Understanding European Electricity Consumption and Spending Pipeline

Nicholas Lee, Mendelina Lopez, & Grigor Tashchyan

Shiley-Marcos School of Engineering, University of San Diego

ADS 508: Data Science Cloud Computing

Professor Sean Coyne

April 17, 2023

Github Link: https://github.com/nlee98/ADS-508-AWS-Cloud-Computing

Understanding European Electricity Consumption and Spending Pipeline
Design Document

Authors: Nicholas Lee, Mendelina Lopez, and Grigor Tashchyan
Company Name: London Energy
Company Industry: Electric Power Industry
Company Size: 100 employees

Abstract:
Our company is interested in understanding the possible relationship electricity usage has with weather, price and consumption specifically looking into data from London.

Problem Statement:

Electricity plays a major role in modern society as the implementation of technological advancements become more common for the average household. As an individual's energy consumption increases the cost grows as well. The objective of this project is to understand the level of energy usage that could be dependent on factors such as weather, price, and consumption. This could possibly help the company assess the effects weather conditions have on electricity consumption in order to execute conservation plans and assist households as needed. Our company strives to provide lasting energy to homes and businesses proving to be both consistent and reliable.

Goals:

1. Predict future energy consumption rates and corresponding prices in London.
2. Identify seasons/periods of high energy usage to indicate seasons where London natives should attempt to conserve on energy.
3. Assess whether future demands will exceed the output capabilities of current resources.

Non-Goals:

We as a group decided that we would not compare the future energy consumption rates and prices against other cities in England and European cities. We will not make recommendations on how to conserve energy. We will not recommend any pricing adjustments based on seasons. We will not investigate any correlations based on currency.

Data Sources:

The data sources selected (Hourly_European_Market_Data.csv, London_Energy.csv, London_Weather_1979_2021.csv, shown in Table 1) were all retrieved from Kaggle and formatted as CSV files. As some of the data sources contain millions of records, the physical sizes of the files exceed GitHub's 25 MB and 100 MB limits. Thus, GitHub could not be used to house the raw data. Instead, to meet the memory requirements, an AWS S3 bucket, with a

memory limit of 5 TB, was utilized to store the raw CSV files. The three CSV files were downloaded from their respective Kaggle sources and manually uploaded to an AWS S3 bucket (https://s3.console.aws.amazon.com/s3/buckets/ads-508-spring2023-team3?region=us-east-1&tab=accesspoint).

Table 1

Data Sources and Metadata

Data Source Name	File Size	Number of Records	Number of Columns
Hourly_European_Market_Data.csv	100.3 MB	1,831,554	9
London_Energy.csv	91.2 MB	3,510,433	3
London_Weather_1979_2021.csv	795.3 KB	15,341	10

Note. The names of the CSV files were changed from their original sources for easier identification.

Data Exploration:

The raw csv files are stored in their own public S3 bucket, located at s3://ads-508-spring2023-team3. To ease user accessibility, a new folder in the user's default S3 bucket is created to house all data related to the project (s3://*user's-default-bucket*/london_data). Then, each csv will be copied from the raw source S3 bucket to a unique folder in the user's personal bucket. An example of such a path is s3://*user's-default-bucket*/london_data/market_data/Hourly_European_Market_Data.csv. A similar path is created for the other two data sources. Creating unique folders per CSV file is key in ensuring that PyAthena has a unique data location to create unique relational tables. In addition, to optimize downstream query performance, the data can be partitioned by year, using Apache Parquet file formatting. With that said, the unique folders created per CSV file creates an organizational system for housing each CSV with its respective Parquet-partitioned files.

Tools Utilized for Data Exploration

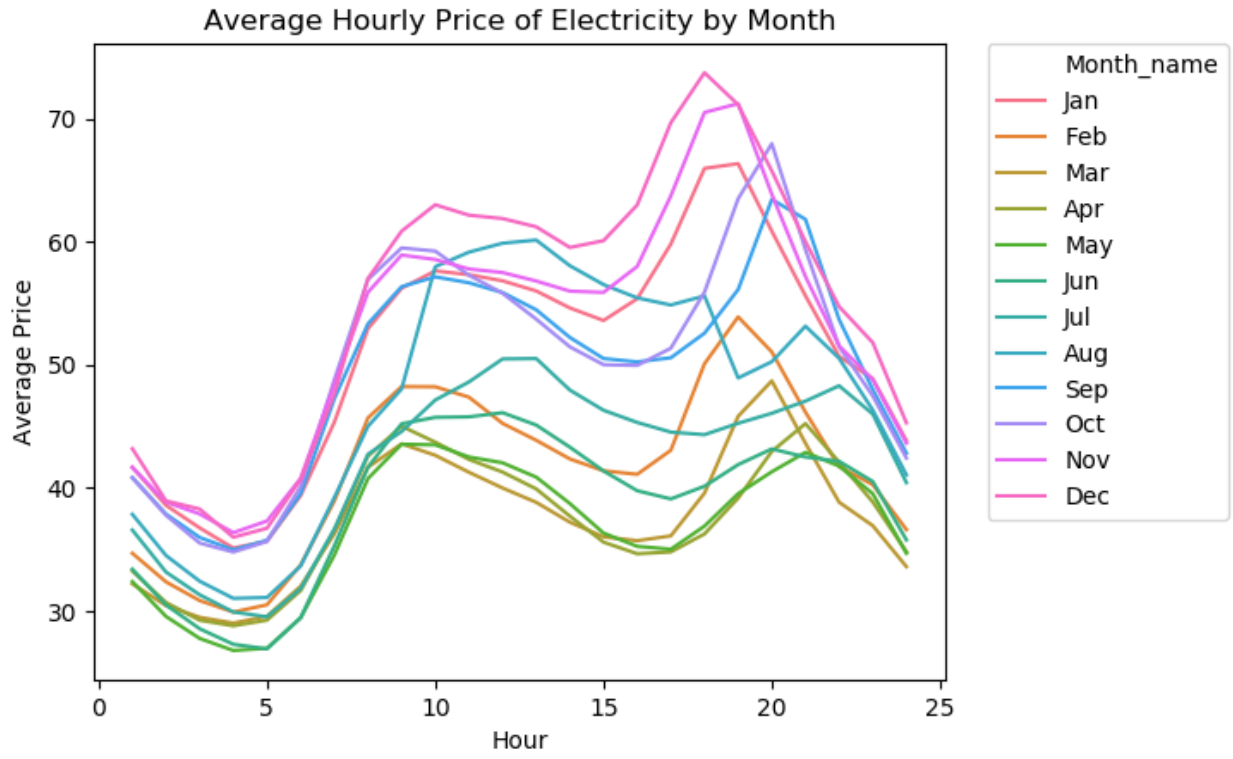
For initial temporary querying, a staging bucket is created in the user's personal S3 bucket, located at s3://*user's-default-bucket*/athena/staging. Utilizing PyAthena, a relational database called *london_data* and tables for each CSV file is created in the staging bucket. SQL is used to query the newly created tables in the user's staging S3 bucket. The result set from the queries will be saved as Panda dataframes and plotted via the *matplolib.pyplot* and *seaborn* dependencies in Python.

Data Exploration Related to Business Goals

One of the initial goals was to identify periods or seasons where electricity consumption can be minimized or is cheaper for the consumer. Figure 1, below, shows how electricity varies by the time of day and by months. It was observed that electrical costs have historically increased in months associated with winter (November, December, January, and February) and around 10 A.M. and between 15:00/3 P.M. and 20:00/8:00 P.M. Thus, a preliminary conservation plan would most make recommendations such as limiting electrical usage in the winter months and during peak hours all year-round.

Figure 1

Average Hourly Price of Electricity by Month (Using Data from July 1979 - Jan. 2022)



Data Exploration Related to the Data

To gain a better understanding of the data sources and so that each data set is within the same time frame, each data set was pared down to range between November 2011 and February 2014. Figure 2 shows a box plot of the distribution of energy consumption within this condensed time frame. As displayed below, the data is positively skewed as the majority of consumption is less than 50 KWh. It is apparent that those instances that are greater than 50 KWh of energy consumption are not as dense, but still seems to reach up to around 350 KWh. For future conservation, a reduction in energy consumption could be encouraged for users who might exceed a specific allotment.

Figure 2

Distribution of Energy Consumption (Using Data from Nov. 2011 - Feb. 2014)

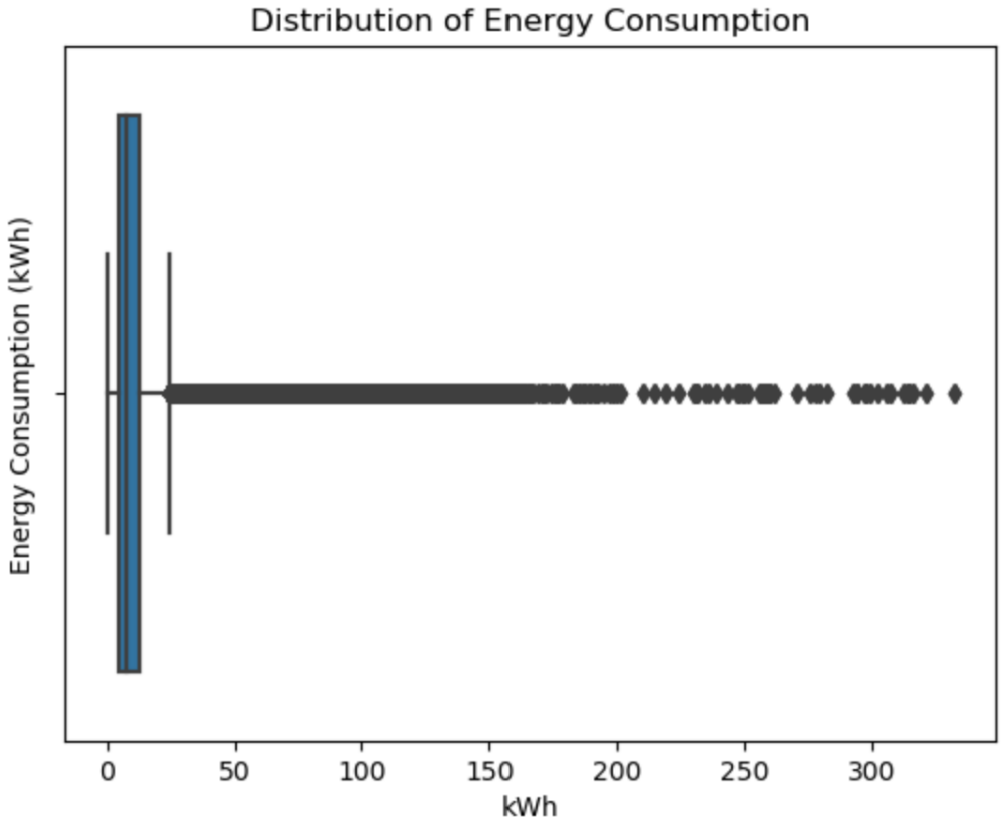


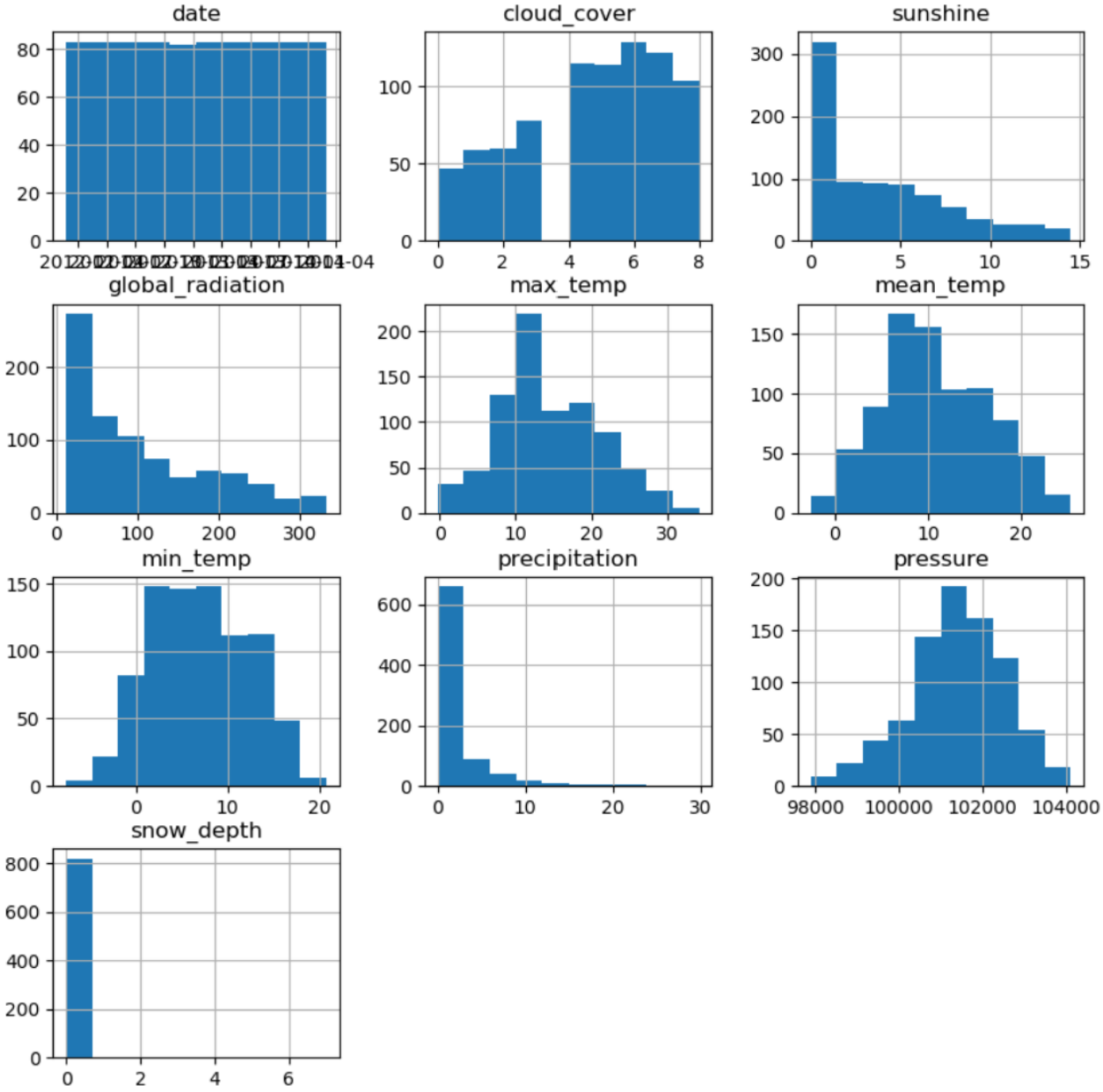
Figure 3 highlights the distribution of variables of the weather data set within the consolidated time frame of interest. The ‘date’ graph shows a generally uniform distribution suggesting the records are evenly distributed across time. The ‘cloud_cover’ histogram is measured in oktas and shows that the majority of the distribution is between 4 - 8 oktas. This means that skies are primarily partly cloudy to overcast. The ‘sunshine’ graph measures the number of hours the sun was present; as the histogram plot is skewed right, the plot for which indicates a minimal amount of sunshine. The ‘global_radiation’ graph (measured in Watts per square meter) is also skewed to the right. The variables for ‘max_temp’, ‘mean_temp’, and ‘min_temp’ are recorded in degrees Celsius and show somewhat of a bell curve. As we analyze the distribution graphs within the weather data set, we are able to determine the variables that might be meaningful to our project.

Of the variables explored, key variables include those in the weather dataset, energy consumption measured in kWh (found in the energy dataset), and the date, hour, and price columns of the hourly market dataset. Data types for each column can be found in the data description notebook, in the GitHub repository. It should be noted that missing values made up 0% of the data for each subset.

In terms of data quality, the skewness of the variables shown in Figures 2 and 3 highlight the need for normalization and scaling prior to training a machine learning algorithm. Training a model on the data in its current state may induce statistical bias, and therefore lead to inaccurate predictions and forecasts.

Figure 3

Distribution of Weather Data (Using Data from Nov. 2011 - Feb. 2014)



Location of Data Ingestion and Exploration Notebook

The Jupyter Notebook (.ipynb) file for data ingestion and data exploration can be found at the following link: https://github.com/nlee98/ADS-508-AWS-Cloud-Computing/blob/main/ADS-508_Final_Notebook.ipynb.

Data Preparation:

Data Scrubbing

Prior to altering any imputation or feature engineering, it is key to ensure that all the datasets are within the same time frame. In its current state, the weather dataset spans from 1979 to 2021, while the energy dataset ranges from 2011 to 2014. If a model was trained on the data in its current state, there would be an excess of missing values at the time points where data for certain features is not known. Therefore, all datasets were parsed down to range between November 23, 2011 and February 28, 2014, where data is available for every time point, across all datasets.

The datasets were subsetted further to ensure that all dates of interest were represented equally. There are 829 days between November 23, 2011 and February 28, 2014. After initially subsetting the weather dataset to range between the two dates, the dataset had 829 rows, one record per day of interest. However, both the market data and energy consumption data still contained an excess of records. Because the market data was recorded on an hourly scale, there should have been 19,896 rows (829 days * 24hr./day). The *sistema* column indicated that market prices were recorded from multiple systems. When grouped by system type, only one system, system HU, contained 19,896 records, one record per hour per day of interest. Thus, the hourly market data was parsed down to only include rows where the value for *sistema* was "HU." Similarly, after the initial subset, the energy consumption dataset still contained its original three million records. Like the weather dataset, data for energy consumption was collected daily. However, the energy consumption sampled daily usage from multiple households, represented by the *LCLid* column. Thus, to ensure that all 829 days were represented per household in the energy consumption dataset, the energy consumption dataset was trimmed to only include *LCLid* values that occurred 829 times in the dataset which resulted in parsing the dataset from 3,510,443 records to 9,119 records.

Within the time frame of interest, there was one missing value in the weather dataset, in the field *cloud_cover*. As weather between days is not known to vary dramatically, the missing value was imputed by examining the values of the days prior to and after the missing value. For the days immediately before and after the missing value, the value for *cloud_cover* was five. Thus, it was deemed reasonable that the missing value was also five or close to five. For simplicity, a value of five was used to fill the missing value.

Feature Selection

In preparation for model training, only certain features were determined as necessary; all other features were dropped from the subsetted datasets. From the hourly market data, the fields *id* (which contained arbitrary identification numbers), *sistema* (which only had the value "HU" at this step), *bandera* (which was an internal flag that had no meaning), *tipo_moneda* (which now only had the value of one for every record), *origen_dato* (which was a repeat of the date/fecha column), and *fecha_actualizacion* (which indicated when the dataset was last updated) contained no information that could be utilized for forecasting electrical consumption. Thus, the above columns were removed from the dataset. Similarly, the field *LCLid*, which

contained arbitrary identifications given to the sampled households, was also removed for similar reasons to the fields listed above.

The features from the raw datasets carried forward were: *fecha* (relabeled as *Date*), *hora,* and *precio* from the hourly market dataset, *Date* and *Kwh* from the energy consumption dataset, and all features from the weather dataset.

Feature Creation

The two features (month and day_of_week) were extracted from Date as numerical values to avoid the need for encoding. There is debate as to whether such features should be treated as numerical values or categorical factors. Many in the discussion post, led by Funkwecker (2017), argue that the variables should be treated as numerical and transformed with cyclical functions, such as sine and cosine functions, to represent the cyclical nature of time, such that Monday is as close to Tuesday as it is Sunday. However, others in the discussion also state that cyclical functions introduce bias depending on the predictive algorithm utilized. Numerical data is favorable, for this study, in that it is compatible with most statistical analysis methods; and, thus is the most used among researchers. Categorical data on the other hand does not support such analytical methods. In addition, categorizing time-related data would decrease the significance in distance between time points. With time-series data, there may be some debate as to how to bin time periods; thus, to reduce subjectivity, data that is numerical in nature is preferred. At this stage of the project, the optimal predictive algorithm is not known. Thus, for the reasons above and for simplicity, this study will treat such variables as numeric.

In addition to the above features, a new feature *temp_range*, representing the daily range in temperatures was created from the difference between the daily maximum and daily minimum temperatures. It is possible that households consume more electricity during volatile seasons to keep temperatures stable and comfortable. Thus, to account for that potential factor, the *temp_range* feature was created.

Feature Transformation

All transformations were performed on the columns that are numerical for standardization. The only column that did not fit this criteria was the *Date* Column. A column transformer was utilized for implementing minimum-maximum (min-max) scaling on the numerical columns. By doing so, all numerical columns were scaled to range between zero and one. As a result, the distribution of every feature became approximately normally distributed. For statistical purposes in downstream model fitting, it was key that the distribution of features were approximately normal.

Data Balancing

Balancing a data set is necessary for ensuring that the accuracy given is true to the problem at hand. Dealing with an imbalanced data set leads to bias and can be dangerous when left with making decisions. This is due to the fact that an imbalanced data set is representative of the overall population, in most cases. Based on our data exploration, it was found that there are an equal number of records per date, as shown in Figure 3. For that reason, the dataset was deemed to be balanced and no further re-sampling would need to be performed to rebalance the data.

Data Partitioning

Before the data sets were merged, the hourly market price was averaged into a per day figure to ensure that the time interval between records were consistent across all datasets. The three data sets consisting of market, weather, and energy information were joined together using the *Date* variable as the index. This was done by implementing the *merge()* function found in the pandas library and specifying an inner join on the data. After the datasets were merged, the newly joined dataset was partitioned into training, validation, and testing sets. 80% of the data was allotted for training, spanning from November 2011 to August 2013. The validation set consisted of 10% of the data, ranging from September 2013 to November 2013. Lastly, the last 10% comprised the test set, which ranged from December 2013 to February 2014.

Model Training:

The built-in SageMaker DeepAR forecasting algorithm was used to train a model for this study. DeepAR is an algorithm based on recurrent neural networks (RNNs) designed for time-series forecasting. To load the algorithm into the notebook, the SageMaker image *forecasting-deepar* was retrieved. It should be noted that SageMaker Jumpstart would have been the preferred method, as the pre-trained models would have greatly accelerated and eased the training process. However, the AWS account utilized for this study did not have the permission "/service-role/AmazonSageMakerServiceCatalogProductsLaunchRole" required for launching a pre-built SageMaker machine learning model. Instead, a SageMaker Jumpstart notebook for a machine learning model on forecasting energy consumption was opened in a read-only format. From which, the parameters and hyperparameters of the Jumpstart model were manually transferred into the imported built-in algorithm. In addition, the SageMaker Jumpstart notebook was adapted for preparing the pre-processed data in an algorithm-acceptable JSON format.

Parameters and Hyperparameters

When implementing our *deepar_estimator*, we decided to include certain hyperparameters in order to tune the model effectively. The *time_freq* hyperparameter that determines the frequency we would like to consider was identified as 'per day'. We made sure to specify the *context_length* as 90 to indicate how far back in days we would like to predict. The *prediction_length* was then noted as a string of 91 as there are 91 days in the validation set. The number of *epochs* was set to 400 as this hyperparameter is dependent on the size of the data and learning rate. As a *learning_rate*, or the speed at which a model learns, the team decided to go with "5E-4" for an effective rate. Our *mini_batch_size* was determined to be 64 since we wanted to incorporate an optimal gradient descent. The *early_stopping_size* was then set to 40, indicating that the algorithm would halt training if no improvement was made to the training metrics after 40 epochs. Lastly, the parameter *num_dynamic_feat* was listed as "auto" so that the algorithm may automatically detect the number of dynamic features being passed through. Aside from the *context_length* and *prediction_length*, the other parameters were determined based on a notebook provided by JumpStart, which also created a model to forecast electrical consumption.

Dynamic features were all other features that were not electricity consumption. The model was trained on 13 different time series, attempting to fit the best model for forecasting electrical consumption given one additional predictor. For example, where one iteration of the model was supplemented with daily price data, another iteration attempted to find the best

model using data on the daily maximum temperature. Given more time, this study would ideally train the model on one time series, where all 13 exogenous variables are included.

Instance Size and Count

Because of the restrictions placed on the AWS accounts utilized for this project, the maximum number of instance counts that could be used was one. To increase the availability of resources in other manners, the instance type was increased from the default ml.t3.medium to ml.m5.xlarge. Further studies would benefit from increasing the instance size and optimizing the instance type.

Model Evaluation

An Amazon SageMaker training job is an iterative process that teaches a model to make predictions by presenting examples from a training dataset. By passing both a training and validation (referred to as the test set, by the algorithm) data set through the DeepAR algorithm, metrics, such as test root mean square error (RMSE) and training negative log-likelihood loss, are calculated. These metrics aid in determining whether the model is learning well or not. The training algorithm writes the value of these metrics to logs which are then sent to Amazon CloudWatch in real time. In Cloudwatch, graphs of the performance metrics are generated as each training iteration is completed. Once the training job is completed, a list of the performance metrics, computed in the final iteration, can be obtained by calling the DescribeTrainingJob operation. In this study, one metric used to evaluate the model is the root mean square error (RMSE). RMSE is the square root of the average of the squared differences between the estimated and the actual value of the variable/feature (Mulani, 2020). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. Thus, the objective is to minimize RMSE. In addition to RMSE, the training set's loss metric is utilized to measure the model's performance during training. Like RMSE, the objective is to minimize the value of the loss metric. SageMaker TrainingJobAnalytics was utilized to pull the above metrics from AWS CloudWatch Metrics.

Measuring Impact:

As the study is time series-oriented, three of the metrics of interest are the root mean square error (RMSE), adjusted-R^2 (R-squared), and mean absolute error (MAE). RMSE will indicate the average distance between predicted values and observed values. Optimal models will minimize the RMSE value. The adjusted-R^2 value measures the proportion of variance in the response variable that can be explained by the predictor variables in a linear regression model, while accounting for the number of input variables. Well-performing models will have an adjusted-R^2 value of 0.8 or higher. Both of which will aid in ensuring that the fitted model will be able to create an accurate future forecast. Mean absolute error (MAE), like RMSE, measures the average error. However, MAE is less sensitive to outliers, whereas RMSE is more sensitive to outliers. Because electrical usage can have sudden spikes, due to a number of factors, which may result in outliers, it is key to utilize performance metrics that account for and equally weigh outliers.

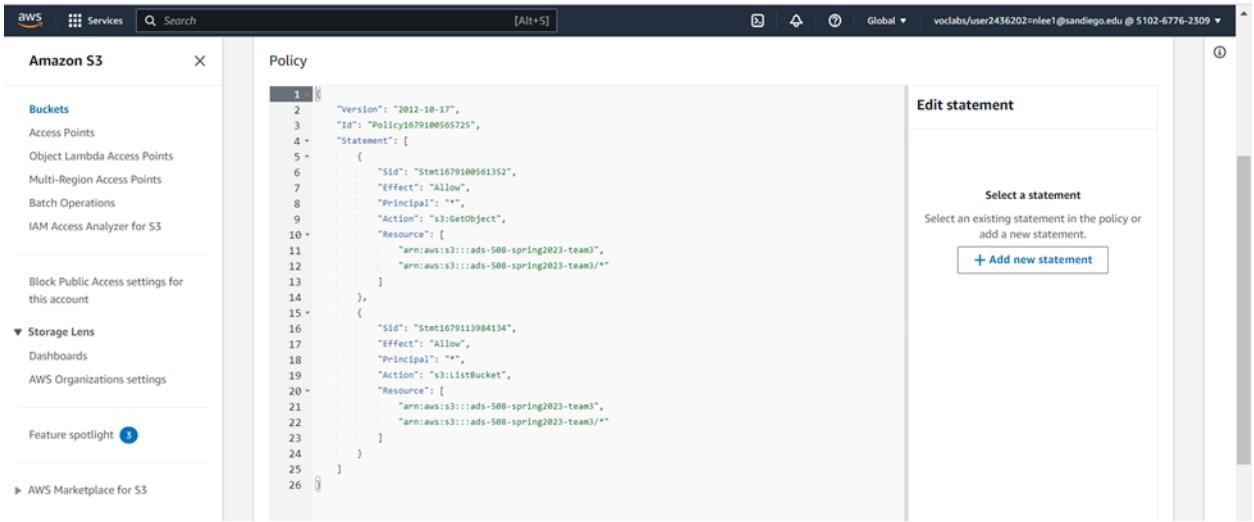
Security Checklist, Privacy and Other Risks:

With regards to data security, much of the data used is made publicly available. For example, a quick internet search reveals that data pertaining to energy usage is readily available through the government. Similarly, weather data is also publicly available through many online resources. None of the data used contains any sensitive information, such as personal health information (PHI) or personal identifiable information (PII) (such as credit card information or user tracking). Thus, in terms of security for data storage, the S3 bucket housing the raw data is made public, following the principle of least privilege.

While the S3 bucket is public, outside users are only granted the permissions s3:ListBucket and s3:GetObject, which allows the users to read the bucket contents and pull the data into their own personal S3 bucket. Figure 4 shows the exact policies added to the bucket, granting all users only the permissions to read and copy the bucket contents. These permissions are deemed acceptable by security standards as they allow the user no further permissions than if they were to pull the data from the direct, online sources themselves. All modifications done by outside users will be done in their personal S3 buckets; thus, the public source S3 bucket will remain unmodified.

Figure 4

Policy Script for Public S3 Bucket



Data Biases

In the context of this study, the primary foreseeable influential bias is outlier data. Outliers are values that deviate greatly, outside the average range of the dataset. Typical causes for outliers include: natural variability, human error, and poor sampling. For example, in the industry-specific context, outliers in electrical consumption, such as massive spikes in usage, may have external contributing factors, such as a large winter storm. Outliers can have a large effect on statistical calculations, such as mean and standard deviation. As a result,

machine learning algorithms and forecasting techniques perform significantly worse in the presence of such data. It is paramount to properly handle outliers to ensure proper representation of the data without introducing bad data into the machine learning algorithm.

We must also be wary of potential availability bias. The data collected for this study is limited to the three data sources utilized; there is a strong possibility that there are more predictive factors unaccounted for (Agarwal, 2020).

Ethical Considerations

One ethical concern to be wary of is any bias that may be due to socio-economic issues. This may unjustly skew the data in a way that could negatively impact the customers. The business model that is set in place should contribute to the improvement of the environment as well as having the best intentions for the clients.

Future Enhancements:

Obtainment of More Recent Data

1. In this study, one of the datasets was limited to data between 2011 and 2014. Because of which, forecasts to the present year, 2023, and the next year, 2024, were not feasible and out of the range of accurate predictions. In addition, the scope of data limited the availability of data for training the model. Thus, the trained model in this study is unsuited for forecasting ten years from the end of the data, to the present day. To improve upon the results here, more recent data should be acquired such that all datasets contain data as close to the time of prediction as possible. Doing so would allow for the business to obtain more accurate forecasts in a more relevant time period.

Hyperparameter Tuning

2. The hyperparameters utilized in this study were derived from an AWS JumpStart Machine Learning Model, designed for a similar purpose. While the intentions of this study and the JumpStart model are similar, the data used here was not identical, in terms of features, to that used to train the JumpStart model. Thus, while the pre-trained JumpStart model provided a foundation for training the algorithm, the model in this study would benefit from hyperparameter tuning as the optimal hyperparameters most likely deviate from those currently implemented.

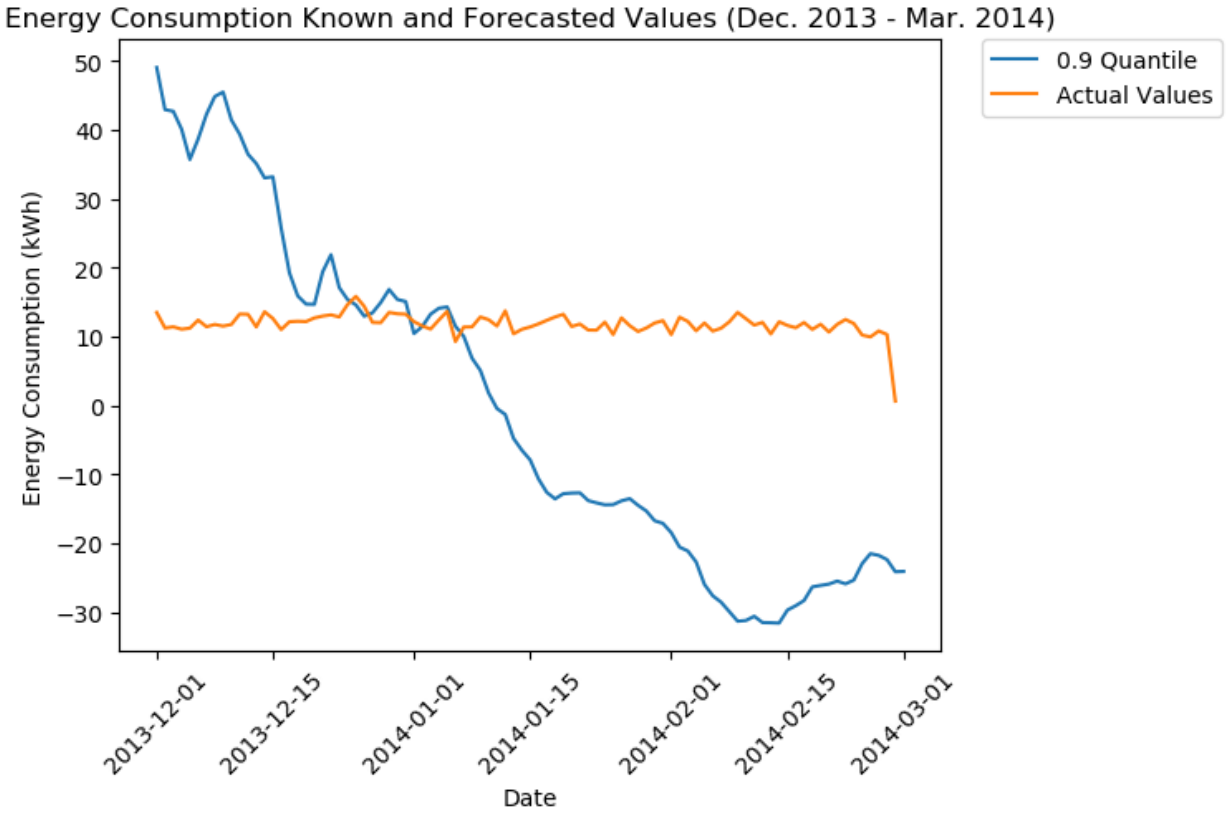
Test Set and Feature Optimization

3. Currently, the RMSE value on the validation set is 0.1597. While this is indicative of a well-performing model, the plot of forecasts for the test set data suggests otherwise (shown in Figure 5). This is most likely due to incorrect formatting of the test set data. In the training and validation set, 13 time series were passed, each attempting to forecast electricity consumption with one exogenous feature. The test set data only includes one time series on electricity consumption with the feature *precio* (price) as the exogenous predictor. Given more time, this study would optimize the formatting of the test data set

to include as many exogenous variables as the training and validation sets; or, alternatively, the model would be trained on one time series without exogenous variables.

Figure 5

Plot of 90th Percentile Electricity Consumption Forecasts and Known Values



Pipeline Automation

4. To meet business forecasting needs, a future project could implement batch predictions to forecast the upcoming period's electrical usage. This would be the case, whether that is next-day, -week, or -month. Batch prediction is an option that could prove to be most helpful as it is cost-effective and optimal for large amounts of data. In order to incorporate automation into the project at hand, an event-based trigger might be used. This would begin a pipeline once a specified event has occurred. A time-based trigger could also be included as it is further activated upon a given period of time. (Fregly & Barth, 2021, p. 386).

References

Agarwal, R. (2020). *Five cognitive biases in data science (and how to avoid them)*. Retrieved from https://towardsdatascience.com/five-cognitive-biases-in-data-science-and-how-to-avoid-them-2bf17459b041.

Fregly, C., & Barth, A. (2021). *Data science on AWS*. O'Reilly.

Funkwecker. (2017). *Re: Encoding features like month and hour as categorical or numeric?* [Online forum post]. StackExchange. https://datascience.stackexchange.com/questions/17759/encoding-features-like-month-and-hour-as-categorial-or-numeric.

Mulani, S. (2020). *RMSE - Root mean Square error in Python - AskPython*. AskPython. https://www.askpython.com/python/examples/rmse-root-mean-square-error.

Windy Weather World Inc, USA. (n.d.). *Okta - must-know measurement unit of clouds amount*. WINDY.APP. From https://windy.app/blog/okta-cloud-cover.html.

Import Needed Packages

In [93]:
```
import calendar
import json
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")

import boto3
import sagemaker
from sagemaker import get_execution_role
from sagemaker.serializers import IdentitySerializer
from sagemaker.tuner import IntegerParameter, CategoricalParameter
from sagemaker.tuner import ContinuousParameter, HyperparameterTuner
!pip install --disable-pip-version-check -q PyAthena==2.1.0
from pyathena import connect
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and conflict
ing behaviour with the system package manager. It is recommended to use a virtual env
ironment instead: https://pip.pypa.io/warnings/venv
```

Connect to S3 Bucket

In [94]:
```
# Get execution permissions
iam_role = get_execution_role()
```

In [95]:
```
# Specify Bucket Name
raw_data_bucket_name = "ads-508-spring2023-team3"

# Set up session parameters
session = boto3.session.Session()
region = session.region_name
sagemaker_session = sagemaker.Session()
personal_bucket = sagemaker_session.default_bucket()

# Establish an S3 connection
s3 = boto3.Session().client(service_name="s3", region_name=region)
```

In [96]:
```
# Set path to buckets (raw data and new personal bucket)
raw_data_path = 's3://{}'.format(raw_data_bucket_name)
personal_s3 = "s3://{}/london_data".format(personal_bucket)
```

In [97]:
```
# Check if connection to new personal s3 bucket was successfully established
try:
    bucket_contents = s3.list_objects(Bucket = personal_bucket)["Contents"]
    print("Connected to bucket successfully!\n")
    #print("Files in S3 bucket:")
    #for i in bucket_contents:
```

```
        #print(i["Key"])
except:
    # If connection failed, print error message
    print("##### ERROR #####")
    print("Could NOT connect to bucket: {} ({})".format(
        raw_data_bucket_name, personal_bucket))
```

```
Connected to bucket successfully!
```

In [98]:
```
# Create Unique Paths for Each CSV File
hourly_market_data_path = personal_s3 + "/market_data/Hourly_European_Market_Data.csv"
london_energy_path = personal_s3 + "/energy_data/London_Energy.csv"
london_weather_path = personal_s3 + "/weather_data/London_Weather_1979_2021.csv"
```

Athena will later create an external table based on the files of a bucket. Because all three files will act as a table for querying, all the files will need a unique path/folder.

Copy Raw CSV Files to Personal S3 Bucket

In [99]:
```
!aws s3 cp --recursive $raw_data_path/ $hourly_market_data_path/ --exclude "*" --inclu
!aws s3 cp --recursive $raw_data_path/ $london_energy_path/ --exclude "*" --include "L
!aws s3 cp --recursive $raw_data_path/ $london_weather_path/ --exclude "*" --include "
```

```
copy: s3://ads-508-spring2023-team3/Hourly_European_Market_Data.csv to s3://sagemaker
-us-east-1-510267762309/london_data/market_data/Hourly_European_Market_Data.csv/Hourl
y_European_Market_Data.csv
copy: s3://ads-508-spring2023-team3/London_Energy.csv to s3://sagemaker-us-east-1-510
267762309/london_data/energy_data/London_Energy.csv/London_Energy.csv
copy: s3://ads-508-spring2023-team3/London_Weather_1979_2021.csv to s3://sagemaker-us
-east-1-510267762309/london_data/weather_data/London_Weather_1979_2021.csv/London_Wea
ther_1979_2021.csv
```

In [100…
```
# Confirm that CSV files were copied to personal S3 bucket
!aws s3 ls $hourly_market_data_path/
!aws s3 ls $london_energy_path/
!aws s3 ls $london_weather_path/
```

```
2023-04-03 17:03:52  105185070 Hourly_European_Market_Data.csv
2023-04-03 17:03:53   95649585 London_Energy.csv
2023-04-03 17:03:55     814426 London_Weather_1979_2021.csv
```

Create Athena Database

In [101…
```
database_name = "london_data"
```

In [102…
```
# Create a staging S3 directory - a temporary directory for querying
stage_dir = "s3://{}/athena/staging".format(personal_bucket)
```

In [103…
```
# Connect to Staging Directory
try:
    pyathena_conn = connect(
        region_name = region,
        s3_staging_dir = stage_dir)
```

```
    print("Connected to S3 staging directory!")
except:
    print("##### ERROR #####")
    print("##### Could NOT connect to S3 staging directory #####")
```

```
Connected to S3 staging directory!
```

In [104…

```
# Create new Database - Print an error if there is a failure
try:
    sql_statement = "CREATE DATABASE IF NOT EXISTS {}".format(database_name)
    pd.read_sql(sql_statement, pyathena_conn)
    print("Database {} succesfully created!".format(database_name))
    sql_statement = "SHOW DATABASES"
    df_show = pd.read_sql(sql_statement, pyathena_conn)
    print(df_show.head(5))
except:
    print("##### ERROR #####")
    print("##### Could NOT create database #####")
```

```
Database london_data succesfully created!
  database_name
0       default
1        dsoaws
2   london_data
```

Create Tables from CSV files

In [105…

```
# SQL Create Table from Hourly Market Data CSV
table_name = "hourly_european_market_data_csv"
path_to_data = personal_s3 + "/market_data"

csv_table_statement = """
    CREATE EXTERNAL TABLE IF NOT EXISTS {}.{}(
        Id int,
        Fecha date,
        Hora int,
        Sistema string,
        Bandera string,
        Precio float,
        Tipo_moneda     string,
        Origen_dato     string,
        Fecha_actualizacion     timestamp
    )
    ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    LOCATION '{}'
    TBLPROPERTIES ('skip.header.line.count'='1')
    """.format(database_name, table_name, path_to_data)

# Execute Create Table Statement
pd.read_sql(csv_table_statement, pyathena_conn)
print("Succesfully made market_data table")
```

```
Succesfully made market_data table
```

In [106…

```
# SQL Create Table from Energy CSV
table_name = "london_energy_csv"
path_to_data = personal_s3 + "/energy_data"

csv_table_statement = """
```

```
    CREATE EXTERNAL TABLE IF NOT EXISTS {}.{}(
        LCLid string,
        Date date,
        Kwh float
    )
    ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    LOCATION '{}'
    TBLPROPERTIES ('skip.header.line.count'='1')
    """.format(database_name, table_name, path_to_data)

# Execute Create Table Statement
pd.read_sql(csv_table_statement, pyathena_conn)
print("Successfully created energy_data table")
```

```
Successfully created energy_data table
```

In [107…

```
# SQL Create Table from Weather CSV
table_name = "london_weather_csv"
path_to_data = personal_s3 + "/weather_data"

csv_table_statement = """
    CREATE EXTERNAL TABLE IF NOT EXISTS {}.{}(
        Date string,
        Cloud_cover float,
        Sunshine float,
        Global_radiation float,
        Max_temp float,
        Mean_temp float,
        Min_temp float,
        Precipitation float,
        Pressure float,
        Snow_depth float
    )
    ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    LOCATION '{}'
    TBLPROPERTIES ('skip.header.line.count'='1')
    """.format(database_name, table_name, path_to_data)

# Execute Create Table Statement
pd.read_sql(csv_table_statement, pyathena_conn)
print("Successfully created weather_data table")
```

```
Successfully created weather_data table
```

Note: The date in London_energy is formated as YYYYMMDD. It will have to be converted to the format (YYYY-MM-DD) for later joins.

In [108…

```
# Show Created Tables
statement = "SHOW TABLES in {}".format(database_name)

df_show = pd.read_sql(statement, pyathena_conn)
df_show.head(5)
```

Out[108]:

	tab_name
0	hourly_european_market_data_csv
1	london_energy_csv
2	london_weather_csv

In [109…

```
# Show the first few rows to ensure that the data looks correct
query = """
    SELECT *
    FROM {}.{}
    LIMIT 5
    """.format(database_name, "hourly_european_market_data_csv")

pd.read_sql(query, pyathena_conn)
```

Out[109]:

	id	fecha	hora	sistema	bandera	precio	tipo_moneda	origen_dato	fecha_actualizacion
0	589846	2016-03-20	22	LV	0	30.20	1	2	2021-10-01 12:39:53
1	589847	2016-03-20	22	NO1	0	21.54	1	2	2021-10-01 12:39:53
2	589848	2016-03-20	22	NO2	0	21.54	1	2	2021-10-01 12:39:53
3	589849	2016-03-20	22	NO3	0	21.54	1	2	2021-10-01 12:39:53
4	589850	2016-03-20	22	NO4	0	21.54	1	2	2021-10-01 12:39:53

In [110…

```
query = """
    SELECT *
    FROM {}.{}
    LIMIT 5
    """.format(database_name, "london_energy_csv")

pd.read_sql(query, pyathena_conn)
```

Out[110]:

	lclid	date	kwh
0	MAC003839	2013-06-14	7.034
1	MAC003839	2013-06-15	6.203
2	MAC003839	2013-06-16	2.780
3	MAC003839	2013-06-17	6.583
4	MAC003839	2013-06-18	6.038

In [111…

```
query = """
    SELECT *
    FROM {}.{}
    LIMIT 5
    """.format(database_name, "london_weather_csv")
```

```
pd.read_sql(query, pyathena_conn)
```

Out[111]:

	date	cloud_cover	sunshine	global_radiation	max_temp	mean_temp	min_temp	precipitation
0	19790101	2.0	7.0	52.0	2.3	-4.1	-7.5	0.4
1	19790102	6.0	1.7	27.0	1.6	-2.6	-7.5	0.(
2	19790103	5.0	0.0	13.0	1.3	-2.8	-7.2	0.(
3	19790104	8.0	0.0	13.0	-0.3	-2.6	-6.5	0.(
4	19790105	6.0	2.0	29.0	5.6	-0.8	-1.4	0.(

Data Exploration in SQL via PyAthena

Monthly Average Electricity Price

In [112…

```
# Monthly average electricity price
query = """
    SELECT YEAR(fecha) AS Year,
        MONTH(fecha) AS Month,
        AVG(precio) AS Avg_Price
    FROM {}.hourly_european_market_data_csv
    GROUP BY YEAR(fecha), MONTH(fecha)
    ORDER BY YEAR(fecha), MONTH(fecha) asc;
    """.format(database_name)

monthly_avg_price = pd.read_sql(query, pyathena_conn)
monthly_avg_price.head(10)
```

Out[112]:

	Year	Month	Avg_Price
0	2010	7	42.636627
1	2010	8	76.112564
2	2010	9	46.093450
3	2010	10	50.410633
4	2010	11	47.319530
5	2010	12	49.341072
6	2011	1	50.033813
7	2011	2	51.160927
8	2011	3	53.950410
9	2011	4	52.013780

In [113…

```
# Modify output
avg_price = monthly_avg_price
```

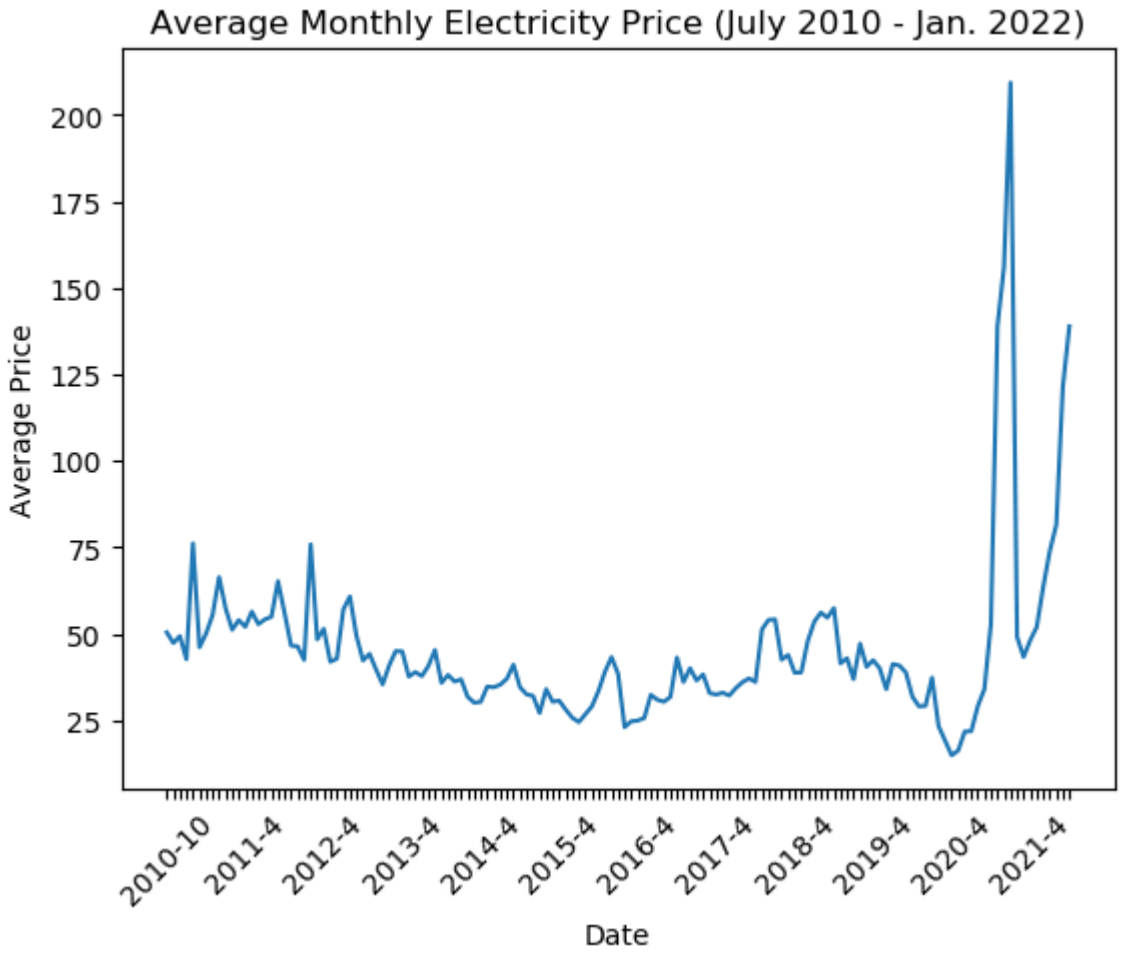
```
# Merge Year and month to create a date column
avg_price["Date"] = avg_price["Year"].map(str) + '-' + avg_price["Month"].map(str)

# Create a Month_name column
avg_price["Month_name"] = avg_price["Month"].apply(lambda x: calendar.month_abbr[x])
```

In [114…

```
# Plot of monthly average electricity price
ax = plt.gca()
sns.lineplot(
    data = avg_price,
    x = "Date", y = "Avg_Price")
for index, label in enumerate(ax.get_xticklabels()):
   if index % 12 == 0:
      label.set_visible(True)
   else:
      label.set_visible(False)
plt.xticks(rotation=45)
plt.ylabel("Average Price")
plt.title("Average Monthly Electricity Price (July 2010 - Jan. 2022)")
plt.show()
```

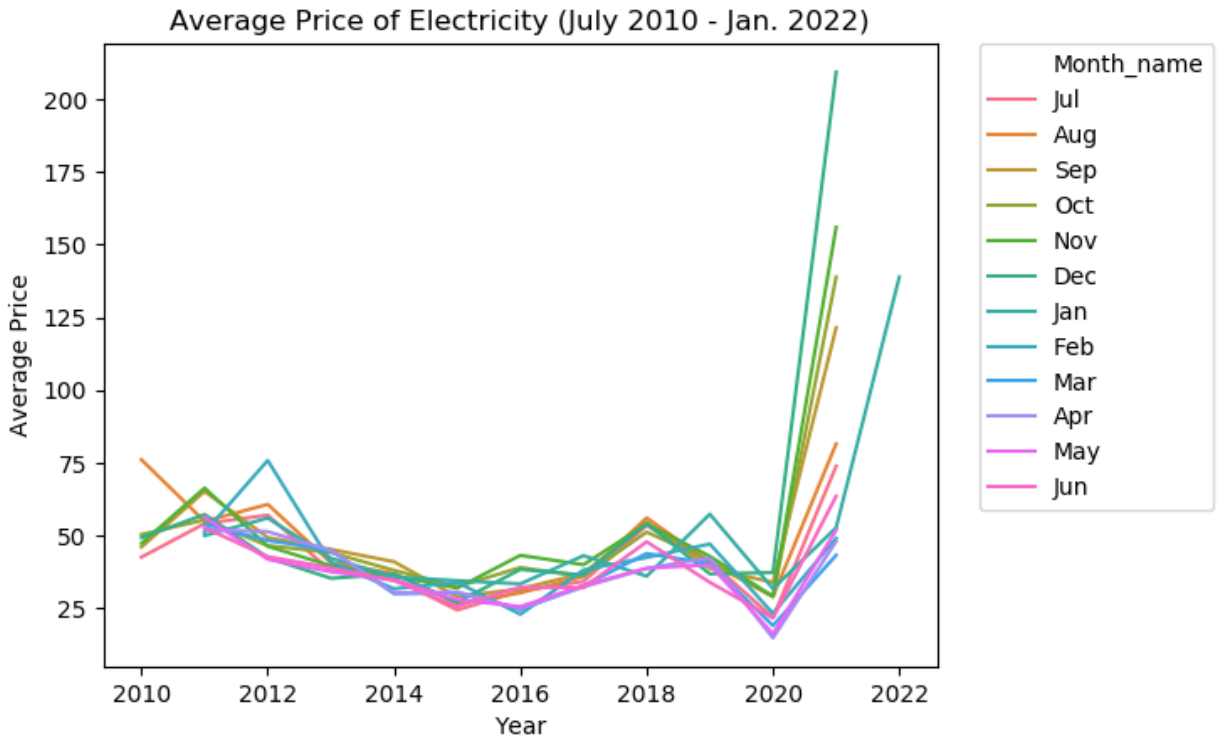
```
INFO:matplotlib.category:Using categorical units to plot a list of strings that are a
ll parsable as floats or dates. If these strings should be plotted as numbers, cast t
o the appropriate data type before plotting.
INFO:matplotlib.category:Using categorical units to plot a list of strings that are a
ll parsable as floats or dates. If these strings should be plotted as numbers, cast t
o the appropriate data type before plotting.
```



There was a large spike in the monthly average electricity price in late 2020, most likely due to increased usage by customers.

In [115…

```
# Plot Monthly Average Price of Electricity Across Years
sns.lineplot(
    data = avg_price,
    x = "Year", y = "Avg_Price",
    hue = "Month_name")
plt.legend(bbox_to_anchor = (1.05, 1), loc = "upper left", borderaxespad = 0)
plt.ylabel("Average Price")
plt.title("Average Price of Electricity (July 2010 - Jan. 2022)")
plt.show()
```



By plotting each average monthly price as its own line, the sudden increase in price began in 2021 and persisted to the start of 2022.

Average Hourly Electricity Price

In [116…

```
query = """
    SELECT YEAR(fecha) AS Year,
        MONTH(fecha) AS Month,
        hora AS Hour,
        AVG(precio) AS Avg_Price
    FROM {}.hourly_european_market_data_csv
    GROUP BY 1, 2, 3
    ORDER BY 1, 2, 3 asc;
    """.format(database_name)

avg_hourly_price_df = pd.read_sql(query, pyathena_conn)
avg_hourly_price_df.head(10)
```

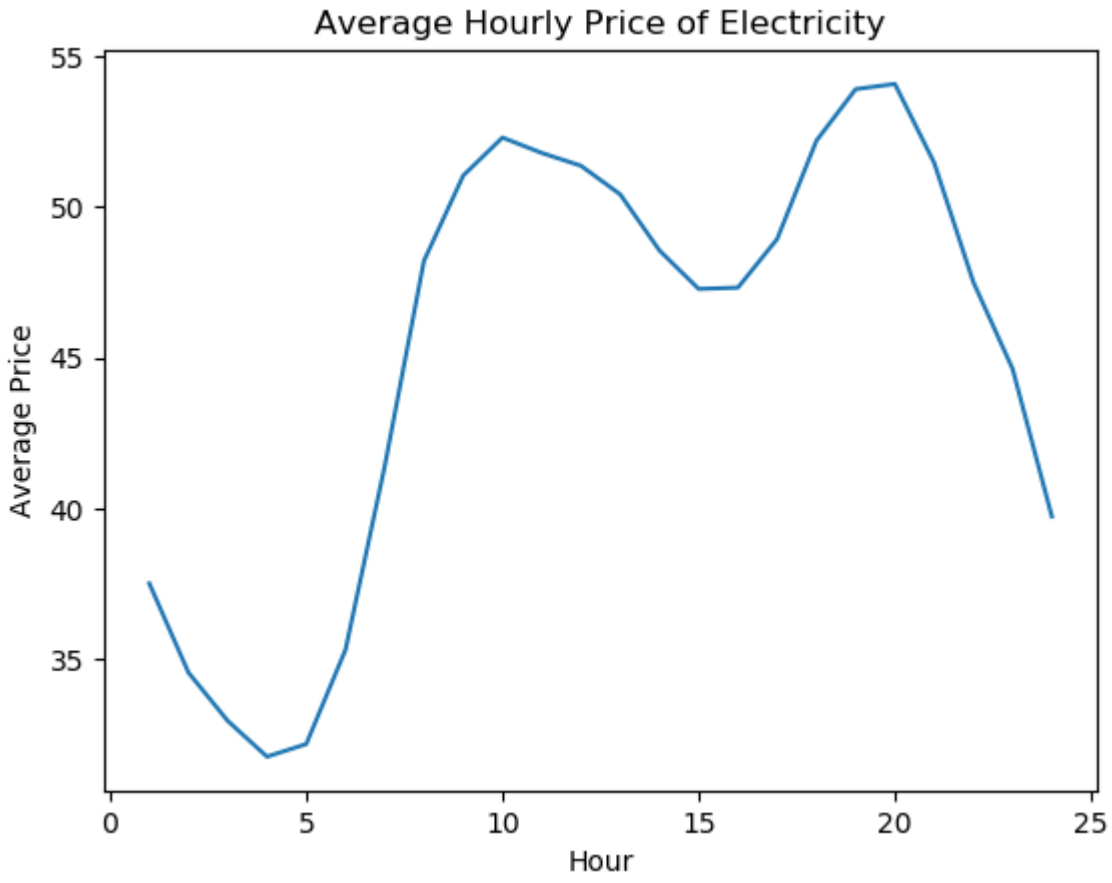
Out[116]:

	Year	Month	Hour	Avg_Price
0	2010	7	1	34.444000
1	2010	7	2	30.008001
2	2010	7	3	28.882545
3	2010	7	4	25.865637
4	2010	7	5	26.308000
5	2010	7	6	30.040090
6	2010	7	7	36.611908
7	2010	7	8	44.018180
8	2010	7	9	43.738728
9	2010	7	10	46.889730

In [117…

```
# Plot the averaged hourly prices of electricity
avg_hourly_price = avg_hourly_price_df.groupby(["Hour"]).mean("Avg_Price")

sns.lineplot(
    data = avg_hourly_price,
    x = avg_hourly_price.index, y = "Avg_Price")
plt.ylabel("Average Price")
plt.title("Average Hourly Price of Electricity")
plt.show()
```



The above plot shows that the most expensive times of the day are around 10:00 and 20:00 (or 10 a.m. and 8 p.m. respectively).

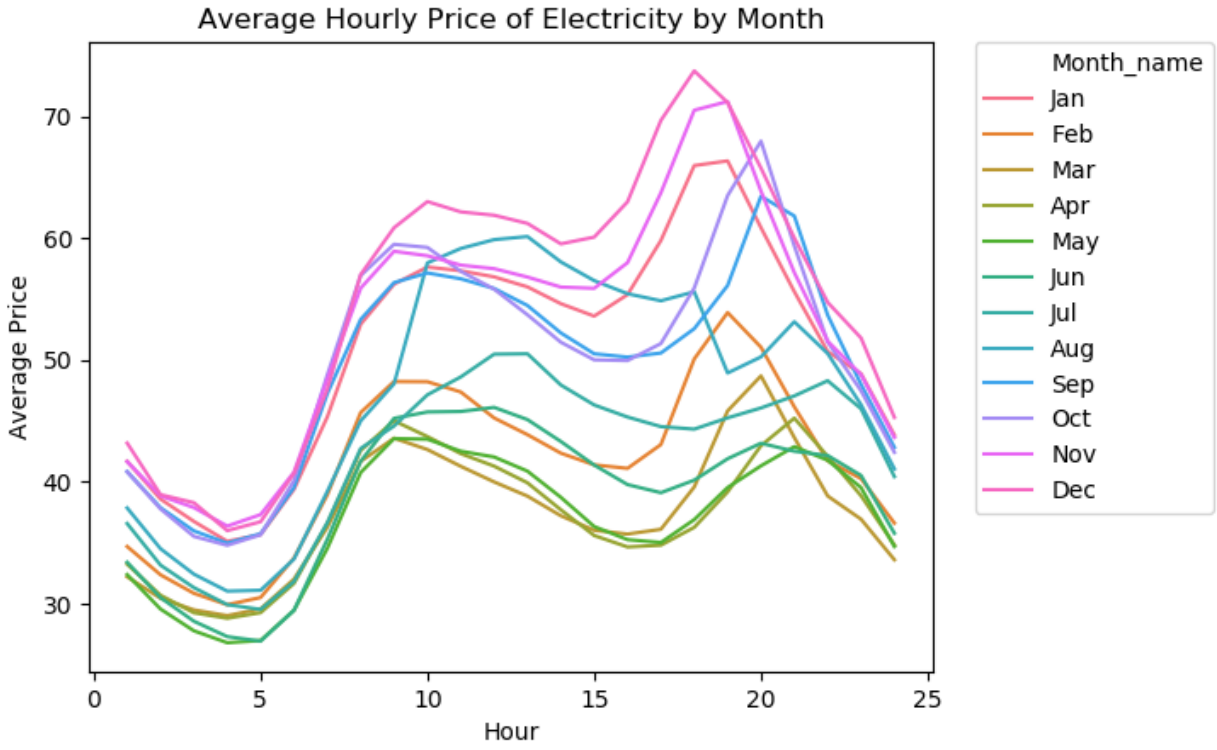
In [118…

```
# Group first by month then by hour
avg_monthly_hourly_price = avg_hourly_price_df.groupby(
    ["Month", "Hour"]).mean(["Avg_Price"]).reset_index()

# Create a Month_name column
avg_monthly_hourly_price["Month_name"] = avg_monthly_hourly_price["Month"].apply(
    lambda x: calendar.month_abbr[x])

# Plot Hourly Data with a line per month
sns.lineplot(
    data = avg_monthly_hourly_price,
    x = "Hour", y = "Avg_Price",
    hue = "Month_name", legend = "full")

plt.legend(bbox_to_anchor = (1.05, 1), loc = "upper left", borderaxespad = 0)
plt.ylabel("Average Price")
plt.title("Average Hourly Price of Electricity by Month")
plt.show()
```



The above plot indicates that the cost of electricity increases with months associated with winter (November, December, January, and February). More specifically, electricity costs rise in the night (between 15:00/3 P.M. and 20:00/8 P.M.).

Pre-Processing

Ingest Data to Pandas Dataframes and Trim All Tables to be within the Same Time Range

The source details that energy dataset ranges from November 23, 2011 to Februrary 28, 2014.

All The datasets will be pared down to this range to create a singular, comprehensive table.

In [119…
```
market_data_df = pd.read_csv(
    "s3://ads-508-spring2023-team3/Hourly_European_Market_Data.csv")
energy_data_df = pd.read_csv(
    "s3://ads-508-spring2023-team3/London_Energy.csv")
weather_data_df = pd.read_csv(
    "s3://ads-508-spring2023-team3/London_Weather_1979_2021.csv")
```

In [120…
```
# Trim data so that all dataframes are in the same time frame
date1 = "2011-11-23"
date2 = "2014-02-28"

# Cast column to datetime format
market_data_df["fecha"] = pd.to_datetime(
    market_data_df["fecha"], format = "%Y-%m-%d")

date_rows = (market_data_df["fecha"] >= date1) & (market_data_df["fecha"] <= date2)
market_data_subset = market_data_df.loc[date_rows]
```

In [121…
```
energy_data_df["Date"] = pd.to_datetime(energy_data_df["Date"], format = "%Y-%m-%d")
date_rows = (energy_data_df["Date"] >= date1) & (energy_data_df["Date"] <= date2)
energy_data_subset = energy_data_df.loc[date_rows]
```

In [122…
```
weather_data_df["date"] = pd.to_datetime(weather_data_df["date"], format = "%Y%m%d")
date_rows = (weather_data_df["date"] >= date1) & (weather_data_df["date"] <= date2)
weather_data_subset = weather_data_df.loc[date_rows]
```

Determining Missing Values

In [123…
```
# Counting number of missing values for each variable in the Market Subset
market_data_subset.isna().sum()
```

Out[123]:
```
Unnamed: 0            0
fecha                 0
hora                  0
sistema               0
bandera               0
precio                0
tipo_moneda           0
origen_dato           0
fecha_actualizacion   0
dtype: int64
```

In [124…
```
# Counting number of missing values for each variable in the Energy Subset
energy_data_subset.isna().sum()
```

Out[124]:
```
LCLid    0
Date     0
KWH      0
dtype: int64
```

In [125…
```
# Counting number of missing values for each variable in the Weather Subset
weather_data_subset.isna().sum()
```

Out[125]:
```
date                0
cloud_cover         1
sunshine            0
global_radiation    0
max_temp            0
mean_temp           0
min_temp            0
precipitation       0
pressure            0
snow_depth          0
dtype: int64
```

In [126…
```
# The below line reveals that cloud_cover is missing a value at row 12114
# weather_data_subset.loc[weather_data_subset["cloud_cover"].isna()]

weather_data_subset.loc[12111:12117]
```

Out[126]:

	date	cloud_cover	sunshine	global_radiation	max_temp	mean_temp	min_temp	precipitatic
12111	2012-02-28	8.0	0.0	31.0	13.0	10.9	8.6	0
12112	2012-02-29	8.0	0.0	32.0	14.4	10.8	8.5	0
12113	2012-03-01	5.0	5.2	105.0	11.7	9.1	3.7	0
12114	2012-03-02	NaN	3.4	88.0	15.0	7.5	3.2	0
12115	2012-03-03	5.0	3.4	89.0	9.4	10.8	6.6	2
12116	2012-03-04	8.0	0.0	37.0	10.3	8.0	6.5	7
12117	2012-03-05	4.0	7.8	135.0	8.9	6.8	3.2	0

As weather usually does not vary dramatically between sequential days, it was deemed appropriate to impute the missing value with the average value of the days prior to and after the missing value. As both of the surrounding values are five, the missing value was filled with a value of 5.

In [127…
```
weather_data_subset.loc[12114, "cloud_cover"] = 5
weather_data_subset.loc[12114, :]["cloud_cover"]
```

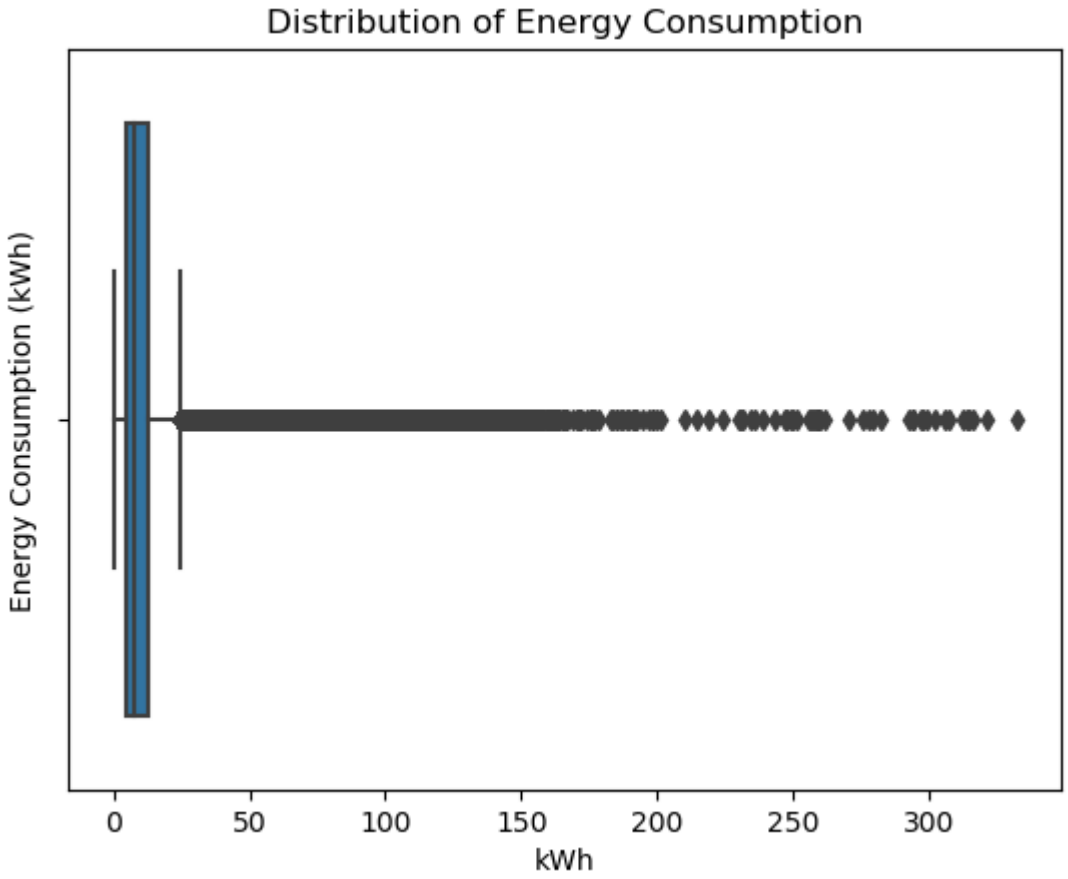
Out[127]:
```
5.0
```

Quick Data Exploration - Distribution of Variables

In [128…

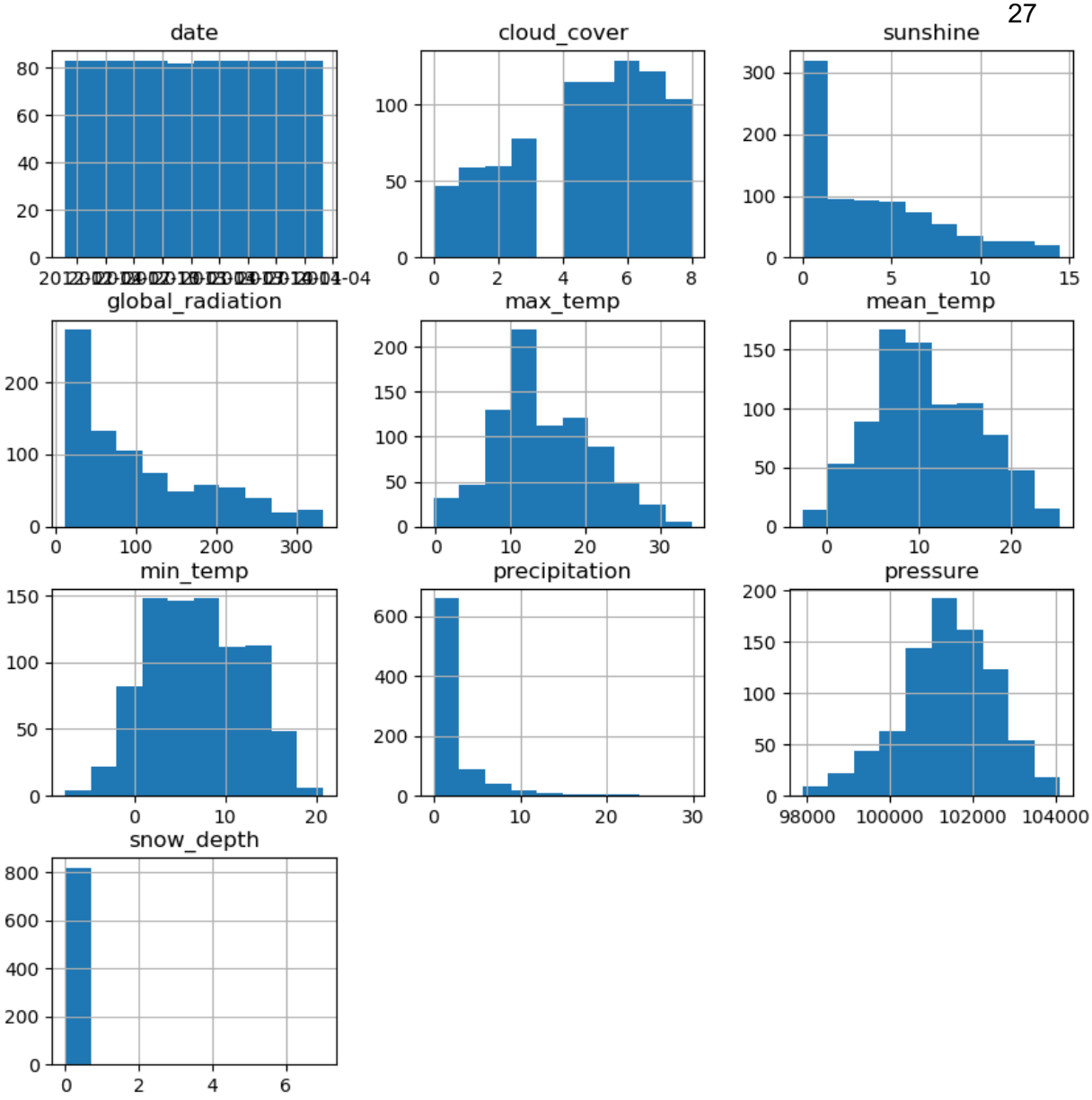
```
# Distribution of Energy Consumption
sns.boxplot(energy_data_subset["KWH"])

plt.title("Distribution of Energy Consumption")
plt.xlabel("kWh")
plt.ylabel("Energy Consumption (kWh)")
plt.show()
```



In [129…

```
weather_data_subset.hist(figsize = (10,10))
plt.show()
```



Drop unnecesary columns

In [130…

```
# Unnamed: 0 = id column (the raw data source did not label the column)
market_data_drop_col = market_data_subset.drop(
    ["Unnamed: 0", "bandera", "origen_dato", "fecha_actualizacion"], axis = 1)
# Rename Date column in market_data to be uniform with other date column names
market_data_drop_col.columns = [
    "Date" if x == "fecha" else x for x in market_data_drop_col.columns]

# Rename Date column in weather_data to be uniform with other date columns
weather_data_subset.columns = [
    "Date" if x == "date" else x for x in weather_data_subset.columns]
```

Subset market_data to sistema = HU

The column *sistema* is the system from which the hourly rates were read from. Only one system is needed for this study. As there are 829 days in the time frame of interest (indicated by the

number of rows in the weather data, after parsing), an hourly data set of the 829 days would have 19,896 values (829 days * 24 hr./day). Indicated below, the only system with all values are system HU. Therefore, the market data subset will be subsetted to only include readings from system HU.

In [131…
```
market_data_drop_col["sistema"].value_counts().head(5)
```

Out[131]:
```
HU     19896
NO3    10176
SYS    10176
SE4    10176
SE3    10176
Name: sistema, dtype: int64
```

In [132…
```
market_data_hu_subset = market_data_drop_col.loc[
    market_data_drop_col["sistema"] == "HU"]

# Drop sistema column - all values are HU
market_data_hu_subset = market_data_hu_subset.drop("sistema", axis = 1)
```

Removing Incomplete Household Records

LCLid in the energy data indicates the households where energy consumption readings were taken from. However, not all households provide all 829 days worth of records. To keep the data consistent and ensure that every day is represented in the training data, LCLid values with a total count of less that 829 will be removed.

In [133…
```
energy_data_id_subset = energy_data_subset.groupby("LCLid").filter(
    lambda x : len(x) == 829)
energy_data_id_subset["LCLid"].value_counts()
```

Out[133]:
```
MAC000145    829
MAC000147    829
MAC000148    829
MAC000149    829
MAC000150    829
MAC000151    829
MAC000152    829
MAC000153    829
MAC000155    829
MAC000156    829
MAC000157    829
Name: LCLid, dtype: int64
```

Only 11 households provided all 829 days of data.

In [134…
```
market_data_hu_subset["tipo_moneda"].value_counts()
```

Out[134]:
```
1    19896
Name: tipo_moneda, dtype: int64
```

All values for tipo_moneda are one. This will not be very informative during model training. Therefore, the column can be removed from the dataframe.

In [135…
```
market_data_hu_subset = market_data_hu_subset.drop("tipo_moneda", axis = 1)
```

Inner Join All Dataframes

Form one large input dataframe for modeling

In [136…
```
# Group hourly market data by date to create one average daily price
market_data_daily_avg = market_data_hu_subset.groupby(
    ["Date"]).mean("precio").reset_index()
```

Creating a daily average price will ensure all value are recorded at the same time interval, daily.

In [137…
```
# Merge the average daily price with the energy consumption data set
merged_df = pd.merge(
    market_data_daily_avg,
    energy_data_id_subset,
    how = "inner",
    on = "Date"
)
```

In [138…
```
# Merge the above joined data frame with the daily weather dataset
merged_df = pd.merge(
    merged_df,
    weather_data_subset,
    how = "inner",
    on = "Date"
)
```

In [139…
```
# Groupby Date to average sampled households energy consumption
# Drop the hour column since data is daily now
full_joined_df = merged_df.groupby(["Date"]).mean().drop(
    "hora", axis = 1).reset_index()
print(full_joined_df.shape)
print(full_joined_df.head(5))
```

```
(829, 12)
        Date     precio        KWH  cloud_cover  sunshine  global_radiation  \
0 2011-11-23  87.079792   7.178909          7.0       2.0              35.0
1 2011-11-24  79.130875  10.911727          3.0       2.0              35.0
2 2011-11-25  71.704208  11.023909          3.0       5.0              52.0
3 2011-11-26  50.857750  11.419455          4.0       0.7              24.0
4 2011-11-27  41.124167  11.972273          3.0       5.9              55.0

   max_temp  mean_temp  min_temp  precipitation  pressure  snow_depth
0      13.5        6.8       2.6            0.2  102720.0         0.0
1      12.5        8.6       3.7            0.2  102710.0         0.0
2      14.0       11.0       9.5            0.0  102450.0         0.0
3      13.9       10.2       6.3            0.0  102580.0         0.0
4      13.2       11.8       9.7            0.0  102130.0         0.0
```

Feature Creation

- Extract Month and Day_of_Week (ex. Mon., Tues., etc.) from the Date column

In [140…
```
# Extract the numerical month value from the date value
full_joined_df["Month"] = full_joined_df["Date"].dt.month
```

In [141…
```
# Create a day of the week column based on the date
# Weekdays are numeric, where 0 represents Monday and 6 represents Sunday
full_joined_df["Day_of_Week"] = full_joined_df["Date"].dt.dayofweek
```

In [142…
```
# Create a daily temperature_range column
# Evaluate the difference betweeen daily max. and min. temperatures
full_joined_df["Temp_range"] = abs(
    full_joined_df["max_temp"] - full_joined_df["min_temp"])
```

Feature Transformation

- Standard scaling is applied to ensure that all features follow an approximate normal distribution.

In [143…
```
# Select Columns that are Numerical for Standardization
# All but the Date Column
num_cols = list(full_joined_df.select_dtypes(include = ["float64", "int64"]).columns)
```

In [144…
```
# Set up column transformer as a pipeline for standardizing numerical columns
minmax_scale = ColumnTransformer([
        ("minmax_scale", MinMaxScaler(), num_cols)
    ],
    remainder = "passthrough"
)

# Standardize the numerical features
scaled_df = pd.DataFrame(minmax_scale.fit_transform(full_joined_df[num_cols]),
            columns = num_cols)
```

In [145…
```
# Insert Date Column to Transformed Dataframe
scaled_df["Date"] = full_joined_df["Date"]
```

Data Partitioning

The data is split using an 80:10:10 train:validation:test partition scheme. Of the 27 months of data, the first 80% of the data lies between November 2011 and Auguest 2013 (inclusively). The validation set will contain records between September 2013 and November 2013; and, the test partition will contain date from December 2013 through February 2014.

In [146…
```
# Partition Data into appropriate time frames
# Training Set: Nov. 2011 - Aug. 2013
train_data = scaled_df.loc[scaled_df["Date"] < "2013-09-01"]

# Validation Set: Sept. 2013 - Nov. 2013
validation_data = scaled_df.loc[
    (scaled_df["Date"] >= "2013-09-01") & (scaled_df["Date"] < "2013-12-01")]
```

```
# Test Set: Dec. 2013 - Feb. 2014
test_data = scaled_df.loc[scaled_df["Date"] >= "2013-12-01"]
```

In [147…

```
# Confirm proportion of data partitions
full_data_size = scaled_df.shape[0]
train_prop = round(train_data.shape[0]/full_data_size * 100, 2)
valid_prop = round(validation_data.shape[0]/full_data_size * 100, 2)
test_prop = round(test_data.shape[0]/full_data_size * 100, 2)

print("Training data is {}% of the dataset".format(train_prop))
print("Validation data is {}% of the dataset".format(valid_prop))
print("Testing data is {}% of the dataset".format(test_prop))
```

```
Training data is 78.17% of the dataset
Validation data is 10.98% of the dataset
Testing data is 10.86% of the dataset
```

Write Partitions of Dataframes to S3 as .CSV Files

In [148…

```
train_data.to_csv(
    "{}/data_partition/training_data/{}".format(
        personal_s3, "train_data.csv"))
validation_data.to_csv(
    "{}/data_partition/validation_data/{}".format(
        personal_s3, "validation_data.csv"))
test_data.to_csv(
    "{}/data_partition/testing_data/{}".format(
        personal_s3, "test_data.csv"))
```

Model Training

SageMaker's built-in algorithm for forecasting, DeepAR, will be utilized as the model of this study.

Code chunks for this section were obtained and interpreted from:

- https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/deepar_synthetic/deepa
- https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/deepar_electricity/Deep Electricity.html
- https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_applying_machine_learning/deepar_chicago

In [149…

```
# Call the forecasting DeepAR image
image_name = sagemaker.image_uris.retrieve("forecasting-deepar", region)
```

```
INFO:sagemaker.image_uris:Same images used for training and inference. Defaulting to
image scope: inference.
INFO:sagemaker.image_uris:Defaulting to the only supported framework/algorithm versio
n: 1.
INFO:sagemaker.image_uris:Ignoring unnecessary instance type: None.
```

In [150…

```
# The frequency of the dataset is 1 Day
freq = "1D"

# Set Dataset Starting Date and Training Data Ending Date
start_dataset = pd.Timestamp("2011-11-23", freq = freq)
end_training = pd.Timestamp("2013-09-01", freq = freq)

prediction_length = 12 * 30 #12 months * 30 days a month
```

Prepare the Data for Training

In [151…

```
# Set Data Index as Date
train_timeindex = train_data.set_index("Date")
validation_timeindex = validation_data.set_index("Date")
test_timeindex = test_data.set_index("Date")
```

In [152…

```
# Re-format Pandas dataframes
## as a List of dictionaries in preparation for JSON file creation
train_list = []
for i in range(train_timeindex.drop("KWH", axis = 1).shape[1]):
    train_list.append(train_timeindex.drop("KWH", axis = 1).iloc[:, i])


training_data = [
    {
        "start" : str(start_dataset),
        "target" : train_timeindex["KWH"][
            start_dataset : end_training].tolist(),
        "dynamic_feat" : [train_list[i][start_dataset : end_training].tolist()]
    }
    for i in range(len(train_list))
]
```

In [153…

```
start_validation = pd.Timestamp("2013-09-01", freq = freq)
end_validation = pd.Timestamp("2013-12-01", freq = freq)
valid_list = []
for i in range(validation_timeindex.drop("KWH", axis = 1).shape[1]):
    valid_list.append(validation_timeindex.drop("KWH", axis = 1).iloc[:, i])


validation_data = [
    {
        "start" : str(start_validation),
        "target" : validation_timeindex["KWH"][
            start_validation : end_validation].tolist(),
        "dynamic_feat" : [valid_list[i][start_validation : end_validation].tolist()]
    }
    for i in range(len(valid_list))
]
```

Write DeepAR JSON-Line Formatted Datasets As .JSON files to S3

In [154…
```
# Function to write the data to .JSON files
def write_dicts_to_file(path, data):
    with open(path, "wb") as fp:
        for d in data:
            fp.write(json.dumps(d).encode("utf-8"))
            fp.write("\n".encode("utf-8"))
```

In [155…
```
# Convert the data to a .JSON extension
write_dicts_to_file("training_data.json", training_data)
write_dicts_to_file("validation_data.json", validation_data)
```

In [156…
```
json_train_bucket = "{}/json_files/training_data".format(personal_s3)
json_valid_bucket = "{}/json_files/validation_data".format(personal_s3)
```

Newly created json files were manually loaded from GtiHub to the personal S3 bucket.

PATH to training.json file: _personal*s3*/json_files/training_data/training_data.json

PATH to validation.json file: _personal*s3*/json_files/validation_data/validation_data.json

Set up Model

In [157…
```
s3_output_path = "{}/DeepAR_Model/Output".format(personal_s3)
```

In [158…
```
deepar_estimator = sagemaker.estimator.Estimator(
    image_uri = image_name,
    sagemaker_session = sagemaker_session,
    role = iam_role,
    instance_count = 1,
    instance_type="ml.m5.xlarge",
    base_job_name = "deepar-train-model",
     output_path = s3_output_path
)
```

In [159…
```
# Set up base hyperparameters
hyperparameters = {
    "time_freq" : freq,
    #How far back to look to predict, apprx. equal to prediction length
    "context_length" : str(90),
    #There are 91 days in the validation set
    "prediction_length" : str(91),
    "epochs" : "400",
    "learning_rate" : "5E-4",
    "mini_batch_size" : "64",
    "early_stopping_patience" : "40",
    "num_dynamic_feat" : "auto"
}

deepar_estimator.set_hyperparameters(**hyperparameters)
```

In [160…
```
# Fit and Train the Model
deepar_estimator.fit(
    inputs = {
        "train" : "{}/json_files/training_data/".format(personal_s3),
        "test" : "{}/json_files/validation_data/".format(personal_s3)
    },
    logs = True,
    wait = False,
)
```

```
INFO:sagemaker:Creating training-job with name: deepar-train-model-2023-04-03-17-04-32-432
```

In [161…
```
training_job_name = deepar_estimator.latest_training_job.name
print("Training Job Name:  {}".format(training_job_name))
```

```
Training Job Name:  deepar-train-model-2023-04-03-17-04-32-432
```

Note: The model takes about 30 minutes to train.

In [162…
```
deepar_estimator.latest_training_job.wait(logs = False)
```

```
2023-04-03 17:04:32 Starting - Starting the training job...
2023-04-03 17:04:53 Starting - Preparing the instances for training........
2023-04-03 17:05:40 Downloading - Downloading input data....
2023-04-03 17:06:05 Training - Downloading the training image........
2023-04-03 17:06:51 Training - Training image download completed. Training in progres
s.....................................................................................
2023-04-03 17:29:03 Uploading - Uploading generated training model..
2023-04-03 17:29:19 Completed - Training job completed
```

View Training and Validation Metrics

In [163…
```
from sagemaker.analytics import TrainingJobAnalytics

training_job_name = training_job_name
metric_name = "test:RMSE"

metrics_dataframe = TrainingJobAnalytics(
    training_job_name = training_job_name,
    metric_names = [metric_name, "train:final_loss"]).dataframe()

metrics_dataframe
```

Out[163]:

	timestamp	metric_name	value
0	0.0	test:RMSE	0.159672
1	0.0	train:final_loss	-1.932093

Create an End Point and Predictor

Create Endpoint

A single instance endpoint on a ml.m5.xlarge instance type.

The following chunk, creating a DeepARPredictor class object, was obtained from the SageMaker DeepAR Demo notebook https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/deepar_electricity/DeepAR-Electricity.html).

In [165…

```
class DeepARPredictor(sagemaker.predictor.Predictor):
    def __init__(self, *args, **kwargs):
        super().__init__(
            *args,
            # serializer=JSONSerializer(),
            serializer=IdentitySerializer(content_type="application/json"),
            **kwargs,
        )

    def predict(
        self,
        ts,
        cat=None,
        dynamic_feat=None,
        num_samples=100,
        return_samples=False,
        quantiles=["0.1", "0.5", "0.9"],
    ):
        """Requests the prediction of for the time series listed in `ts`, each with th
        corresponding category listed in `cat`.

        ts -- `pandas.Series` object, the time series to predict
        cat -- integer, the group associated to the time series (default: None)
        num_samples -- integer, number of samples to compute at prediction time (defau
        return_samples -- boolean indicating whether to include samples in the respons
        quantiles -- list of strings specifying the quantiles to compute (default: ["0

        Return value: list of `pandas.DataFrame` objects, each containing the predicti
        """
        prediction_time = ts.index[-1] + ts.index.freq
        quantiles = [str(q) for q in quantiles]
        req = self.__encode_request(ts, cat, dynamic_feat, num_samples, return_samples
        res = super(DeepARPredictor, self).predict(req)
        return self.__decode_response(res, ts.index.freq, prediction_time, return_samp

    def __encode_request(self, ts, cat, dynamic_feat, num_samples, return_samples, qua
        instance = series_to_dict(
            ts, cat if cat is not None else None, dynamic_feat if dynamic_feat else No
        )

        configuration = {
            "num_samples": num_samples,
            "output_types": ["quantiles", "samples"] if return_samples else ["quantile
            "quantiles": quantiles,
        }

        http_request_data = {"instances": [instance], "configuration": configuration}
```

```
        return json.dumps(http_request_data).encode("utf-8")

    def __decode_response(self, response, freq, prediction_time, return_samples):
        # we only sent one time series so we only receive one in return
        # however, if possible one will pass multiple time series as predictions will
        predictions = json.loads(response.decode("utf-8"))["predictions"][0]
        prediction_length = len(next(iter(predictions["quantiles"].values())))
        prediction_index = pd.date_range(
            start=prediction_time, freq=freq, periods=prediction_length
        )
        if return_samples:
            dict_of_samples = {"sample_" + str(i): s for i, s in enumerate(predictions
        else:
            dict_of_samples = {}
        return pd.DataFrame(
            data={**predictions["quantiles"], **dict_of_samples}, index=prediction_inc
        )

    def set_frequency(self, freq):
        self.freq = freq


def encode_target(ts):
    return [x if np.isfinite(x) else "NaN" for x in ts]


def series_to_dict(ts, cat=None, dynamic_feat=None):
    """Given a pandas.Series object, returns a dictionary encoding the time series.

    ts -- a pands.Series object with the target time series
    cat -- an integer indicating the time series category

    Return value: a dictionary
    """
    obj = {"start": str(ts.index[0]), "target": encode_target(ts)}
    if cat is not None:
        obj["cat"] = cat
    if dynamic_feat is not None:
        obj["dynamic_feat"] = dynamic_feat
    return obj
```

In [166…

```
deepar_endpoint = deepar_estimator.deploy(
    initial_instance_count = 1,
    instance_type = "ml.m5.xlarge",
    predictor_cls = DeepARPredictor
)
```

```
INFO:sagemaker:Creating model with name: deepar-train-model-2023-04-03-17-30-35-659
INFO:sagemaker:Creating endpoint-config with name deepar-train-model-2023-04-03-17-30
-35-659
INFO:sagemaker:Creating endpoint with name deepar-train-model-2023-04-03-17-30-35-659
----!
```

Predict

In [167…
```
prediction_valid_data = validation_timeindex

# Add an arbitrary id column - acts as an indicator of rows for prediction
prediction_valid_data.insert(0, "Id", range(0, 0 + len(prediction_valid_data)))
# Set the frequency of the index to Days
prediction_valid_data.index.freq = "D"
```

In [174…
```
# Test Set Predictions
test_set_predictions = deepar_endpoint.predict(
    ts = prediction_valid_data["Id"],
    # Dynamic_feat = Training Range + prediction_length(91 days = validation set)
    dynamic_feat = [train_timeindex["precio"].append(prediction_valid_data["precio"]).
    quantiles = [0.1, 0.5, 0.9],
)
```

In [183…
```
test_set_predictions
```

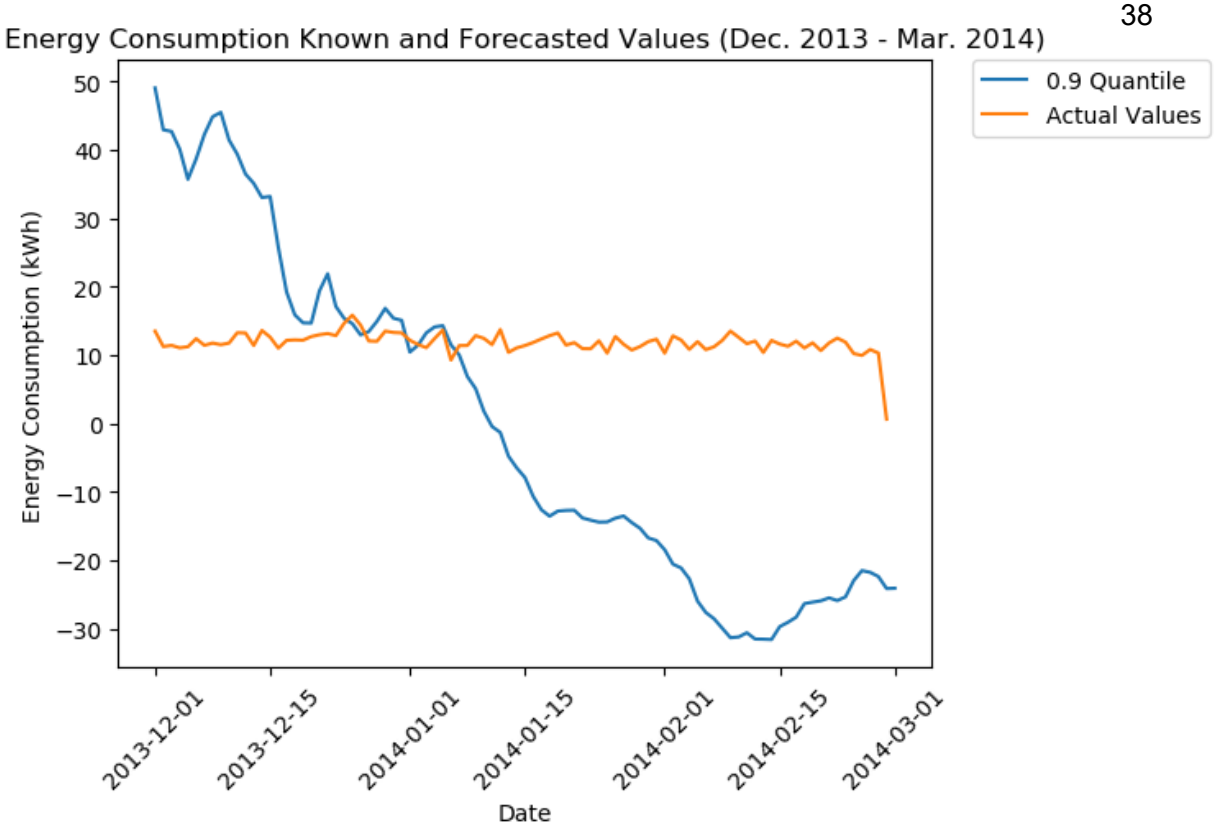
Out[183]:

	0.1	0.5	0.9
2013-12-01	38.733883	44.470448	49.011688
2013-12-02	35.935284	39.588326	42.903229
2013-12-03	35.739975	39.126102	42.663624
2013-12-04	32.361252	36.227840	40.071182
2013-12-05	27.953800	31.979671	35.670059
...	...	...	...
2014-02-25	-29.657372	-25.893055	-21.491856
2014-02-26	-30.506039	-26.783165	-21.744022
2014-02-27	-31.024557	-26.858809	-22.358946
2014-02-28	-32.172989	-28.004890	-24.113199
2014-03-01	-31.948000	-27.911474	-24.068684

91 rows × 3 columns

In [207…
```
plt.plot(test_set_predictions["0.9"],
         label = "0.9 Quantile")
plt.plot(
    full_joined_df.loc[full_joined_df["Date"] >= "2013-12-01"]["Date"],
    full_joined_df.loc[full_joined_df["Date"] >= "2013-12-01"]["KWH"],
    label = "Actual Values"
)

plt.xticks(rotation = 45)
plt.xlabel("Date")
plt.ylabel("Energy Consumption (kWh)")
plt.title("Energy Consumption Known and Forecasted Values (Dec. 2013 - Mar. 2014)")
plt.legend(bbox_to_anchor = (1.05, 1), loc = "upper left", borderaxespad = 0)
plt.show();
```



Delete Model and Endpoints

In [ ]:
```
#deepar_estimator.delete_model()
sagemaker_session.delete_endpoint(deepar_endpoint)
```

Release Resources

In [ ]:
```
%%html

<p><b>Shutting down your kernel for this notebook to release resources.</b></p>
<button class="sm-command-button" data-commandlinker-command="kernelmenu:shutdown" sty

<script>
try {
    els = document.getElementsByClassName("sm-command-button");
    els[0].click();
}
catch(err) {
    // NoOp
}
</script>
```

In [ ]:
```
%%javascript

try {
    Jupyter.notebook.save_checkpoint();
    Jupyter.notebook.session.delete();
}
```

```
catch(err) {
    // NoOp
}
```