Lab 3.1: Deterministic Prediction Methods

Inverse Distance Weighting

Inverse distance weighting (IDW) is one of the simplest deterministic spatio-temporal interpolation methods. It can be implemented easily in `R` using the function `idw` in the package **gstat**, or from scratch, and in this Lab we shall demonstrate both approaches. We require the following packages.

```
library("dplyr")
library("fields")
library("ggplot2")
library("gstat")
library("RColorBrewer")
library("sp")
library("spacetime")
library("STRbook")
```

We consider the maximum temperature field in the NOAA data set for the month of July 1993. These data can be obtained from the data `NOAA_df_1990` using the `filter` function in **dplyr**.

```
data("NOAA_df_1990", package = "STRbook")
Tmax <- filter(NOAA_df_1990,       # subset the data
              proc == "Tmax" &     # only max temperature
              month == 7 &         # July
              year == 1993)        # year of 1993
```

We next construct the three-dimensional spatio-temporal prediction grid using `expand.grid`. We consider a 20×20 grid in longitude and latitude and a sequence of 6 days regularly arranged in the month.

```
pred_grid <- expand.grid(lon = seq(-100, -80, length = 20),
                         lat = seq(32, 46, length = 20),
                         day = seq(4, 29, length = 6))
```

The function in **gstat** that does the inverse distance weighting, `idw`, takes the following arguments: `formula`, which identifies the variable to interpolate; `locations`, which identifies the spatial and temporal variables; `data`, which can take the data in a data frame; `newdata`, which contains the space-time grid locations at which to interpolate; and `idp`, which corresponds to α in (3)below. The larger α (`idp`) is, the less the smoothing. This parameter is typically set using cross-validation, which we explore later in this Lab; here we fix $\alpha = 5$. We run `idw` below with the variable `Tmax`, omitting data on 14 July 1993.

```
Tmax_no_14 <- filter(Tmax, !(day == 14))         # remove day 14
Tmax_July_idw <- idw(formula = z ~ 1,            # dep. variable
                     locations = ~ lon + lat + day, # inputs
                     data = Tmax_no_14,           # data set
                     newdata = pred_grid,         # prediction grid
                     idp = 5)                     # inv. dist. pow.
```

The output `Tmax_July_idw` contains the fields `lon`, `lat`, `day`, and `var1.pred` corresponding to the IDW interpolation over the prediction grid. This data frame can be plotted using **ggplot2** commands as follows.

```
ggplot(Tmax_July_idw) +
    geom_tile(aes(x = lon, y = lat,
                  fill = var1.pred)) +
    fill_scale(name = "degF") +     # attach color scale
    xlab("Longitude (deg)") +           # x-axis label
    ylab("Latitude (deg)") +            # y-axis label
    facet_wrap(~ day, ncol = 3) +       # facet by day
    coord_fixed(xlim = c(-100, -80),
                ylim = c(32, 46))  +    # zoom in
    theme_bw()                          # B&W theme
```

Notice how the day with missing data is "smoothed out" when compared to the others. As an exercise, you can redo IDW including the 14 July 1993 in the data set, and observe how the prediction changes for that day.

Implementing IDW from First Principles

It is often preferable to implement simple algorithms, like IDW, from scratch, as doing so increases code versatility (e.g., it facilitates implementation of a cross-validation study). Reducing dependence on other packages will also help the code last the test of time (as it becomes immune to package changes).

It can be shown that the IDW interpolator is given by

$$\widehat{Z}(\mathbf{s}_0;t_0) = \sum_{j=1}^{T}\sum_{i=1}^{m_j} w_{ij}(\mathbf{s}_0;t_0) Z(\mathbf{s}_{ij};t_j), \tag{1}$$

where

$$w_{ij}(\mathbf{s}_0;t_0) \equiv \frac{\widetilde{w}_{ij}(\mathbf{s}_0;t_0)}{\sum_{k=1}^{T}\sum_{\ell=1}^{m_k}\widetilde{w}_{k\ell}(\mathbf{s}_0;t_0)}, \tag{2}$$

and

$$\widetilde{w}_{ij}(\mathbf{s}_0;t_0) \equiv \frac{1}{d(\{\mathbf{s}_{ij},t_j\},\{\mathbf{s}_0,t_0\})^{\alpha}}. \tag{3}$$

Moreover, we can treat IDW as a kernel predictor. That is, in (3) we can let

$$\widetilde{w}_{ij}(\mathbf{s}_0; t_0) = k(\{\mathbf{s}_{ij}; t_j\}, \{\mathbf{s}_0; t_0\}; \theta),$$

where $k(\{\mathbf{s}_{ij}; t_j\}, \{\mathbf{s}_0; t_0\}; \theta)$ is a *kernel function* (i.e., a function that quantifies the similarity between two locations) that depends again on the distance between $\{\mathbf{s}_{ij}; t_j\}$ and $\{\mathbf{s}_0; t_0\}$ and some *bandwidth* parameter, θ (in this case equal to α). To construct these kernel weights we first need to find the distances between all prediction locations and data locations, take their reciprocals and raise them to the power (`idp`) of α. Pairwise distances between two arbitrary sets of points are most easily computed using the **`rdist`** function in the package **fields**. Since we wish to generate these kernel weights for different observation and prediction sets and different bandwidth parameters, we create a function **`Wt_IDW`** that generates the required kernel-weights matrix.

```
pred_obs_dist_mat <- rdist(select(pred_grid, lon, lat, day),
                           select(Tmax_no_14, lon, lat, day))
Wt_IDW <- function(theta, dist_mat) 1/dist_mat^theta
Wtilde <- Wt_IDW(theta = 5, dist_mat = pred_obs_dist_mat)
```

The matrix `Wtilde` now contains all the $\widetilde{w}_{ij}$ described in (3); that is, the (k, l)th element in `Wtilde` contains the distance between the kth prediction location and the lth observation location, raised to the power of 5, and reciprocated.

Next, we compute the weights in (2). These are just the kernel weights normalized by the sum of all kernel weights associated with each prediction location. Normalizing the weights at every location can be done easily using **`rowSums`** in `R`.

```
Wtilde_rsums <- rowSums(Wtilde)
W <- Wtilde/Wtilde_rsums
```

The resulting matrix `W` is the weight matrix, sometimes known as the *influence matrix*. The predictions are then given by (1), which is just the influence matrix multiplied by the data.

```
z_pred_IDW <- as.numeric(W %*% Tmax_no_14$z)
```

One can informally verify the computed predictions by comparing them to those given by **`idw`** in **gstat**. We see that the two results are very close; numerical mismatches of this order of magnitude are likely to arise from the slightly different way the IDW weights are computed in **gstat** (and it is possible that you get different, but still small, mismatches on your computer).

```
summary(Tmax_July_idw$var1.pred - z_pred_IDW)

##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -1.53e-12 -1.70e-13  0.00e+00  0.00e+00  1.70e-13  9.80e-13
```

Generic Kernel Smoothing and Cross-Validation

One advantage of implementing IDW from scratch is that now we can change the kernel function to whatever we want and compare predictions from different kernel functions. We implement a kernel smoother below, where the kernel is a Gaussian radial basis function given by

$$k(\{\mathbf{s}_{ij};t_j\},\{\mathbf{s}_0;t_0\};\theta)\equiv\exp\left(-\frac{1}{\theta}d(\{\mathbf{s}_{ij};t_j\},\{\mathbf{s}_0;t_0\})^2\right), \tag{4}$$

with $\theta = 0.5$.

```
theta <- 0.5                             # set bandwidth
Wt_Gauss <- function(theta, dist_mat) exp(-dist_mat^2/theta)
Wtilde <- Wt_Gauss(theta = 0.5, dist_mat = pred_obs_dist_mat)
Wtilde_rsums <- rowSums(Wtilde)          # normalizing factors
W <- Wtilde/Wtilde_rsums                 # normalized kernel weights
z_pred2 <- W %*% Tmax_no_14$z            # predictions
```

The vector `z_pred2` can be assigned to the prediction grid `pred_grid` and plotted using **ggplot2** as shown above. Note that the the predictions are similar, but not identical, to those produced by IDW. But which predictions are the best in terms of squared prediction error? A method commonly applied to assess goodness of fit is known as *cross-validation* (CV). CV also allows us to choose bandwidth parameters (i.e., α or θ) that are optimal for a given data set.

To carry out CV, we need to fit the model using a subset of the data (known as the training set), predict at the data locations that were omitted (known as the validation set), and compute a *discrepancy*, usually the squared error, between the predicted and observed values. If we leave one data point out at a time, the procedure is known as leave-one-out cross-validation (LOOCV). We denote the sum of the discrepancies for a particular bandwidth parameter θ as the LOOCV score, $CV_{(m)}(\theta)$ (note that m, here, is the number of folds used in the cross-validation; in LOOCV, the number of folds is equal the number of data points, m).

The LOOCV for simple predictors, like kernel smoothers, can be computed analytically without having to refit.Since the data set is reasonably small, it is feasible here to do the refitting with each data point omitted (since each prediction is just an inner product of two vectors). The simplest way to do LOOCV in this context is to compute the pairwise distances between *all* observation locations and the associated kernel-weight matrix, and then to select the appropriate rows and columns from the resulting matrix to do prediction at a left-out observation; this is repeated for every observation.

The distances between all observations are computed as follows.

```
obs_obs_dist_mat <- rdist(select(Tmax, lon, lat, day),
                          select(Tmax, lon, lat, day))
```

A function that computes the LOOCV score is given as follows.

```
LOOCV_score <- function(Wt_fun, theta, dist_mat, Z) {
  Wtilde <- Wt_fun(theta, dist_mat)
  CV <- 0
  for(i in 1:length(Z)) {
    Wtilde2 <- Wtilde[i,-i]
    W2 <- Wtilde2 / sum(Wtilde2)
    z_pred <- W2 %*% Z[-i]
    CV[i] <- (z_pred - Z[i])^2
  }
  mean(CV)
}
```

The function takes as arguments the kernel function that computes the kernel weights `Wt_fun`; the kernel bandwidth parameter `theta`; the full distance matrix `dist_mat`; and the data `Z`. The function first constructs the kernel-weights matrix for the given bandwith. Then, for the ith observation, it selects the ith row and excludes the ith column from the kernel-weights matrix and assigns the resulting vector to `Wtilde2`. This vector contains the kernel weights for the ith observation location (which is now a prediction location) with the weights contributed by this ith observation removed. This vector is normalized and then cross-multiplied with the data to yield the prediction. This is done for all $i = 1, \dots, n$, and then the mean of the squared errors is returned. To see which of the two predictors is "better," we now simply call **`LOOCV_score`** with the two different kernel functions and bandwidths.

```
LOOCV_score(Wt_fun = Wt_IDW,
            theta = 5,
            dist_mat = obs_obs_dist_mat,
            Z = Tmax$z)

## [1] 7.8

LOOCV_score(Wt_fun = Wt_Gauss,
            theta = 0.5,
            dist_mat = obs_obs_dist_mat,
            Z = Tmax$z)

## [1] 7.5
```

Clearly the Gaussian kernel smoother has performed marginally better than IDW in this case. But how do we know the chosen kernel bandwidths are suitable? Currently we do not, as these were set by simply "eye-balling" the predictions and assessing visually whether they looked suitable or not. An objective way to set the bandwidth parameters is to put them equal to those values that minimize the LOOCV scores. This can be done by simply computing `LOOCV_score` for a set, say 21, of plausible bandwidths and finding the minimum. We do this below for both IDW and the Gaussian kernel.

```
theta_IDW <- seq(4, 6, length = 21)
theta_Gauss <- seq(0.1, 2.1, length = 21)
CV_IDW <- CV_Gauss <- 0
for(i in seq_along(theta_IDW)) {
  CV_IDW[i] <- LOOCV_score(Wt_fun = Wt_IDW,
                           theta = theta_IDW[i],
                           dist_mat = obs_obs_dist_mat,
                           Z = Tmax$z)

  CV_Gauss[i] <- LOOCV_score(Wt_fun = Wt_Gauss,
                             theta = theta_Gauss[i],
                             dist_mat = obs_obs_dist_mat,
                             Z = Tmax$z)
}
```

The plots showing the LOOCV scores as a function of α and θ for the IDW and Gaussian kernels, respectively, exhibit clear minima when plotted, which is very typical of plots of this kind.

```
par(mfrow = c(1,2))
plot(theta_IDW, CV_IDW,
     xlab = expression(alpha),
     ylab = expression(CV[(m)](alpha)),
     ylim = c(7.4, 8.5), type = 'o')
plot(theta_Gauss, CV_Gauss,
     xlab = expression(theta),
     ylab = expression(CV[(m)](theta)),
     ylim = c(7.4, 8.5), type = 'o')
```

The optimal inverse-power and minimum LOOCV score for IDW are

```
theta_IDW[which.min(CV_IDW)]

## [1] 5

min(CV_IDW)

## [1] 7.8
```

The optimal bandwidth and minimum LOOCV score for the Gaussian kernel smoother are

```
theta_Gauss[which.min(CV_Gauss)]

## [1] 0.6

min(CV_Gauss)

## [1] 7.5
```

Our choice of $\alpha = 5$ was therefore (sufficiently close to) optimal when doing IDW, while a bandwidth of $\theta = 0.6$ is better for the Gaussian kernel than our initial choice of $\theta = 0.5$. It is clear from the results that the Gaussian kernel predictor with $\theta = 0.6$ has, in this example, provided superior performance to IDW with $\alpha = 5$, in terms of mean-squared-prediction error.

Bibliography