Bill Foote

Making Quantiles: Goal Programming and Bayesian Approaches

Contents

1. Making quantiles

Imagine we have the problem of buying a used asset. By used we mean hours spent operating the asset. We would like to somehow impute pricing on an 80/20 split, that is, average pricing with 80% plausibility versus 20%.

A quantile regression

One rough and ready approach would be to estimate the parameters of the simple linear regression at a quantile $\tau = 0.80$, or some other split, of the dependent variable y_i conditional on independent variable x_i.

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

At quantile τ we define the model as $y_i > \alpha + \beta x_i$ with positive deviations of y_i from the model $\alpha + \beta x_i$ as

$$\begin{aligned} \varepsilon_i^+ &= y_i - \alpha + \beta x_i, & (1) \\ \text{for } y_i &> \alpha + \beta x_i. & (2) \end{aligned}$$

At quantile $1 - \tau$ we define the model as $y_i \leq \alpha + \beta x_i$ with negative deviations of y_i from the model $\alpha + \beta x_i$ as

$$\varepsilon_i^- = y_i - \alpha + \beta x_i, \tag{3}$$

$$\text{for } y_i \leq \alpha + \beta x_i. \tag{4}$$

Our objective, should we choose to accept it, is to find those *alpha*s and βs to minimize the sum of weighted deviations on both sides of the τ quantile divide. The weights are simply τ and $1 - \tau$. This ought to look like a standard goal programming problem. And it does.

We can stack our constraints as columns of 1's, x_i values, as well as positive and negative deviations, all equal, row i by row i, to y_i as a hard goal. There is one wrinkle in this otherwise easy to digest story. When the y_is split at the τ quantile, so does the model. We already recognize the ε split into positive and negative deviations (y_i above and below the regression line). What we also need to do is realize that the regression parameters will also follow the same split. We will have α^- and α^+ and the same for β, and thus two regression lines. This will allow for the linear program to use all positive parameters in the basis, the solution set. If we simply subtract one regression line from the other, we can recognize this differential. Thus the estimator of $\alpha = \alpha^+ - \alpha^-$ and $\beta = \beta^+ - \beta^-$.[1]

Implementing the model

We now set up a standard (if not canonical) goal programming model.

$$min_{\{\alpha^{+,-}, \beta^{+,-}, \varepsilon^{+,-}\}} \Sigma_{i=1}^N [\tau\varepsilon_i^+ + (1-\tau)\varepsilon_i^-]$$

We can succinctly write

$$min_\theta \, c^T\theta$$

where the θ is a $N+2$ vector of the $\alpha^{+,-}$, $\beta^{+,-}$, $\varepsilon_i^{+,-}$ parameters, including the deviations as parameters, N the number of observations and thus constraints; and c is a $N+2$ vector of objective function coefficients, 0's for the α and β parameters, and τ or $1-\tau$ for positive or negative deviations. The expression $c^T\theta$ is known in mathematics as an inner product. The computation will produce a scalar from the two vectors, perfect for an objective function value – and that's that! Phew!

We will concatenate blocks of intercept and x_i vectors with positive identity matrices (1's on the diagonal, 0's off the diagonal) and negative identity

[1] See Koenker (2005) for several details and simplifications. In the R package **quantreg** he uses the Karmarkar (1984) interior point algorithm implemented as a call to Rational FORTRAN as opposed to our rendering with the **lpSolve** Simplex algorithm here.

matrices (-1's on the diagonal, 0's off the diagonal) all into matrix A with dimension $N \times (N+4)$. The N rows are the rows of observations and the $2N+4$ columns are the $2N$ positive and negative deviations with 2 positive and 2 negative intercepts and slopes. Lots of bookkeeping will keep us busy! The right-hand side N vector b contains the y_i observations.

$$A\theta = b$$

Here the solution vector $\theta \geq 0$. constrained quantile regressions can be set up with non-data constraints as well.

Some dimensions are much in order. There are N rows representing the number of observations. There are K columns in the regression: a column of 1's for the intercept (very old school) and, just one column for the x_i vector of observations of the independent variable. However there are N columns each of positive and negative deviations, one for each observation for a total of $2N$ deviational columns. The matrix A thus has dimension (rows × columns) of $N \times 2K + 2N$, remembering positive and negative regression parameters. In our case, $N = 5$ and $K = 2$, so that A has dimension 5×14. One can imagine that the proliferation of deviations can seem to get a bit out of hand when we suppose a data set with $N = 1000$ observations with intercept and 3 explanatory variables and thus an A coefficient matrix of dimension 1000×2006.

```
library(tidyverse)
library(lpSolve)
# Data
y <- c(12500, 13350, 14600, 15750, 17500)/1000 # price
x <- c( 43890, 37750, 27300, 15500, 8900 )/1000 # odometer miles
d <- tibble(
  price = y,
  odo = x
)
# Variates including intercept as 1s
X <- cbind( 1, d$odo )
K <- ncol(X)
N <- nrow(X)
# Build constraints: note the X and -X
# to estimate the alpha+ and alpha- parameters, as well as beta+ and beta-
# The identity matrices 1*diag(N)
# and -1*diag(N) are the positive
# and negative deviations.
A <- cbind( X, -X, diag(N), -diag(N) )
const_type <- rep( "=", N )
```

In this example we will set $\tau = 0.8$ while $1 - \tau = 0.2$. Relatively the positive deviations of y_i from the model are 4 times as *likely* in the analyst's mind than the negative deviations in this thought experiment. Are they really? Not at all necessarily! What the model does for us is explore what the intercept and slope look like if this is the likely or preferentially set weighting.

```
# Define objective function
# Set 0.2, 0.8 split in cumulative probability:
# odds ratio = 8/2 = 4:1 in favor of positive deviations;
# nice prior
tau <- 0.8
# 0's for the positive and negative parameter blocks
# tau*1 for positive and (1-tau)*1
# for negative deviations
c <- c( rep(0, 2*K),
        tau*rep(1,N),
        (1-tau)*rep(1,N)
)

# RHS variable is y_i
b <- d$price

quant_reg <- lp( "min",
                 c,
                 A,
                 const_type,
                 b
                 )
theta <- quant_reg$sol[1:K] -  quant_reg$sol[(1:K+K)]
theta
```

```
## [1] 18.771792 -0.142898
```

```
# Compare with Koenker's quantreg::rq()
library(quantreg)
rq(price ~ odo,
   tau=tau,
   data=d)
```

```
## Call:
## rq(formula = price ~ odo, tau = tau, data = d)
##
## Coefficients:
## (Intercept)         odo
##   18.771792   -0.142898
##
## Degrees of freedom: 5 total; 3 residual
```

Again, we are on target with our data and models.

2. Inference

There is a possible correspondence between the quantile regression approach and inference using probabilistic simulation as in Bayesian analysis. If we think about the τ parameter as an hierarchical input with distribution $tau \sim \text{BETA}(0,1)$, then the dependent variate $y \sim \text{QR}(\tau, q_\tau)$ with $q_\tau = \alpha + \beta\, x$, where q_τ is the τth quantile dependent on independent variate x and parameters $\alpha,\ \beta$.

Benoit and Poel (2017) deploy the *asymmetric Laplace distribution (ALD)* of Yu and Zhang (n.d.) as the observational model for detecting quantile likelihoods.

$$f(x; q, \sigma, \tau) = \frac{\tau(1-\tau)}{\sigma} \begin{cases} \exp\left(-((\tau-1)/\sigma)(x-q)\right) & \text{if } x \leq q \\ \exp(-(\tau/\sigma)(x-q)) & \text{if } x > q \end{cases} \tag{5}$$

The two branches of the density locate the quantile split in the x variate around the τth quantile q, the new marker of relevant location.[2]

Here is a procedure we can use to incorporate this distribution into a quadratic

[2] The motivation for quantile regression is usually that there is some departure from the assumption of independent and identically distributed (IID) error terms and the case of error heteroskedasticity has been a prominant reason to invoke quantile regression. Heteroskedasticity means for us that there are multiple behaviors at work in the realm of the known-unknown of statistical residuals. Each behavior is somehow parametrized by a different σ. In this case, the OLS assumptions would not hold and quantile regression can offer additional flexibility. Yet the Bayesian ALD approach also does assume IID errors and this might seem to contrast with one of the key reasons for using distribution-free (so it seems) quantile regression. The motivation for the approach follows from the fact that the maximum likelihood estimate from the ALD likelihood is equivalent to solving the goal programming minimization problem. Whatever the issues, we must remember that assumptions obtain only within the model, as McElreath (2020) points out, in the *golem*, the *robot* we are programming. These assumptions are part of the abstraction from the reality of what is represented through our measured (more assumptions!) use of observations. Our epistemological (how we know what, ontologically, we know) probability specifications might seem even much stronger than the reality we model. But we must remember Edwin T. Jaynes (2004) and his warning that we also must not confuse assumptions about the world we observe with assumptions we make to represent uncertainty in our model. He called this confusion the *mind projection fallacy*, also known as logical *reification*. Reality comes first and is out there to be known for us. Our model is one way of finding intelligibility, patterns, in what we observe in the wild. What is true is objective reality and it is intelligible. And if is not intelligible we would abolish every body of knowledge in the natural and human sciences! Probability is only an artifice we use to represent our view and version of what patterns we can eek out of very limited knowledge about reality. Probability possesses what W. Norris Clarke, SJ called *mental being*, that is, something, not objectively true being we observe as subjects. This approach well aligns with Finetti (1974) and Nau (2001).

approximation of the posterior distribution conditional on data and a prior distribution.

1. Define the ALD density as the vectorized function dALD_vec.
2. Specify a simple linear model of price and odometer readings as the mean with standard deviation with price normally, Gaussian that is, distributed.
3. Specify normal priors for the intercept and slope of the linear model, exponential prior for the standard deviation of price, and a beta density to bind the quantile parameter between 0 and 1.
4. Write a negative log-likelihood function.
5. Find the point values of the parameters that minimize the negative log-likelihood function and retrieve the gradient Hessian matrix to calculate the variance-covariance matrix and standard deviations of the parameters.
6. Extract samples for inference.

The vectorized ALD density

Here is the vectorized ALD density. This one is based on the `ALD` R package.

```r
dALD_0 <- function(y, mu = 0, sigma = 1, tau=0.5, log=FALSE) {
    if (length(y) == 0)
        stop("Please input y.")
    if (sum(y[is.na(y) == TRUE]) > 0)
        stop("There are some NA values in y.")
    if (sigma <= 0)
        stop("sigma must be a positive number.")
    if (tau >= 1 | tau <= 0)
        stop("Be sure that tau is a real number in (0,1), eg, 0.25.")
    if (abs(mu) == Inf)
        stop("mu must be a finite real number.")
    dALD <- ifelse(test = y < mu,
                   yes = (tau * (1 - tau)/sigma) *
                     exp((1 - tau) * (y - mu)/sigma),
                   no = (tau * (1 - tau)/sigma) *
        exp(-(tau) * (y - mu)/sigma)
        )
    dALD <- ifelse(test = (log == TRUE),
                   yes = log( dALD ),
                   no = dALD
        )
    return(dALD)
}
# Vectorize dALD_0
dALD_vec <- Vectorize(dALD_0, vectorize.args = c("y", "mu"))
```

The functions contain error handling rules for each parameter. They help immensely to shoot trouble and are simply good program engineering practice. They also force us to think closely and clearly about the ranges and types of admissible values. Here are some computations and a stubby-fingered mistake to test our functions.

```
dALD_0( 200, mu = 220, sigma = 21, tau = 0.5)
```

```
## [1] 0.007394585
```

```
# try this
# dALD_0( 200, mu = 220, sigma = 21, tau = 5)
#to see the function catch an inadmissable value for tau:
#  Error in dALD_0(200, mu = 220, sigma = 21, tau = 5) :
#  Be sure that tau is a real number in (0,1), eg, 0.25.
#
dALD_vec( y=1:5, mu=5-0.1*6:10, sigma=2, tau=0.5)
```

```
## [1] 0.05342687 0.07033811 0.09260228 0.12191374 0.09735010
```

The vectorized density will be more than a little useful in sampling with data series.

McElreath's parser

Logic, causation and an algorithm are needed to estimate our quantile model. We start with formulas. They encapsulate our logic.[3]

Formulas in R are binary operators with left- (LHS) and right-hand (RHS) sides as operands and an operator in the middle. For example, the formula $y \sim dnorm(mu, sigma)$ parses as LHS operator RHS, where LHS $= y$, RHS $= dnorm(mu, sigma)$ and operator $= \sim$. The formula class stores the formula, seemingly in reverse polish notation (RPN), as a list where f[[1]] = operator,f[[2]] = LHS, and f[[3]] = RHS. Other examples of operators include=,<-,%*%, as well as custom operators like the piping operator S`|>`.

Here is an example of a simple model deposited into a list.

```
f_list <- list(
  price ~ dALD_vec(a+b*odo,sigma,tau),
```

[3] Richard McElreath shared with us on his twitter account an early version of his `rethinking::quap()` function with parser which will look very much like the one here – because it is, however with some modifications and much explanation of the inner workings of such modeling.

```
  a ~ dnorm(0,10),
  b ~ dnorm(0,1),
  sigma ~ dexp(1),
  tau ~ dunif(0.009,0.999)
)
f_list
```

```
## [[1]]
## price ~ dALD_vec(a + b * odo, sigma, tau)
##
## [[2]]
## a ~ dnorm(0, 10)
##
## [[3]]
## b ~ dnorm(0, 1)
##
## [[4]]
## sigma ~ dexp(1)
##
## [[5]]
## tau ~ dunif(0.009, 0.999)
```

```
str(f_list)
```

```
## List of 5
##  $ :Class 'formula'  language price ~ dALD_vec(a + b * odo, sigma, tau)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language a ~ dnorm(0, 10)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language b ~ dnorm(0, 1)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language sigma ~ dexp(1)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language tau ~ dunif(0.009, 0.999)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

There are four elements to the list. We just displayed the first of the four. Our net step is to parse[4] Already we see with `str(f_list)` that R understands the list as class formula, simply because of the dyadic structure `LHS ~ RHS` and delimiter `~`.

Our goal is to use quadratic approximation of the posterior curve to simulate

[4] Parse, from the Latin *pars orationis* simply meant to analyze a speech, *oratio*, into its parts. We use this word to mean analyze as well, as in a hyper-detailed way. The details will include functions, operators, parameters, data, all the parts of the speech we speak.

the various posterior distributions of parameters. We will be minimizing the log likelihood conditional on the probability of parameters.

The logarithm of the Gaussian distribution is essentially quadratic in structure.

$$\log\ g(x;\mu,\sigma) \propto (1/\sigma^2)(x^2 - 2\mu x + \mu^2) \tag{6}$$

where the $g()$ is the Gaussian density. Laplace, Truscott, and Emory (1902) used this fact and his observation that likelihood functions looked quadratic over certain domains of the posterior distribution as the basis of an approximation. This meant he would find the parameters which would effectively be compatible with the minimum (or maximum) of a parabola. Then, he would use these parameters to simulate the distribution around the modeled observations.

Here is the general form of the log likelihood function with log prior densities we will use.

$$\begin{aligned}
\log \mathcal{L}(\alpha,\beta,\sigma \mid y,x) = &\log f(y,x;\alpha,\beta,\sigma,\tau) + && (7)\\
&\log g(\alpha) + \log g(\beta) + && (8)\\
&\log h(\sigma) + \log h(\tau). && (9)
\end{aligned}$$

Where $f()$ is the ALD density, $g()$ is the Gaussian density, and $h()$ is a non-zero density for σ and τ.

We now need to fill in the blanks for the following R optimization routine.

```
fit <- optim( par=parameters, fn=neg_loglik, flist=f_list_1, data=d, hessian=TRUE, ...)
```

And the Hessians are coming! We will invert this matrix to estimate the variance-covariance matrix of parameters `parameters` which includes this list with starting values for the numerical optimization scheme. Next is the list of the parameters, each with a starting value to help the optimization routine to get to a possibly more reasonable answer. We must remember we are searching a four-dimensional space, with not a few rules, for that needle in a hay stack of a solution.

```
parameters <- list( a=14, b=0, sigma=1, tau=0.5 )
```

We specify the formulas we just devised and will modify into this list of distributions which all can be recognized by R. Along with the list of formulae, we will insert parameters and data into a log likelihood function, all for the `optim()` specification.

```
formula2text <- function( f_list ){
    RHS <- f_list[[3]]
    LHS <- f_list[[2]]
    n_args <- length( RHS )
    args_list <- as.list( RHS )
    args_list[[1]] <- LHS
    args_list[[n_args+1]] <- "log=TRUE"
    args_text <- paste( args_list, collapse=", "  )
    paste0( RHS[[1]], "( ", args_text, " )" )
}
f_list_0 <- as.list( f_list )
# Vectorize formula2text with l(list)apply
f_list_1 <- lapply( f_list_0, formula2text )
f_list_1
```

```
## [[1]]
## [1] "dALD_vec( price, a + b * odo, sigma, tau, log=TRUE )"
##
## [[2]]
## [1] "dnorm( a, 0, 10, log=TRUE )"
##
## [[3]]
## [1] "dnorm( b, 0, 1, log=TRUE )"
##
## [[4]]
## [1] "dexp( sigma, 1, log=TRUE )"
##
## [[5]]
## [1] "dunif( tau, 0.009, 0.999, log=TRUE )"
```

The `formula2text` function iterates through each line of the formula in `flist`. The whole purpose of this process is to get data into the density functions we specified. The so-called data are observed `y` and `x`, and as yet unobserved intercept `a`, slope `b`, and even the usually user-specified `tau`.

The data is what we used for the goal programming approach.

```
library(tidyverse)
# Data
y <- c(12500, 13350, 14600, 15750, 17500)/1000 # price
x <- c( 43890, 37750, 27300, 15500, 8900 )/1000 # odometer miles
d <- tibble(
  price = y,
  odo = x
)
```

Last, and definitely not the least, we specify the objective function to be minimized, the default action in `optim()`. Our objective is to find parameter values which maximize the log-likelihood of `dALD_0()` conditioned (since logs we add) by the logarithms of `dnorm(a)` and `dnorm(b)`. To minimize we simply multiply the log-likelihood by -1.

```
# Use of optim requires us to use its
# argument list here: par, flist, data
  neg_loglik <- function(par, flist, data){
    e <- list( as.list(par), as.list(data) )
    e <- unlist( e, recursive = FALSE )
    LL <- sapply( flist, function(i) sum(eval(parse(text=i), envir=e)) )
    return( -sum(LL) )
  }
#Here is one run
neg_loglik(parameters, f_list_1, d)
```

```
## [1] 17.04188
```

The next sub-section dives much more deeply into the McElreath primitive `quap()` structure. Perhaps we go feet first instead of head first?

Inside the parser (very optional)

While this is (very) optional, bushwacking our way through the woods of a parser will 1) provide always needed practice with R lists, 2) insert new R functions into our growing memory, 3) introduce us to environments unique to functions and not shared with the global environment of this workspace.

We start with our model. We will play with the `dbeta()` version of a very uninformative, relatively, prior.

```
f_list <- list(
  price ~ dALD_vec(a+b*odo,sigma,tau),
  a ~ dnorm(0,10),
  b ~ dnorm(0,1),
  sigma ~ dexp(1),
  tau ~ dbeta(1,1)
)
#
f_list
```

```
## [[1]]
## price ~ dALD_vec(a + b * odo, sigma, tau)
##
## [[2]]
## a ~ dnorm(0, 10)
##
## [[3]]
## b ~ dnorm(0, 1)
##
```

```
## [[4]]
## sigma ~ dexp(1)
##
## [[5]]
## tau ~ dbeta(1, 1)
```

```
str( f_list )
```

```
## List of 5
##  $ :Class 'formula'  language price ~ dALD_vec(a + b * odo, sigma, tau)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language a ~ dnorm(0, 10)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language b ~ dnorm(0, 1)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language sigma ~ dexp(1)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language tau ~ dbeta(1, 1)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

So far we have a great way to depict, quasi-mathematically, our model. But R does not see these formulas the same way. From `str()` we see that R interprets the list as class `'formula'`, and this is because we use the formula delimiter `~`.

For example let's isolate `a ~ dnorm(0,10)` as the second item in `f_list`, `f_list[[2]]`. We will then translate our formula into `dnorm( a, 0, 10, log=TRUE )` so that R can evaluate the log-normal density of `a`. We will use the logarithmic evaluation of densities to compute the negative log likelihood of our model. The RHS of this binary relationship is `dnorm(0,10)`; the LHS is `a`; the delimiter of the two strings is `~`. By typing `str(f_list[[2]]` see that `f_list[[2]]` is a string of class `formula` with language simply because we use R's formula delimiter `~`. This allows us to pick our the RHS and LHS of the delimiter `~`. In the `formula` class, the default delimiter is `~` in `f_list[[1]]`. Positions 2 and 3 will yield the RHS and LHS strings. So much from so many lists of lists!

```
# We access list contents using [[number]]
f_2 <- f_list[[2]]
f_2 # for example
```

```
## a ~ dnorm(0, 10)
```

```
RHS <- f_2[[3]]
LHS <- f_2[[2]]
RHS
```

```
## dnorm(0, 10)
```

```
LHS
```

```
## a
```

```
# build the list of arguments (a, 0, 10, log=TRUE) for dnorm()
n_args <- length( RHS )
# 3: RHS[[1]]=dnorm, RHS[[2]]=0, RHS[[3]]=10
# as in this list
args_list <- as.list( RHS )
args_list[[1]] <- LHS # Insert parameter a first
args_list[[n_args+1]] <- "log=TRUE" # last
args_list
```

```
## [[1]]
## a
##
## [[2]]
## [1] 0
##
## [[3]]
## [1] 10
##
## [[4]]
## [1] "log=TRUE"
```

Now we have all of the arguments for a formula which R might be able to evaluate. All we have to do is strap the arguments from a list back into a string with commas, then paste the R function name `dnorm`, already in the `RHS[[1]]` position, along with appropriately placed parentheses `( )` to yield a string we might evaluate in R.

```
args_text <- paste( args_list, collapse=", "  )
args_text
```

```
## [1] "a, 0, 10, log=TRUE"
```

```
result <- paste0( RHS[[1]], "( ", args_text, " )" )
```

We have thus compiled, perhaps more like interpreted, math-English into R code. This function will perform everything we just witnessed for a list of formulae. Here is a function to help us.

```
formula2text <- function( f_list ){
  RHS <- f_list[[3]]
  LHS <- f_list[[2]]
  n_args <- length( RHS )
  args_list <- as.list( RHS )
  args_list[[1]] <- LHS #Replace ~ with parameter a
  args_list[[n_args+1]] <- "log=TRUE"
  args_text <- paste( args_list, collapse=", "  )
  return( paste0( RHS[[1]], "( ", args_text, " )" ) )
}
formula2text( f_list[[2]] )
```

```
## [1] "dnorm( a, 0, 10, log=TRUE )"
```

Next we build an environment `e` within which we evaluate the strings, aka R functions. The environment is native, not to the workspace within which the console operates, but only within which the function moves data. Before we get to a function-specific environment, here we have the raw workflow.

```
#
parameters <- list( a=15, b=-.1, sigma=0.3, tau=0.50 )
#
f_list_0 <- as.list( f_list )
flist_1 <- lapply( f_list_0, formula2text )
e <- list( as.list(parameters), as.list(d) )
e <- unlist( e, recursive = FALSE)
e
```

```
## $a
## [1] 15
##
## $b
## [1] -0.1
##
## $sigma
## [1] 0.3
##
## $tau
## [1] 0.5
##
## $price
## [1] 12.50 13.35 14.60 15.75 17.50
##
## $odo
## [1] 43.89 37.75 27.30 15.50  8.90
```

```
LL <- sapply( f_list_1, function(i) sum(eval(parse(text=i), e)))
LL
```

```
## [1] -20.96827445  -4.34652363  -0.92393853  -0.30000000   0.01005034
```

```
-sum( LL )
```

```
## [1] 26.52869
```

Now that we see an instance of one objective function value for the starting parameters, the model, the data, we can pull this together into a function for `optim()`.

```r
parameters <- unlist( parameters )
neg_loglik <- function(par, flist, data){
  # establish neg_loglik (numerator of conditional probability) environment
  e <- unlist( list( as.list(par), as.list(data) ), recursive=FALSE )
  LL <- sum( sapply( flist, function(i) sum(eval(parse(text=i), envir=e)) ) )
  return(-sum( LL ) )
}
parameters <- list( a=18, b=-0.13, sigma=0.27, tau=0.5 )
neg_loglik( par=parameters, flist=f_list_1, data=d )
```

```
## [1] 9.199223
```

```r
# just as before and in a format
# to which optim() will be receptive
```

Now we can optimize. But we have one more wrinkle: optimization is often very sensitive to parameter choices. Byrd et al. (1995) updated the BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm from the 1970's, among other things, to include lower and upper bounds, and thus the `method="L-BFGS-B"`. An upshot of this choice is that we can specify a value of `tau` and incorporate this value into an upper bound as an increment. Lower and upper bounds are further useful as they will focus the gradient search onto a more narrow parameter space. These bounds are also in the spirit of regularizing prior distributions of parameters.

```r
# declare starting values for parameters
parameters <- list( a=17, b=-0.13, sigma=0.27, tau=0.5 )
parameters <- unlist( parameters )
tau_L <- 0.5; tau_U <- 0.505
lower <- c(1,-1,0.27,tau_L)
upper <- c(20,1,0.271,tau_U)
fit <- optim(
  par=parameters,
  fn=neg_loglik,
  flist=f_list_1,
  data=d,
  hessian=TRUE, method="L-BFGS-B",
  lower=lower, upper=upper
  )
# Report results
fit$par
```

```
##          a          b      sigma         tau
## 17.4223536 -0.1078882  0.2700000  0.5000000
```

```
fit$value
```

```
## [1] 8.805948
```

```
fit$vcov <- solve( fit$hessian )
fit$stdev <- sqrt( abs(diag( fit$vcov)) )
fit$vcov
```

```
##                   a             b         sigma           tau
## a      3.210886e-04 -1.046481e-05  0.0007085620  2.561351e-04
## b     -1.046481e-05  1.011962e-05 -0.0004872505 -4.266492e-05
## sigma  7.085620e-04 -4.872505e-04 -0.1967831190 -7.142019e-02
## tau    2.561351e-04 -4.266492e-05 -0.0714201948  7.347534e-04
```

```
fit$stdev
```

```
##           a           b       sigma         tau
## 0.017918945 0.003181135 0.443602434 0.027106334
```

Playing around (what modeling ends up being) with the parameter starting values will reveal a strong sensitivity in this data to minute changes in σ as well.

Altogether now

In the spirit of tying this massive set of considerations and unity among all of us readers, here is the entire workflow in one place.

```
library(tidyverse)
#
# 0. Sample data
#
y <- c(12500, 13350, 14600, 15750, 17500)/1000 # price
x <- c( 43890, 37750, 27300, 15500, 8900 )/1000 # odometer miles
d <- tibble(
  price = y,
  odo = x
)
#
# 1. Formulate model
#
# 1A. Write vectorized dALD
dALD_0 <- function(y, mu = 0, sigma = 1, tau=0.5, log=FALSE) {
    if (length(y) == 0)
        stop("Please input y.")
    if (sum(y[is.na(y) == TRUE]) > 0)
        stop("There are some NA values in y.")
    if (sigma <= 0)
        stop("sigma must be a positive number.")
    if (tau >= 1 | tau <= 0)
        stop("Be sure that tau is a real number in (0,1), eg, 0.25.")
    if (abs(mu) == Inf)
        stop("mu must be a finite real number.")
    dALD <- ifelse(test = y < mu,
                   yes = (tau * (1 - tau)/sigma) * exp((1 - tau) * (y - mu)/sigma),
                   no = (tau * (1 - tau)/sigma) * exp(-(tau) * (y - mu)/sigma)
        )
    dALD <- ifelse(test = (log == TRUE),
                   yes = log( dALD ),
                   no = dALD
        )
    return(dALD)
}
# Vectorize dALD_0
dALD_vec <- Vectorize(dALD_0, vectorize.args = c("y", "mu", "sigma"))
# 1B. Build model
f_list <- list(
  price ~ dALD_vec(a+b*odo,sigma,tau),
  a ~ dnorm(0,10),
  b ~ dnorm(0,1),
  sigma ~ dexp(1)
)
#
# 2. Compile model into R
#
formula2text <- function( f_list ){
    RHS <- f_list[[3]]
    LHS <- f_list[[2]]
    n_args <- length( RHS )
    args_list <- as.list( RHS )
    args_list[[1]] <- LHS
    args_list[[n_args+1]] <- "log=TRUE"
    args_text <- paste( args_list, collapse=", "  )
    paste0( RHS[[1]], "( ", args_text, " )" )
}
f_list_0 <- as.list( f_list )
# Vectorize formula2text with l(list)apply
f_list_1 <- lapply( f_list_0, formula2text )
#
# 3. Evaluate objective function
#
```

```
neg_loglik <- function(par, flist, data){
  # use same arg names here as in optim
  # establish neg_loglik (numerator of conditional probability) environment
  e <- unlist( list( as.list(par), as.list(data) ), recursive=FALSE )
  LL <- sum( sapply( flist, function(i) sum(eval(parse(text=i), envir=e)) ) )
  return(-sum( LL ) )
}
# declare starting values for parameters
parameters <- list( a=17, b=-0.13, sigma=0.27, tau=0.5 )
#
#  4. Optimize
#
parameters <- unlist( parameters )
tau_L <- 0.495; tau_U <- 0.505
fit <- optim(
  par=parameters,
  fn=neg_loglik,
  flist=f_list_1,
  data=d,
  hessian=TRUE, method="L-BFGS-B",
  lower=c(1,-1,0.269,tau_L),
  upper=c(20,1,0.271,tau_U)
  )
#
# 5. Report results
#
fit$par
```

```
##           a           b       sigma         tau 
## 17.08757091 -0.09895663  0.26900000  0.49809346
```

```
fit$value
```

```
## [1] 9.8542
```

```
fit$vcov <- solve( fit$hessian )
fit$stdev <- sqrt( abs(diag( fit$vcov)) )
fit$vcov
```

```
##                  a             b         sigma          tau
## a     -0.078202771  2.087239e-03  0.0117263982 -0.004570872
## b      0.002087239 -3.904672e-05 -0.0001414178  0.000414822
## sigma  0.011726398 -1.414178e-04  0.0484420595  0.028080567
## tau   -0.004570872  4.148220e-04  0.0280805668  0.042149946
```

```
fit$stdev
```

```
##           a           b       sigma         tau
## 0.279647584 0.006248738 0.220095569 0.205304520
```

3. Comparisons

Here we attempt some crude comparisons. The `quantreg` and `bayesQR` packages report results that differ with our use of ALD and quadratic approximation. We will use $\tau = 0.50$ here.

```
#
# Goal programming model LP
#
theta
```

```
## [1] 18.771792 -0.142898
```

```
#
# ALD quadratic approximation
#
fit$par
```

```
##           a           b       sigma         tau
## 17.08757091 -0.09895663  0.26900000  0.49809346
```

```
#
# bayesQR MCMC ALD
#
library(bayesQR)
fit_bayesQR <- bayesQR(price ~ odo, data=d, quantile=c(0.5), alasso=TRUE, ndraw=500)
```

```
## Current iteration :
## [1] 500
```

```
summary( fit_bayesQR )
```

```
##
## Type of dependent variable: continuous
## Lasso variable selection: yes
## Normal approximation of posterior: yes
## Estimated quantile:  0.5
## Lower credible bound:  0.025
## Upper credible bound:  0.975
## Number of burnin draws:  0
## Number of retained draws:  500
##
##
## Summary of the estimated beta:
##
```

```
##               Bayes Estimate  lower  upper adj.lower adj.upper
## (Intercept)         16.3230  0.123 20.913   -11.119    43.765
## odo                 -0.0749 -0.240  0.337    -0.858     0.708
```

```
#
# The original quantreg
#
library(quantreg)
summary( rq( price ~ odo, data=d, tau=c(.5)))
```

```
##
## Call: rq(formula = price ~ odo, tau = c(0.5), data = d)
##
## tau: [1] 0.5
##
## Coefficients:
##               coefficients   lower bd       upper bd
## (Intercept)    1.805570e+01 -1.797693e+308  1.797693e+308
## odo           -1.265800e-01 -1.797693e+308  1.797693e+308
```

4. Stan's view of it all

Yet another approach rears its head. Similar to the ALD above, we can define for this regression this density and likelihood.

$$ALD(y \mid) = \frac{\tau(1-\tau)}{\sigma} \exp\left[-\left(\frac{\Sigma_{i=1}^{N}\rho_\tau(u)(y_i - X\beta(\tau))}{\sigma}\right)\right] \tag{10}$$

$$\rho_\tau(u) = \frac{|u| + (2\tau - 1)}{2} \tag{11}$$

First of all we now have a multiple quantile regression with X as an N row (observations) P column (features) matrix. Second, we implement an algebraic splitting of the y observations into τ and $1-\tau$ regions, instead of if-then logic. Third, we implement all of this in the Stan probabilistic programming language. We follow Sean Pinkney's implementation quite closely now.[5]

```
model <- "
functions{
real q_u(real q, vector u){
    if (fabs(u) > 1e-15)
      return sum(q*u);
    else
      return sum((1-q)*u);
```

[5] See Pinkney (n.d.) posts.

```
  // return sum(abs(u) + (2 * q - 1) * u);
}                   // similar to the goal programming
                    // loss objective of deviations from
                    // constraint hard goals
                    // but why 1/2?
real ald_lpdf(vector y, real q, real sigma, vector q_est){
  int N = num_elements(y);
  return N * (log(q) + log1m(q) - log(sigma)) - q_diff(q, y - q_est) / sigma;
}

}

data {
  int N;                   // Number of observation
  int P;                   // Number of predictors
  real<lower=0, upper=1> q;
  vector[N] y;             // Response variable sorted
  matrix[N, P] x;
}
parameters {
  vector[P] beta;
  real<lower=0> sigma;
}
model {
  beta ~ normal(0, 4);
  sigma ~ exponential(1);
  y ~ ald(q, sigma, x * beta);
}
"
```

```
library(Brq)
library(rstan)
library(tidyverse)
library(cmdstanr)
#
#Compile once and use for a long time!
ALD_model <- cmdstan_model( "ald-quantreg.stan")
#save( ALD_model, file="ALD_model")
#
load(file="ALD_model")

ALD_model_1 <- ALD_model$sample(
  data = list(N = nrow(d),
            P = 2,
            q = 0.5,
              y = d$price,
              x = as.matrix(data.frame(
                alpha = 1,
                x = d$odo))
              ),
  seed = 12123123,
  parallel_chains = 4,
  iter_warmup = 500,
  iter_sampling = 1000,
  refresh = 0,
  show_messages = FALSE
)
```

```
## Running MCMC with 4 parallel chains...
##
## Chain 3 finished in 0.8 seconds.
## Chain 1 finished in 0.9 seconds.
## Chain 4 finished in 1.0 seconds.
## Chain 2 finished in 1.5 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 1.0 seconds.
## Total execution time: 1.9 seconds.
```

Some results are in order.

```
#ImmunogG_model_1$summary()
ALD_model_1$summary()
```

```
## # A tibble: 4 x 10
##   variable     mean  median     sd    mad      q5     q95  rhat ess_bulk ess_t~1
##   <chr>       <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl> <dbl>    <dbl>   <dbl>
## 1 lp__     -18.7    -18.0   2.40   1.99   -23.5   -16.2    1.02     174.    381.
## 2 beta[1]   16.7     17.7   2.90   0.730    9.59   18.7    1.02     229.    234.
## 3 beta[2]   -0.0874  -0.116 0.0854 0.0210  -0.145   0.118  1.02     231.    244.
## 4 sigma      0.760    0.442 0.852  0.270    0.181   2.78   1.03     156.    307.
## # ... with abbreviated variable name 1: ess_tail
```

```
#ALD_model_1$output()
```

5. How far we have come!

Quite a run! The way we model, think about parameters, even choices about numerical methods like the simplex algorithm or the modified BFGS gradient optimization, all matter! There appears to be no one true answer either. We can hear Bruno De Finetti yelling in the background: PROBABILITY DOES NOT EXIST![6]

References

Benoit, Dries F., and Dirk Van den Poel. 2017. "bayesQR: A Bayesian Approach to Quantile Regression." *Journal of Statistical Software* 76 (7).

Byrd, Richard H., Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. "A Limited Memory Algorithm for Bound Constrained Optimization." *SIAM Journal on Scientific Computing* 16 (5): 1190–1208. https://doi.org/10.1137/0916069.

Edwin T. Jaynes, ed. G. Larry Bretthorst. 2004. *Probability Theory: Logic of Science*. Cambridge, U.K.: Cambridge University Press.

Finetti, B. de. 1974. *Theory of Probability*. Wiley.

Karmarkar, N. 1984. "A New Polynomial-Time Algorithm for Linear Programming." *Combinatorica* 4: 373–95.

Koenker, Roger. 2005. *Quantile Regression (Econometric Society Monographs)*. Cambridge, U.K.: Cambridge University Press.

Laplace, P. S. de, F. W. Truscott, and F. L. Emory. 1902. *A Philosophical Essay on Probabilities*. A Philosophical Essay on Probabilities. Wiley. https://books.google.com/books?id=WxoPAAAAIAAJ.

McElreath, Richard. 2020. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Second edition. CRC Press. https://xcelab.net/rm/statistical-rethinking/.

Nau, Robert F. 2001. "DE FINETTI WAS RIGHT: PROBABILITY DOES NOT EXIST." *Theory and Decision* 51: 89–124.

Pinkney, Sean. n.d. "Quantile Regressions in Stan: Part i (Retrieved 2023-03-29)." https://spinkney.github.io/posts/post-2-quantile-reg-part-I/quantile-reg.html.

Yu, K., and J. Zhang. n.d. "A Three-Parameter Asymmetric Laplace Distribution and Its Extension." *Communications in Statistics-Theory and Methods* 34 (9-10).

[6] "My thesis, paradoxically, and a little provocatively, but nonetheless genuinely, is simply this: PROBABILITY DOES NOT EXIST … The abandonment of superstitious beliefs about the existence of the Phlogiston, the Cosmic Ether, Absolute Space and Time, … or Fairies and Witches was an essential step along the road to scientific thinking. Probability, too, if regarded as something endowed with some kind of objective existence, is no less a misleading misconception, an illusory attempt to exteriorize or materialize our true probabilistic beliefs." Finetti (1974), p. x.