The Bernstein Mechansim

diffpriv team

November 3, 2026

Abstract

This vignette presents a short tutorial on the application of the generic Bernstein mechanism `DPMechBernstein` for differentially-private release of functions in the diffpriv R package[1].

1 Introduction

The primary use case of the Bernstein mechanism is releasing real-valued functions on $[0,1]^d$. The typical function released will depend on an arbitrary sensitive dataset (which could be numeric or otherwise), and after release the function may be evaluated on unlimited points.

If you make use of the mechanism in your work, please consider citing the original paper `https://arxiv.org/abs/1507.04499` in subsequent writeups:

> Francesco Aldà and Benjamin I. P. Rubinstein. "The Bernstein Mechanism: Function Release under Differential Privacy", in Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'2017), pp. 1705–1711, 2017.

2 Bernstein Polynomial Approximation

Like the more common Taylor polynomial approximation, Bernstein approximations of a target function $f : [0,1]^d \to \mathbb{R}$ involve a weighted sum of basis polynomials. We'll refer to these weights as coefficients, and introduce the Bernstein approximation for the simple one-dimensional $d = 1$ case. For details on the multidimensional case (implemented in diffpriv see the reference paper above).

The $k+1$ Bernstein basis polynomials of degree k are defined as $b_{\nu,k}(x) = \binom{k}{\nu} x^\nu (1-x)^{k-\nu}$ for ν ranging over $0, \ldots, k$. Fixing ν and for varying $x \in [0,1]$, the basis function corresponds to the probability that k coin tosses results in ν heads, where the chance of a head is x. Taken as a whole, the set of basis functions therefore makes up the entire probability mass for the $Binomial(k, x)$ distribution.

The coefficients of the Bernstein approximation of target f are simply the evaluations of f on the $(k+1)$-point regular grid covering $[0,1]$: at points $\{0, 1/k, \ldots, k/k\}$.

Together, then, $f(x)$ is approximated as $\tilde{f}(x) = \sum_{\nu=0}^{k} f(\nu/k) b_{\nu,k}(x)$ which can be interpreted as the expectation of $f(X/k)$ for $X \sim Binomial(k, x)$. We note in passing that nice guarantees exist about the closeness of $\tilde{f}$ to f, with natural conditions on smoothness of f.

[1] The diffpriv can be found at `https://github.com/brubinstein/diffpriv`

2.1 Example

To see Bernstein approximation in action in diffpriv, consider approximating the function $f(x) = x\sin(10x)$ on $x \in [0, 1]$ with a Bernstein polynomial of degree $k = 25$.

```
library(diffpriv)
targetF <- function(x) x * sin(10 * x)
bernsteinF <- bernstein(targetF, dims = 1, k = 25)
```

The returned value is an S3 object of class `bernstein`, a list with various slots including one that holds the $k + 1$ coefficients of the approximation.

```
bernsteinF$coeffs

#>  [1]  0.00000000  0.01557673  0.05738849  0.11184469
#>  [5]  0.15993178  0.18185949  0.16211116  0.09379668
#>  [9] -0.01867973 -0.15930736 -0.30272100 -0.41870491
#> [13] -0.47815901 -0.45939642 -0.35350932 -0.16764930
#> [17]  0.07459149  0.33599708  0.57144086  0.73561895
#> [21]  0.79148660  0.71786308  0.51472713  0.20505872
#> [25] -0.16735371 -0.54402111
```

Predictions $\tilde{f}(x)$ can be made for objects of type `bernstein` using the `predict.bernstein()` function implementing the S3 generic `predict()`.

```
predict(bernsteinF, D = 0.2)    # approximate f(0.5)

#> [1] 0.1101786

targetF(0.2)                    # actual f(0.5)

#> [1] 0.1818595
```

Evaluation on a collection of points is also easy.

```
xs <- seq(from = 0, to = 1, length = 50)
plot(xs, targetF(xs), xlim = c(0,1), ylim = c(-1,1), lty = "dashed", lwd = 2,
     col = "red", type="l", xlab="x", ylab="y",
     main="Bernstein polynomial approximation")
lines(xs, predict(bernsteinF, xs), col = "blue", lwd = 2)
```

3 Differential Privacy with the Bernstein Mechanism

The S4 class `DPMechBernstein` subclasses the virtual `DPMech` within the diffpriv package, implementing the generic Bernstein mechanism. The mechanism

- First instantiates the target function, itself a function of sensitive input data (such as a classifier or statistical model).

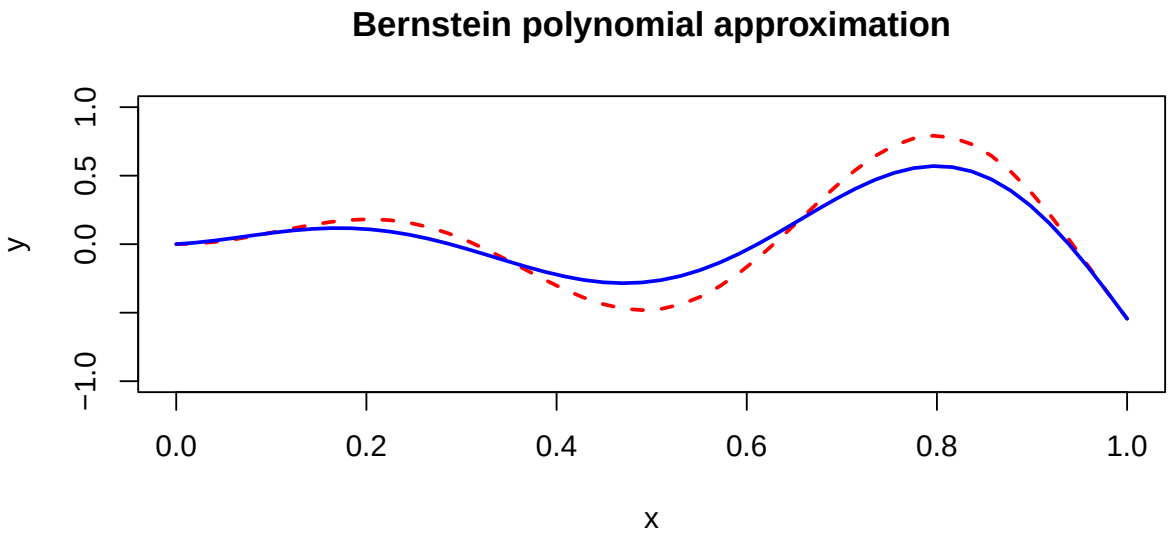


Figure 1: Bernstein polynomial approximation (blue) vs target (red).

- It then forms a Bernstein polynomial approximation as described above.
- The Laplace mechanism [DMNS06] `DPMechLaplace` is used to perturb the Bernstein approximation coefficients. As these are the only component of the approximation that depends on the target function (and hence input dataset; the basis polynomials are target/data-independent), this is sufficient for preserving differential privacy [DMNS06].
- Subsequent evaluations of the perturbed approximation function are simply sums of the basis polynomials, weighted by these perturbed coefficients.

A sufficient level of Laplace noise depends on the global sensitivity of the target function, required as an argument to `DPMechBernstein` construction unless the sensitivity sampler is used (demonstrated in the example below). Intuitively, targets that are more volatile—vary more with perturbed input data—require more smoothing by Laplace noise.

3.1 Example

Suppose we want to fit a sensitive dataset `D` with Priestly-Chao kernel regression, using the Gaussian kernel with a `bandwidth` hyperparameter specifying kernel smoothness. For simplicity, we'll consider a single co-variate. A fitting function for the estimator is given as follows. It takes codeD a 2-column matrix with examples in rows, and returns a function for making predictions on new data.

```
pck_regression <- function(D, bandwidth = 0.1) {
  K <- function(x) exp(-x^2/2)
  ids <- sort(D[,1], decreasing = FALSE, index.return = TRUE)$ix
```

```
  D <- D[ids, ]
  n <- nrow(D)
  ws <- (D[2:n,1] - D[1:(n-1),1]) * D[2:n,2]
  predictor <- function(x) {
    sum(ws * sapply((x - D[2:n,1]) / bandwidth, K)) / bandwidth
  }
  return(predictor)
}
```

We have the following (synthetic) sensitive dataset, as a 250×2 matrix with the first column representing co-variates/features and the second column representing dependent variables/labels.

```
N <- 250
D <- runif(N)
D <- cbind(D, sin(D*10)*D + rnorm(N, mean=0, sd=0.2))
```

Let's fit three models for comparison:

- A non-private exact Priestly-Chao regression given by `model`;
- A non-private Bernstein approximation of the exact regression `bmodel`; and
- A privatized regression produced by `DPMechBernstein`, `pmodel`.

```
## Non private fitting
model <- pck_regression(D)

## Bernstein non private fitting
K <- 25
bmodel <- bernstein(model, dims=1, k=K)

## Private Bernstein fitting
m <- DPMechBernstein(target=pck_regression, latticeK=K, dims=1)
P <- function(n) {  # a sampler of random, "plausible", datasets
  Dx <- runif(n)
  Dy <- rep(0, n)
  if (runif(1) < 0.95) Dy <- Dy + Dx
  if (runif(1) < 0.5) Dy <- Dy * sin(Dx)
  if (runif(1) < 0.5) Dy <- Dy * cos(Dx)
  cbind(Dx, Dy + rnorm(n, mean=0, sd=0.2))
}
m <- sensitivitySampler(m, oracle=P, n=N, gamma=0.20, m=500)
R <- releaseResponse(m, privacyParams=DPParamsEps(epsilon=5), X=D)
pmodel <- R$response
```

The private model is produced as described above. `sensitivitySampler()` probes the non-private model with 500 random pairs of datasets, sampled from `P()`, to estimate the target's sensitivity. The resulting perturbed private model preserves random differential privacy with level $\epsilon = 5$ and confidence $\gamma = 0.2$. In practice we could easily take γ much smaller (much higher confidence) by increasing sensitivity sample size `m`.

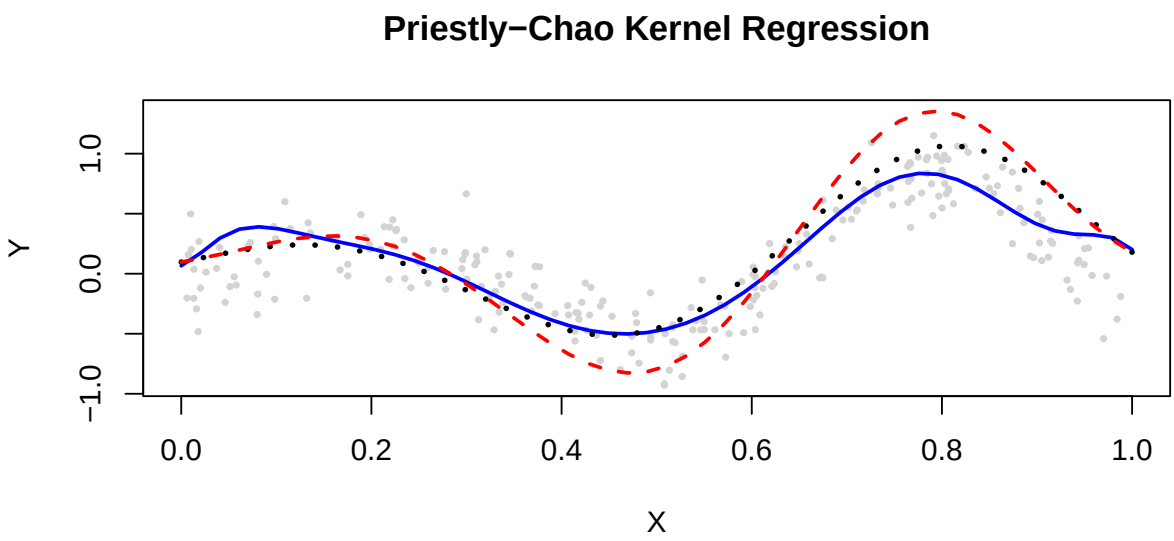


Figure 2: Kernel regression on 1D training data (gray points): non-private model (red dashed); non-private Bernstein polynomial approximation (black dotted); private Bernstein mechanism (blue solid).

Let's now take our three fitted models, and predict the dependent variable/label across a range of covariates/features.

```
xs <- seq(from=0, to=1, length=50)
yhats   <- sapply(xs, model)
yhats.b <- predict(bmodel, xs)
yhats.p <- R$response(xs)
```

We can now finally visually compare the three fitted models, alongside the original training dataset.

```
xlim <- c(0, 1)
ylim <- range(c(yhats.b, yhats.p, yhats, D[,2]))
plot(D, pch=20, cex=0.6, xlim=c(0,1), ylim=ylim, xlab="X", ylab="Y",
    main="Priestly-Chao Kernel Regression", col="lightgrey")
lines(xs, yhats.p, col="blue",  type="l", lty="solid", lwd = 2)
lines(xs, yhats.b, col="black", type="l", lty="dotted", lwd = 3)
lines(xs, yhats,   col="red",   type="l", lty="dashed", lwd =2)
```

We could safely release the model `pmodel` but not the other non-private models. Also note that while a target's sensitivity can be computed/bounded manually in many cases, when the target is more complex sensitivity analysis can be prohibative. The sensitivity sampler offers a pragmatic approach to such situations, replacing exact bounds with random probing and estimation. For the resulting random differential privacy to make sense, the sampling distribution (passed as argument 'oracle' to the sensitivity sampler) should reflect public knowledge about the dataset. This could be

noninformative (like a uniform/normal distribution), it could be a public Bayesian prior, it could even be the result of density estimation on a real dataset (potentially privately estimated).

If using the sensitivity sampler, we suggest citing the original paper:

> Benjamin I. P. Rubinstein and Francesco Aldà. "Pain-Free Random Differential Privacy with Sensitivity Sampling", to appear in the 34th International Conference on Machine Learning (ICML'2017), 2017.

Further details on the sampler can be found there.

References

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.