1 Introduction

The United States electrical grid produces a lot of power. In order to prevent blackouts, it must make available enough power to meet demand at all times. However, Storing exccess electrical power is often costly and inefficient, and if excess cannot be stored, it is wasted. So, authorities and power companies would like to limit the amount of excess power generated in order to save on storage costs and reduce waste generation. The U.S. Energy Information Agency provides data on energy production and demand as well as forecast data on how much demand is expected to be 24 hours in advance. We will use this information to try to model the electrical demand and attempt to out perform the forecast provided.

To start off, we import several libraries and set some constants needed to pull the data for the E.I.A's website. A couple of different functions were defined to streamline collection of data from different data series on the E.I.A website.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import TimeSeriesSplit
from sklearn import linear_model
from sklearn import metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error
import datetime
from sklearn.tree import DecisionTreeRegressor
from fbprophet import Prophet
from sklearn import metrics
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from scipy.stats import uniform
import copy
import requests
import warnings
warnings.filterwarnings('ignore')
EIA_KEY = 'API_KEY'
DEMAND_CAT_ID = '2122628' #found these IDs manually
FORECAST_CAT_ID = '2122627'
NETGEN_CAT_ID = '2122629'
SERIES_URL = 'http://api.eia.gov/series/'
CAT_URL = 'http://api.eia.gov/category/'
REGIONS = ["US48", "CAL", "CAR", "CENT", "FLA", "MIDA", "MIDW", "NE", "NY", "NW", "SE", "SW",
    "TEN", "TEX"]
```

```python
def get_series_ids(id):
    """returns a list of series ids with some supplementary information from the given category
       id"""
    data_series = []
    params = {'api_key': EIA_KEY, 'category_id': id}
    ses = requests.get(CAT_URL, params = params)
    cat_params = {'api_key': EIA_KEY, 'category_id': ''}
    for cat in ses.json()['category']['childcategories']:
        cat_params['category_id'] = cat['category_id']
        req = requests.get(CAT_URL, params = cat_params)
        for series in req.json()['category']['childseries']:
            if series['series_id'][-1] == 'H':
                name = series['name']
                is_region = False
                if name[name.index('(')+1:name.index(')')] == 'region':
                    is_region = True
                    name = cat['name'][cat['name'].index('(')+1:cat['name'].ind ex(')')]
                else:
                    name = name[name.index('(')+1:name.index(')')]
            data_series.append((series['series_id'], name, is_region, series['units']))
    return data_series
```

```python
def get_series(target_series):
    """returns series data from given series ids and then aggregates all data
    by date and converts the date column from the ISO standard to a python datetime"""
    result = []
    series_params = {'api_key':EIA_KEY}
    for series in target_series:
        series_params['series_id'] = series[0]
        req = requests.get(SERIES_URL, params = series_params)
        result.append(pd.DataFrame(req.json()['series'][0]['data'], columns = ['date', series[1]]))
    m = map(lambda x: x.set_index('date'), result)
    df = pd.concat(m).reset_index().groupby('date').sum()
    type_dir = {}
    for col in df.columns[1:]:
        type_dir[col] = float
    df = df.astype(type_dir)
    df = df.reset_index()
    df['date'] = pd.to_datetime(df['date'])
    return df
```

```python
    demand_df = get_series(get_series_ids(DEMAND_CAT_ID))
    forecast_df = get_series(get_series_ids(FORECAST_CAT_ID))
    net_gen_df = get_series(get_series_ids(NETGEN_CAT_ID))
```

The data is extremely clean, and does not require much in terms of removing or interpolating null values. Which means we can begin exploratory data analysis.

2 EDA

First we look at how well the forecast from the EIA does at predicting demand by looking at its r^2 score.

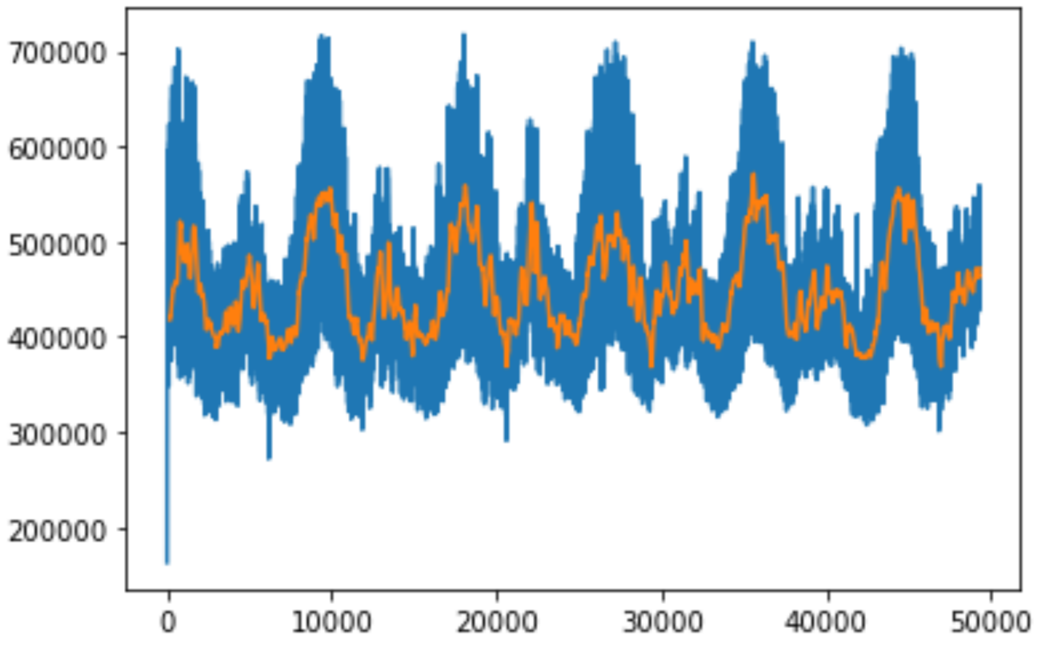
```python
diff = forecast.shape[0] - demand.shape[0] #there is more forecast data than actual demand data
sklearn.metrics.r2_score(demand['US48'], forecast['US48'].iloc[:-diff])
```

Running the above code yields an r^2 score of $0.9509745723874736 \approx 0.951$. This is pretty good, but there is room for improvement on the model. We then use this function to convert the datetime objects into a format that the various models in sklearn can use. That is, we extract the numerical values for the month, day of month, year, etc.

```
def extract_dates(X):
   """Extract numerical values from the date column and add the appropriate columns to the data
       frame"""
   #dates is a dictionary that maps the month to the number of days in the month. It does not
       account for leap years.
   dates = {1:31, 2:28, 3:31, 4:30, 5:31, 6:30, 7:31, 8:31, 9:30, 10:31, 11:30, 12:31}
   X['date'] = pd.to_datetime(X.date)
   X['month'] = X.date.apply(lambda x : x.month)
   X['year'] = X.date.apply(lambda x : x.year)
   X['dayOfMonth'] = X.date.apply(lambda x : x.day)
   X['dayOfWeek'] = X.date.apply(lambda x : x.isoweekday())
   X['hour'] = X.date.apply(lambda x : x.hour)
   X['month.p'] = 2*np.pi*X['month']/12.0
   X['daysInMonth'] = X.month.replace(dates)
   X['dayOfMonth.p'] = 2*np.pi*X['dayOfMonth']/X['daysInMonth']
   X['dayOfWeek.p'] = 2*np.pi*X['dayOfWeek']/7.0
   X['hour.p'] = 2*np.pi*X.hour/24.0
   return X
```

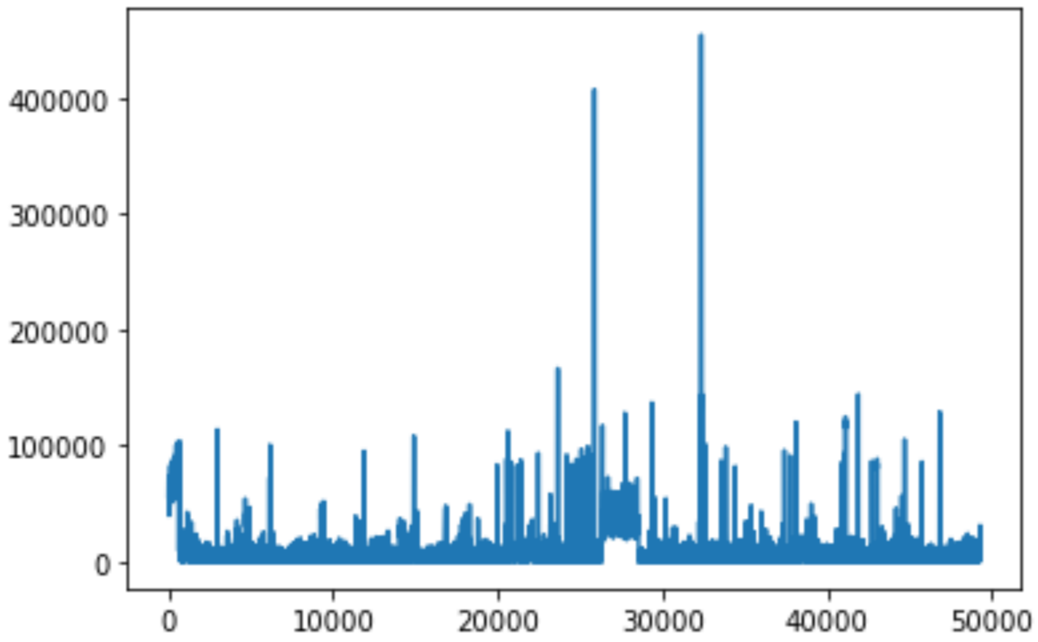
We also created some basic plots of the data.

```
roll = forecast['US48'].rolling(days*24).mean()
r2score(demand['US48'].iloc[167-11:49297],roll.iloc[167:49297])
demand['US48'].plot()
roll.plot()
```



And a graphic

of a forecast error.



As we can see, there are some large spikes in the error.

We used the following to add shifted columns to our demand data frame at various time intervals.

```
us_demand = demand[['date', 'US48']]
for i in [1,2,3,12,24,48,72]:
   us_demand['US48.H' + str(i)] = us_demand['US48'].shift(i)
us_demand = us_demand.iloc[72:]
```

And We created a simple linear regression model based on these shifts.

```
model = linear_model.LinearRegression()
model.fit(pd.DataFrame(X[['US48.H24', 'year', 'month', 'dayOfMonth', 'dayOfWeek',
     'hour']].iloc[:10000]),pd.DataFrame(y.iloc[:10000]))
model.score(pd.DataFrame(X[['US48.H24', 'year', 'month', 'dayOfMonth', 'dayOfWeek',
     'hour']].iloc[10000:]), pd.DataFrame(y.iloc[10000:]))
```

Yielding a score of 0.9268866378364472 $\approx$ 0.927. So even a very simple unoptimized model does quite well. This is encouraging as it means it might not be too difficult to squeeze out better performance than the current forecast. Running a Random Forest Regressor in a similar manner yields a score of approximately 0.947 which is almost on par with the provided forecast. So, we will look into optimizing the Random Forest Regressor in addition to other options.

Upon researching other options it was decided to look into Facebook's Prophet model. it is an open source project that is targeted at predicting time series on high frequency data. Our data is collected hourly, so it should qualify. Preprocessing the data for fbprophet is quite simple.

```
#fbprophet really just needs a ds column of datetimes and a y column of data to fit.
df = demand[['date','US48']]
df.columns = ['ds','y']
#The datetimes do need to have there timezone information removed as fbprophet does not support
     this information.
df['ds'] = pd.to_datetime(df['ds'])
df['ds'] = df['ds'].dt.tz_convert(None)
```

Facebook designed prophet to be compatible with sklearn, so training the model uses the same method call as any of the other models shown thus far. Upon training a fbprophet model. It achieved an r^2 score of approximately 0.688 which is disappointing given how well simpler models performed. Attempting to combine it with the RandomForestRegressor by means of a weighted average based on the individual models scores actual resulted in worse results. So, we will simple go with the random forest regressor and optimize it on just two features being the demand 24 and 48 hours in the past.

```
sdf = pd.DataFrame(df)
sdf['sy'] = sdf.y.shift(-24)
sdf['ssy'] = sdf.y.shift(-48)
sdf = sdf.iloc[:-48]
spoint = int(0.75*sdf.shape[0])
X_train, X_test, y_train, y_test = sdf[['y', 'sy']].iloc[:spoint], sdf[['y',
    'sy']].iloc[spoint:], sdf.ssy.iloc[:spoint]
rfr = RandomForestRegressor()
rf_params = {'n_estimators' : list(range(100, 501, 100)),
   'criterion' : ['mse','mae'],
   'max_depth' : [10],
   'max_features' : ['auto', 'sqrt']}
rscv = RandomizedSearchCV(rfr, rf_params, n_iter=2)
rscv.fit(X_train,y_train)
rfr = rscv.best_estimator_
rf_score = rscv.best_score_
metrics.r2_score(rfr.predict(X_test), y_test)
```

Results in an r^2 score of $0.9873837279672111 \approx 0.987$ which is an improvement to the model. Because we used a proper test split to predict the data. We can recommend adopting the optimized RandomForestRegressor for future use.