict_proba(x)[0][0]
             else:
                 p = clf1.predict_proba(x)[0][1]
             return p
```

1.3 Testing the accuracy of Markov Models

This function runs the Markov Model. Since most timesteps are not charging, the initial probabilities were set to $P(0_{t=0}) = 1$, $P(1_{t=0}) = 0$.

```
In [14]: def run_MC(house):

             p0s = [1.0]
             p1s = [0.0]

             p0 = 1.0
             p1 = 0.0
             #for x in zip(d0[h][1:], d1[h][1:], d2[h][1:], d3[h][1:]):
             for x in np.array([d[house] for d in derivatives]).T[1:]:
                 p0 = p0 * transition_prob(x, 0, 0) + p1 * transition_prob(x, 1, 0)
                 p1 = p0 * transition_prob(x, 0, 1) + p1 * transition_prob(x, 1, 1)
                 # normalize (shouldn't drift, but in case it does for some reason)
                 s = p0 + p1
                 p0 = p0 / s
                 p1 = p1 / s

                 p0s.append(p0)
                 p1s.append(p1)
             return [np.array(p0s), np.array(p1s)]
```

```
In [15]: def MC_acc_info(house):
             true_values = when_charging[house]
             (prob_0, prob_1) = run_MC(house)
             pred = (prob_0 < 0.5).astype(float)
             info = acc_info(pred, true_values)
             info['true'] = true_values
             info['pred'] = pred
```

```
        info['p0'] = prob_0
        info['p1'] = prob_1
        return info
```

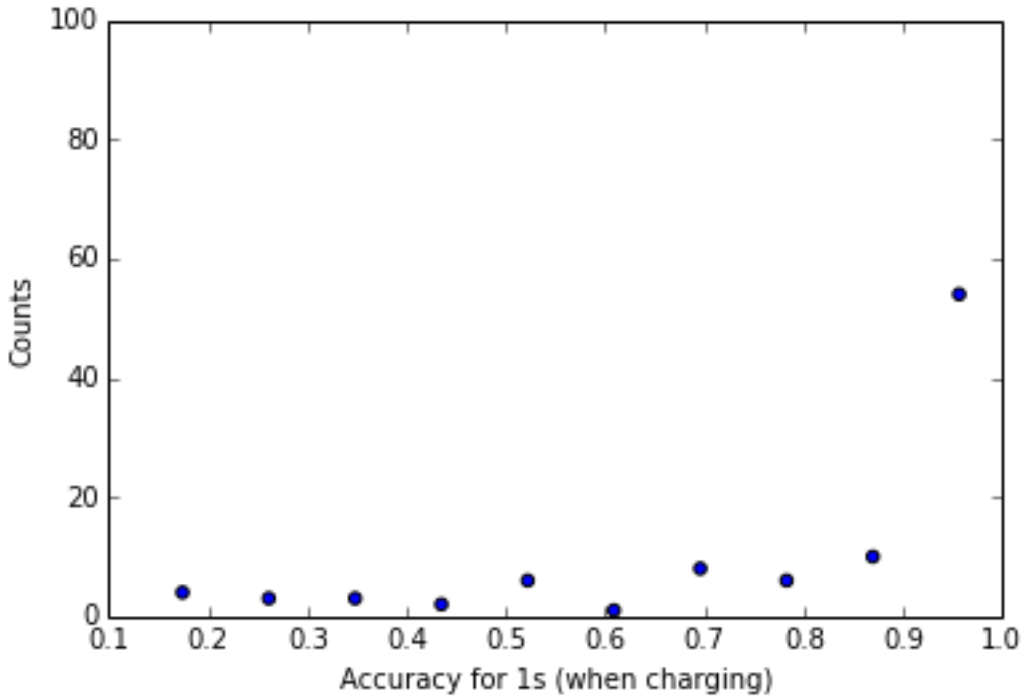
Gets the result of running the Markov Model for each house individually. This part takes a while on my computer (maybe around 5 minutes).

In [16]:
```
MC_infos = {}
i = 0
for h in test_markov:
    #print "currently on: " + str(i)
    i += 1
    MC_infos[h] = MC_acc_info(h)
```

Next up are some plots/averages for the accuracy of the Markov Model. What's interesting is the distribution of accuracies among different houses - some houses are consistently more accurate than others.

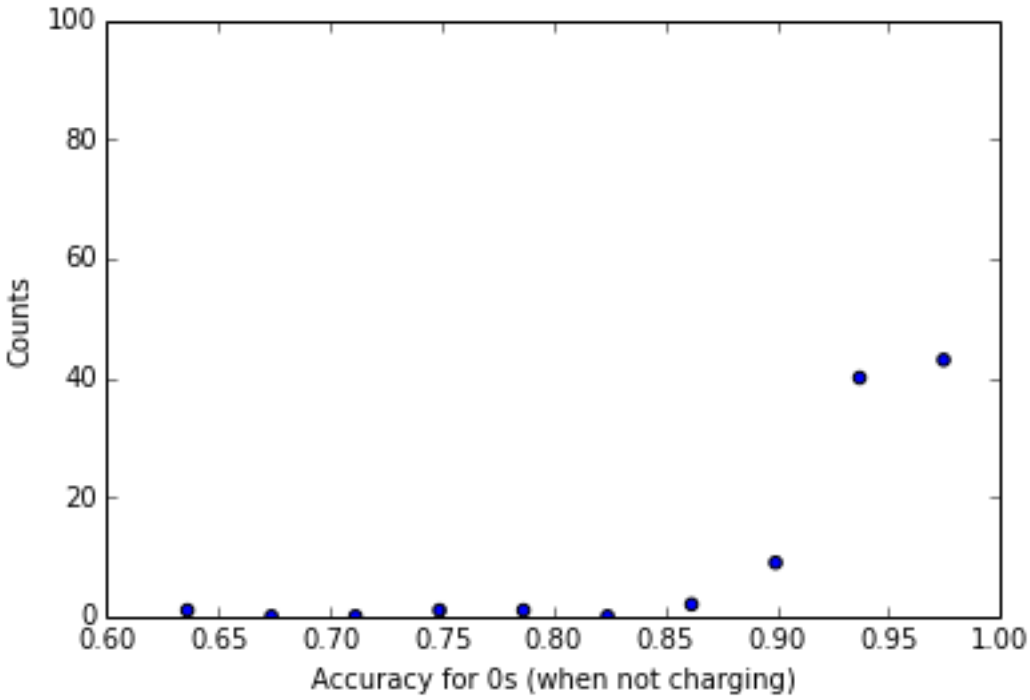
In [17]:
```
neg_accs = []
neg_acc_uncs = []
pos_accs = []
pos_acc_uncs = []
accs = []
acc_uncs = []
for info in MC_infos.values():
    pos_accs.append(info['pos_acc'])
    pos_acc_uncs.append(info['pos_acc_unc'])
    neg_accs.append(info['neg_acc'])
    neg_acc_uncs.append(info['neg_acc_unc'])
    accs.append(info['acc'])
    acc_uncs.append(info['acc_unc'])
```

In [18]:
```
h = np.histogram(pos_accs)
xs = (h[1][1:] + h[1][:-1]) / 2
ys = h[0]
pl.scatter(xs, ys)
pl.xlabel('Accuracy for 1s (when charging)')
pl.ylabel('Counts')
pl.ylim([0, 100])
pl.show()

avg = np.average(np.array(pos_accs))
unc = np.std(np.array(pos_accs)) / np.sqrt(len(pos_accs))
print "average accuracy for 1s: %.4f (%.4f)" % (avg, unc)
```



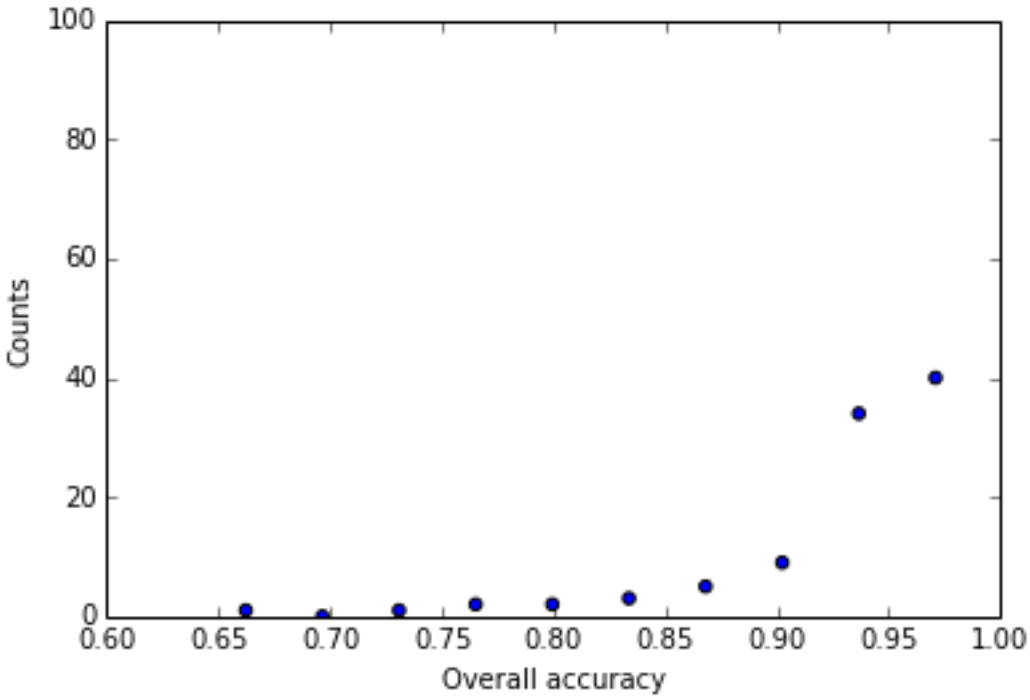

```
average accuracy for 1s: 0.8175 (0.0251)
```

```
In [19]: h = np.histogram(neg_accs)
         xs = (h[1][1:] + h[1][:-1]) / 2
         ys = h[0]
         pl.scatter(xs, ys)
         pl.xlabel('Accuracy for 0s (when not charging)')
         pl.ylabel('Counts')
         pl.ylim([0, 100])
         pl.show()

         avg = np.average(np.array(neg_accs))
         unc = np.std(np.array(neg_accs)) / np.sqrt(len(neg_accs))
         print "average accuracy for 0s: %.4f (%.4f)" % (avg, unc)
```




```
average accuracy for 0s: 0.9432 (0.0052)
```

```
In [20]: h = np.histogram(accs)
         xs = (h[1][1:] + h[1][:-1]) / 2
         ys = h[0]
         pl.scatter(xs, ys)
         pl.xlabel('Overall accuracy')
         pl.ylabel('Counts')
         pl.ylim([0, 100])
         pl.show()

         avg = np.average(np.array(accs))
         unc = np.std(np.array(accs)) / np.sqrt(len(accs))
         print "average accuracy overall: %.4f (%.4f)" % (avg, unc)
```

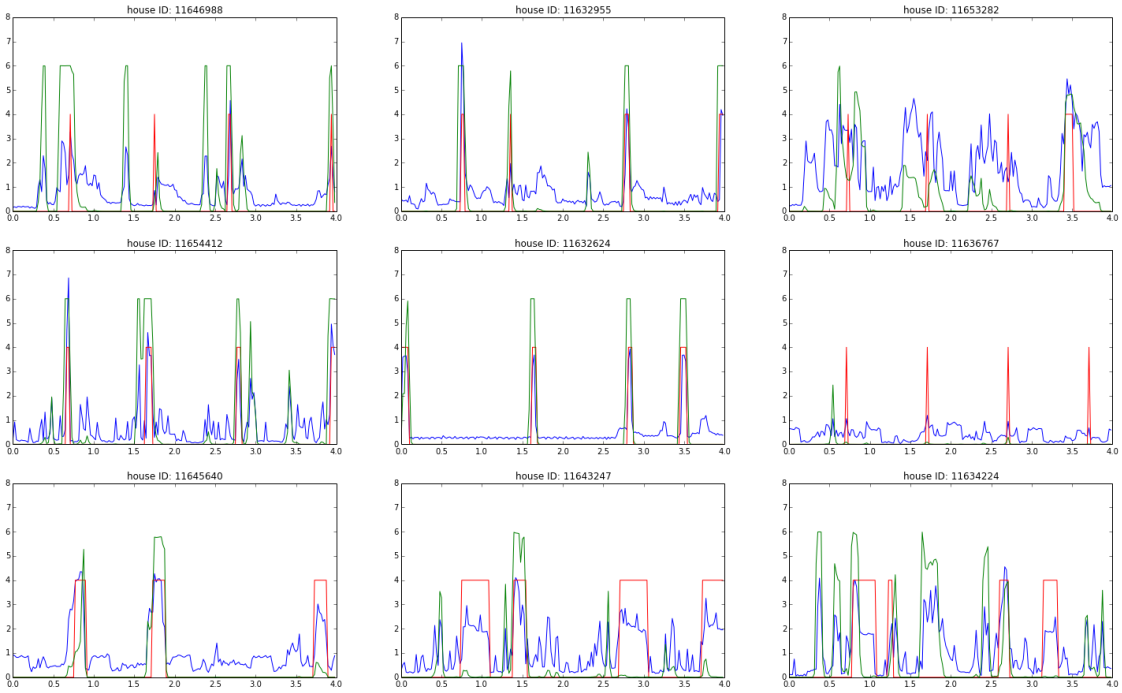



```
average accuracy overall: 0.9299 (0.0061)
```

Finally, some usage plots. The green spikes represent the probability of charging inside the Markov Model, while the red spikes are when an EV is actually charging. It's interesting to see how the Markov Model will follow spikes, even if it wasn't caused by a charge, and how the Markov Model will miss a charge if there isn't a corresponding a spike.

(Note the "usages" aren't true usages, rather the normalized usages from before).

```
In [21]: random.seed(0)
         small_interval = range(192)
         grid_size = 3
         xs = (original_usages.index.values / 48.)[small_interval]
         test_house_sample = random.sample(test_markov, grid_size**2)
         f = pl.figure(figsize=(25, 15), dpi=80)
         for i,house_id in enumerate(test_house_sample):
             curr_usage = original_usages[house_id][small_interval]
             curr_charging_data = when_charging[house_id][small_interval]
             curr_preds = MC_infos[house_id]['p1']
             pl.subplot(grid_size, grid_size, i+1) # subplots have 1-based indexing
             pl.plot(xs,curr_usage, color='blue')
             pl.plot(xs,curr_charging_data*4, color='red')
             pl.plot(xs,curr_preds[small_interval] * 6, color='green')
             pl.title('house ID: ' + str(house_id))
             #pl.xlabel('day')
             #pl.ylabel('Usage (kWh) / 30-minute interval')
             pl.ylim([0, 8])
         pl.show()
```



1.4 Conclusions

Overall, the Markov Model is accurate to about 9.0%. There are a few ways the Markov Model could be improved: - Use a higher-order Markov Model (eg, include the derivatives at the preceding time, or the probabilities going back further than one timestep) - Improve the classifier for calculating transition probabilities (eg, SVMs, neural networks, etc.).