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 Loading and Structure

```r
# Aggregate total
# Frequency is 30 minutes, so each day has 48 ticks
agg <- rowSums(indCons)
```

1.3 Feature engineering

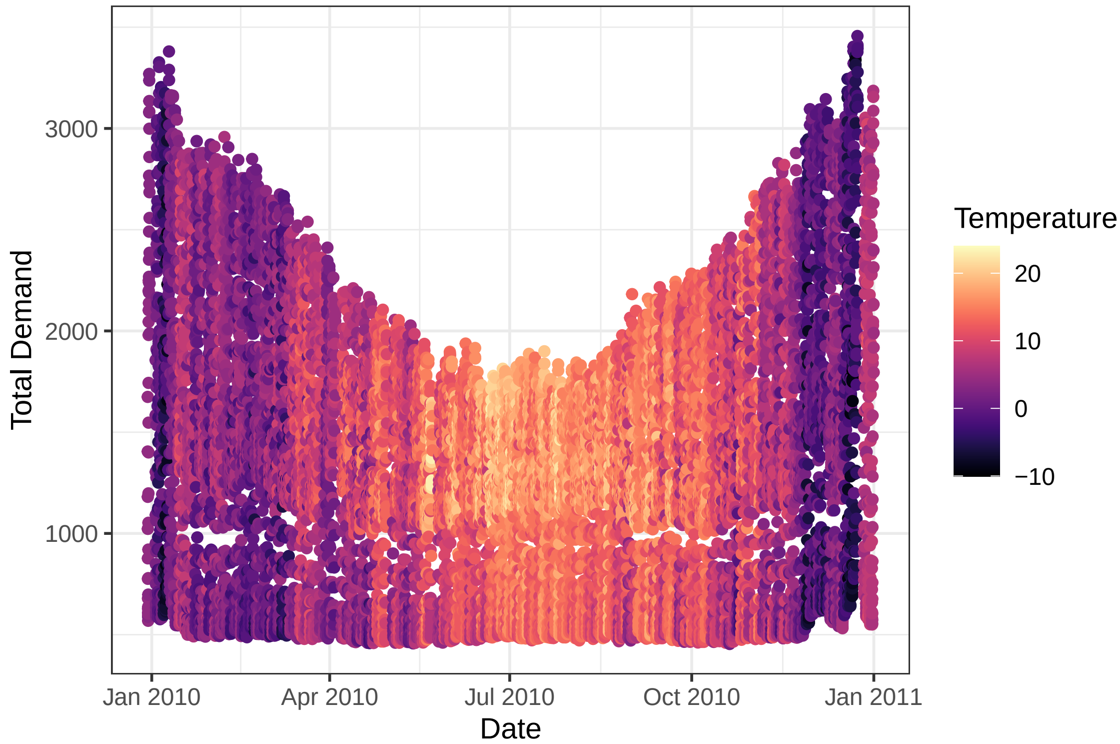
Based on exploratory data analysis, we created some features from the dataset to model demand.

1.3.1 Temperature

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

```r
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.

1.3.2 Day of the week

Since we have the categorical variable day of the week for each date, we used one-hot encoding to include this information in our models.

1.3.3 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.

We used Fourier terms to model the daily and annual seasonality in the data.

1.4 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.

2 Simple Regression

2.1 Theory

2.2 Model

2.3 Implementation

3 Ridge Regression

3.1 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).

3.2 Model

3.3 Implementation

4 Gaussian Process Regression

4.1 Theory

A gaussian process is a collection of random variables, which have a joint Gaussian distribution. A Gaussian process is completely specified by its mean function and covariance function. We build the following model:

Let $y_i = f(x_i) + \varepsilon_i$, where $f(x) \sim \text{GP}(0, k(x, x'))$ and $\varepsilon_i \sim N(0, \sigma^2)$. Then we can find the posterior distribution of $f(x_*)$ given y as:

$$f(x_*)|y \sim N(\mu(x_*), \sigma^2(x_*))$$

where $\mu(x_*) = k(x_*, x)^T (K + \sigma^2 I)^{-1} y$ and $\sigma^2(x_*) = k(x_*, x_*) - k(x_*, x)^T (K + \sigma^2 I)^{-1} k(x_*, x)$. In pratice, to find the posterior distribution, we maximise the marginal log-likelihood.

4.2 Model

4.3 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. By convention we will use the rbf kernel:

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

where l is another hyperparameter to be tuned. Below is our code which fits the gaussian process regression model to given data:

```
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))
}
```

Since we are working as bayesians to find the ideal hyperparameters l and σ, we will minimise the negative log likelihood of the model. On one hand this skirts the issues of cross validation, but on the other hand we are left with a non convex optimisation problem. To get around this issue we outsoucre to C++, making use of the C++ optimising library optim. Then parallizing with RcppArmadillo to parallelise the optimisation over a two dimensional grid of intiial choices. Below is the `RcppArmadillo` code we used to optimise the hyperparameters:

```
#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;
}
```

5 Results