Assignment 1, STK-IN4300

Oskar Våle

2024-09-06

```
knitr::opts_chunk$set(echo = TRUE)
library(reticulate)
use_python("/usr/bin/python3")
```

Power consumption dataset

The dataset I chose is the "Individual Household Electric Power Consumption" dataset from the UCI machine learning repository. The dataset contains measurements of electric power consumption in a single household over a period of almost 4 years. The data includes variables such as global active power, global reactive power, voltage, and global intensity, as well as sub-metering values corresponding to different household areas like the kitchen, laundry room, and other appliances. Additionally, the data includes timestamps that have been used to extract the time of day and the month in which the data was recorded.

This is the variable information from UCI's page:

- date: Date in format dd/mm/yyyy
- time: time in format hh:mm:ss
- global_active_power: household global minute-averaged active power (in kilowatt)
- global_reactive_power: household global minute-averaged reactive power (in kilowatt)
- voltage: minute-averaged voltage (in volt)
- global_intensity: household global minute-averaged current intensity (in ampere)
- sub_metering_1: energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).
- sub_metering_2: energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.
- sub_metering_3: energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

I decided to make an 'Hour' column, which is just the hour of the day. This is because energy consumption is likely to vary depending on the time of day, and hour integers will probably be easier for the model to interpret than unix timestamps. I also made a new column 'Month', and let this one just be integers 1 - 12. Due to temperature, holidays and other factors, energy consumption is likely to vary from month to month. There is no point in training on days of the month though, as, except for weekends and holidays, there is no reason one day should have more consumption than the next.

```
import pandas as pd
from ucimlrepo import fetch_ucirepo

# Fetch dataset
individual_household_electric_power_consumption = fetch_ucirepo(id=235)
```

```
## /home/oskarva/.local/lib/python3.10/site-packages/ucimlrepo/fetch.py:97: DtypeWarning: Columns (2,3,4
##   df = pd.read_csv(data_url)
```

```python
# Data (as pandas dataframes)
X = individual_household_electric_power_consumption.data.features
y = individual_household_electric_power_consumption.data.targets

# Variable information
print(individual_household_electric_power_consumption.variables)
```

```
##                    name     role         type  ... description units missing_values
## 0                  Date  Feature         Date  ...        None  None             no
## 1                  Time  Feature  Categorical  ...        None  None             no
## 2   Global_active_power  Feature   Continuous  ...        None  None             no
## 3  Global_reactive_power  Feature   Continuous  ...        None  None             no
## 4               Voltage  Feature   Continuous  ...        None  None             no
## 5      Global_intensity  Feature   Continuous  ...        None  None             no
## 6        Sub_metering_1  Feature   Continuous  ...        None  None             no
## 7        Sub_metering_2  Feature   Continuous  ...        None  None             no
## 8        Sub_metering_3  Feature   Continuous  ...        None  None             no
##
## [9 rows x 7 columns]
```

```python
# Combine features and targets for easier manipulation
data = pd.concat([X, y], axis=1)
# NOTE: The above code lines are taken from UCI's "import in python" function. This
# is a button you can press on the respective dataset's profile on UCI's website
# that gives you all the code you need to import the dataset via UCI's API.

# Convert 'Date' and 'Time' into a single datetime column
data['Datetime'] = pd.to_datetime(data['Date'] + ' ' + data['Time'], format='%d/%m/%Y %H:%M:%S')

# Function to categorize time of day
def categorize_time_of_day(hour):
    return int(hour)

# Apply the function to create a new column 'Hour'
data['Hour'] = data['Datetime'].dt.hour.apply(categorize_time_of_day)

# Extract the month and create a new column 'Month'
data['Month'] = data['Datetime'].dt.month

# Drop the original Date, Time, and Datetime columns if not needed
data = data.drop(columns=['Date', 'Time', 'Datetime'])

# As the output shows, some of the data is still objects. We therefore need to
# convert it to numerical values.
print(data.dtypes)
```

```
## Global_active_power       object
## Global_reactive_power     object
## Voltage                   object
## Global_intensity          object
## Sub_metering_1            object
## Sub_metering_2            object
## Sub_metering_3           float64
## Hour                       int64
```

```
## Month                      int64
## dtype: object
```

```
cols_to_convert = ['Global_active_power', 'Global_reactive_power', 'Voltage',
                   'Global_intensity', 'Sub_metering_1', 'Sub_metering_2']

for col in cols_to_convert:
    data[col] = pd.to_numeric(data[col], errors='coerce')

#Check for NaN values
print(data.isna().sum())
```

```
## Global_active_power      25979
## Global_reactive_power    25979
## Voltage                  25979
## Global_intensity         25979
## Sub_metering_1           25979
## Sub_metering_2           25979
## Sub_metering_3           25979
## Hour                         0
## Month                        0
## dtype: int64
```

```
#Check that all columns now have numerical values
print(data.dtypes)
```

```
## Global_active_power      float64
## Global_reactive_power    float64
## Voltage                  float64
## Global_intensity         float64
## Sub_metering_1           float64
## Sub_metering_2           float64
## Sub_metering_3           float64
## Hour                       int64
## Month                      int64
## dtype: object
```

```
# Drop rows that contain NaN values
data.dropna(axis=0, inplace=True)
```

Statistics on dataset

```
summary_stats = data.describe(include="all")

# Combine the summaries
pd.set_option('display.max_columns', None)
summary_stats.loc['count'] = len(data)
print(summary_stats)
```

```
##        Global_active_power  Global_reactive_power       Voltage  \
## count         2.049280e+06           2.049280e+06  2.049280e+06
## mean          1.091615e+00           1.237145e-01  2.408399e+02
## std           1.057294e+00           1.127220e-01  3.239987e+00
## min           7.600000e-02           0.000000e+00  2.232000e+02
## 25%           3.080000e-01           4.800000e-02  2.389900e+02
```

```
## 50%           6.020000e-01           1.000000e-01  2.410100e+02
## 75%           1.528000e+00           1.940000e-01  2.428900e+02
## max           1.112200e+01           1.390000e+00  2.541500e+02
##
##        Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3  \
## count      2.049280e+06    2.049280e+06    2.049280e+06    2.049280e+06
## mean       4.627759e+00    1.121923e+00    1.298520e+00    6.458447e+00
## std        4.444396e+00    6.153031e+00    5.822026e+00    8.437154e+00
## min        2.000000e-01    0.000000e+00    0.000000e+00    0.000000e+00
## 25%        1.400000e+00    0.000000e+00    0.000000e+00    0.000000e+00
## 50%        2.600000e+00    0.000000e+00    0.000000e+00    1.000000e+00
## 75%        6.400000e+00    0.000000e+00    1.000000e+00    1.700000e+01
## max        4.840000e+01    8.800000e+01    8.000000e+01    3.100000e+01
##
##                Hour         Month
## count  2.049280e+06  2.049280e+06
## mean   1.150391e+01  6.454433e+00
## std    6.925189e+00  3.423209e+00
## min    0.000000e+00  1.000000e+00
## 25%    5.000000e+00  3.000000e+00
## 50%    1.200000e+01  6.000000e+00
## 75%    1.800000e+01  9.000000e+00
## max    2.300000e+01  1.200000e+01
```

Possible use case

A possible use case for this dataset is predicting the power consumption of this user on a given day. Power consumption tends to be quite similar from one person to another, so if we can achieve a good model here, then it can act as a proof-of-concept for a more general application of predicting energy consumption patterns. With the new "smart meters" installed in all houses in Norway, the possibility of getting your own power consumption data is already here. With that in mind, one could create a dataset similar to this one with one's own data, and train a machine learning model on it. With an accurate model, one could roughly predict one's own consumption, and thereby the price of one's power bill, which could be valuable for creating budgets for example.

Bad plot:

Let us first make a bad plot. We will plot the mean daily energy usage of each month.

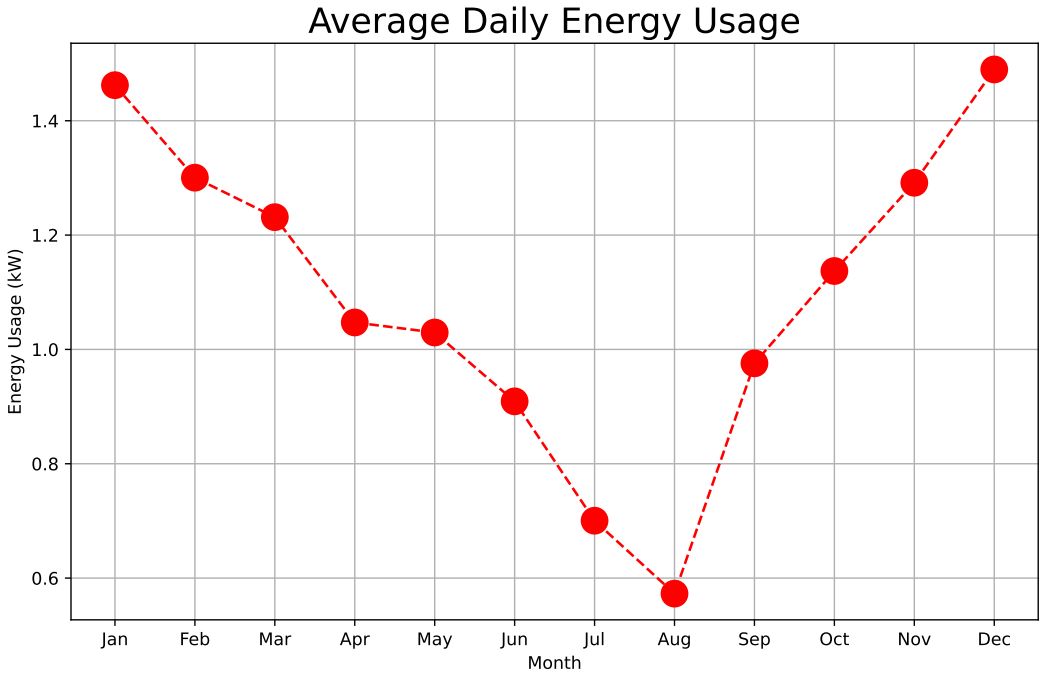
```
import matplotlib.pyplot as plt
import numpy as np

# Group data by month and calculate the mean for Global_active_power
monthly_avg = data.groupby('Month')['Global_active_power'].mean()

# Plot the data
plt.figure(figsize=(10, 6))
plt.plot(monthly_avg.index, monthly_avg.values, color='red', linestyle='--', marker='o', markersize=15)
plt.title('Average Daily Energy Usage', fontsize=20)
plt.xlabel('Month')
plt.ylabel('Energy Usage (kW)')
plt.xticks(ticks=np.arange(1, 13, 1), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
```

```
## ([<matplotlib.axis.XTick object at 0x7f4374dc4d00>, <matplotlib.axis.XTick object at 0x7f4374dc4bb0>
```

```
plt.grid(True)
plt.show()
```



Why this is a bad plot:

- Misleading Title: The title suggests daily usage, but we're actually plotting the average monthly usage.
- Excessive Marker Size: The markers are too large, which can obscure the trend in the data.
- Poor Line Style: The dashed line with large markers isn't the best choice for a time series plot.
- Inadequate Labels: The x-axis labels are abbreviated, which might not be clear for everyone.
- No Error Bars: There are no error bars to show the variability in daily usage within each month.

Good plot

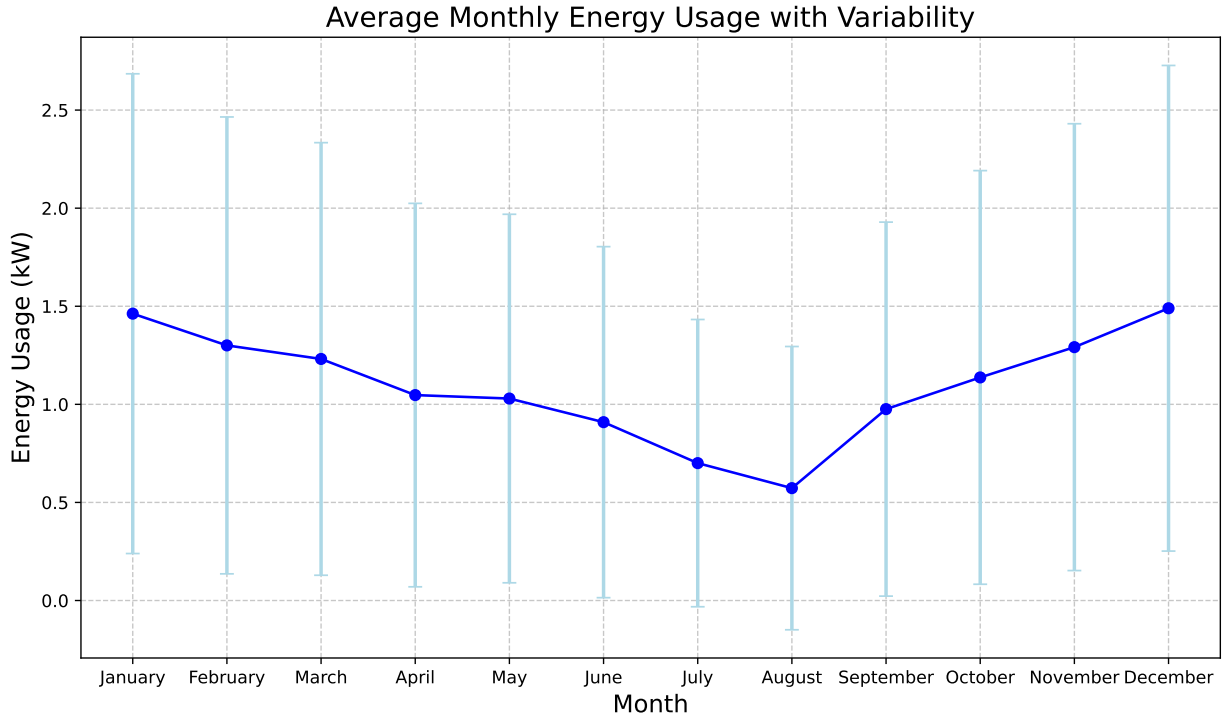
```
import matplotlib.pyplot as plt
import numpy as np

# Group data by month and calculate the mean and standard deviation for Global_active_power
monthly_avg = data.groupby('Month')['Global_active_power'].mean()
monthly_std = data.groupby('Month')['Global_active_power'].std()

# Plot the data
plt.figure(figsize=(10, 6))
plt.errorbar(monthly_avg.index, monthly_avg.values, yerr=monthly_std.values, fmt='-o',
color='blue', ecolor='lightblue', elinewidth=2, capsize=4)
plt.title('Average Monthly Energy Usage with Variability', fontsize=16)
plt.xlabel('Month', fontsize=14)
plt.ylabel('Energy Usage (kW)', fontsize=14)
plt.xticks(ticks=np.arange(1, 13, 1), labels=['January', 'February', 'March', 'April', 'May',
'June', 'July', 'August', 'September', 'October', 'November', 'December'])
```

```
## ([<matplotlib.axis.XTick object at 0x7f4372134820>, <matplotlib.axis.XTick object at 0x7f4372136230>
```

```
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



Why the above plot is good:

- Accurate Title: The title clearly states what the plot shows and includes a mention of variability.
- Appropriate Line and Marker Style: The solid line with smaller markers provides a clean look, making trends easy to follow.
- Error Bars: The error bars represent the variability within each month, giving the viewer a better understanding of the data's spread.
- Clear and Descriptive Labels: The axis labels are detailed, and the month names are fully written out for clarity.
- Improved Color Scheme: The blue line with light blue error bars provides a visually pleasing and easily interpretable color contrast.

A simple analysis

Finally, I will perform a simple analysis of the dataset using linear regression.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error


# Define the features and target variable
X = data[['Global_reactive_power', 'Voltage', 'Global_intensity', 'Sub_metering_1', 'Sub_metering_2', \
'Sub_metering_3', 'Hour', 'Month']]
y = data['Global_active_power']
```

```
possible_features = X.columns

# Split the data into training, validation and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=4)
# Using 50% of the test set as validation set
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test, test_size=0.5, random_state=4)

# Feature selection
from itertools import combinations

# Function to generate all combinations of a list
def generate_combinations(lst):
    all_combinations = []
    for r in range(1, len(lst) + 1):
        comb = list(combinations(lst, r))
        all_combinations.extend(comb)
    return all_combinations

# Generate all combinations
all_combinations = generate_combinations(possible_features)

# Convert to list of lists and print the total count
all_combinations_list = [list(comb) for comb in all_combinations]

# Now we will do a training loop where we fit a model for each combination of features,
# using MSE as selection criteria
best_mse = float('inf')
best_features = None
best_model = None
for i, comb in enumerate(all_combinations_list):
    # Select the features
    X_train_subset = X_train[comb]
    X_val_subset = X_val[comb]

    # Fit the model
    model = LinearRegression()
    model.fit(X_train_subset, y_train)

    # Predict on the validation set
    y_pred = model.predict(X_val_subset)

    # Calculate the mean squared error
    mse = mean_squared_error(y_val, y_pred)

    # Update the best model if the current model is better
    if mse < best_mse:
        best_mse = mse
        best_features = comb
        best_model = model
```

```
    # Uncomment the below line to see training loop progress and MSE values
    #print(f'Iteration {i+1}/{len(all_combinations_list)} - MSE: {mse}')
```

```
import textwrap
wrapped_output = textwrap.fill(str(best_features), width=100)
print(f"Best features: {wrapped_output}")
```

```
## Best features: ['Global_reactive_power', 'Voltage', 'Global_intensity', 'Sub_metering_1', 'Sub_meter
## 'Sub_metering_3', 'Hour', 'Month']
```

```
print(f'Best MSE on validation set: {best_mse}')
```

```
## Best MSE on validation set: 0.0016434470015159795
```

```
# Now it is time to evaluate the best model on the test set to see how good our model really is.
X_test_subset = X_test[best_features]
y_pred = model.predict(X_test_subset)
mse_test = mean_squared_error(y_test, y_pred)
print(f'MSE on test set: {mse_test}')
```

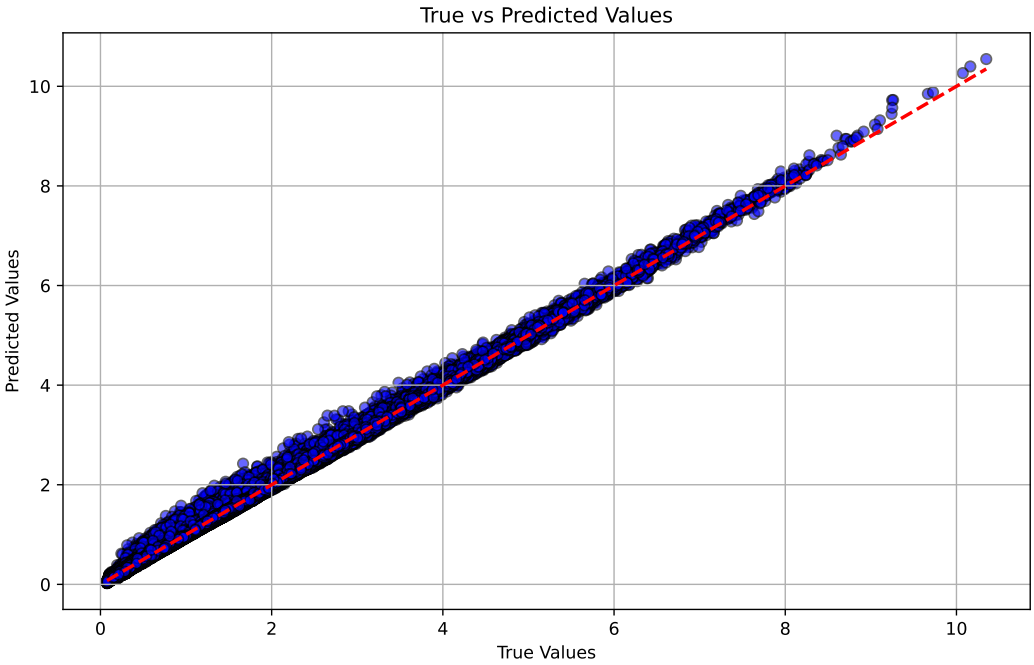
```
## MSE on test set: 0.0016535981595665237
```

Given the results, is a linear model sufficient?

The final MSE we got is very low, which indicates that a linear model is indeed enough in this case. The below plot shows the relationship between the predictions and true values, in terms of how much the predicted values over/undershoot the true values. The red line indicates where the blue dots should optimally be. Given the MSE and the plot showing that the prediction errors are relatively stable linear model can actually be deemed sufficient in this case. Especially considering how explainable linear models are in comparison to some more complicated machine learning methods.

```
import matplotlib.pyplot as plt

# Plotting true vs predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, color='blue', edgecolors='k', alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.title('True vs Predicted Values')
plt.grid(True)
plt.show()
```


True vs Predicted Values
Predicted Values
True Values
0
2
4
6
8
10