Part III - Predicting Which Houses Own EVs

September 24, 2016

1 Predicting Which Houses Own EVs

The goal of this section is the following: Given the power consumption, is it possible to predict which houses own EVs?

This section takes two main approaches: - In part a, predictions are made using aggregate statistics - In part b, predictions are made using the Markov Model from Part II

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

import util

# 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()[0:1

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

# Series where index = house ids, values = Bool
houses_with_ev = houses[when_charging.sum() > 0]
houses_without_ev = houses[when_charging.sum() == 0]
```

1.1 Part III a - Using Aggregate Statistics

As noted in Part I, houses with EVs tend to use more power on average, and also tend to have higher variances in power consumption. This section elaborates by looking for similar aggregate statistics, but with a focus on separating houses with EVs from those without EVs as reliably as possible. Much of this section is experimental and guided by intuition.

First split into train/test data

In [2]:
```
random.seed(0)
train_houses, validate_houses, test_houses = util.split_data(houses, ratios=[0.6, 0.2, 0.2])
```

Next, get some statistics for each house. The main statistics here are averages of each derivative, weighted by some polynomial k. The statistics are also split by normalization, to see if it helps or not.

In [3]:
```
derivatives = util.derivatives(usages)

normed_usages = util.normalize_usages(usages)
normed_derivatives = util.derivatives(normed_usages)

house_stats = pd.DataFrame(index = houses)
house_stats['owns_ev'] = when_charging.sum() > 0
house_stats['avg'] = usages.apply(np.average)
house_stats['std'] = usages.apply(np.std)
for k in range(1, 6):
    for i,d in enumerate(derivatives):
        key = "d%i_k%i" % (i, k)
        values = d.apply(abs).apply(lambda x: np.power(x, k)).apply(np.average).apply(lambda x:
        house_stats[key] = values
for k in range(1, 6):
    for i,d in enumerate(normed_derivatives):
        key = "normed_d%i_k%i" % (i, k)
        values = d.apply(abs).apply(lambda x: np.power(x, k)).apply(np.average).apply(lambda x:
        house_stats[key] = values
```

In [4]:
```
for (feature, values) in house_stats.iteritems():
    if feature != 'owns_ev':
        values = (values - np.mean(values)) / np.std(values)
        house_stats[feature] = values
```

Out of all the features generated above, see which statistic is the most accurate using 1D KNN.

In [5]:
```
from sklearn.neighbors import KNeighborsClassifier

accs_1D = {}
stats = house_stats.columns[1:] # take out the 'owns_ev' column
for stat in stats:

    X = house_stats.T[train_houses].T[stat].values
    y = house_stats.T[train_houses].T['owns_ev'].values.astype('int')

    clf = KNeighborsClassifier(n_neighbors=20)
    clf.fit(np.array([X]).T, y)

    X = house_stats.T[validate_houses].T[stat].values
    y = house_stats.T[validate_houses].T['owns_ev'].values.astype('int')

    pred = clf.predict(np.array([X]).T)

    accs_1D[stat] = util.acc_info(pred, y)

accs_1D = [(k,v) for k,v in accs_1D.iteritems()]
accs_1D.sort(key = lambda (k,v): -v['acc'])
```

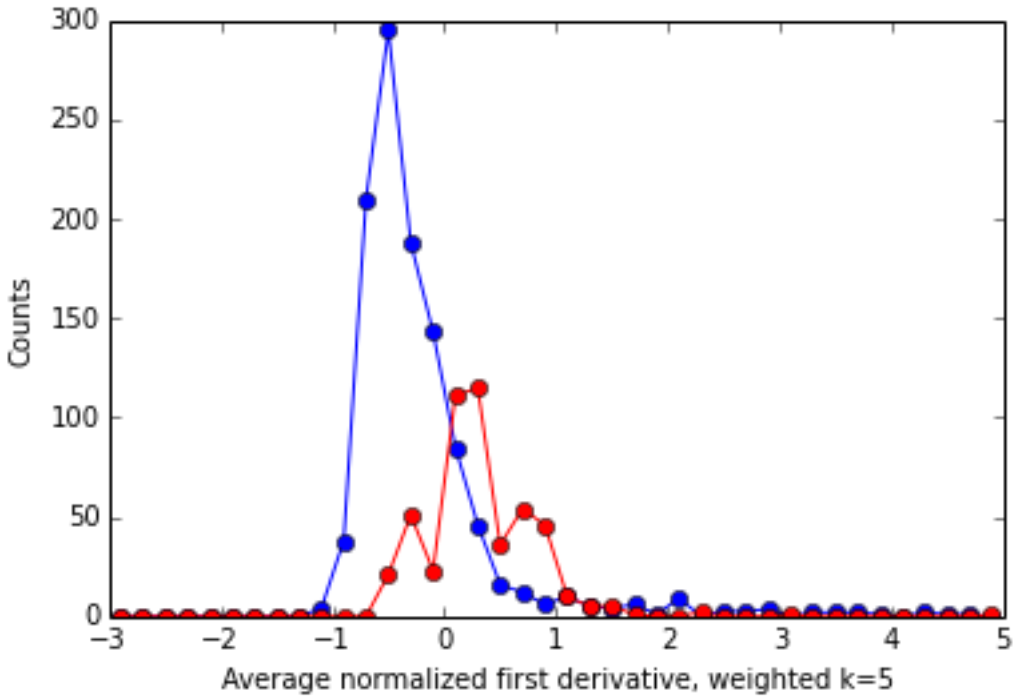
```
In [6]: for (k,v) in accs_1D[:10]:
            print "key: %s, acc: %.3f (%.3f), pos acc: %.3f (%.3f), neg acc: %.3f (%.3f)" % (k, v['acc'
```

```
key: d1_k5, acc: 0.840 (0.021), pos acc: 0.724 (0.045), neg acc: 0.891 (0.021)
key: d1_k4, acc: 0.827 (0.021), pos acc: 0.684 (0.047), neg acc: 0.891 (0.021)
key: d3_k5, acc: 0.818 (0.022), pos acc: 0.704 (0.046), neg acc: 0.868 (0.023)
key: d2_k5, acc: 0.805 (0.022), pos acc: 0.684 (0.047), neg acc: 0.859 (0.023)
key: d1_k3, acc: 0.802 (0.022), pos acc: 0.643 (0.048), neg acc: 0.873 (0.022)
key: d3_k4, acc: 0.789 (0.023), pos acc: 0.582 (0.050), neg acc: 0.882 (0.022)
key: d1_k2, acc: 0.767 (0.024), pos acc: 0.582 (0.050), neg acc: 0.850 (0.024)
key: d0_k5, acc: 0.764 (0.024), pos acc: 0.592 (0.050), neg acc: 0.841 (0.025)
key: d2_k3, acc: 0.761 (0.024), pos acc: 0.643 (0.048), neg acc: 0.814 (0.026)
key: d2_k4, acc: 0.755 (0.024), pos acc: 0.571 (0.050), neg acc: 0.836 (0.025)
```

```
In [7]: X = house_stats.T[train_houses].T['d1_k5'].values
        y = house_stats.T[train_houses].T['owns_ev'].values.astype('int')
        clf = KNeighborsClassifier(n_neighbors=20)
        clf.fit(np.array([X]).T, y)
        X = house_stats.T[test_houses].T['d1_k5'].values
        y = house_stats.T[test_houses].T['owns_ev'].values.astype('int')
        pred = clf.predict(np.array([X]).T)
        acc = util.acc_info(pred, y)
        print util.acc_str(acc)
```

```
acc: 0.8365 (0.0207), pos acc: 0.6429 (0.0523), neg acc: 0.9060 (0.0191)
```

```
In [8]: bins = np.linspace(-3, 5, 41)
        xs = (bins[1:] + bins[:-1])/2
        h0 = np.histogram(house_stats[house_stats['owns_ev']]['d1_k5'], bins=bins)
        h1 = np.histogram(house_stats[~house_stats['owns_ev']]['d1_k5'], bins=bins)
        pl.plot(xs, h1[0], marker='o', color='blue', label='No EV')
        pl.plot(xs, h0[0], marker='o', color='red', label='Owns EV')
        pl.xlabel('Average normalized first derivative, weighted k=5')
        pl.ylabel('Counts')
        pl.show()
```



It would appear that high k-values provide the most robust classifiers. It would also appear that first and second derivatives provide the most information.

Next, see which combination of statistics is most accurate using 2D KNN.

```
In [9]: from sklearn.neighbors import KNeighborsClassifier

        accs_2D = {}
        stats = house_stats.columns[1:] # take out the 'owns_ev' column
        for s1 in range(0, len(stats)-1):
            for s2 in range(s1, len(stats)):
                stat1 = stats[s1]
                stat2 = stats[s2]

                X = zip(house_stats.T[train_houses].T[stats[s1]].values, house_stats.T[train_houses].T[s
                y = house_stats.T[train_houses].T['owns_ev'].values.astype('int')

                clf = KNeighborsClassifier()
                clf.fit(X, y)

                X = zip(house_stats.T[validate_houses].T[stats[s1]].values, house_stats.T[validate_house
                y = house_stats.T[validate_houses].T['owns_ev'].values.astype('int')

                pred = clf.predict(X)

                accs_2D[(stat1,stat2)] = util.acc_info(pred, y)

        accs_2D = [(k,v) for k,v in accs_2D.iteritems()]
        accs_2D.sort(key = lambda (k,v): -v['acc'])
```

```
In [10]: for (k,v) in accs_2D[:10]:
             print "key: %s, acc: %.3f (%.3f), pos acc: %.3f (%.3f), neg acc: %.3f (%.3f)" % (k, v['acc
```

```
key: ('d3_k4', 'd2_k5'), acc: 0.874 (0.019), pos acc: 0.714 (0.046), neg acc: 0.945 (0.015)
key: ('d1_k3', 'd1_k5'), acc: 0.865 (0.019), pos acc: 0.765 (0.043), neg acc: 0.909 (0.019)
key: ('d1_k4', 'd1_k5'), acc: 0.862 (0.019), pos acc: 0.796 (0.041), neg acc: 0.891 (0.021)
key: ('d1_k3', 'd1_k4'), acc: 0.852 (0.020), pos acc: 0.745 (0.044), neg acc: 0.900 (0.020)
key: ('d2_k1', 'd1_k5'), acc: 0.852 (0.020), pos acc: 0.724 (0.045), neg acc: 0.909 (0.019)
key: ('d1_k1', 'd1_k3'), acc: 0.852 (0.020), pos acc: 0.724 (0.045), neg acc: 0.909 (0.019)
key: ('d1_k5', 'normed_d2_k2'), acc: 0.852 (0.020), pos acc: 0.776 (0.042), neg acc: 0.886 (0.021)
key: ('d3_k1', 'd1_k4'), acc: 0.852 (0.020), pos acc: 0.735 (0.045), neg acc: 0.905 (0.020)
key: ('d2_k1', 'd1_k3'), acc: 0.852 (0.020), pos acc: 0.724 (0.045), neg acc: 0.909 (0.019)
key: ('d2_k5', 'd3_k5'), acc: 0.852 (0.020), pos acc: 0.765 (0.043), neg acc: 0.891 (0.021)
```

```
In [11]: X = zip(house_stats.T[train_houses].T['d3_k4'].values, house_stats.T[train_houses].T['d2_k5'].
         y = house_stats.T[train_houses].T['owns_ev'].values.astype('int')
         clf = KNeighborsClassifier()
         clf.fit(X,y)
         X = zip(house_stats.T[test_houses].T['d3_k4'].values, house_stats.T[test_houses].T['d2_k5'].va
         y = house_stats.T[test_houses].T['owns_ev'].values.astype('int')
         pred = clf.predict(X)
         acc = util.acc_info(pred, y)
         print util.acc_str(acc)
```

```
acc: 0.8145 (0.0218), pos acc: 0.5476 (0.0543), neg acc: 0.9103 (0.0187)
```

After doing cross-validation, it appears the accuracy of using 2D is only around 81%. It turns out 1D KNN is better than any pair of 2D KNN!

1.2 Part b - Using the Markov Model

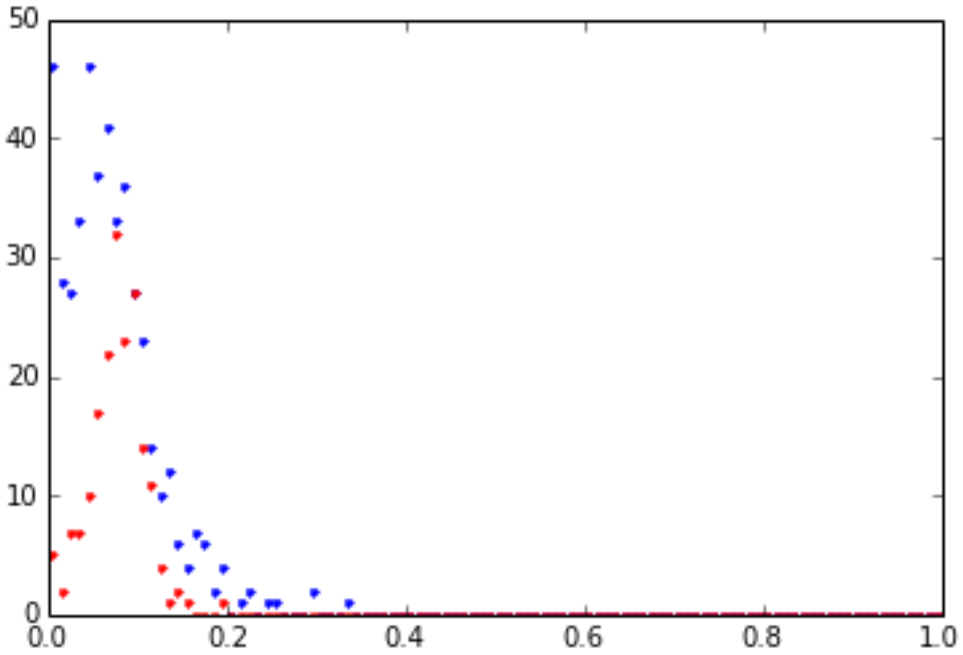
The Markov Model from Part II was put into a file `markov.py`, which provides an easy interface for training and running.

```
In [12]: from markov import EVMarkovModel

         random.seed(0)
         # want to only train the Markov Model with houses that own EVs, while also keeping the ratio (
         train_markov, train_classifier, test_classifier = util.split_data(houses, ratios=[0.4, 0.4, 0.1
         run_markov = train_classifier.append(test_classifier)
         #train_markov, train_classifier, test = util.split_data(houses, ratios=[0.4, 0.4, 0.2])

         # train the markov model transition probabilities
         model = EVMarkovModel(usages, when_charging)
         model.train(train_markov, normed=True)

         # run the markov model on houses to be classified (this part takes a while)
         preds = pd.DataFrame()
         probs = pd.DataFrame()
         for i,house in enumerate(run_markov):
             #print "house %i out of %i" % (i+1, len(run_markov))
             (p0, p1) = model.run(house)
             probs[house] = p1
             preds[house] = (p1 > 0.5).astype('float')
```

Now that the probabilities of charging are known at each timestep, it should be possible to perform aggregate statistics to see which houses own EVs. For example, houses with EVs should report more charging than houses without EVs. The plot below looks at a histogram of the average fraction of time the Markov Chain reported charging, separated into houses with or without EVs.

In [13]:
```
avg_probs_with_ev = probs[train_classifier.intersection(houses_with_ev)].apply(np.average)
avg_probs_without_ev = probs[train_classifier.intersection(houses_without_ev)].apply(np.averag

bins = np.linspace(0, 1, 101)
xs = (bins[1:] + bins[:-1])/2
ys1 = np.histogram(avg_probs_with_ev, bins=bins)[0]
ys2 = np.histogram(avg_probs_without_ev, bins=bins)[0]
pl.plot(xs, ys2, linestyle='None', marker='.', color='blue')
pl.plot(xs, ys1, linestyle='None', marker='.', color='red')
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0xae00eb4c>]
```



Although houses with EVs do report more charging, it doesn't seem signficant enough for an accurate classifier. Below is an attempt at classification using this statistic with 1D KNN, but it is not as accurate as the aggregate models used in part a.

In [14]:
```
from sklearn.neighbors import KNeighborsClassifier

X = preds[train_classifier].apply(np.sum).values
y = (when_charging[train_classifier].sum() > 0)

clf = KNeighborsClassifier()
clf.fit(np.array([X]).T, y)

X = preds[test_classifier].apply(np.sum).values
y = (when_charging[test_classifier].sum() > 0)
```

```
        pred = clf.predict(np.array([X]).T)

        accs = util.acc_info(pred, y)

        print "acc: %.3f (%.3f), pos acc: %.3f (%.3f), neg acc: %.3f (%.3f)" % (accs['acc'], accs['acc
```

```
acc: 0.642 (0.027), pos acc: 0.330 (0.048), neg acc: 0.772 (0.028)
```

1.3 Conclusions

The most accurate model so far for predicting which houses own EVs has an accuracy of about 84% with an uncertainty of 2%. This was the 1D KNN with first derivative (weighted k=5).

All of the most accurate 1D KNN or 2D KNN pairs seemed to rely on first or second derivatives with high k-values. This implies the tails of the distributions had the most significance. This isn't too surprising, since a house with an EV would have more sudden jumps, hence it would have many timeseries with large first or second derivatives. A different approach which extracts or fits the tails could result in a higher accuracy.

Using the Markov Models were not that accurate, but the Markov Model was only used in one way (average time spent charging). It could be possible to use the results from the Markov Models in a different way, for example looking at distributions for how long each charge lasted (turn on to turn off), or the total number of charge groups, or the frequency of the charges. Real charges might have different behavior, which could still be encoded in the results of the Markov Model.

Finally, combining the two approaches could result in a more accurate multi-dimensional classifier. For example, it might be worth re-running the first 2D KNN with the Markov Model's output as one of the statistics.