SC-2 Electric Boogalo

Regression on the Irish datset agregated by class

Kieran Morris, Cecina Babich Morrow and Sherman Kjo

Contents

1 Cleaning the Data

1.1 Data overview

We are analyzing a set of Irish household electricity demand available from the `electBook` package. We have three datasets:

- `indCons`: 16799 x 2672 matrix of individual household electricity consumption. Each column corresponds to a household and each row to a time point. Demand is observed every half hour, so there are 48 observations per day per household.
- `survey`: 2672 row dataframe of household survey data. This dataset contains household level data on variables such as social class, renting vs. owning, appliances, etc.
- `extra`: 16799 row dataframe of time-related variables. This dataset contains the date-time of each demand observation, time of year, day of week, time of day, whether the day was a holiday, and external temperature.

```r
# Extract individual dataframes
library(electBook)
library(tidyverse)
data(Irish)
indCons <- Irish[["indCons"]]
survey <- Irish[["survey"]]
extra <- Irish[["extra"]]
```

1.2 Social class

We wanted to investigate demand patterns across different social classes. The dataset includes 5 social classes, defined by the occupation of the head of household:

- AB: managerial roles, administrative or professional
- C1: supervisory, clerical, junior managerial
- C2: skilled manual workers
- DE: semi-skilled and unskilled manual workers, state pensioners, casual workers
- F: farmers

** Insert plot of demand patterns for different classes **

We modeled the average demand for each social class separately.

1.3 Loading and Structure

We cleaned the data using the code available in `data_cleaning.R`, which writes out three dataframes: `df_halfhr`, `df_hr`, and `df_day`. These dataframes contain the half-hourly, hourly, and daily average demand data for each social class, as well as temperature (also aggregated to the relevant time scale), time of year (the time of year with 1st January represented as 0 and 31st December represented as 1), and day of the week. These dataframes are written to the `data` folder.

1.4 Feature engineering

Based on exploratory data analysis, we created some features from the dataset to model demand. Feature engineering (with the exception of the addition of the Fourier terms) was performed in the `feature_engineering.R` script, and the resulting dataframes `df_halfhr_scaled`, `df_hr_scaled`, and `df_day_scaled` were saved to the `data` folder.

1.4.1 Time-related features

We extracted the hour of the day and the month from the date-time variable. We also included a quadratic term for temperature to capture the non-linear relationship between temperature and demand. We also used one-hot encoding to include the day of the week in our models.

1.4.2 Temperature

We can visualize the relationship between temperature and the aggregate demand over time across all households:

```r
agg <- colMeans(indCons)

temp_demand <- data.frame(demand = agg) %>%
  bind_cols(Irish[["extra"]])

ggplot(temp_demand, aes(x = dateTime, y = demand, color = temp)) +
  geom_point() +
  viridis::scale_color_viridis(option = "magma") +
  labs(x = "Date", y = "Total Demand", color = "Temperature") +
  theme_bw()
```

We can see that during the warmer summer months, demand dips, although the pattern is messy. We included linear and quadratic terms for temperature in our models.

1.4.3 Fourier terms

We used Fourier terms to capture the patterns of seasonality in the data. Fourier terms are a set of sine and cosine functions with different frequencies that can be used to model periodic patterns. For a given period P, the Fourier terms are defined as follows:

$$\sin_k(t) = \sin\left(\frac{2\pi kt}{P}\right), \quad \cos_k(t) = \cos\left(\frac{2\pi kt}{P}\right)$$

where k is the frequency and t is the time. Then a partial Fourier sum can be written as

$$y(t) = \beta_0 + \sum_{k=1}^{K}\left(\beta_{1k}\sin\left(\frac{2\pi kt}{P}\right) + \beta_{2k}\cos\left(\frac{2\pi kt}{P}\right)\right)$$

where K is the maximum order of Fourier terms to use.

We used Fourier terms to model the daily and annual seasonality in the data. Since our dataset is missing data for some days, we needed to adjust the t used to calculate the Fourier terms to account for these gaps. In the `feature_engineering.R` script, we added the variable `counter` to all dataframes. This variable increments by one for each time step (half hour, hour, or day) in the dataframe, reflecting the fact that some days are missing. These `counter` variables were then used later on to generate Fourier terms for our models. We wrote an `Rcpp` function in our package called `generate_fourier_terms`, which creates a matrix of Fourier terms given a time counter variable representing t, the maximum order of the Fourier terms K, and the number of time increments in a period P.

Insert example of `generate_fourier_terms` function

1.5 Scaling

We centered and scaled all variables used in the models.

1.6 Training and testing

Because this is time series data, we needed to make sure the training and testing data were both contiguous. We trained the models on the first 90% of the rows and tested on the final 10%.

2 Ridge Regression

2.1 Theory

Ridge regression is a method for penalized regression. Consider the model

$$Y_i^0 = \alpha + \beta x_i^0 + \epsilon_i, \quad i = 1, ..., n$$

where $\beta \in \mathbb{R}^p$, $\alpha \in \mathbb{R}$, and for all $i, l \in \{1, ..., n\}$, $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i \epsilon_l] = \sigma^2 \delta_{il}$ for some $\sigma^2 > 0$. Then the ridge regression estimator is defined as the minimizer of the following objective function:

$$(\hat{\alpha}_\lambda, \hat{\beta}_\lambda) = \text{argmin}_{\alpha \in \mathbb{R}, \beta \in \mathbb{R}^p} \|y^0 - \alpha - X^0 \beta\|_2^2 + \lambda \|\beta\|_2^2$$

where $\lambda > 0$ is a tuning parameter and $\|\cdot\|_2$ denotes the Euclidean norm. Ridge regression is thus imposing a penalty on the size of β, with the strength of that penalty determined by the choice of λ. The coefficients will be shrunk towards zero, but will not be set to zero (as opposed to in lasso regression).

2.2 Model

We performed ridge regression using the model detailed above, with the following input variables:

- Temperature and squared temperature
- Time of year
- Hour
- Month
- One-hot encoded variables for each day of the week
- Daily Fourier terms $\sin_k(t) = \sin\left(\frac{2\pi kt}{P}\right), \quad \cos_k(t) = \cos\left(\frac{2\pi kt}{P}\right)$ for $k \in \{1, ..., K\}$
- Annual Fourier terms likewise

We had two hyperparameters to tune: the standard ridge regression parameter λ and the maximum order of the Fourier terms included K. For sake of simplicity, we elected to use the same K value for both the daily and the annual terms.

2.3 Implementation

We implemented ridge regression using `RcppArmadillo`. To select λ, we used k-fold cross-validation, also implemented in `RcppArmadillo` using `RcppParallel` to parallelise the cross-validation process. These functions can be found in our package in the `ridge_regression.h` file.

3 Gaussian Process Regression

3.1 Theory

A gaussian process $W = (W(x))_{x \in \mathcal{X}}$ is a collection of random variables, which have a joint Gaussian distribution. One useful fact is that a Gaussian process is completely specified by its mean ($\mu : \mathcal{X} \to \mathbb{R}$) and covariance ($k : \mathcal{X}^2 \to \mathbb{R}$) functions - meaning the objective of GPR is to obtain the mean and covariance. This allows us to express any finite collection of W as follows:

$$(W(x_1), W(x_2), ..., W(x_n)) \sim N((\mu(x_1), ...\mu(x_n)), (k(x_i, x_j))_{ij})$$

We build the following model:

Let $y_i = f(x_i) + \varepsilon_i$, where $\varepsilon_i \sim N(0, \sigma^2)$ and we let f be a gaussian process. It can be shown that

$$f \mid y^0 \sim \mathrm{GP}(f_n, k_n)$$

where

$$f_n(x) = k^n(x)^T(K + \sigma^2 I_n)^{-1}y^0,$$
$$k(x, x*) = k(x, x^*) - k^n(x)^T(K + \sigma^2 I_n)^{-1}k^n(x^*)$$

and

$$K = (k(x_i^0, x_j^0))_{ij}$$

$$k^n(x) = (k(x_1^0, x), k(x_n^0, x)).$$

Although if it looks scary just remember a computer can do it! This gives us a way to then model

$$Y^0 \sim \mathrm{N}(0, K_\lambda I_n),$$

to simulate our results.

In practice, to find the posterior distribution, we maximise the marginal log-likelihood.

3.2 Tuning hyperparameters

For our needs we will use a radial basis function kernel, which has a single hyperparemter l, often called the bandwidth.

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2l}\right)$$

We also have the hyperparameter σ which is the standard deviation of our noise. In order to find the ideal hyperparameters we will attempt to maximize the marginal log likelihood. The marginal log likelihood is given by:

$$\log p(y^0 \mid x^0, l, \sigma^2) = -\frac{1}{2}y^0(K + \sigma^2 I_n)^{-1}y^0 - \frac{1}{2}\log|K + \sigma^2 I_n|$$

where we remove a constant which is irrelevant to the optimisation. Since this is computationaly intense, we outsource the optimisation to C++, and integrate it via RcppArmadillo. Below is the C++ code we used, which can also be found `gauss_process_reg.cpp`:

```cpp
#include <RcppArmadillo.h>
#include <optim.hpp>

// [[Rcpp::depends(RcppArmadillo)]]

// Define the RBF kernel
arma::mat rbf_kernel(const arma::mat& X, double l, double sigma) {
    arma::mat K = arma::zeros(X.n_rows, X.n_rows);
    for (unsigned i = 0; i < X.n_rows; ++i) {
        for (unsigned j = 0; j < X.n_rows; ++j) {
            double dist = arma::norm(X.row(i) - X.row(j), 2);
            K(i, j) = std::pow(sigma, 2) * std::exp(-std::pow(dist, 2) / (2 * std::pow(l, 2)));
        }
    }
    return K;
}

// Define the negative log marginal likelihood
double neg_log_marginal_likelihood(const arma::vec& theta, arma::vec* grad_out, void* opt_data) {
    // Extract data and parameters
    arma::mat X = *static_cast<arma::mat*>(opt_data);
    arma::vec y = *static_cast<arma::vec*>(opt_data);
    double l = theta(0);
    double sigma = theta(1);

    // Calculate the kernel matrix
    arma::mat K = rbf_kernel(X, l, sigma);

    // Calculate the log marginal likelihood

    double log_likelihood = -0.5 * arma::as_scalar(y.t() * arma::solve(K, y)) - 0.5 * arma::log_det(K +

    // Return the negative log marginal likelihood
    return -log_likelihood;
}

// [[Rcpp::export]]
arma::vec optimise_gaussian_process(arma::mat X, arma::vec y) {
    // Initial guess for the parameters
    arma::vec theta = arma::ones(2);

    // Optimise the negative log marginal likelihood
    bool success = optim::de(theta, neg_log_marginal_likelihood, &X);

    // Return the optimised parameters
    return theta;
}
```

We parallize over the grid of initial choices to speed up the optimisation. As we going to be performing this on different sized datasets (half hourly, hourly and daily) it is unwise to perform parallisation over those. Instead we created an wrapper function in `R` which can take in an arbitrary dataset. We display it below but it can be found in `gauss_process_reg.R`:

```
find_optimal_params <- function(X, y) {
  # Source the C++ file
  sourceCpp("gauss_process_reg.cpp")

  # Call the C++ function
  theta <- optimise_gaussian_process(X, y)

  # Return the optimal parameters
  return(theta)
}
```

3.3 Implementation

For the gaussian process regression, we will use the `kernlab::gausspr` function, which can perform multivariate gaussian process regression, allowing us to add our additional features as covariates. Since we have our ideal hyperparameters l and σ for each dataset, we can now fit the gaussian process regression model fairly easily. Source code for the following can be found in `gauss_process_reg.R`:

```r
gaussian_process_reg <- function(data,
                                 class = "DE",
                                 kernel = "rbfdot",
                                 plot = FALSE,
                                 sigma = 100) {

  #Training and test set, first 90% of data is training, last 10% is test
  train_index <- round(nrow(data) * 0.9)
  train_set <- data[1:train_index, ]
  test_set <- data[(train_index + 1):nrow(data), ]

  # Define the Gaussian process model
  gpr_model <- kernlab::gausspr(as.matrix(train_set[, c("toy", "temp")]),
                                as.vector(train_set[[class]]),
                                kernel = kernel, kpar = list(sigma = sigma))

  # Predict the mean function for plotting
  mean_func <- predict(gpr_model, as.vector(data$toy))
  prediction <- data.frame(
                           toy = data$toy,
                           mean = mean_func)

  data_with_pred <- left_join(data, prediction, by = "toy")

  # Evaluate performance
  test_mean_func <- predict(gpr_model, as.vector(test_set$toy))
  performance <- postResample(pred = test_mean_func, obs = test_set$DE)

  # Include Plots ?
  if (plot){
    pl <- ggplot(data_with_pred, aes(x = toy)) +
      geom_point(aes(y = get(class)), color = "#414141") +
      geom_line(aes(y = mean), color = class_colours[class]) +
      labs(title = "GPR model predictions",
           x = "Date", y = "Demand")
    pl
  } else {
    pl <- NULL
  }
  return(list(model = gpr_model,
              data <- data_with_pred,
              performance,
              plot = pl))
}
```

4 Results