Uber Demand prediction challenge

First, let's load our dataset into a usable form. We'll be taking advantage of Pandas `DataFrames` in this Notebook due their great time-series and visualization abilities.

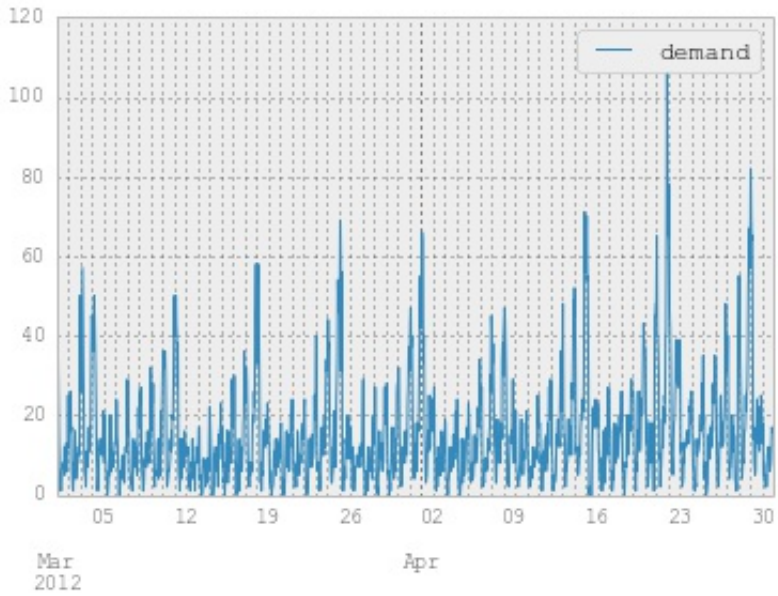
The first thing we do is import numpy and pandas, as well as set some pretty graphing settings. Once we've read in the JSON file as `raw` discrete time-series events, we bucket the data into demand by hour as a `ts` time-series by counting the discrete events within an hour span. Finally, I bring the `TimeSeries` into a `DataFrame` (which will allow us to operform multi-dimensional analysis later) and plot the resulting, binned data.

In [12]:

```
#Set up Matplotlib magic
%matplotlib inline

#Some common data analysis imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
pd.options.display.mpl_style = 'default'


# Data Acquisition
raw = pd.read_json('../data.json', typ='series')
ts = pd.TimeSeries(1, raw).resample('H', how='count')
df = pd.DataFrame(ts.values, index=ts.index, columns=['demand'])
df.plot()
```

Out[12]:

```
<matplotlib.axes.AxesSubplot at 0x10825d290>
```



Before we dive into any statistical or machine learning methods for predicting future data, let's

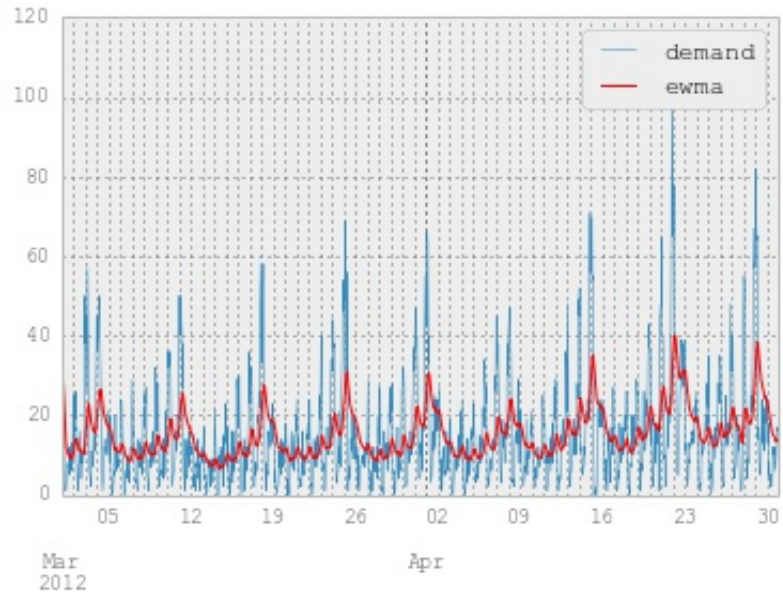
Before we dive into any statistical or machine learning methods for predicting future data, let's take a look at the data we already have. I'm a visual learner, so let's transform the data and see if we can pick out any immediate trends.

In [13]:

```
ax = df['demand'].plot(lw=.5)
pd.ewma(df['demand'], halflife=12).plot(style='r', label='ewma',ax=ax)
plt.legend()
```

Out[13]:

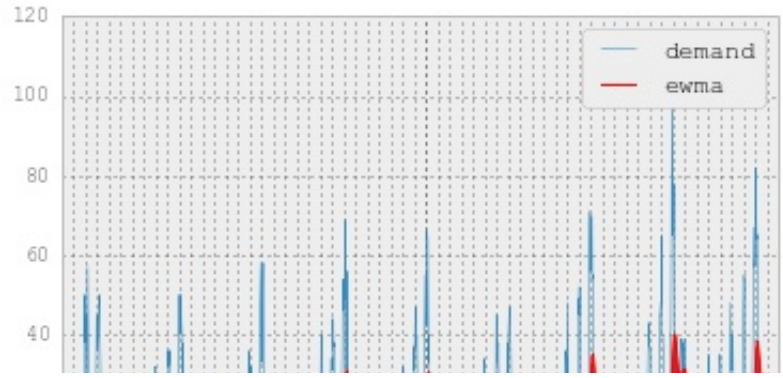
```
<matplotlib.legend.Legend at 0x1083f3950>
```

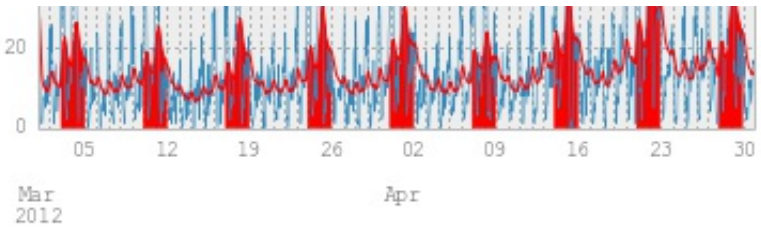


There's a pretty stong cyclic component to the data, and I'd be willing to bet that the peaks are weekend demand. Let's see if that's true.

In [24]:

```
ax = df['demand'].plot(lw=.5)
moving_average = pd.ewma(df['demand'], halflife=12)
moving_average.plot(style='r', label='ewma',ax=ax)
plt.legend()

ymin, ymax = ax.get_ylim()
day = df.index.map(lambda x: x.day)
weekends = df.index.map(lambda x: x.weekday() in [5,6])
ax.fill_between(df.index,moving_average,where=weekends,alpha=1,color='r')
plt.show()
```





So it's pretty clear that the weekends have higher demand, it's likely that the day of the week and the hour is enough for us to make a solid prediction for the future. Let's expore that.

We'll use a support vector machine, trained on the hour and day of the week as inputs. While we have day and hour in continuous values, SVMs will train more accurately on highly-dimensional binary data. For that, we will encode the discrete `hour` and `weekday` values using a `OneHotEncoder`.

The `extract_date_feature` function creates appropriately encoded input data, along with an encoder that we can use later in `vectorize_date_feature` to automatically pre-process input data in future predictions.

In [15]:

```
# Imports
from sklearn import svm
from sklearn import grid_search
from sklearn import metrics
from sklearn import preprocessing
from sklearn import cross_validation

# Preprocessing
def extract_date_feature(df):
    df['hour'] = df.index.map(lambda idx: idx.hour)
    df['weekday'] = df.index.map(lambda idx: idx.weekday())

    encoder = preprocessing.OneHotEncoder()
    encoder.fit_transform(df[['hour','weekday']])

    encoded = pd.concat([pd.get_dummies(df[col], prefix=col) for col in df.c
olumns if not col in ['demand']], axis=1)
    return (pd.concat([df, encoded], axis=1), encoder)

def vectorize_date_feature(date, encoder):

    datestamp = pd.to_datetime(date)
    return encoder.transform([[datestamp.hour,datestamp.weekday()]]).toarray
()[0]

# Encode the continuous data
dataset, encoder = extract_date_feature(df)

# We only want to train on encoded data
X_cols = [col for col in dataset.columns if not col in [u'demand', u'hour',
u'weekday']]
y_cols = 'demand'
```

Once we have input data in a format that is ready to feed into our regressor, we split up the data

into train and test datasets. Since we're doing time-series prediction, we'll attempt to simulate the problem of predicting future data by limiting our view to the first half of the dataset.

In order to optimize our regressor, we'll perform a grid search of the model's hyperparameters. This trains the model on a set of hyperparameter combinations and find the mose accurate model. We can access this optimized model via the `best_estimator_` property.

In [16]:

```
# Split the dataset into train and test sets
split = int(dataset.shape[0] * 0.5)

# Get the test and training sets into a format that scikit-learn likes
X, y = dataset[X_cols], dataset[y_cols]
(X_train, X_test, y_train, y_test,) = X[0:split], X[split:], y[0:split], y[s
plit:]
# (X_train, X_test, y_train, y_test,) = cross_validation.train_test_split(X,
y)

params = {
    'C': [1,10,100],
    'kernel': ['rbf','sigmoid'],
    'gamma': [0.0, 0.1],
}

optimizer = grid_search.GridSearchCV(svm.SVR(), param_grid=params, verbose=T
rue)
optimizer.fit(X_train, y_train)
best_params = optimizer.best_params_;
model = optimizer.best_estimator_
```

```
[Parallel(n_jobs=1)]: Done   1 jobs       | elapsed:    0.0s
[Parallel(n_jobs=1)]: Done  36 out of  36 | elapsed:    1.5s finished

Fitting 3 folds for each of 12 candidates, totalling 36 fits
```

Now that we've created a trained model with optimized hyperparamaters, we can use it to predict data. Let's see how the regressor does on the unseen 'future' data from the our dataset.

In [17]:

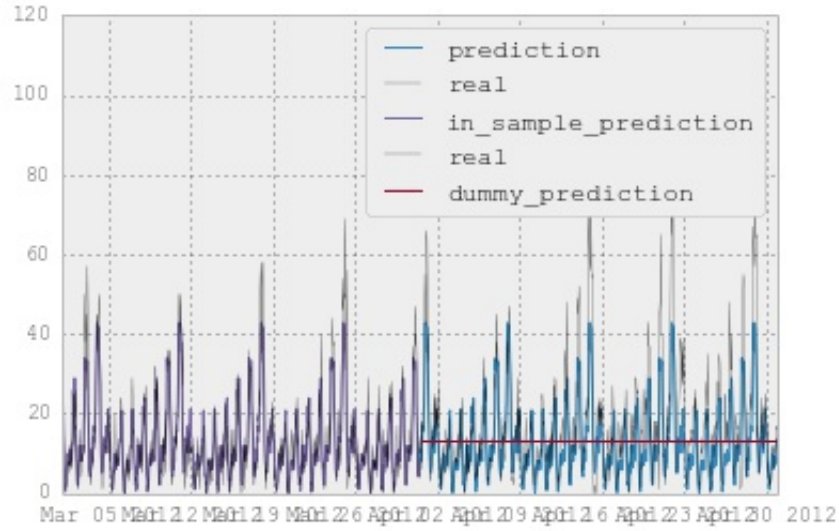
```
# Check the performance of out of sample predictions ( on the test dataset )
prediction = model.predict(X_test)
print("Out of sample MSE:{0}".format(metrics.mean_squared_error(y_test,predi
ction)))
plt.plot(X_test.index, prediction, label='prediction')
plt.plot(y_test.index, y_test, lw=0.2, color='k',label='real')
plt.legend()

# Compare against the performance of the model on the training dataset.
in_sample_prediction = model.predict(X_train)
print("In sample MSE: {0}".format(metrics.mean_squared_error(y_train,in_samp
le_prediction)))
plt.plot(X_train.index, in_sample_prediction, label='in_sample_prediction')
plt.plot(y_train.index, y_train, lw=0.2, color='k',label='real',)
```

```
plt.legend()

# Sanity check, check how much better we're doing than a mean-estimator
from sklearn import dummy
r = dummy.DummyRegressor()
r.fit(X_train,y_train)
dummy_prediction = r.predict(X_test)
print("Dummy MSE: {0}".format(metrics.mean_squared_error(y_test,dummy_predic
tion)))
plt.plot(X_test.index, dummy_prediction, label='dummy_prediction')
plt.legend()
```

```
Out of sample MSE:84.9854124836
In sample MSE: 27.2478339621
Dummy MSE: 226.862365254
```

Out[17]:

```
<matplotlib.legend.Legend at 0x107d95490>
```



So the MSE is OK, but not great. Because we're treating the samples as IID, we're aren't taking advantage of the intrinsic trending nature of the dataset.

Let's try a different method that ignores explicitely learning on the time of day and instead trains directly from historical data. To help visualize the dataset, we'll see what the two-week autocorrelation plot of the data looks like.

In [18]:

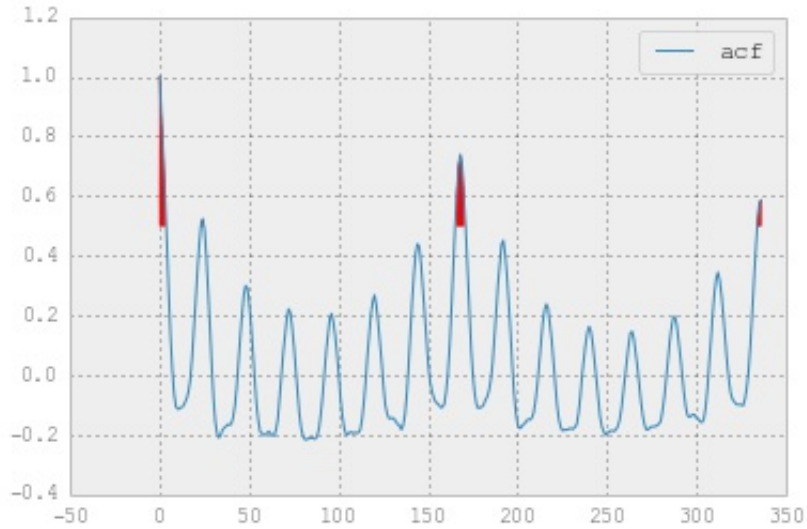
```
# Autocorrelation
import statsmodels.api as sm
from statsmodels.graphics import tsaplots

nlags=14*24
correlation_threshold = .5
acf = sm.tsa.acf(df['demand'], nlags=nlags)
acf = pd.Series(data=acf, index=np.arange(nlags+1), name='acf')
lags = acf[acf >= correlation_threshold].index
print("Lags with highest autocorrelation {l}".format(l=lags))

ax = acf.plot()
ymin, ymax = ax.get_ylim()
```

```
high_acf = acf.index.map(lambda x: x in lags)
ax.fill_between(acf.index,correlation_threshold, acf,where=high_acf,alpha=1,
color='r',label='high acf')
plt.legend()
```

```
Lags with highest autocorrelation Int64Index([0, 1, 2, 3, 23, 24, 166, 167,
168, 169, 170, 334, 335, 336], dtype='int64')
```

Out[18]:

```
<matplotlib.legend.Legend at 0x107c84a10>
```



As we could have guessed, the data has a high autocorrelation with small lags, with smaller peaks one week and two weeks back. We've captured the lags associated with the highest correlation factor and can use those to construct an input dataset later on.

We'll now create some lagged data, where we derive the lags from the autocorrelation values before. The idea is that, given lagged data, we will be able to predict future demand. *Note*: high autocorrelation will not automatically give us the best inputs to train on, but is a good starting point!

In [19]:

```
# Preprocessing
window_size = lags.shape[0]
target_size = 1
input_size = window_size-target_size

shift = pd.concat([df['demand'].shift(n) for n in lags], axis=1).dropna()
target_columns = ['demand_0']
input_columns  = ['demand_{0}'.format(n) for n in xrange(target_size, window
_size)]
shift.columns  = target_columns + input_columns

# Train/Test Split
split = int(shift.shape[0] * .8)
X,y = shift[input_columns], shift[target_columns]
X_train,X_test,y_train,y_test = X[0:split], X[split:], y[0:split], y[split:]
```

Now that we've created some lagged data, let's use another common machine learning model, Artificial Neural Networks, to create a predictor. We've already performed some preprocessing of

the data by generating lagged samples, but ANNs also benefit from scaling the input data before training.

Because the ANN backpropegation takes so long, we pickle the the trained network so that it can be reused. To re-train, remove the `net.pickle` file.

In [20]:

```
# ANN Imports
from pybrain.datasets import SupervisedDataSet, UnsupervisedDataSet
from pybrain.tools.shortcuts import buildNetwork
from pybrain.supervised import BackpropTrainer
from pybrain.structure import TanhLayer
from sklearn import preprocessing

try:
    import cPickle as pickle
except ImportError:
    import pickle

scaler = preprocessing.StandardScaler().fit(df['demand'].values)

try:
    net = pickle.load(open('net.pickle','rb'))
except Exception as e:
    print(e)
    # Build and Train
    ds = SupervisedDataSet(inp=input_size, target=target_size)
    ds.setField('input', scaler.transform(X_train.values))
    ds.setField('target', scaler.transform(y_train.values))

    hidden_size = input_size * 5
    net = buildNetwork( input_size, hidden_size, target_size, bias = True, h
iddenclass=TanhLayer)
    trainer = BackpropTrainer( net, ds )
    errors = trainer.trainUntilConvergence( verbose = True, validationPropor
tion = 0.1, maxEpochs = 1000, continueEpochs = 5 )
    ret = pickle.dump(net, open('net.pickle','wb'))
```

```
[Errno 2] No such file or directory: 'net.pickle'
train-errors: [  1.450011  0.351219  0.223002  0.160609  0.139232  0.123732
0.111233  0.103610  0.100896  0.095955  0.091103  0.092614  0.090886  0.0853
71  0.085404  0.083293  0.080322  0.081471  0.076556  0.079348  0.076092  0.
073791  0.071052  0.074917  0.071358  0.070820  0.071193  0.071143  0.071076
0.068716  0.068917  0.068615  0.068285  0.066512  0.068463  0.066765  0.0650
31  0.066258  0.065989  0.063490  0.065362  0.063688  0.060348  0.062662  0.
061342  0.060582  0.059991  0.059466  0.060250  0.057950  0.059310  0.063061
]
valid-errors: [  11.341764  0.525972  0.353855  0.240270  0.217873  0.224007
0.157787  0.198166  0.178405  0.136138  0.202482  0.116380  0.174815  0.1169
90  0.104187  0.107205  0.113216  0.121140  0.137015  0.107659  0.114651  0.
107539  0.153824  0.123577  0.159957  0.108128  0.113670  0.118221  0.145906
0.164147  0.111195  0.107467  0.108753  0.152430  0.135882  0.137453  0.1238
32  0.110889  0.135328  0.118752  0.144383  0.129467  0.117672  0.114076  0.
123438  0.125203  0.125685  0.127960  0.129745  0.241116  0.153134  0.145481
]
```

Now that we've created and trained a network on our training dataset, we can use that network to predict on our test dataset.

Like in the previous pybrain example, we need to coerce the Pandas `DataFrame` into a pybrain Dataset before the network can consume our data.

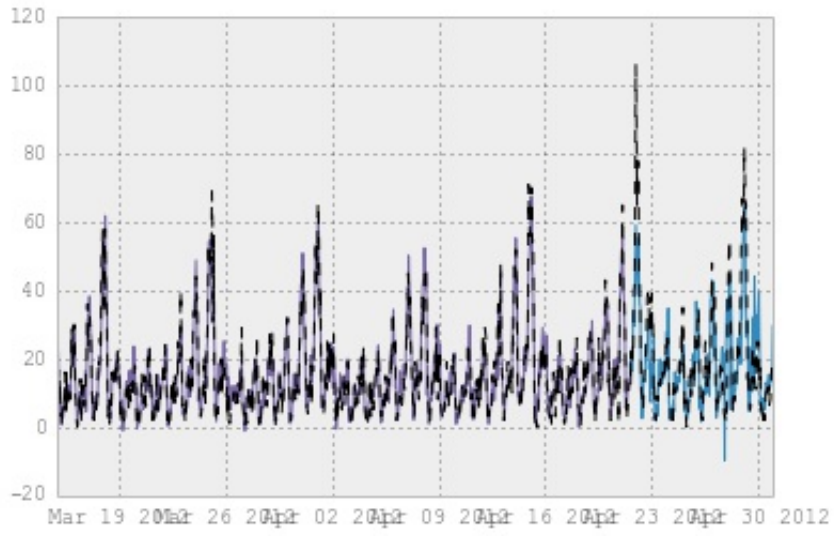
In [25]:

```
# Validate
from pybrain.tools import validation

tds = UnsupervisedDataSet(dim=input_size)
tds.setField('sample', scaler.transform(X_test.values))
prediction = scaler.inverse_transform(net.activateOnDataset(tds))

print("Out of sample MSE: {0}".format(validation.Validator.MSE(prediction, y
_test.values)))
plt.plot(X_test.index, prediction, label='prediction')
plt.plot(y_test.index, y_test, 'k--',label='real',)

tds = UnsupervisedDataSet(dim=input_size)
tds.setField('sample', scaler.transform(X_train.values))
in_sample_prediction = scaler.inverse_transform(net.activateOnDataset(tds))

print("In sample MSE: {0}".format(validation.Validator.MSE(in_sample_predict
ion, y_train.values)))
plt.plot(X_train.index, in_sample_prediction, label='in_sample_prediction')
plt.plot(y_train.index, y_train, 'k--',label='real',)
```

```
Out of sample MSE: 121.975447083
In sample MSE: 23.7175995812
```

Out[25]:

```
[<matplotlib.lines.Line2D at 0x105dbfa10>]
```



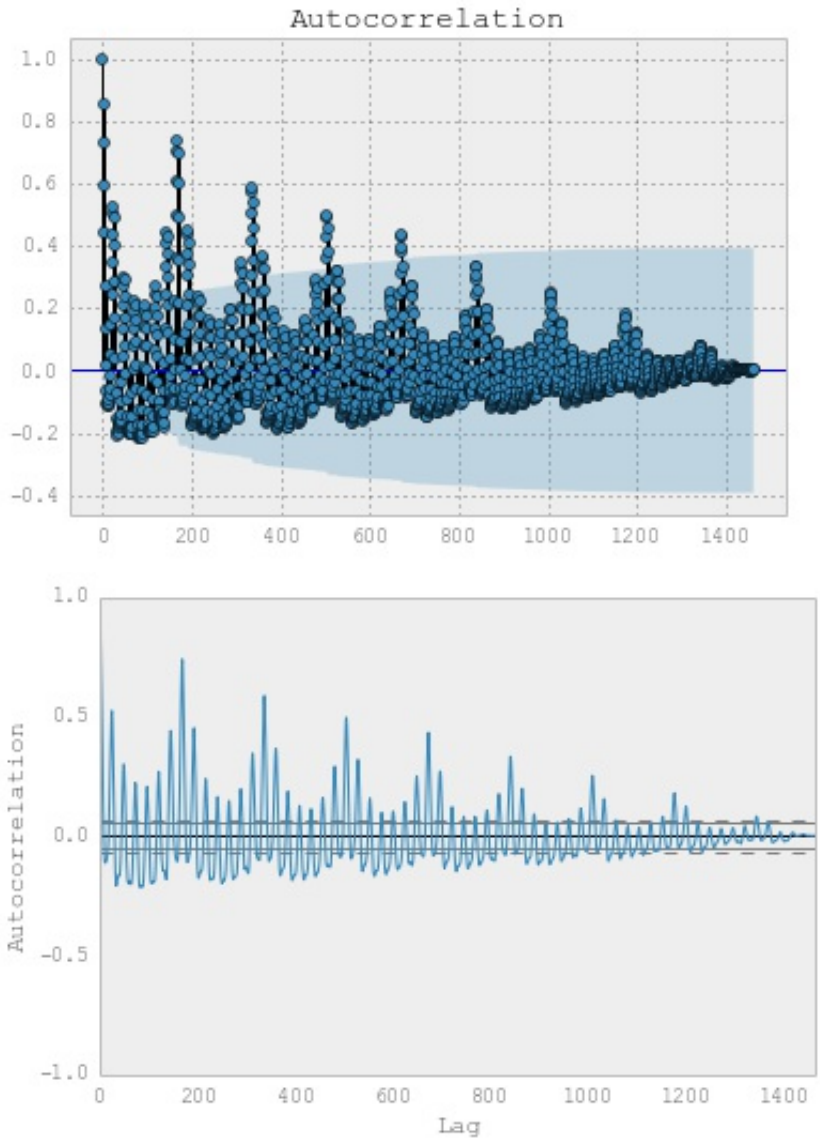
Appendix of investigations that looked promising, but I never finished exploring

In [41]:

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(df['demand'], use_vlines=True, label='acf')

plt.figure();
from pandas.tools.plotting import autocorrelation_plot
autocorrelation_plot(df['demand'])
```

Out[41]:

```
<matplotlib.axes.AxesSubplot at 0x10856b4d0>
```

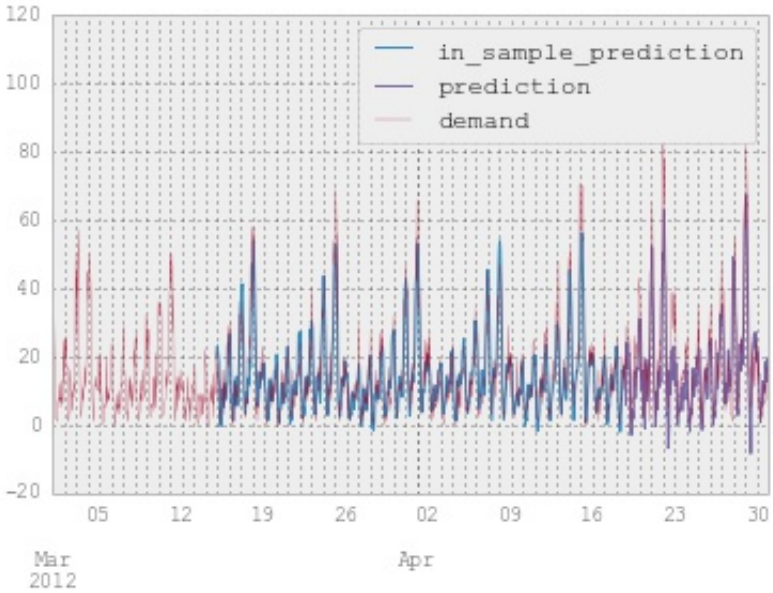


In [38]:

```
import statsmodels.api as sm
from sklearn import metrics

# Train
maxlag=14*24
split = int(df.shape[0] * .8)
train, test = df['demand'][0:split], df['demand'][split:]

model = sm.tsa.AR(endog=train).fit(maxlag=maxlag)

# In sample Prediction
start = train.index[model.k_ar].isoformat()
```

```
start = train.index[-1].isoformat()
end = train.index[-1].isoformat()
rng = pd.date_range(start=start,end=end,freq='H')

in_sample_prediction = model.predict(start=start,end=end,dynamic=True)

print("In sample MSE: {0}".format(metrics.mean_squared_error(df['demand'][rn
g], in_sample_prediction)))
in_sample_prediction.plot(label='in_sample_prediction')

# Out of sample Predict
start = train.index[-1].isoformat()
end = test.index[-1].isoformat()
rng = pd.date_range(start=start,end=end,freq='H')

prediction = model.predict(start=start,end=end,dynamic=True)

print("Out of sample MSE: {0}".format(metrics.mean_squared_error(df['demand'
][rng], prediction)))
prediction.plot(label='prediction')
df['demand'].plot(lw=.2)
plt.legend()
```

```
In sample MSE: 27.0814085951
Out of sample MSE: 99.9878807026
```

Out[38]:

```
<matplotlib.legend.Legend at 0x1091a42d0>
```