INNOVATION IN ENERGY MEASURE CONSUMPTION

DESCRIPTION:

This document outlines the AI powered energy measuring system. The system primary objective is Smart Grids with AI-Enabled Controls represent a technologically advanced and adaptive electrical grid system that leverages Artificial Intelligence (AI) for more efficient and sustainable energy management.

Here's an overview of how it works:

Advanced Metering Infrastructure (AMI):

Smart grids begin with the implementation of smart meters that provide real-time data on energy consumption and production. This two-way communication allows for more accurate monitoring and control.

Real-Time Monitoring and Communication:

- Smart grid components continuously monitor various aspects of the grid, including electricity flow, voltage levels, and demand patterns.
- AI algorithms process this real-time data to make dynamic adjustments in grid operations.

AI-Enabled Decision Making:

AI algorithms analyze the data collected from smart meters, sensors, and other grid components. This analysis allows the system to make intelligent decisions in response to changing conditions.

Load Balancing and Demand Response:

AI can dynamically balance the load by redistributing energy resources in real-time, ensuring that supply meets demand.

In times of high demand, AI can trigger demand response programs to encourage consumers to reduce their energy usage.

Optimized Energy Distribution:

AI algorithms determine the most efficient pathways for electricity distribution, minimizing losses and maximizing the utilization of renewable energy sources.

Integration of Renewable Energy Sources:

Smart grids with AI can seamlessly integrate intermittent renewable energy sources like solar and wind, adapting to their variable output and maximizing their contribution to the grid.

Predictive Maintenance:

AI algorithms can predict when grid components are likely to fail, allowing for proactive maintenance, reducing downtime, and preventing potential outages.

Fault Detection and Self-Healing:

Smart grids equipped with AI can quickly detect faults or disruptions in the grid and take corrective actions to isolate the affected area and restore power.

Cybersecurity and Resilience:

AI can play a crucial role in identifying and mitigating cybersecurity threats, ensuring the integrity and security of the grid.

Voltage Regulation and Power Quality:

AI algorithms can adjust voltage levels in real-time to maintain consistent power quality, ensuring that electrical devices operate efficiently and safely.

Grid Planning and Expansion:

AI can assist in long-term planning by analyzing data trends to forecast future energy demands. This helps utilities make informed decisions about grid expansion and infrastructure upgrades.

Dynamic Pricing and Tariffs:

AI can analyze real-time data to implement dynamic pricing models, allowing for more flexible and responsive energy pricing based on supply and demand.

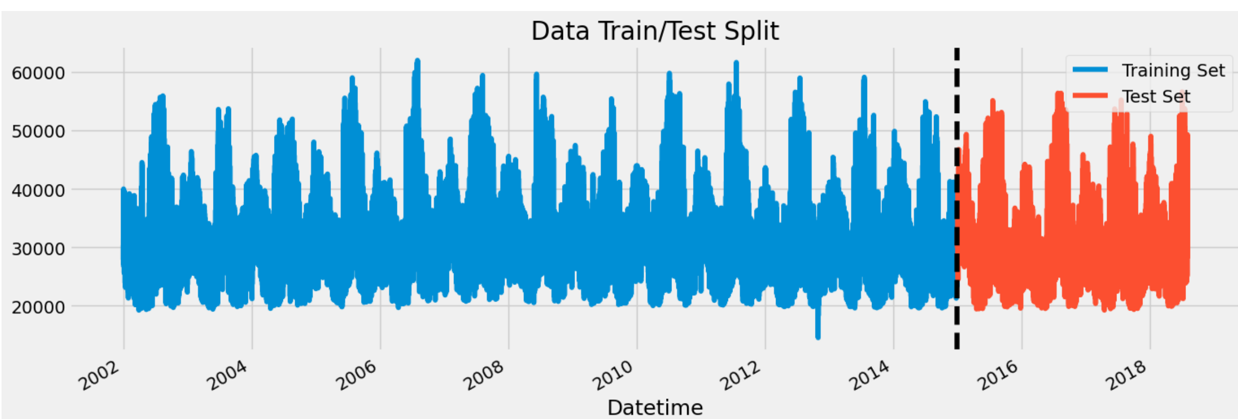
Decentralized Energy Trading:

Smart grids with AI can facilitate peer-to-peer energy trading, enabling consumers to buy and sell excess energy within a localized grid.

Smart Grids with AI-Enabled Controls represent a significant advancement in the modernization of energy infrastructure. They enhance the grid's efficiency, reliability, and adaptability, ultimately contributing to a more sustainable and resilient energy ecosystem.

IMPORT DATA SET :

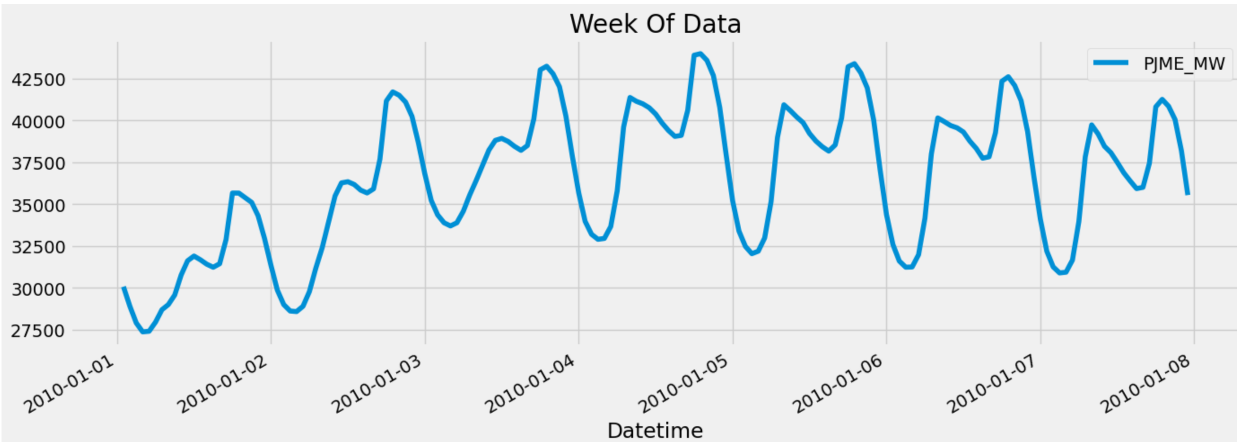
```
            train = df.loc[df.index < '01-01-2015']
test = df.loc[df.index >= '01-01-2015']


fig, ax = plt.subplots(figsize=(15, 5))
train.plot(ax=ax, label='Training Set', title='Data Train/Test Split')
test.plot(ax=ax, label='Test Set')
ax.axvline('01-01-2015', color='black', ls='--')
ax.legend(['Training Set', 'Test Set'])
plt.show()
```



In [5]:

```
df.loc[(df.index > '01-01-2010') & (df.index < '01-08-2010')] \
```

```
.plot(figsize=(15, 5), title='Week Of Data')
plt.show()
```



Feature Creation

In [6]:

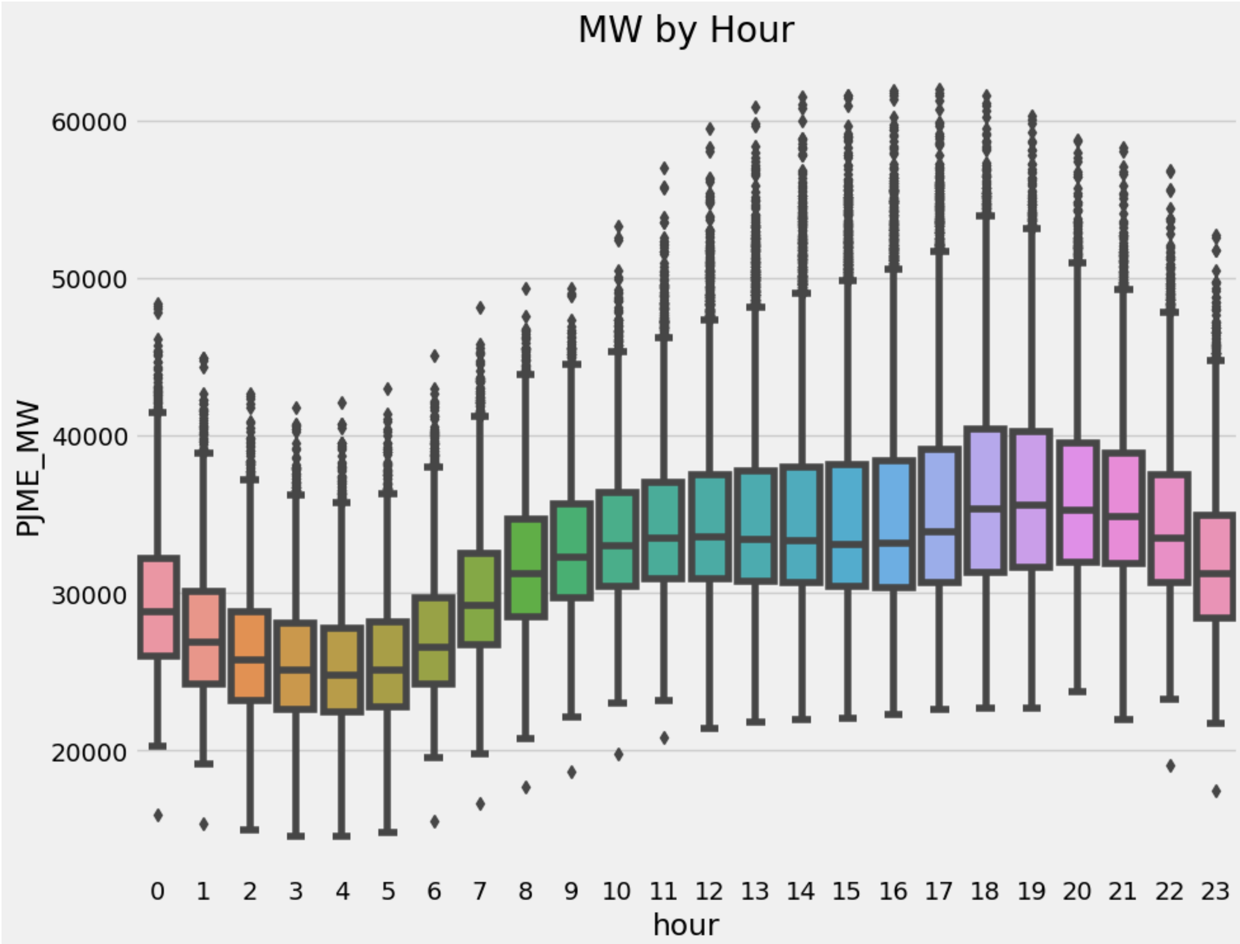
```
def create_features(df):
    """
    Create time series features based on time series index.
    """

    df = df.copy()
    df['hour'] = df.index.hour
    df['dayofweek'] = df.index.dayofweek
    df['quarter'] = df.index.quarter
    df['month'] = df.index.month
    df['year'] = df.index.year
    df['dayofyear'] = df.index.dayofyear
    df['weekofmonth'] = df.index.day
    df['weekofyear'] = df.index.isocalendar().week
```

```
    return df

df = create_features(df)
```

Visualize our Feature / Target Relationship

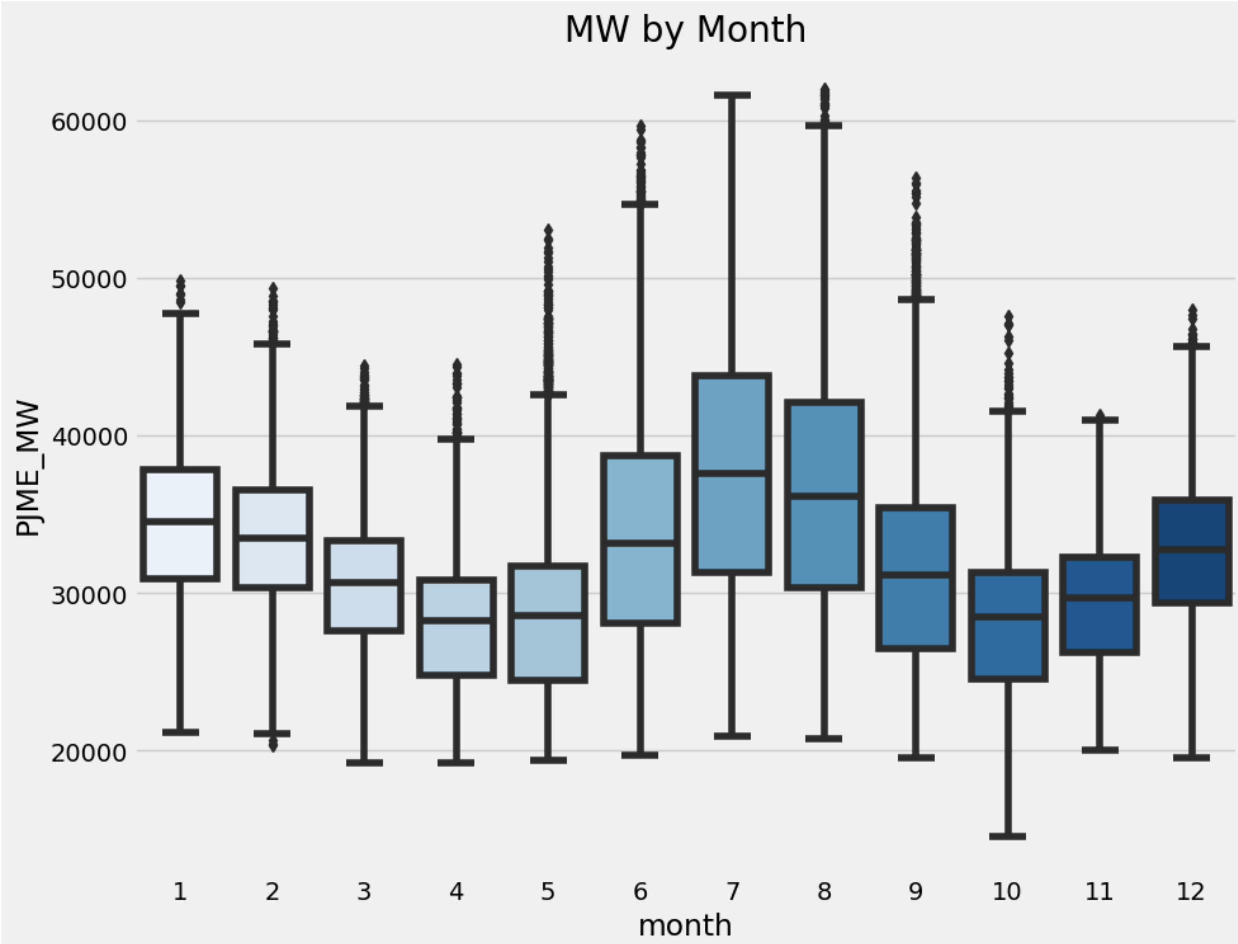
In [7]:

```
fig, ax = plt.subplots(figsize=(10, 8))
sns.boxplot(data=df, x='hour', y='PJME_MW')
ax.set_title('MW by Hour')
plt.show()
```



In [8]:

```
fig, ax = plt.subplots(figsize=(10, 8))

sns.boxplot(data=df, x='month', y='PJME_MW', palette='Blues')

ax.set_title('MW by Month')

plt.show()
```



Create our Model

In [9]:

```
train = create_features(train)
test = create_features(test)

FEATURES = ['dayofyear', 'hour', 'dayofweek', 'quarter', 'month', 'year']
TARGET = 'PJME_MW'

x_train = train[FEATURES]
y_train = train[TARGET]
```

```
x_test = test[FEATURES]
y_test = test[TARGET]
```

In [10]:

```
reg =xgb.XGBRegressor(base_score=0.5, booster='gbtree', n_estimators=1000,
                      early_stopping_rounds=50,
                      objective='reg:linear',
                      max_depth=3,
                      learning_rate=0.01)
reg.fit(x_train, y_train,
       eval_set= [(x_train, y_train), (x_test, y_test)],
       verbose=100)
```

```
[20:46:20] WARNING: ../src/objective/regression_obj.cu:213: reg:linear is now deprecated in favor of reg:squarederror.
[0]	validation_0-rmse:32605.13860	validation_1-rmse:31657.15907
[100]	validation_0-rmse:12581.21569	validation_1-rmse:11743.75114
[200]	validation_0-rmse:5835.12466	validation_1-rmse:5365.67709
[300]	validation_0-rmse:3915.75557	validation_1-rmse:4020.67023
[400]	validation_0-rmse:3443.16468	validation_1-rmse:3853.40423
[500]	validation_0-rmse:3285.33804	validation_1-rmse:3805.30176
[600]	validation_0-rmse:3201.92936	validation_1-rmse:3772.44933
[700]	validation_0-rmse:3148.14225	validation_1-rmse:3750.91108
[800]	validation_0-rmse:3109.24248	validation_1-rmse:3733.89713
[900]	validation_0-rmse:3079.40079	validation_1-rmse:3725.61224
[999]	validation_0-rmse:3052.73503	validation_1-rmse:3722.92257
```

Out[10]:

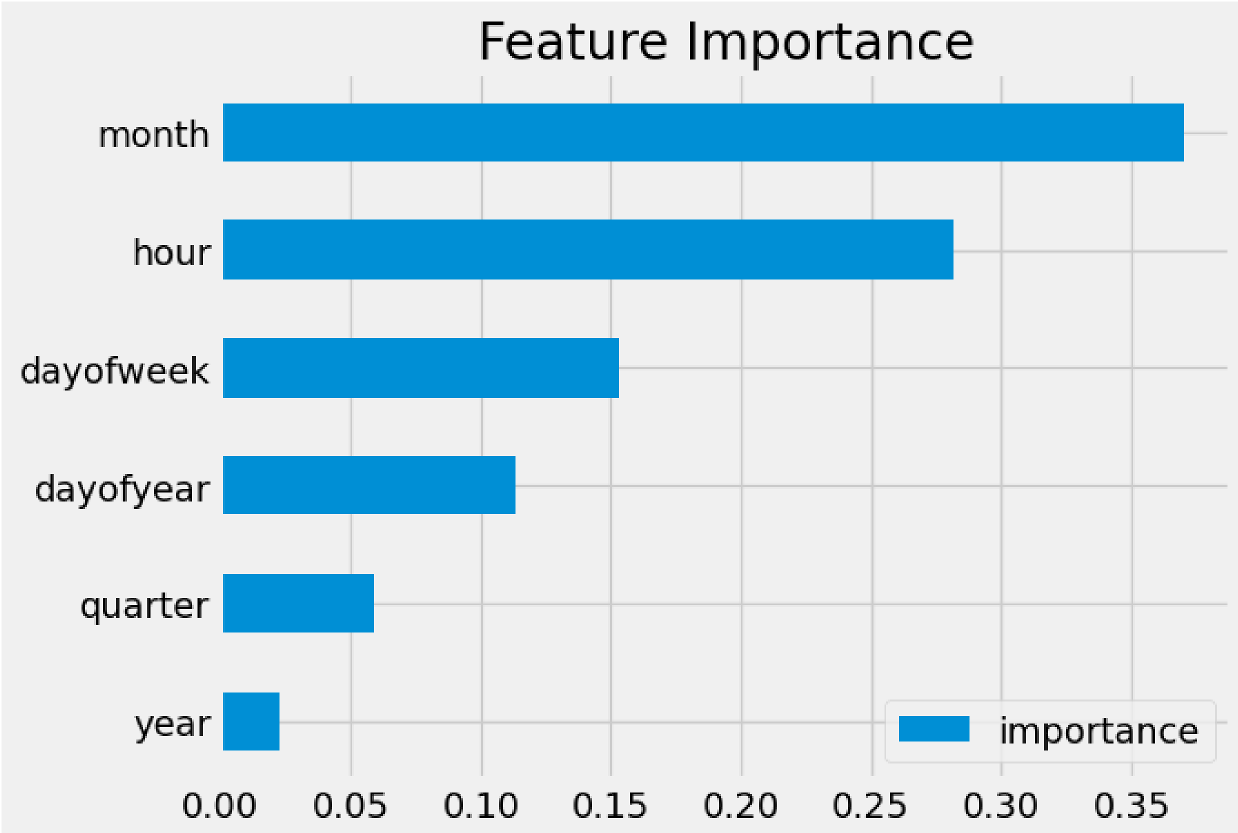
```
XGBRegressor
```

```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=50,
             enable_categorical=False, eval_metric=None,
feature_types=None,
             gamma=None, gpu_id=None, grow_policy=None,
importance_type=None,
             interaction_constraints=None, learning_rate=0.01,
max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=3, max_leaves=None,
             min_child_weight=None, missing=nan,
monotone_constraints=None,
             n_estimators=1000, n_jobs=None, num_parallel_tree=None,
             objective='reg:linear', predictor=None, ...)
```

Feature Importance

In [11]:

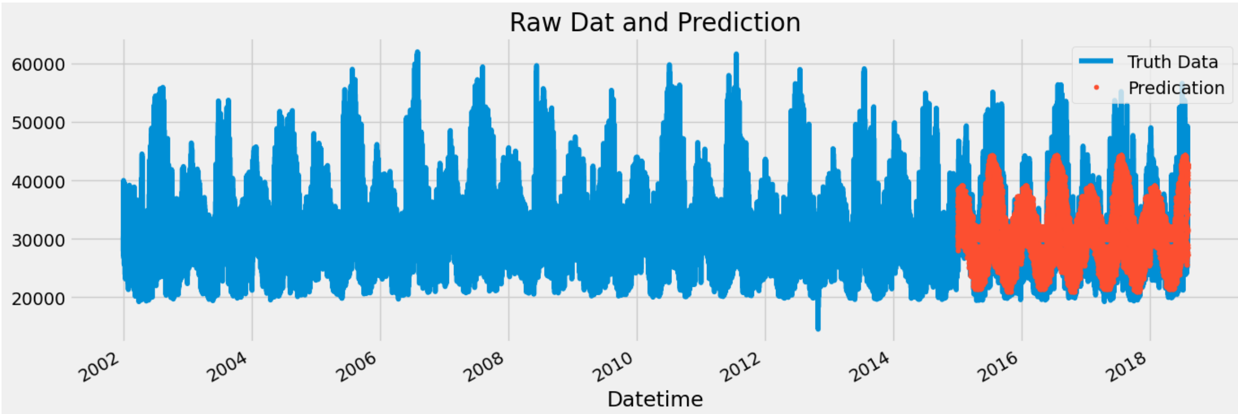
```
fi = pd.DataFrame(data=reg.feature_importances_,
                  index=reg.feature_names_in_,
                  columns=['importance'])
fi.sort_values('importance').plot(kind='barh', title='Feature Importance')
plt.show()
```



Forecast on Test

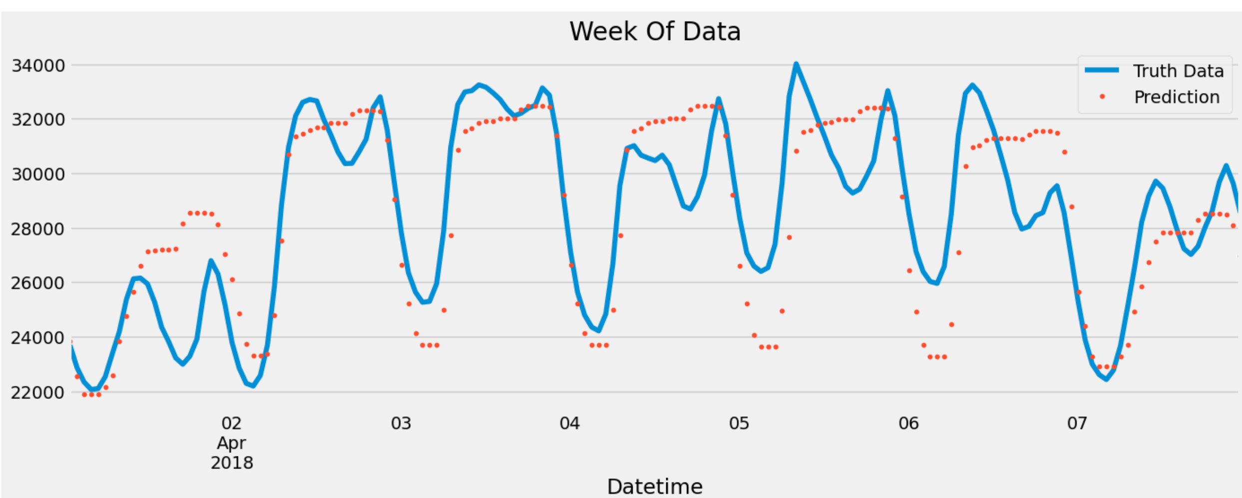
In [12]:

```
test['prediction'] = reg.predict(x_test)

df = df.merge(test[['prediction']], how= 'left', left_index=True,
right_index=True)

ax = df[['PJME_MW']].plot(figsize=(15, 5))

df['prediction'].plot(ax=ax, style='.')

plt.legend(['Truth Data', 'Predication'])

ax.set_title('Raw Dat and Prediction')

plt.show()
```



In [13]:

```
ax = df.loc[(df.index > '04-01-2018') & (df.index <
'04-08-2018')]['PJME_MW'] \
    .plot(figsize=(15, 5), title='Week Of Data')
df.loc[(df.index > '04-01-2018') & (df.index <
'04-08-2018')]['prediction'] \
    .plot(style='.')
plt.legend(['Truth Data','Prediction'])
plt.show()
```



Score (RMSE)

In [14]:

```
score = np.sqrt(mean_squared_error(test['PJME_MW'], test['prediction']))
print(f'RMSE Score on Test set: {score:0.2f}')
```

```
RMSE Score on Test set: 3721.75
```

Calculate Error Look at the worst and best predicted days

In [15]:

```
test['error'] = np.abs(test[TARGET] - test['prediction'])
test['date'] = test.index.date
test.groupby(['date'])['error'].mean().sort_values(ascending=False).head(1
0)
```

Out[15]:

```
date
2016-08-13    12839.597087
2016-08-14    12780.209961
2016-09-10    11356.302979
2015-02-20    10965.982259
2016-09-09    10864.954834
2018-01-06    10506.845622
2016-08-12    10124.051595
2015-02-21     9881.803711
2015-02-16     9781.552246
2018-01-07     9739.144206
Name: error, dtype: float64
```