STA5068Z Machine Learning
Overfitting: Concept Discovery & Generalization

Zach Wolpe - WLPZAC001

19 October 2020

Abstract

In this assignment we study the conditions that allow one to learn a concept from data in a generalized fashion. When learning a concept from data, we wish to approximate an unknown target function given noisy data that represents an instance of the target.

There are two primary conditions that allow one to learn a concept from data: a sufficient signal-to-noisy ratio in the sample; & a sufficient number of examples (sample size) from which to extrapolate some pattern. We will also examine the importance of specifying a suitable hypothesis set. The goal of learning is to generalize - extrapolating to make predictions & perform inference outside of the sample data available. One should make model design decisions accordingly.

The goal of this assignment is to consider the conditions that impede generalization: exorbitant noise in the data generating process; small sample size & limitations of the chosen hypothesis set.

1 Question 1

1.1 Legendre Polynomials

Our target functions will consist of either standard polynomials or Legendre polynomials of order q, Legendre polynomials given by:

$$L_q(x) = 2^q \sum_{k=0}^{q} x^k \binom{q}{k} \binom{\frac{q+k-1}{2}}{q}$$

We begin by visualizing the Legendre polynomials for $q = \{0, 1, 2, 3, 4, 5\}$ over the range $\mathcal{X} \in [-1, 1]$.

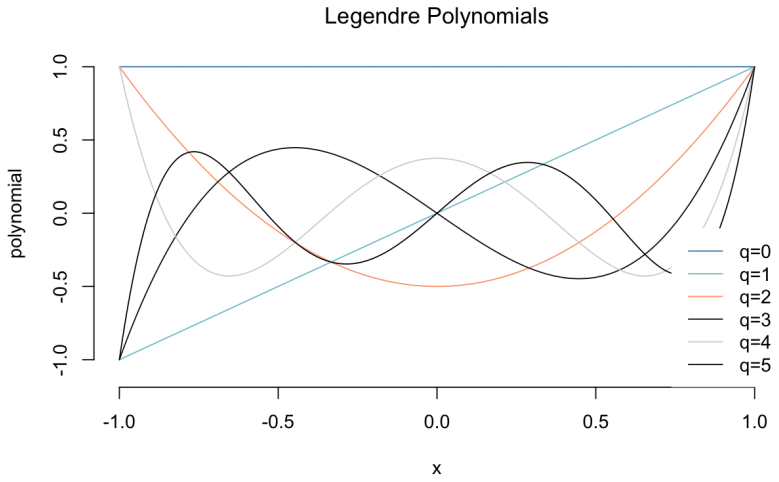

Figure 2: Legendre polynomials for Q

The Legendre polynomials make particularly appealing basis functions for two reasons:

1. They take the form of highly flexible basis functions.
2. They are orthogonal over this $X \in [-1, 1]$ range, thus linearly independent, & when used to generate a target we need not concern ourselves with multicollinearity (dependencies between variables). Each basis then also adds a full dimension - without overlap with other basis functions.

1.2 Random Target Functions

The first target is given by polynomials of the form:

$$f(x) = \sum_{q=0}^{Q_f} \alpha_q x^q$$

We generate random α coefficents to multiply with the x^q basis functions. Figure 3 details 3 independent target functions generated this way with $Q_f = 25$.

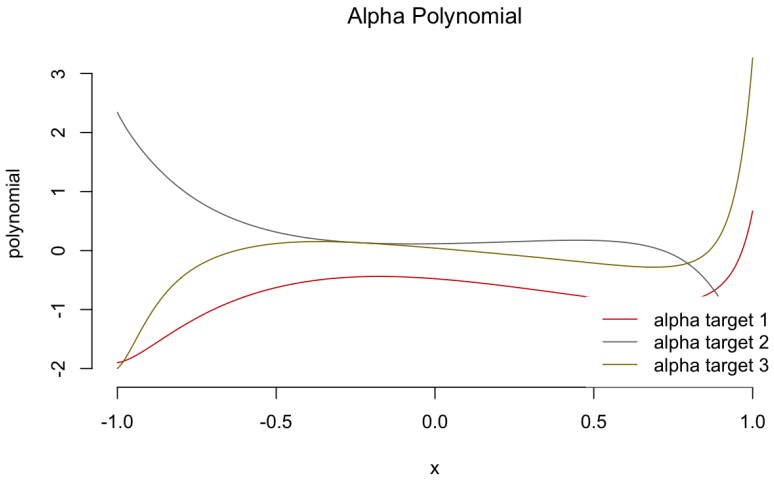

Figure 3: Randomly generated α targets for $Q_f = 25$

Due to the none-orthogonal nature of the polynomials, the function produced does not appear to be a high order polynomial as many of the 'effects' cancel out or repeat one another. The curve produced is thus very smooth.

It may be more interested to look at targets for a range of f values, figure 4 again generates 3 independent targets by randomly samples the coeffients $\alpha \sim UNIFORM[-1, 1]$, but now over 6 different values for $f = \{5, 10, 15, 20, 25, 30\}$.

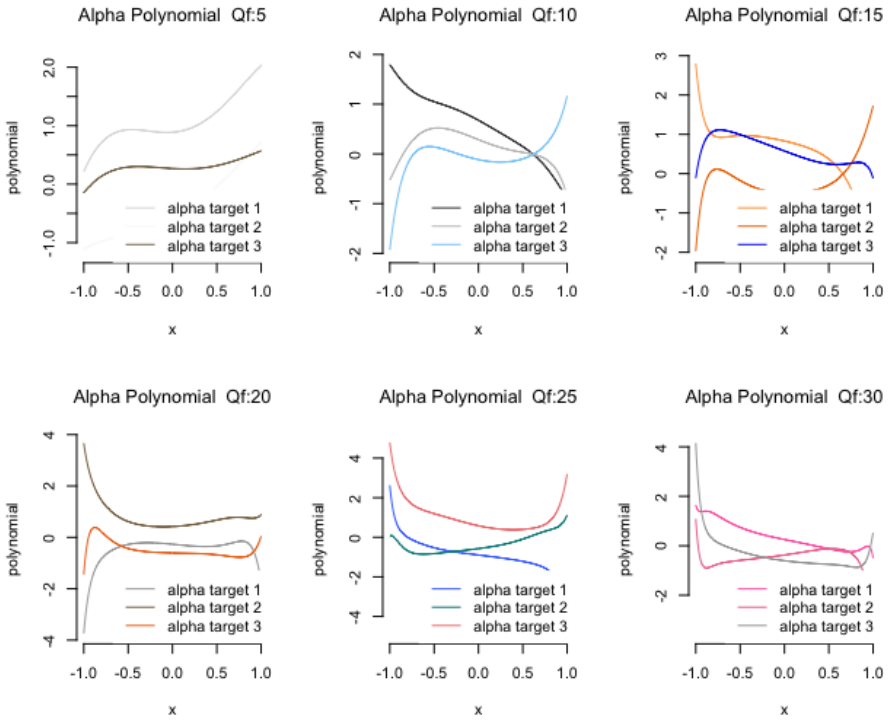

Figure 4: Generate αs over a range of Q_f's

Additionally we consider generating target functions using Legendre polynomials:

$$f(x) = \sum_{q=0}^{Q_f} L_q x^q$$

where:

$$L_q(x) = 2^q \sum_{k=0}^{q} x^k \binom{q}{k} \binom{\frac{q+k-1}{2}}{q}$$

Again, one can see 3 independent targets generated by randomly sampling the β coeffients $\beta \sim UNIFORM[-1, 1]$ & multiplying by the Legendre polynomials basis functions. Figure 5 show the case for $Q_f = 25$ - whilst figure **??** details the same result over $Q_f = \{5, 10, 15, 20, 25, 30\}$.

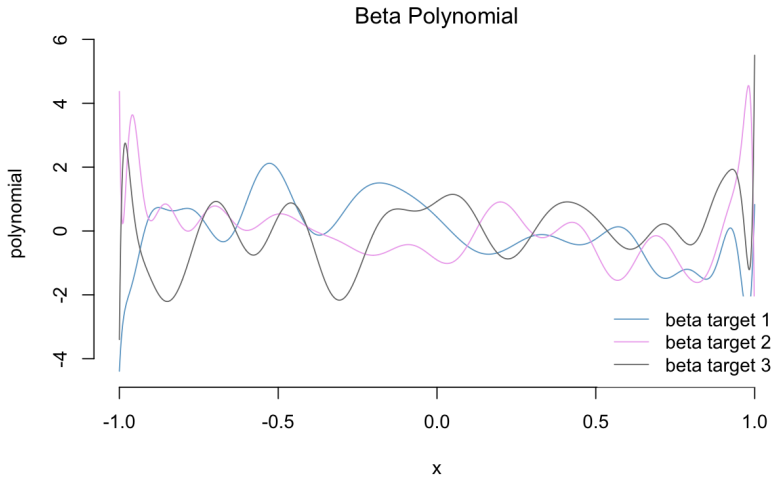

Figure 5: Randomly generated βs targets for $Q_f = 25$

We can immediately see from figure 5 that functions with high variability are produced. These functions are polynomials as their are simple the weighted sum of other polynomials. The orthogonal nature of the Legendre polynomials result in the high order producing great variability - a violent target is produced.

Taking a closer look at the two target generating methods, over the various Q^f values, yields some interesting insight. Again the key distinction between the two methods is the independence of the Legendre polynomials.

Over the range of Q^f, the standard polynomials do not become significantly more chaotic & violent. This is due to great overlap in influence - dependence in the basis functions.

The Legendre polynomials, however, exhibit far more variability at higher order of Q^f, similarly so at lower orders of Q^f compared to the same order of the standard polynomial implementations.

It is intuitive to see that attempting to learn such a highly variable function could be problematic even in the absence of severe noise disturbance. Moreover, even if fitted correctly, in a real setting one would suspect overfitting on visualization.

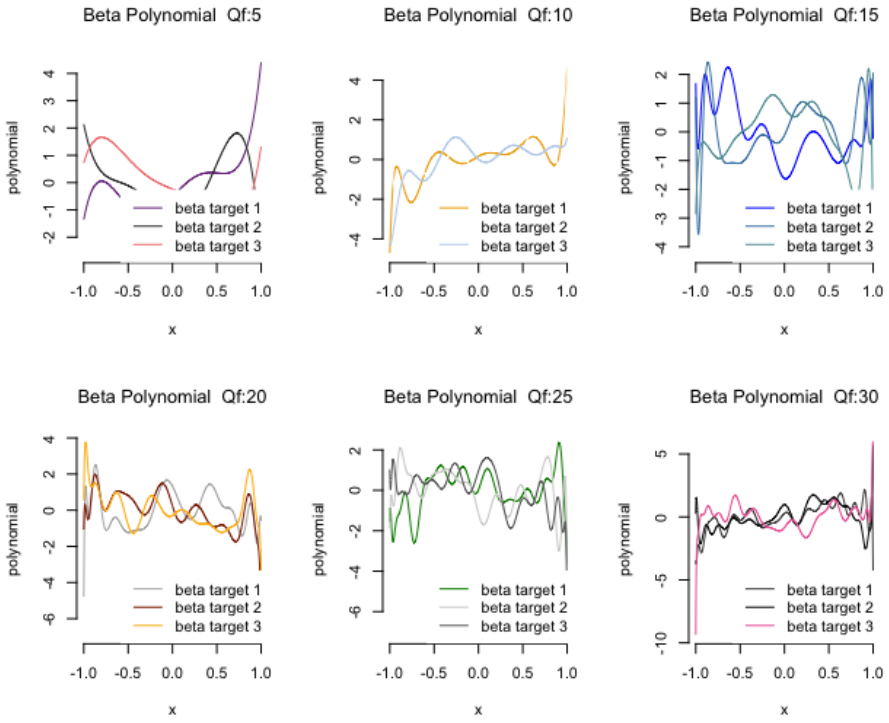

Figure 6: Generate βs over a range of Q_f's

We note the aggregate Legendre polynomials become increasingly chaotic with Q^f - which is not observed in the standard polynomial.

2 Question 2

2.1 Overfitting: Fitting Noise

Overfitting can simply be thought fitting the data more than is warranted. Here we contrast the relative performance of the two hypothesis by estimating the overfit measure:

$$\mathbf{E}_{\mathcal{D}}[E_{out}g_{10}^{\mathcal{D}} - E_{out}g_{2}^{\mathcal{D}}]$$

In each iteration we generate a target function - which is scaled to unit variance, simulate data over a range of N samples sizes, & over a range of σ noise levels. Figure 7 displays an instance of generating the target - in this case having a large sample size and high noise level - illustrating the data generating process.

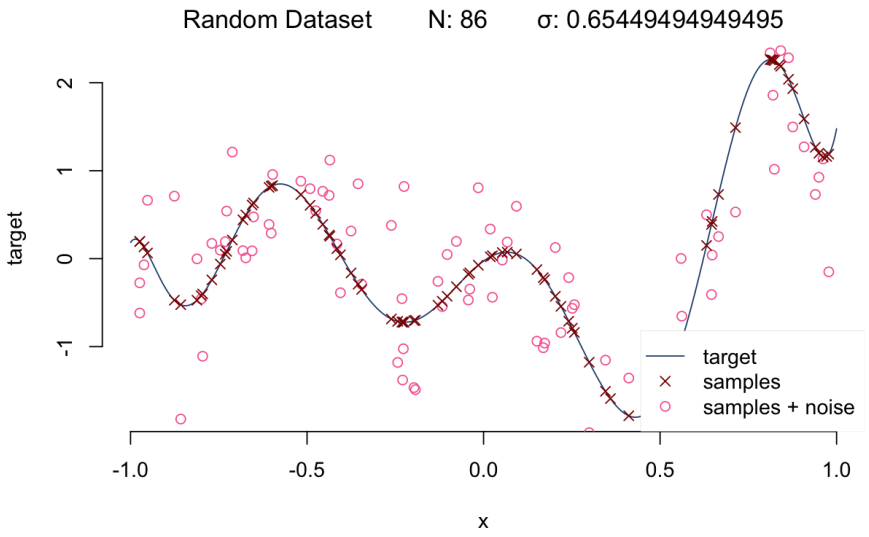

Figure 7: An illustration of the data generating process.

This process was repeated 100 times, each time the 2^{nd} & 10^{th} order Legendre polynomials are fit to the data analytically by generating the basis functions & solving for the coefficients. The 100 results are the averaged to produce an estimate for the overfit measure - over the range of the data generating process.

$E_{out}(g_{Q}^{\mathcal{D}})$ is computed by integrating over the squared difference between the true target function - without noise - & fitted model $g_{Q}^{\mathcal{D}}$ over the $[-1, 1]$ range. That is:

$$E_{out}(g_{Q}^{\mathcal{D}}) = \int_{-1}^{1} (f(x) - g_{Q}^{\mathcal{D}}(x))^2$$

The difference then details an overfit measure, as large discrepancy between $E_{out}(g_{10}^{\mathcal{D}})$ and $E_{out}(g_2^{\mathcal{D}})$ explain the degree of overfit. Large positive values indicate severe overfitting - over parameterization in the larger model has led to far inferior out of sample performance, whilst negative values indicate that the larger model has been able to more accurately represent the target function out of sample.

This actually captures bias, however the random noise component will be the same in each instance & as such will cancel out & can be ignored.

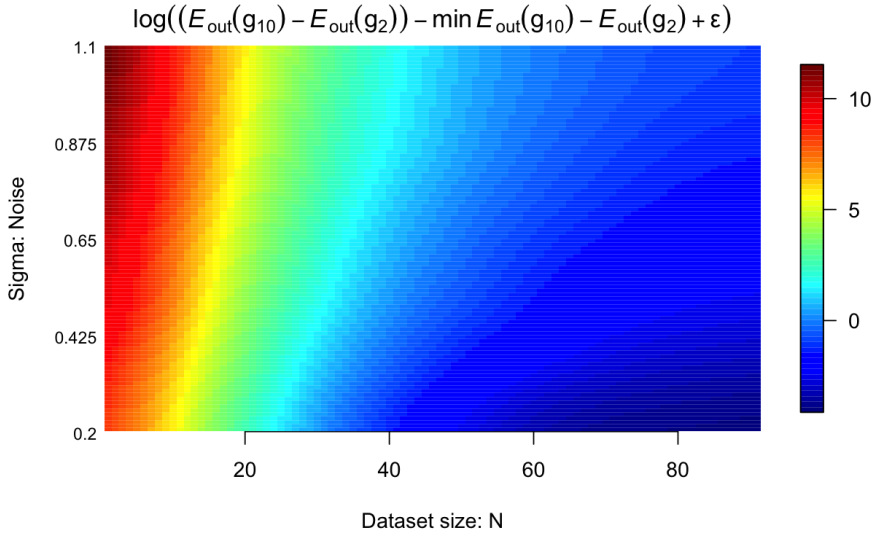

Figure 8: Approximate degree of overfit over various datasets.

Figure 8 is the result of averaging the 100 runs. As theory suggests overfitting is increasing in the noise level & decreasing in the sample size.

The raw output various greatly and is relatively noisey. As such to produce the figure the data was transformed after averaging. The minimum was added to each value - as well as a small epsilon value to avoid logging zero if the minimum were to be subtracted - and thereafter the data is logged. A number of different transformation schemes were attempted, with no theoretical logic but pure trial & error. All including centering and logging (to reduce the large discrepancies) the data. The result was, again, fairly noisy and as such I decided to use a Gaussian kernel to smooth the plot of simpler interpretation. One might expect the same smoothing result to be achieved with averaging many more runs.

Situations with very small datasets are hopeless under almost any condition, even noiseless target functions cannot be learned from very small datasets. This is expected as the data cannot represent complexity. Attempting to fit the 10^{th} order polynomial to such a small dataset perfect interpolates the data, thus is very sensitive to the exact samples given and thus - by definition - overfits to

the sample.

We also observe that the data needed to model the target increases as the data becomes more noisy. Again, intuitive as more data is required to accurately represent the target function. Surplus noise in the data results in fitting the noise, thus overfitting. This is referred to as stochastic noise, it cannot be captured by any model & fitting a model to stochastic noise will always result in overfitting.

It's also worth noting the dark blue region in figure 8: situations containing large datasets with large signal-to-noise ratios are more readily fit with higher order polynomials - keeping in mind the measure of overfit is relative.

We know the true target is given by a $Q_f = 10^{th}$ polynomial, despite this, fitting a 10^{th} order polynomial often yields inferior results. The key take away is that one should fit a model based on the resources available - asking: how much data do we have? - not purely on what is believed to be true about the data. Fitting the 'correct' order of the polynomial is inferior without sufficient data to back it up. When we have sufficient data, fitting the appropriate high order model yields optimal results - thus outperforming the simpler model on the condition that we have sufficient data. The data required increases as a function of noise, this is also intuitively sound as the high noise is to detriment of the quality of the data.

This can be succinctly summarized by figure 4.2 in the Learning From Data textbook: in-sample error is always lower in the higher order model, whilst a better out-of-sample error is only achieved with sufficient data.

2.2 Overfitting: Target Complexity

Now, the same experiment as above is conducted, however the data is generated to vary over target complexity & sample size, with a fixed level of noise. As such the measure of overfitting is now given by:

$$\mathbf{E}_{\mathcal{D},\mathcal{Q}\{}[E_{out}g_{10}^{\mathcal{D}} - E_{out}g_{2}^{\mathcal{D}}]$$

For $q \leq 15$, 100 datasets were generated and fit, whilst for $q > 15$ only 10 models were utilized. This design choice was taken as polynomials of such a high order exhibit more stationary behaviour in this experiment. They are also exceedingly computationally expensive. the results do not seem hindered by this choice.

The results of the experiment are given in figure **??**. Less severe smoothing (a Gaussian kernel with a much smaller radius) was used, & the transformation is given by the title of figure **??**. Again, the transformation was chosen by empirical examination. The colour scheme was chosen to best represent the key features.

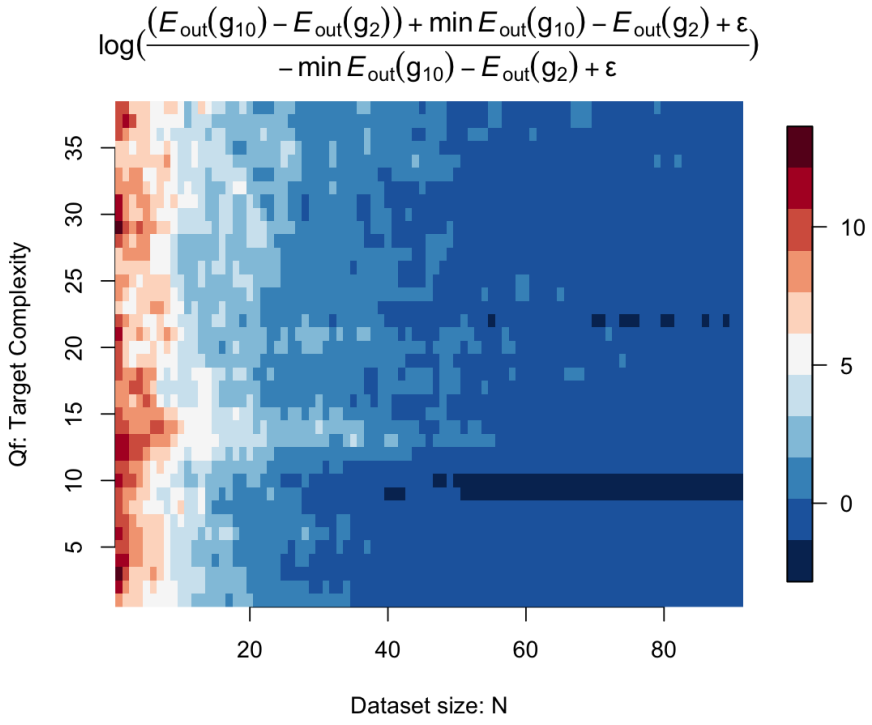

Figure 9: An illustration of deterministic noise.

Less obvious results in this instance, however a few important observations

are present. The model fits very well around the region $Q^f = 10$ - given sufficient data - given by the dark blue 'stripe' in the plot. Most importantly we note that the number of data points needed to achieve generalization is proportional to the target complexity. The more complex the target, the more data we require to fit a higher order model successfully. This phenomena truly begins after $Q^f > 10$ - thus when the target is more complex than the hypothesis set can capture.

The phenomena is described as deterministic noise - a fitting description as it is not a function of stochasticity - as apposed to regular random noise. Deterministic noise is a part of the target that cannot be captured - thus we are not able to generalize. Separate from stochastic noise as it can be captured given sufficient data.

Deterministic noise depends directly on the hypothesis set: the set needs to be sufficiently complex to allow for the complexity of the target. Deterministic noise is a synonym for the bias - it represents the deviation from the best possible hypothesis in the set from the true target. Again illustrating why it is purely dependent on the hypothesis set.

Though this premise may mislead one to believe we can simple apply an arbitrarily flexible hypothesis set that will certainly capture the essence of the target - this logic fails under the need for suitable sample. Going back to the prior argument, complex hypothesis will overfit under small (or low quality/high noise) data conditions.

One flaw in my results, is that the deterministic noise should only truly begin after $Q^f > 10$ as this is indicative of the case where there is more complexity than we can capture in our hypothesis set.

We do, however, observe in figure 9 that the level of overfitting/lack of generalization significantly increases after $Q^f \geq 10$ - visible by the degree of which the overfit protrudes over the N dimension: at $Q^f \geq 10$ far more data points are needed to generalize.

3 Closing Thoughts

It is readily apparent, that both pure stochastic noise & deterministic noise can result in overfitting. Stochastic noise describes the pure random variation in the data, whilst deterministic noise relates to the bias: a consequence of part of the target that cannot be captures by the specific hypothesis set.

In hopes of generalizing adequately, one should choose the hypothesis set as a function of the data available - size & quality of the data - and not the assumed target complexity. The quality (noise ratio) & quantity of the data determine our ability to fit in a generalized fashion. Given sufficient data, one can capture deterministic noise (bias) by increasing the flexibility of the hypothesis set. If, however, one utilizes a flexible model without sufficient data the model will simple capture noise & fail to generalize.

4 Appendix

4.1 question 1 code

```
# Question 1
```

```
# requirements ------x
require(comprehenr)

# 1.i ---------------- Legendre Polynomials Target Functions ----------------

# ----- hyper-parameters -----x
x_min=-1.0; x_max=1.0; n_points=1000
x <- seq(x_min, x_max, length.out=n_points)

# ----- legendre polynomial -----x
legendre_polynomial <- function(q, x) {
  # inner function: parallel computation
  apply_to_k <- function(k, q=q, x=x) x^k * choose(q,k) * choose((q+k-1)/2, q)
  a <- apply(matrix(0:q), MARGIN = 1, FUN = apply_to_k, q=q, x=x)
  a <- 2^q * rowSums(a)
  return(a)
}

# ----- call functions -----x
legendre <- apply(matrix(0:5), MARGIN = 1, FUN = legendre_polynomial, x=x)

# ----- visualization -----x
set.seed(849)
plot(x=x, y=legendre[,1], 'l', frame=F, main='Legendre_Polynomials', col='steelblue'
     ,
     ylim = c(min(legendre)-0.1, max(legendre)+0.1), font.main = 1, ylab='polynomial
       ')
cols <- c('steelblue')
for (i in 2:6) {
  c <- sample(colors(), 1)
  cols <- c(cols, c)
  lines(x=x, y=legendre[,i], col=c)
}
legend('bottomright', legend=c('q=0', 'q=1', 'q=2', 'q=3', 'q=4', 'q=5'),
       col=cols, lty=1, cex=1, box.lwd = 0)



# 1.ii ---------------- Random Target Functions ----------------

# ------ Alpha Target Functions ------x

# ----- hyper-parameters -----x
x_min=-1.0; x_max=1.0; n_points=1000
x <- seq(x_min, x_max, length.out=n_points)

# ----- Random Target Functions: alpha -----x
random_target_alphas <- function(Qf, n_targets, x=x, show_plot=TRUE) {
  targets_alpha <- c()
  for (t in 1:n_targets) {
    alphas <- runif(Qf, min = -1.0, max = 1.0)
    fx <- comprehenr::to_vec(for(q in 0:Qf) alphas[q+1] * x^q)
    fx <- matrix(fx, ncol = Qf+1)
    fx <- rowSums(fx, na.rm = T)
    targets_alpha <- cbind(targets_alpha, fx)
  }

  # ----- visualization -----x
  if (show_plot) {
    c <- sample(colors(), 1)
    cols <- c(c)
    plot(x=x, y=targets_alpha[,1], 'l', main=paste('Alpha_Polynomial__Qf:', Qf, sep=
         ''), frame=F, col=c, font.main=1,
         ylim = c(min(targets_alpha)-0.1, max(targets_alpha)+0.1), ylab='polynomial'
           )
    for (p in 2:n_targets) {
      c <- sample(colors(), 1)
      cols <- c(cols, c)
      lines(x=x, y=targets_alpha[,p], col=c)
    }
    legend('bottomright', legend=paste('alpha_target', 1:n_targets),
           col=cols, lty=1, cex=1, box.lwd = 0)
  }
  return(target=targets_alpha)
}
```

```r
# ----- visualize over many Qf -----x
set.seed(3688)
par(mfrow=c(2,3))
for (i in c(5,10,15,20,25,30)) {
  alphas <- random_target_alphas(Qf = i, n_targets = 3, x = x, show_plot = TRUE)
}
# ----- visualize over many Qf -----x


# ------ Generate Beta Functions ------x

# ----- Random Target Functions: beta -----x
random_target_betas <- function(Qf, n_targets, x=x, show_plot=TRUE) {

  # ----- call functions -----x
  Lqx <- apply(matrix(0:Qf), MARGIN = 1, FUN = legendre_polynomial, x=x)

  targets_betas <- c()
  targets_functions <- c()
  for (t in 1:n_targets) {
    betas <- runif(Qf+1, min = -1.0, max = 1.0)
    fx <- comprehenr::to_vec(for(q in 0:Qf) betas[q+1] * Lqx[,q+1])
    fx <- matrix(fx, ncol = Qf+1)
    fx <- rowSums(fx, na.rm = T)

    # ----- Rescale Target unit variance & adjust betas accordingly -----x
    fx <- (fx - mean(fx))/sd(fx)
    betas <- solve(t(Lqx) %*% Lqx) %*% t(Lqx) %*% fx

    targets_betas <- cbind(targets_betas, betas)
    targets_functions <- cbind(targets_functions, fx)
  }

  # ----- visualization -----x
  if (show_plot) {
    c <- sample(colors(), 1)
    cols <- c(c)
    plot(x=x, y=targets_functions[,1], 'l', main=paste('Beta_Polynomial__Qf:', Qf,
         sep=''), frame=F, col=c, font.main=1,
         ylim = c(min(targets_functions)-0.1, max(targets_functions)+0.1), ylab='
              polynomial')
    if (n_targets > 1) {
      for (p in 2:n_targets) {
        c <- sample(colors(), 1)
        cols <- c(cols, c)
        lines(x=x, y=targets_functions[,p], col=c)
      }
    }
    legend('bottomright', legend=paste('beta_target', 1:n_targets),
           col=cols, lty=1, cex=1, box.lwd = 0)
  }

  return(list(targets_functions=targets_functions, targets_betas=targets_betas))
}


# ----- visualize over many Qf -----x
set.seed(5310)
par(mfrow=c(2,3))
for (i in c(5,10,15,20,25,30)) {
  betas <- random_target_betas(Q = i, n_targets = 3, x = x, show_plot = TRUE)
}
# ----- visualize over many Qf -----x
```

4.2 question 2 code

```
# Question 2
require(comprehenr)
library(RColorBrewer)
library(fields)

# ----- Illustrating a single data generating iteration -------x

# ----- data range -------x
sigma_range <- seq(0.2, 1.0999, length.out = 100)
N_range <- 20:110

# ----- hyper-parameters -------x
x_min=-1.0; x_max=1.0; n_points=1000
x <- seq(x_min, x_max, length.out=n_points)

# ----- Random Target Functions: beta -------x
random_target_betas <- function(Qf, n_targets, x=x, show_plot=TRUE) {

  # ----- call functions -------x
  Lqx <- apply(matrix(0:Qf), MARGIN = 1, FUN = legendre_polynomial, x=x)

  targets_betas <- c()
  targets_functions <- c()
  for (t in 1:n_targets) {
    betas <- runif(Qf+1, min = -1.0, max = 1.0)
    fx <- comprehenr::to_vec(for(q in 0:Qf) betas[q+1] * Lqx[,q+1])
    fx <- matrix(fx, ncol = Qf+1)
    fx <- rowSums(fx, na.rm = T)

    # ----- Rescale Target unit variance & adjust betas accordingly -------x
    fx <- (fx - mean(fx))/sd(fx)
    betas <- solve(t(Lqx) %*% Lqx) %*% t(Lqx) %*% fx

    targets_betas <- cbind(targets_betas, betas)
    targets_functions <- cbind(targets_functions, fx)
  }

  # ----- visualization -------x
  if (show_plot) {
    c <- sample(colors(), 1)
    cols <- c(c)
    plot(x=x, y=targets_functions[,1], 'l', main=paste('Beta_Polynomial__Qf:', Qf,
         sep=''), frame=F, col=c, font.main=1,
         ylim = c(min(targets_functions)-0.1, max(targets_functions)+0.1), ylab='
             polynomial')
    if (n_targets > 1) {
      for (p in 2:n_targets) {
        c <- sample(colors(), 1)
        cols <- c(cols, c)
        lines(x=x, y=targets_functions[,p], col=c)
      }
    }
    legend('bottomright', legend=paste('beta_target', 1:n_targets),
           col=cols, lty=1, cex=1, box.lwd = 0)
  }

  return(list(targets_functions=targets_functions, targets_betas=targets_betas))
}

set.seed(3081842)
target_func <- random_target_betas(Q = 10, n_targets = 1, x = x, show_plot = FALSE)
target <- target_func$targets_functions

# ----- compute datasets -------x
len_target <- length(target)
datasets <- c()
for (s in sigma_range) {
  # ----- for each sigma -------x
  for (n in N_range) {
    # ----- sample point (index) -------x
    ind <- sample(x = 1:len_target, size = n, replace = TRUE)
    sample_y <- target[ind] + s*rnorm(n, mean = 0, sd = 1)
    sample_x <- x[ind]; true_val <- target[ind]
    samp <- cbind(sample_x, sample_y, true_val, s)
    datasets <- c(datasets, list(samp))
  }
}

# ----- visualize datasets -------x
visualize_dataset <- function(dataset) {
  plot(x, target, 'l', frame.plot =F, col='#264f7d',
       main = paste('Random_Dataset________N:', nrow(dataset), '______ :', dataset
           [1,4]), font.main=1)
  points(dataset, col='#f768a1')
  points(dataset[,1], dataset[,3], col='darkred', pch=4)
  legend('bottomright', legend=c('target', 'samples', 'samples_+_noise'),
```

```
        lty=c(1,NA,NA), pch=c(NA,4,1), cex=1, col=c('#264f7d', 'darkred', '#f768a1'
            ), box.lwd = 0.05)
}
visualize_dataset(datasets[[sample(1:length(datasets), 1)]])




# ------ Question 2.i fitting noise ------x
require(comprehenr)
library(RColorBrewer)
library(fields)

# ----- hyper-parameters -----x
x_min=-1.0; x_max=1.0; n_points=1000
x <- seq(x_min, x_max, length.out=n_points)


# ------------------------- Target & Hypothesis Functions -------------------------x
# ----- g2x -----x
g2x <- function(xs) {
  Lqx <- apply(matrix(0:(length(betas_2)-1)), MARGIN = 1, FUN = legendre_polynomial,
      x=xs)
  return(Lqx %*% betas_2)
}
# ----- g10x -----x
g10x <- function(xs) {
  Lqx <- apply(matrix(0:(length(betas_10)-1)), MARGIN = 1, FUN = legendre_polynomial
      , x=xs)
  return(Lqx %*% betas_10)
}
# ----- target function -----x
f_x_target <- function(xs) {
  Lqx <- apply(matrix(0:(length(target_betas)-1)), MARGIN = 1, FUN = legendre_
      polynomial, x=xs)
  return(Lqx %*% target_betas)
}
# ------------------------- Target & Hypothesis Functions -------------------------x


# ------------------------- Compute out of Sample Error Functions
    -------------------------x
# ----- E_out 10 -----x
E_out_10 <- function(xs) {
  E_out <- 1 * (f_x_target(xs) - g10x(xs))^2
  return(E_out)
}
# ----- E_out 2 -----x
E_out_2 <- function(xs) {
  E_out <- 1 * (f_x_target(xs) - g2x(xs))^2
  return(E_out)
}
# ------------------------- Compute out of Sample Error Functions
    -------------------------x


# ------------------------- For each Dataset -------------------------x
all_results <- c()
for (runs in 1:100) {
  print(paste('----------run: ', runs, '----------'))
  # ---------- Generate Target ----------x
  target_func <- random_target_betas(Q = 10, n_targets = 1, x = x, show_plot = FALSE
      )
  target <<- target_func$targets_functions
  target_betas <<- target_func$targets_betas

  # ------ generate samples -----x
  inner_sample <- function(s, n) {
    ind <- sample(x = 1:len_target, size = n, replace = TRUE)
    # sample_y <- target[ind] + rnorm(n, mean = 0, sd = s)
    sample_y <- target[ind] + s*rnorm(n)
    sample_x <- x[ind]; true_val <- target[ind]
    samp <- cbind(sample_x, sample_y, true_val, s)
    return(samp)
  }

  outer_sample <- function(ss) {
    apply(matrix(N_range), 1, inner_sample, s=ss)
  }
  datasets <- unlist(apply(matrix(sigma_range), 1, outer_sample), recursive = F)

  results <- c()
  for (d in datasets) {
    # ----- hyperparameters -----x
    nn <- nrow(d)
    sigma <- d[1,4]
```

```
    x_basis <- d[,1]
    y <- d[,2]

    # ----- basis functions -----x
    z_2 <- apply(matrix(0:2), MARGIN = 1, FUN = legendre_polynomial, x=x_basis)
    z_10 <- apply(matrix(0:10), MARGIN = 1, FUN = legendre_polynomial, x=x_basis)

    # ----- model coefficients -----x
    betas_2 <<- solve(t(z_2) %*% z_2) %*% t(z_2) %*% y
    betas_10 <<-  solve(t(z_10) %*% z_10) %*% t(z_10) %*% y

    # ----- model fit -----x
    # yhat_2 <- z_2 %*% betas_2
    # yhat_10 <- z_10 %*% betas_10

    # ----- Compute E_in -----x
    # E_in_2 <- sum((y - yhat_2)^2)
    # E_in_10 <- sum((y - yhat_10)^2)


    # ----- Compute E_out -----x
    # ----- Compute Integral of Difference -----x

    # bias
    Eout10 <- integrate(E_out_10, lower = -1, upper = 1)
    Eout2 <- integrate(E_out_2, lower = -1, upper = 1)


    # ----- compute generalization error: E_out10 - E_out2 -----x
    gen_error <- Eout10$value - Eout2$value


    # ----- Expected Value [E_out10 - E_out2] over D -----x

    # ----- store results -----x
    results <- rbind(results, cbind(gen_error=gen_error, n=nn, sigma=sigma))


  }
  # ------------------------- For each Dataset -------------------------x


  # store
  all_results <- rbind(all_results, results)

}


# ---------- compute averages ----------x
final_averaged_estimates <- c()
for (s in sigma_range) {
  for (n in N_range) {

    sub <- (all_results[all_results[,2] == n & all_results[,3] == s,])
    final_averaged_estimates <- rbind(final_averaged_estimates, cbind(mean(sub[,1]),
          n, s))

  }
}


saveRDS(final_averaged_estimates, file = "final_averaged_estimates.rds")
final_averaged_estimates <- readRDS(file = "final_averaged_estimates.rds")



# tranfermation ----------x
z <-  matrix(final_averaged_estimates[,1], nrow=100, ncol = 91, byrow = T)
z <- t(z)
zz <- log(z - min(z) + 1e-4)
zz <- (image.smooth(zz, theta = 10))

# visualization -----x
image.plot(zz, main=bquote(log((italic(E)["out"](g['10']) - italic(E)["out"](g['2']))
      ) -
                                  min(italic(E)["out"](g['10']) - italic(E)["out"](g[
                                      '2']))+    )),
           font.main=1, xlab = 'Dataset_size:_N', ylab = 'Sigma:_Noise', frame=F,
                 yaxt = "n")
mtext(text=seq(0.2, 1.1, length.out = 5), side=2, line=0.3, at=seq(0,100,25), las=1,
      cex=0.8)



# ------- Question 2.ii over Qf --------x
```

```
require(comprehenr)
library(RColorBrewer)
library(fields)
library(RColorBrewer)
library(fields)


# ----- hyper-parameters -----x
x_min=-1.0; x_max=1.0; n_points=1000
x <- seq(x_min, x_max, length.out=n_points)
Qf_range <- 1:40
sig <- 0.2


# ------------------------- For each Dataset -------------------------x
all_results_Qf <- c()

#for (q in Qf_range) {
for (q in 1:38) {
  print(paste('----------running_for_Qf:', q, '----------'))

  # ---------- Generate Target ----------x
  target_func <- random_target_betas(Q = q, n_targets = 1, x = x, show_plot = FALSE)
  target <<- target_func$targets_functions
  target_betas <<- target_func$targets_betas

  # ------ generate samples: fast method -----x
  inner_sample <- function(s, n) {
    ind <- sample(x = 1:len_target, size = n, replace = TRUE)
    sample_y <- target[ind] + sig*rnorm(n)
    sample_x <- x[ind]; true_val <- target[ind]
    samp <- cbind(sample_x, sample_y, true_val, q)
    return(samp)
  }

  outer_sample <- function(ss) {
    # just use ss as a empty index to use apply
    apply(matrix(N_range), 1, inner_sample, s=ss)
  }

  if (q <= 15) datasets <<- unlist(apply(matrix(1:100), 1, outer_sample), recursive
      = F)
  if (q > 15) datasets <<- unlist(apply(matrix(1:10), 1, outer_sample), recursive =
       F)

  results <- c()
  for (d in datasets) {
    # ----- hyperparameters -----x
    nn <- nrow(d)
    x_basis <- d[,1]
    y <- d[,2]
    # ----- basis functions -----x
    z_2 <- apply(matrix(0:2), MARGIN = 1, FUN = legendre_polynomial, x=x_basis)
    z_10 <- apply(matrix(0:10), MARGIN = 1, FUN = legendre_polynomial, x=x_basis)

    # ----- model coefficients -----x
    betas_2 <<- solve(t(z_2) %*% z_2) %*% t(z_2) %*% y
    betas_10 <<-  solve(t(z_10) %*% z_10) %*% t(z_10) %*% y

    # ----- Compute E_out -----x
    # ----- Compute Integral of Difference -----x

    # bias
    Eout10 <- integrate(E_out_10, lower = -1, upper = 1)
    Eout2 <- integrate(E_out_2, lower = -1, upper = 1)
    # ----- compute generalization error: E_out10 - E_out2 -----x
    gen_error <- Eout10$value - Eout2$value


    # ----- Expected Value [E_out10 - E_out2] over D -----x
    # ----- store results -----x
    results <- rbind(results, cbind(gen_error=gen_error, n=nn, Qf=q))
  }
  # ------------------------- For each Dataset -------------------------x
  # store
  all_results_Qf <- rbind(all_results_Qf, results)
}


# ---------- compute averages ----------x
final_averaged_estimates_Qf_total <- c()
# for (q in Qf_range) {
for (q in 1:38) {
  for (n in N_range) {
    sub <- (all_results_Qf[all_results_Qf[,2] == n & all_results_Qf[,3] == q,])
    final_averaged_estimates_Qf_total <- rbind(final_averaged_estimates_Qf_total,
        cbind(mean(sub[,1]), n, q))
  }
}
```

```
saveRDS(final_averaged_estimates_Qf_total, file = "final_averaged_estimates_Qf_total
    .rds")
final_averaged_estimates_Qf_total <- readRDS(file = "final_averaged_estimates_Qf_
    total.rds")


# trans 1 ----------------x
z <-  matrix(final_averaged_estimates_Qf_total[,1], ncol=38, nrow = 91, byrow = F)
zz <- log((z - min(z) + 4e-3)/(-min(z) + 4e-3))


# zz <- log((z - min(z) + 4e-3))
zz <- (image.smooth(zz, theta = 1))


image.plot(zz, main=
             bquote(log (frac((italic(E)["out"](g['10']) - italic(E)["out"](g['2']))
                    +
                                 min(italic(E)["out"](g['10']) - italic(E)["out"](g['
                                     2']))+    , -
                                 min(italic(E)["out"](g['10']) - italic(E)["out"](g['
                                     2']))+    ))),
           font.main=1, xlab = 'Dataset_size:_N',
           ylab = 'Qf:_Target_Complexity', frame=F, col=rev(brewer.pal(n = 20, name
               = 'RdBu')))
```

```
```