Problem Set 3

Clinton Tepper

Problem 1

a

- Derive the likelihood
  - Suppose the data point Y_i^* is greater than zero. Then the likelihood for a given point is $\frac{1}{\sigma}\phi\left(\frac{Y_i - X_i'\beta}{\sigma}\right)$, as with a normal distribution.
  - Suppose the data point Y_i^* is less than zero. Then the likelihood is given by the entire region below the cutoff, as dictated by $\Phi\left(\frac{\tau - X_i'\beta}{\sigma}\right)$ where $\tau = 0$ (the cutoff).
    - Intuitively, the $X_i'\beta$ term is the mean of the normal distribution
  - Write down the total likelihood:

$$L\left(\hat{\theta}|X\right) = \prod_{i\in 1:n}\left[\frac{1}{\sigma}\phi\left(\frac{Y_i - X_i'\beta}{\sigma}\right)\right]^{\iota(Y_i>0)}\left[\Phi\left(\frac{-X_i'\beta}{\sigma}\right)\right]^{\iota(Y_i=0)}$$

$$l\left(\hat{\theta}|X\right) = \sum_{i\in 1:n}\ln\left[\frac{1}{\sigma}\phi\left(\frac{Y_i - X_i'\beta}{\sigma}\right)\right]\iota\left(Y_i > 0\right) + \sum_{i\in 1:n}\ln\left[1 - \Phi\left(\frac{X_i'\beta}{\sigma}\right)\right]\iota\left(Y_i = 0\right)$$

b

- We have:

$$\begin{aligned}
E\left(Y_i|X_i = x\right) &= \int_{-\infty}^{\infty} Y_i\left(\frac{1}{\sigma}\phi\left(\frac{Y_i^* - X'\beta}{\sigma}\right)\right)dY_i^* \\
&= \int_{-\infty}^{0}\frac{Y_i}{\sigma}\phi\left(\frac{Y_i^* - X'\beta}{\sigma}\right)dY_i^* + \int_{0^+}^{\infty}\left(\frac{Y_i}{\sigma}\phi\left(\frac{Y_i^* - X'\beta}{\sigma}\right)\right)dY_i^* \\
&= \int_{0^+}^{\infty}\left(\frac{Y_i}{\sigma}\phi\left(\frac{Y_i^* - X'\beta}{\sigma}\right)\right)dY_i^* \\
&= \int_{0^+}^{\infty}\left(\frac{Y_i^*}{\sigma}\phi\left(\frac{Y_i^* - X'\beta}{\sigma}\right)\right)dY_i^*
\end{aligned}$$

  - Let $\gamma = \frac{Y_i^* - X'\beta}{\sigma} \implies d\gamma = \frac{1}{\sigma}dY_i^*$. Also note that $Y_i^* = \sigma\gamma + X'\beta$, and define $\gamma_\tau = \frac{-X'\beta}{\sigma}$

$$\begin{aligned}
&= \int_{\gamma_\tau}^{\infty}\left(\left[\sigma\gamma + X'\beta\right]\phi\left(\gamma\right)\right)d\gamma \\
&= \int_{\gamma_\tau}^{\infty}\left(\sigma\gamma\phi\left(\gamma\right)\right)d\gamma + X'\beta\Phi\left(\gamma\right)|_{\gamma_\tau}^{\infty} \\
&= \int_{\gamma_\tau}^{\infty}\left(\sigma\gamma\phi\left(\gamma\right)\right)d\gamma + X'\beta\left(1 - \Phi\left(\frac{-X'\beta}{\sigma}\right)\right) \\
&= \int_{\gamma_\tau}^{\infty}\left(\sigma\gamma\frac{1}{\sqrt{2\pi}}\exp\left(\frac{-\gamma^2}{2}\right)\right)d\gamma + X'\beta\Phi\left(\frac{X'\beta}{\sigma}\right)
\end{aligned}$$

– Let $\xi = -\frac{\gamma^2}{2} \implies d\xi = -\gamma d\gamma$. Then:

$$
\begin{aligned}
&= X'\beta\Phi\left(\frac{X'\beta}{\sigma}\right) - \int_{\gamma=\gamma_\tau}^{\infty}\left(\sigma\frac{1}{\sqrt{2\pi}}\exp\left(\xi\right)\right)d\xi \\
&= X'\beta\Phi\left(\frac{X'\beta}{\sigma}\right) - \sigma\phi\left(\gamma\right)|_{\gamma_\tau}^{\infty} \\
&= X'\beta\Phi\left(\frac{X'\beta}{\sigma}\right) + \sigma\phi\left(\frac{X'\beta}{\sigma}\right)
\end{aligned}
$$

– And we are done (didn't need the hint....)

c

```
require(ggplot2) #for graphs
```

```
## Loading required package: ggplot2
```

```
require(parallel) #good for bootstrapping
```

```
## Loading required package: parallel
```

```
require(data.table) #this and the below package are needed to work with data
```

```
## Loading required package: data.table
## data.table 1.10.4.3
##   The fastest way to learn (by data.table authors): https://www.datacamp.com/courses/data-analysis-t
##   Documentation: ?data.table, example(data.table) and browseVignettes("data.table")
##   Release notes, videos and slides: http://r-datatable.com
```

```
require(xtable)
```

```
## Loading required package: xtable
```

```
set.seed(10) #A seed for me



#holds constants and program parameters
CONST = list(
    EPSILON = .Machine$double.eps, #machine precision
    MIN_DOUBLE = .Machine$double.xmin,
    NUM_WORKERS = max(round(detectCores() * .5), 2), #just a heuristic for multi-threading
    X_NAMES = c("l.physint", "l.polity2", "l.lneconaid", "l.lnrgdp", "l.lnpop", "colony"),
    Y_NAME = "lneconaid",
    CI95 = 1.96,
    FILE_NAME = "nielsenaid.csv"
)

#This is the tobit loglikelihood function
llikelihoodTobit = function(bs, Y, X) {
    epsilon = CONST$EPSILON
    k = length(bs)
    b = bs[1:(k - 1)]
```

```
    s = max(abs(bs[k]), epsilon)
    #s = b[k]

    likes = ifelse(Y > 0,
        dnorm((Y - (X %*% b)) / s) / s, 1 - pnorm((X %*% b) / s))
    likes = log(likes)
    likes = ifelse(is.finite(likes), likes, -10 ^ 10)

    return(sum(likes))
}



tobitModel = function(Y, X, yname = NULL, xnames = NULL, suppressintercept = FALSE) {

    pnames = xnames

    #make the intercept as needed
    if (!suppressintercept) {
        if (min(X[, ncol(X)]) != 1 || max(X[, ncol(X)]) != 1) {
            X = cbind(X, rep(1, nrow(X))) #bind the intercept column
            if (!is.null(pnames)) {
                pnames = c(pnames, "intercept") #adjust the names
            }
        }
    }
    pnames = c(pnames, "sigma")

    # Get some convenience constants
    R = nrow(Y)
    C = ncol(X)

    #initial value of b
    bs = rep(.001, C + 1)

    #make single argument versions for optim
    ll = function(x) - 1.0 * llikelihoodTobit(x, Y, X)

    #call the optimizer
    opt = optim(bs, fn = ll,
        method = "BFGS", hessian = TRUE)
    if (opt$convergence != 0) print("WARNING! Optimizer did not converge")

    print(opt$par)
    #Efficient matrix inversion
    U = chol(opt$hessian)
    UInv = solve(chol(opt$hessian))
    Sigma = t(UInv) %*% UInv

    #form the info we want into a named list
    tob = list(B = opt$par, llikelihood = opt$value, varB = diag(Sigma),
        seB = diag(Sigma) ^ 0.5, k = length(opt$par))
```

```
    if (!is.null(pnames)) {
        names(tob$B) = pnames #name the coefficients
        names(tob$seB) = pnames
    }

    return(tob)
}

prepsample = function() {
    d = fread(file = CONST$FILE_NAME)

    X = as.matrix(d[, CONST$X_NAMES, with = FALSE])
    Y = as.matrix(d[, CONST$Y_NAME, with = FALSE])

    S = list(Y = Y, X = X, yname = CONST$Y_NAME, xnames = CONST$X_NAMES)

    return(S)
}

#tests the model a single time and prints the results
runTobitModelOnce = function() {

    S = prepsample()
    tob = tobitModel(Y = S$Y, X = S$X, xnames = S$xnames, yname = S$yname)
    #print(tob)

    #set the test vector as the median of the X values
    testvals = as.matrix(sapply(1:ncol(S$X), function(x) median(S$X[, x])))
    testvals[length(testvals)] = 0 #mode for colony
    testvals[length(testvals) + 1] = 1

    predictTobit = function(xb, s) pnorm(xb / s) * xb + dnorm(xb / s) * s

    b = as.matrix(tob$B[1:(tob$k - 1)])
    s = tob$B["sigma"]

    print(b)
    print(s)
    print(testvals)
    print(t(b) %*% testvals)

    results = list(
    predicted = predictTobit(t(b) %*% testvals, s),
    CI = c(predictTobit(t(b) %*% testvals - s * CONST$CI95, s),
        predictTobit(t(b) %*% testvals + s * CONST$CI95, s)))
    print(results)

    outdf = data.table(coef = c(S$xnames, "intercept", "sigma"), value = tob$B, SE = tob$seB)
    xtable(outdf, type="html")
}

runTobitModelOnce()
```

[1] 0.07139795 0.01394888 1.21493561 -0.52425905 0.11768203 -0.12036171 [7] 0.62131182 2.95278682 [,1] l.physint 0.07139795 l.polity2 0.01394888 l.lneconaid 1.21493561 l.lnrgdp -0.52425905 l.lnpop 0.11768203 colony -0.12036171 intercept 0.62131182 sigma 2.952787 [1] 3.000000 6.000000 0.000000 8.450480 9.203427 0.000000 1.000000 [,1][1,] -2.427964 $predicted [,1][1,] 0.3412308

$CI [1] 0.002384895 3.547451068

% latex table generated in R 3.4.3 by xtable 1.8-2 package % Mon Mar 04 19:23:09 2019

	coef	value	SE
1	l.physint	0.07	0.02
2	l.polity2	0.01	0.01
3	l.lneconaid	1.21	0.02
4	l.lnrgdp	-0.52	0.03
5	l.lnpop	0.12	0.07
6	colony	-0.12	0.37
7	intercept	0.62	0.75
8	sigma	2.95	0.08

Problem 2

a

- The loss function is defined as

$$L(Y_i,\, \hat{f}(X_i)) = \left(Y_i - \hat{f}(X_i)\right)^2$$

  - Taking expectations:

$$\begin{aligned} E\left[L(Y_i,\, \hat{f}(X_i))|X_i\right] =& E\left[\left(Y_i - \hat{f}(X_i)\right)^2 |X_i\right] \\ =& E\left[Y_i|X_i\right]^2 - 2\hat{f}\left(X_i\right) E\left[Y_i|X_i\right] + \hat{f}(X_i)^2 \\ =& \left(E\left[Y_i|X_i\right] - \hat{f}(X_i)\right)^2 \end{aligned}$$

  - Since $E\left[L(Y_i,\, \hat{f}(X_i))|X_i\right] \in \Re_+$ and $\left(\hat{f}(X_i) = E\left[Y_i|X_i\right]\right) \iff \left(E\left[L(Y_i,\, \hat{f}(X_i))|X_i\right] = 0\right)$, loss is minimized when $E\left[Y_i|X_i\right] = \hat{f}(X_i)$.

b

- Work through the algebra:

$$
\begin{aligned}
R =& E\left[\left(Y_i - \hat{f}(X_i)\right)^2\right] \\
R =& E\left[\left(f(X_i) + \varepsilon_i - \hat{f}(X_i)\right)^2\right] \\
=& E\left[\left(f(X_i) - \hat{f}(X_i)\right)^2\right] + \sigma^2 \\
=& E\left[f(X_i)^2\right] - 2E\left[\hat{f}(X_i) f(X_i)\right] + E\left[\hat{f}(X_i)^2\right] + \sigma^2 \\
=& E\left[f(X_i)^2\right] - 2E\left[\hat{f}(X_i) f(X_i)\right] + E\left[\hat{f}(X_i)\right]^2 + \left(E\left[\hat{f}(X_i)^2\right] - E\left[\hat{f}(X_i)\right]^2\right) + \sigma^2 \\
=& \left(E\left[f(X_i)\right] - E\left[\hat{f}(X_i)\right]\right)^2 + \left(E\left[\hat{f}(X_i)^2\right] - E\left[\hat{f}(X_i)\right]^2\right) + \sigma^2 \\
=& (\text{bias})^2 + \text{variance} + \sigma^2
\end{aligned}
$$

- The variance here reflects the incremental noise of the approximated outcomes given new draws of X_i. A high variance implies a lack of a precise prediction.
- The bias represents the degree to which the approximation estimates the true data generating process in expectation. High bias implies poor prediction accuracy, even if the prediction is extremely precise.

c

- Learning algorithms with lower bias are generally more flexible. For example, if $\hat{f}(X_i)$ is just the nearest known neighbor, bias is zero in expectation, but the variance could be quite high. In contrast, reducing flexibility will lower variance and increase the similarity between the in- and out-of-sample distributions.

Problem 3

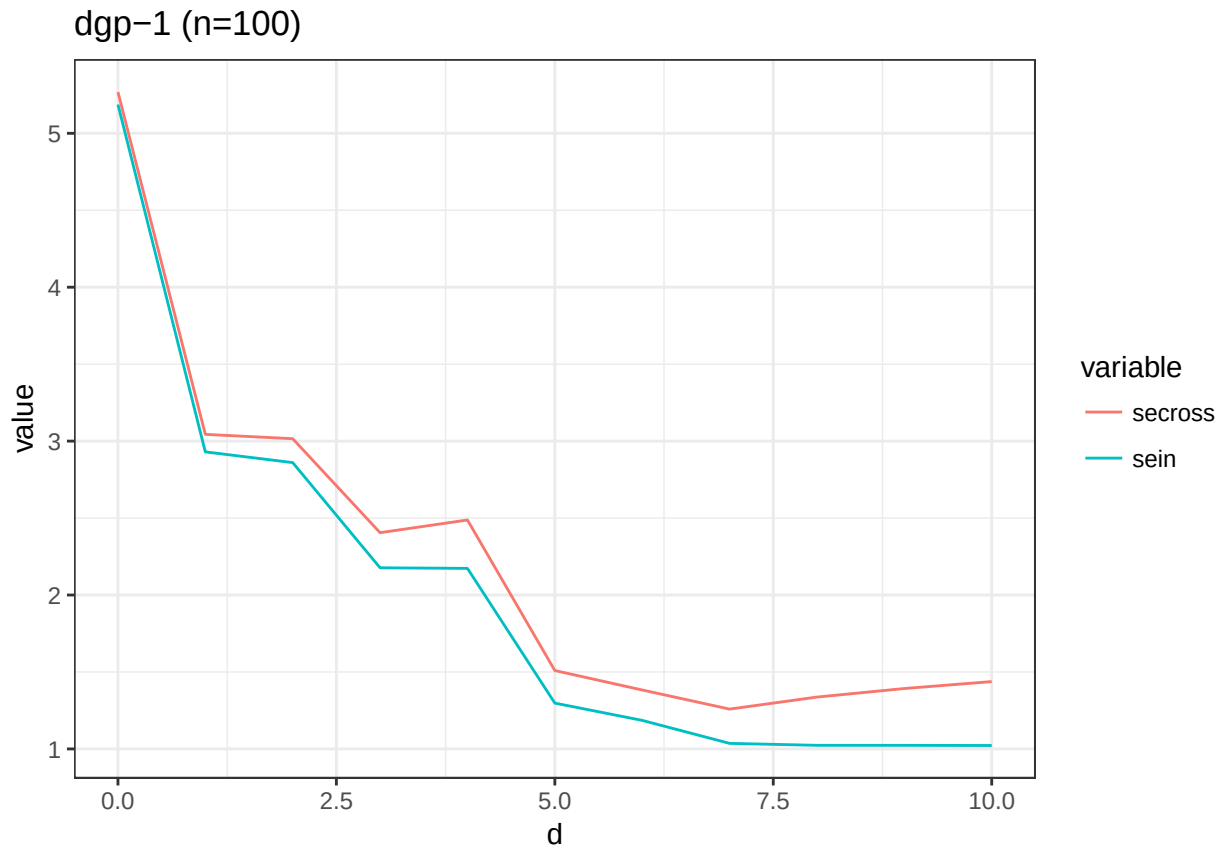
1/2

```
#holds constants and program parameters
CONST = list(
    EPSILON = .Machine$double.eps, #machine precision
    MIN_DOUBLE = .Machine$double.xmin,
    NUM_WORKERS = 7 #max(round(detectCores() * .5), 2) #just a heuristic for multi-threading
)

predict0 = function(m, dt) {
    options(warn = -1) #disable the rank defficient warning
    p = predict(m, dt)
    options(warn = 1)

    return(p)
}

#univariate k-fold polynomial cross-validation
polygnome = function(dt, degree, k, yname = "Y", xname = "X") {
    print("got here")
```

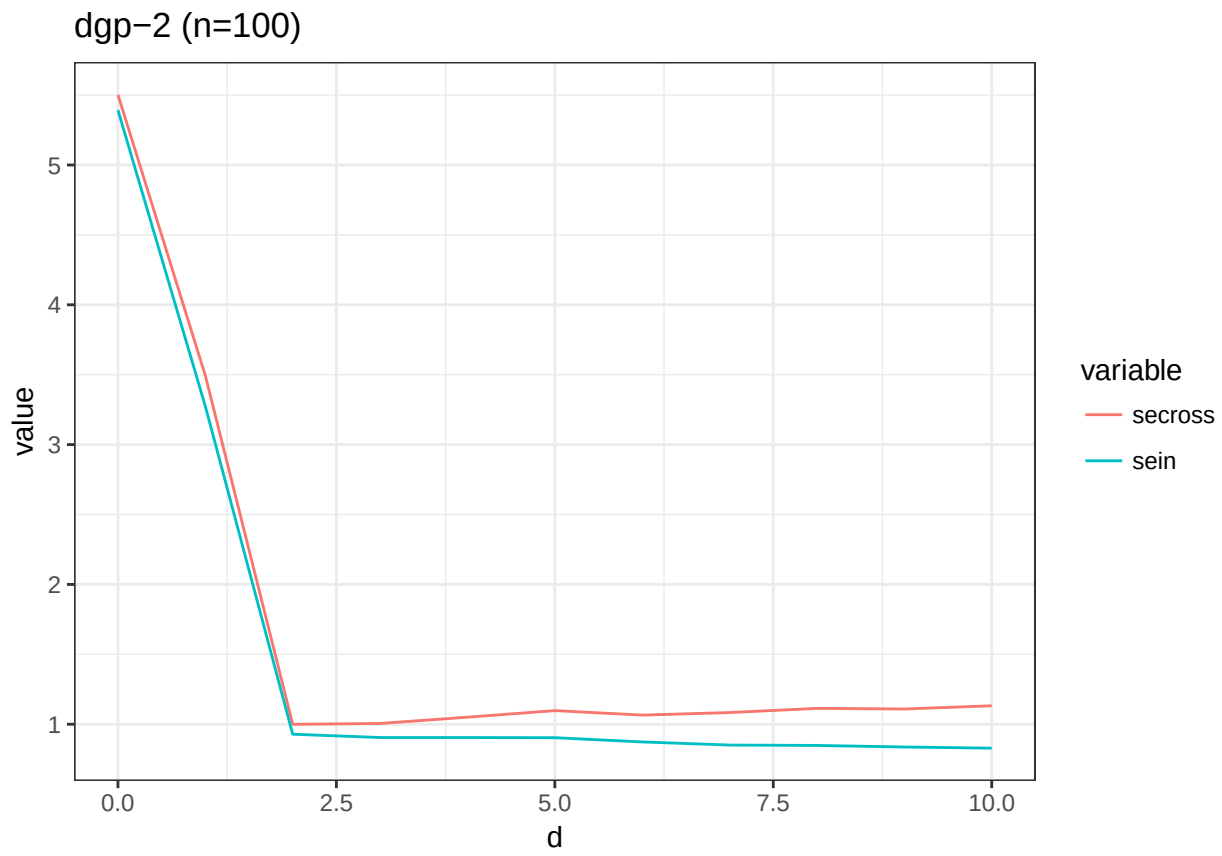
```
X = dt[[xname]]
n = length(X)

#create the formula
rhs = "intercept"
coefs = rhs
if (degree > 0) {
    rhs = paste0(c(rhs, " + ", xname))
    coefs = append(coefs, rhs)
}

if (degree > 1) {
    for (i in 1:degree) {

        coefname = paste0(xname, i, collapse = "")
        dt[, eval(coefname)] = X ^ i
        rhs = paste0(c(rhs, " + ", coefname))
        coefs = append(coefs, rhs)
    }
}

#handle the intercept ourselves
rhs = paste0(c(rhs, " + 0"))

#create the formula
dt$intercept = rep(1, n)
spec = formula(paste0(c(yname, " ~ ", rhs), collapse = ""))

#create the folds
foldwidth = trunc(n / k)
dt$fold = rep(k, n)
r = 1
for (i in 1:(k - 1)) {
    for (j in 1:foldwidth) {
        dt$fold[r] = i
        r = r + 1
    }
}

#cross-validate
folderrors = rep(0.0, k)
for (i in 1:k) {
    m = lm(spec, dt[fold != i])
    p = predict0(m, dt[fold == i])
    folderrors[i] = sum((dt[fold == i, Y] - p) ^ 2)
}
secross = sum(folderrors) / n

#compute the in-sample variance
m = lm(spec, dt)
p = predict0(m, dt)
sein = mean((p - dt[, Y]) ^ 2)
```

```
    return(list(model = m, secross = secross, sein = sein))
}

#form the data
formdgps = function(n) {
    X = runif(n, min = -4, max = 4)
    epsilon = rnorm(n)

    Y1 = -2 * (X < -3) + 2.55 * (X > -2) - 2 * (X > 0) + 4 * (X > 2) - 1 * (X > 3) + epsilon
    Y2 = 6 + .4 * X - .36 * X ^ 2 + .005 * X ^ 3 + epsilon
    Y3 = 2.83 * sin(pi / 2 * X) + epsilon
    Y4 = 4 * sin(3 * pi * X) * (X > 0) + epsilon

    return(list(dgp1 = data.table(Y = Y1, X = X),
      dgp2 = data.table(Y = Y2, X = X),
      dgp3 = data.table(Y = Y3, X = X),
      dgp4 = data.table(Y = Y4, X = X)))
}

testpolignome = function(n, showplots = TRUE) {
    cl = makeCluster(CONST$NUM_WORKERS)
    clusterExport(cl = cl,
        varlist = c("testpolignome", "polygnome", "CONST", "predict0", "formdgps", "data.table"))

    dgpn = formdgps(n)

    processpar = function(dgp) {
        return(parLapply(cl, 0:10, function(d) polygnome(dgp, degree = d, k = 10)))
    }

    #clusterExport(cl=cl, varlist = c("processpar"))

    #run polygnome for each polynomial size and data generating process
    results010list = lapply(dgpn, processpar)

    #recast the results in a usable form
    results010 = lapply(results010list,
        function(results) melt(data.table(d = 0:10,
            secross = sapply(0:10, function(x) results[[x + 1]]$secross),
            sein = sapply(0:10, function(x) results[[x + 1]]$sein)), id.vars = "d"))

    #make the plots
    d = rep(-1, 4)
    for (i in 1:4) {
        p = ggplot(results010[[i]], aes(x = d, y = value, color = variable)) + geom_line() + theme_bw()
            ggtitle(paste0(c("dgp-", i, " ", "(n=", n, ")"), collapse = ""))

        if (showplots) print(p)
        d[i] = which.min(results010[[i]][variable == "secross", value])
        print(paste0("dsg-", i, " min mse: degree=", d[i] - 1, "  for n = ", n))
    }

    #extract the best models
```

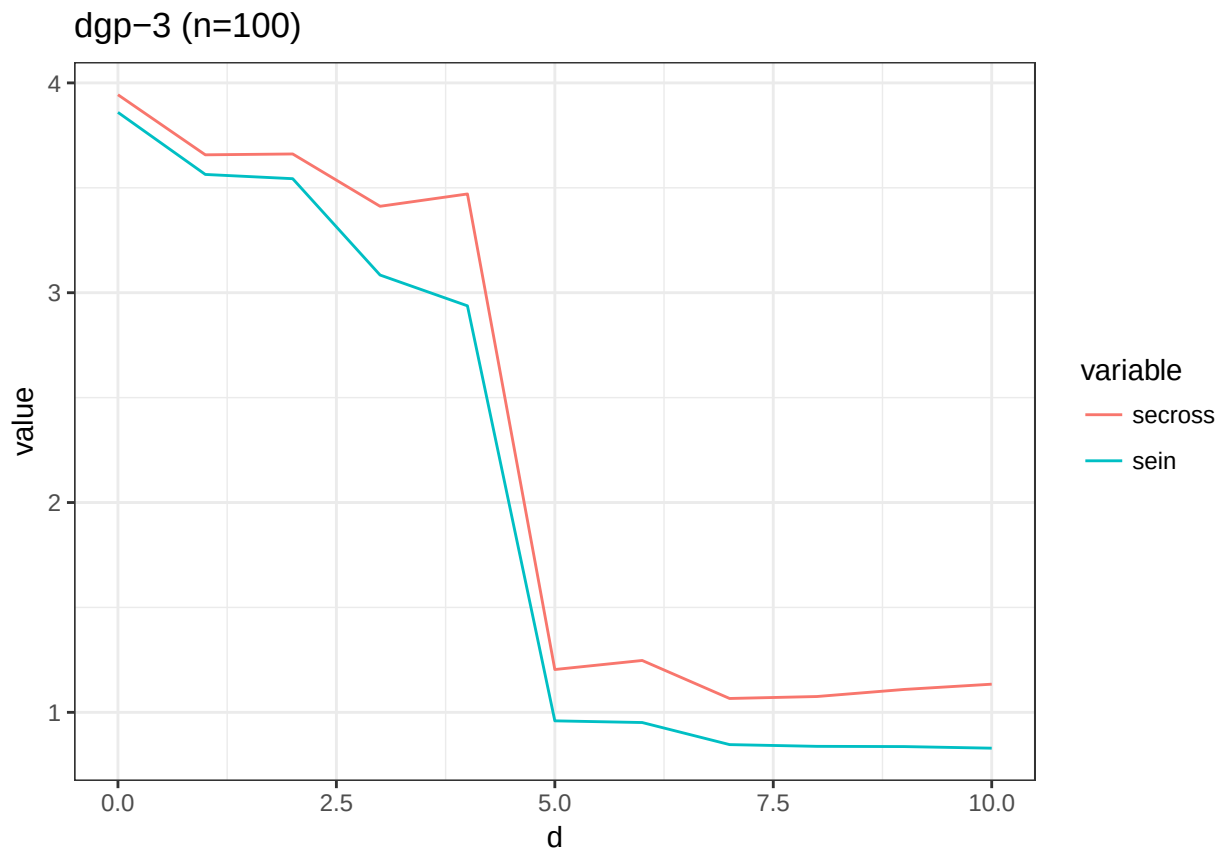
```
    models = mapply(function(resultlist, degree) resultlist[[degree]]$model,
        results010list, d, SIMPLIFY = FALSE)

    #cleanup
    stopCluster(cl)

    return(list(dgpn = dgpn, models = models, d = d))
}

####problem 3.2
p32results = testpolignome(n = 100)
```



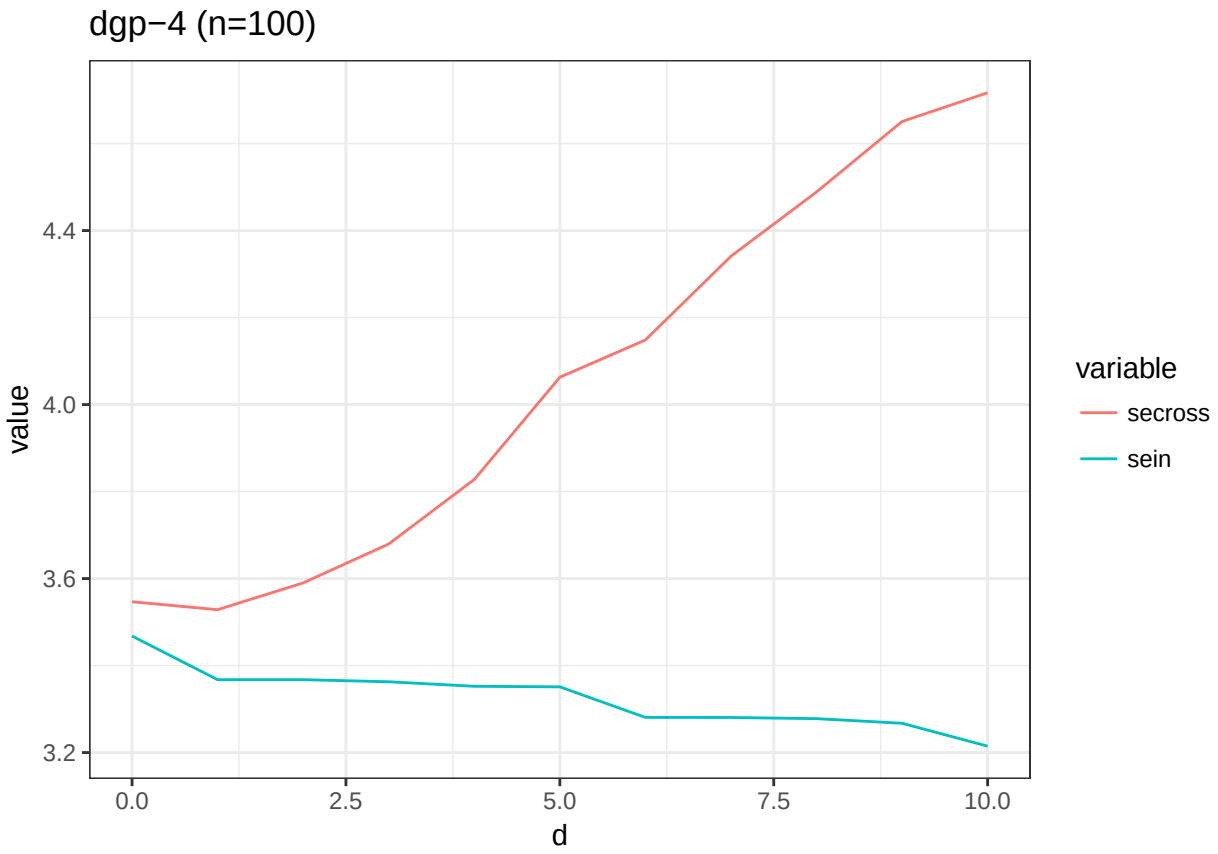

```
## [1] "dsg-1 min mse: degree=7  for n = 100"
```




```
## [1] "dsg-2 min mse: degree=2  for n = 100"
```




```
## [1] "dsg-3 min mse: degree=7  for n = 100"
```



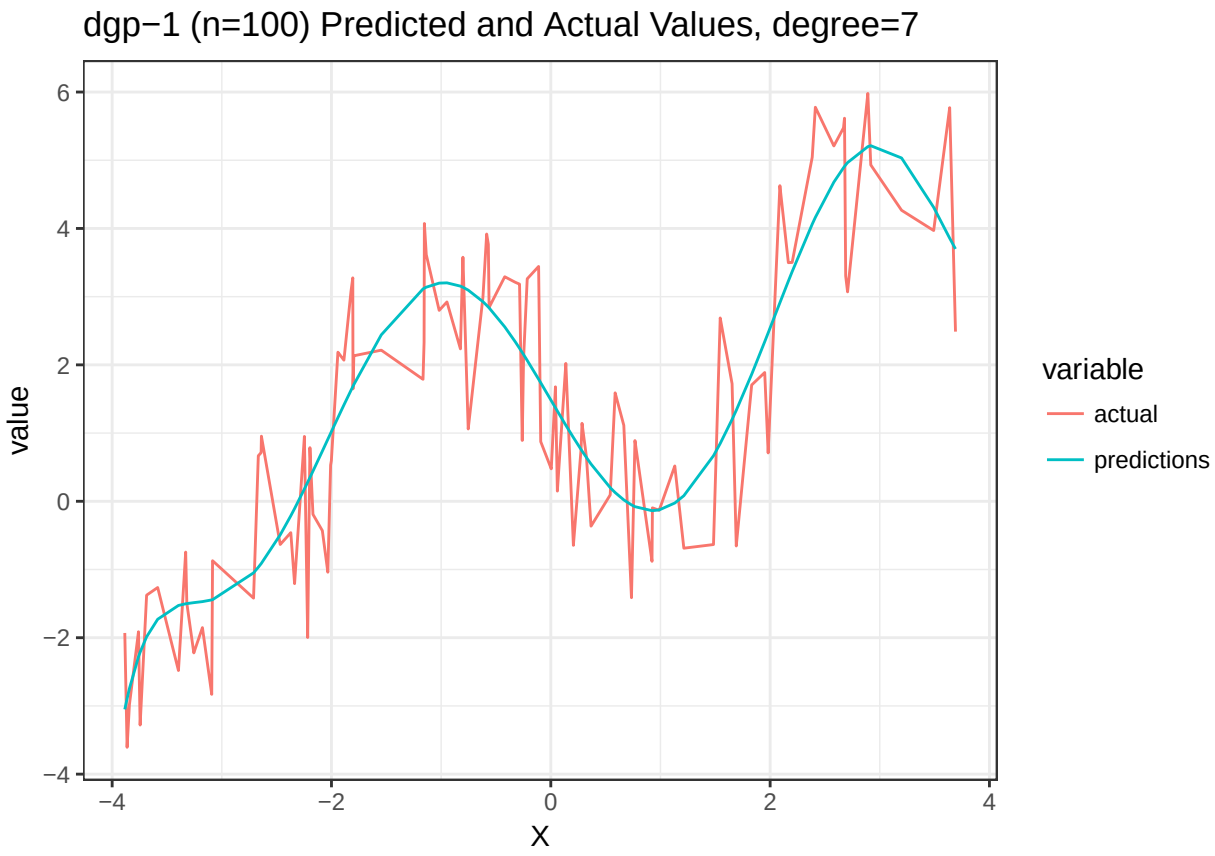

```
## [1] "dsg-4 min mse: degree=1  for n = 100"
```

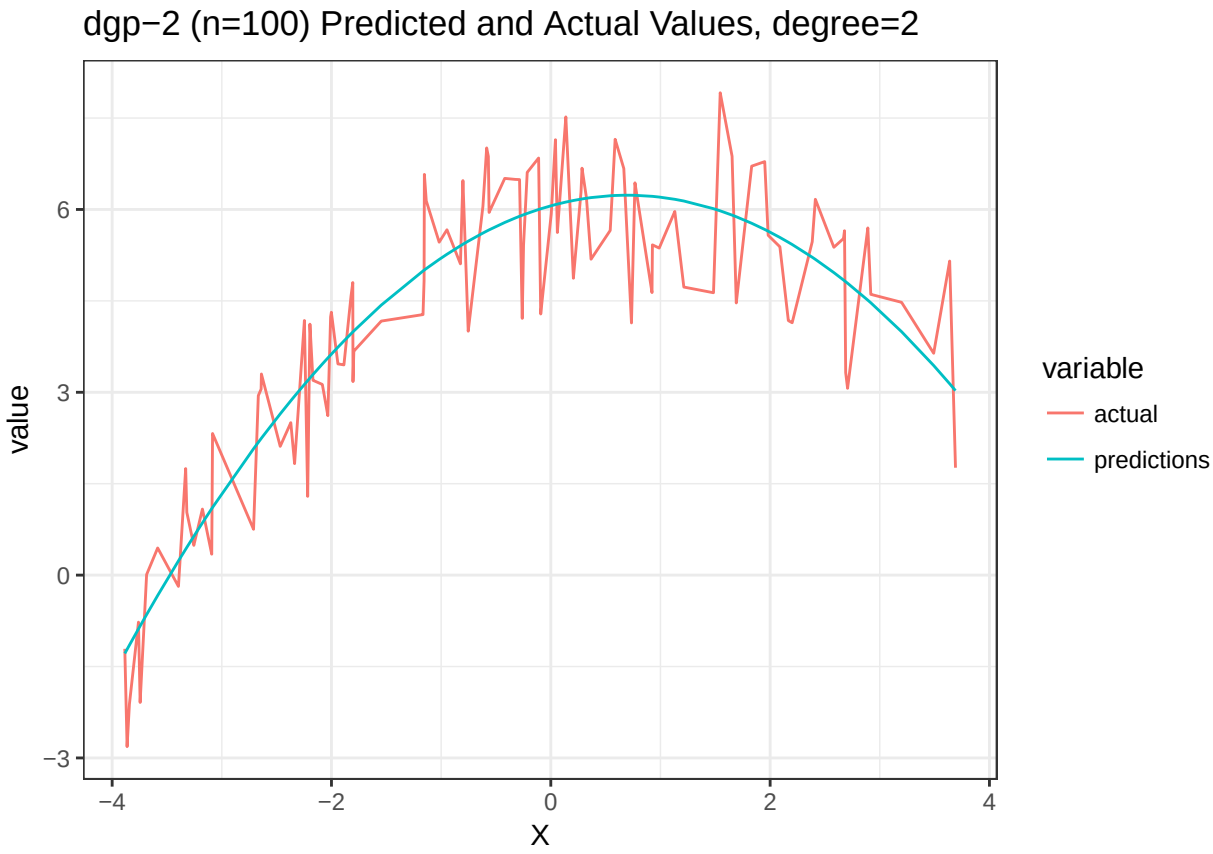
- From the plots:
  - The results from dgp-1 seem somewhat ambiguous, but degree=7 seems like a reasonable choice
  - For dgp-2, degree=2 clearly makes sense
  - For dgp-3, degree=7 seems reasonable, although like with dgp-1 we have a lot of noise.
  - For dgp-4, a linear model (degree=1) looks reasonable.

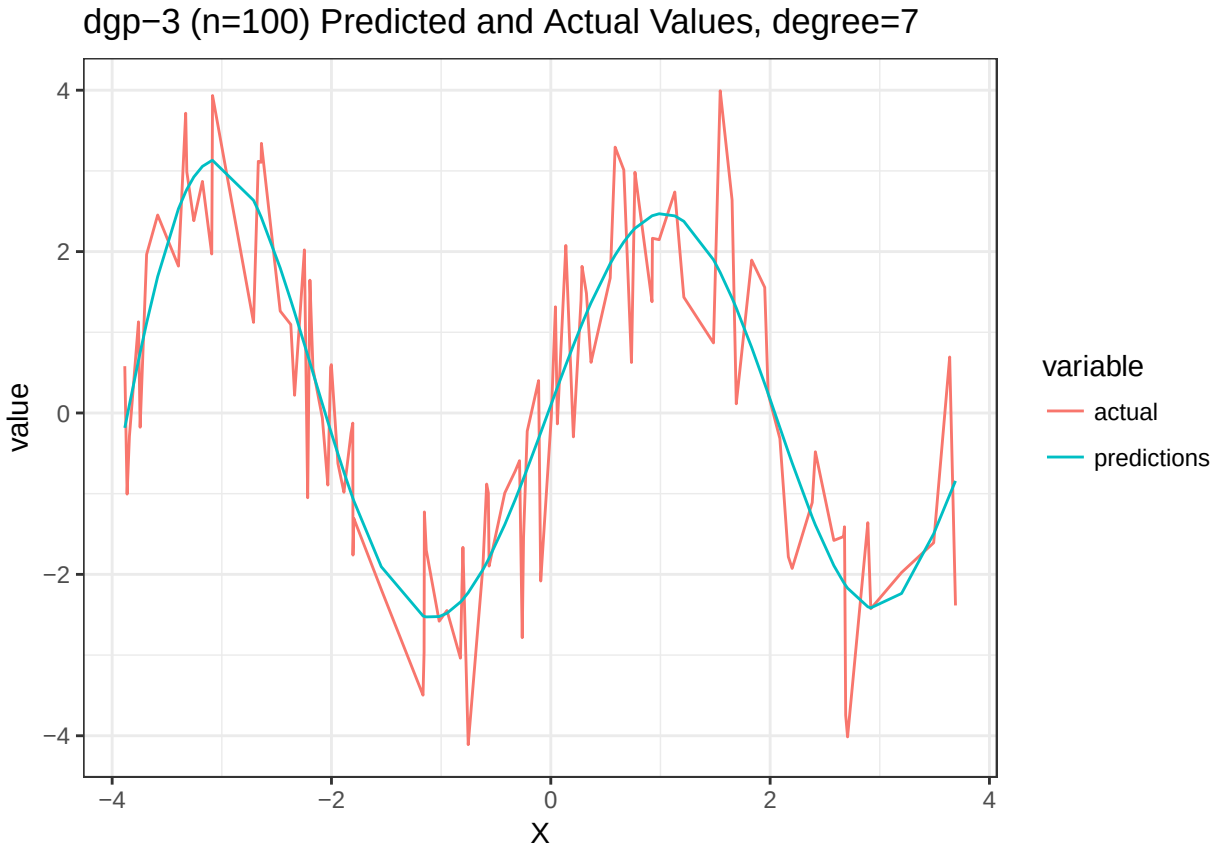
3

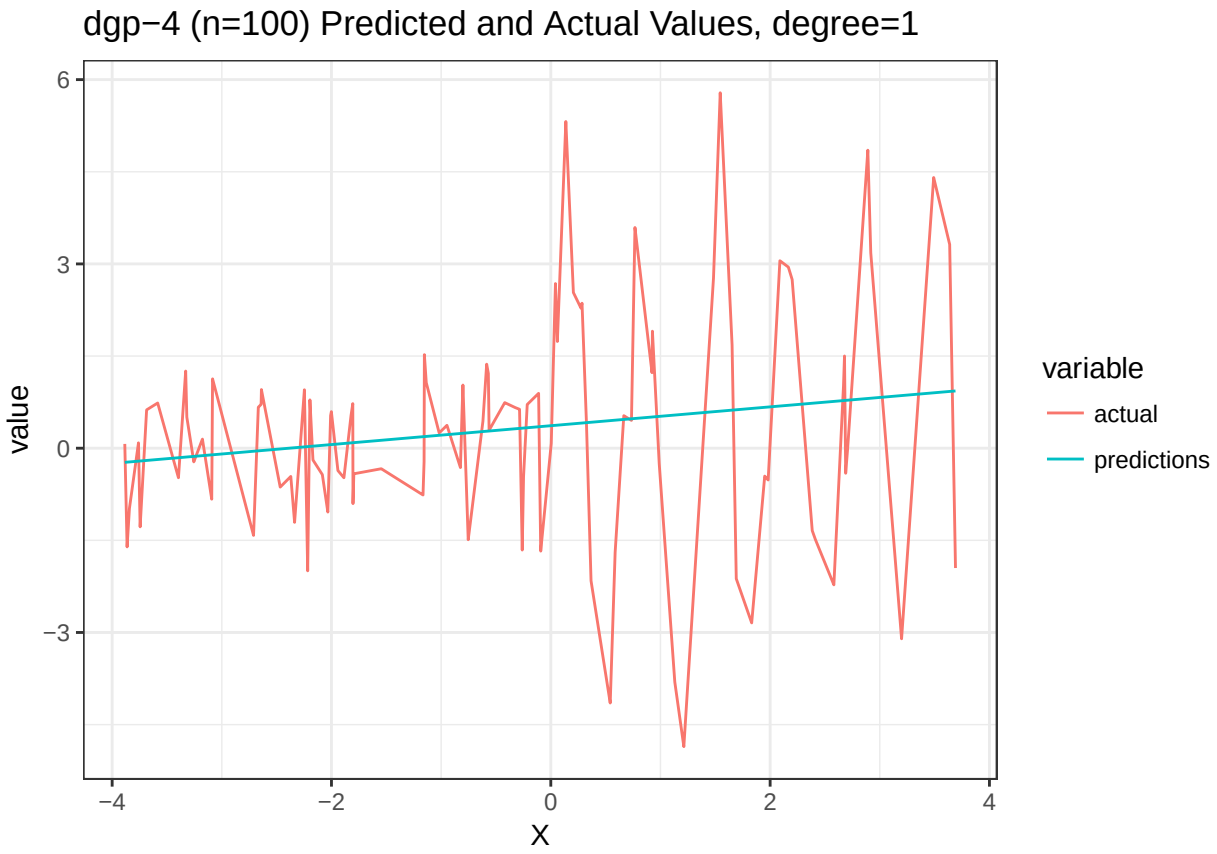
```
plotpredictions = function(dgpn, models, degrees) {

    n = nrow(dgpn[[1]])

    #helper function to make the dt containing the predictions
    predictdt = function(dgp, model, degree) {
        X = dgp[, X]
        if (degree > 1) {
            for (i in 1:degree) {
                coefname = paste0("X", i, collapse = "")
                dgp[, eval(coefname)] = X ^ i
            }
        }

        dgp$intercept = rep(1, n)
```

```
        p = predict0(model, dgp)
        dt = data.table(X = dgp[, X], actual = dgp[, Y], predictions = p)

        return(melt(dt, id.vars = "X"))
    }

    predtables = mapply(predictdt, dgpn, models, degrees, SIMPLIFY = FALSE)

    for (i in 1:4) {
        p = ggplot(predtables[[i]], aes(x = X, y = value, color = variable)) + geom_line() + theme_bw()
            ggtitle(paste0(c("dgp-", i, " ", "(n=", n, ") Predicted and Actual Values, degree=",
                degrees[i] - 1), collapse = ""))
        print(p)
    }

    return(NULL)
}

###problem 3.3
plotpredictions(dgpn = p32results$dgpn, models = p32results$models, degrees = p32results$d)
```




dgp−2 (n=100) Predicted and Actual Values, degree=2
value
6
3
0
−3
−4
−2
0
2
4
X
variable
actual
predictions


dgp−3 (n=100) Predicted and Actual Values, degree=7
value
X
4
2
0
−2
−4
−4
−2
0
2
4
variable
actual
predictions




```
## NULL
```

4

```
finalresults = testpolignome(n = 10000, showplots = FALSE)
```

```
## [1] "dsg-1 min mse: degree=10  for n = 10000"
## [1] "dsg-2 min mse: degree=3  for n = 10000"
## [1] "dsg-3 min mse: degree=10  for n = 10000"
## [1] "dsg-4 min mse: degree=10  for n = 10000"
```

- The optimal degree from cross validation is significantly higher (10) for all of the processes except dgp-2. For dgp-2, the algorithm picks a degree of 3, which is the degree of the true process. Otherwise, the algorithm seems to be picking up higher and higher order terms from the Taylor expansion. This makes sense, as the greater information content of the additional data allows the algorithm to make more precise guesses as to the true structure of the series without sacrificing robustness.