EM algorithm

Maximum Likelihood Estimation of the t-distribution

Lucas Støjko Andersen

University of Copenhagen

November 8, 2022

The t-distribution

The density of the non-standard t-distribution

The density of the t-distribution is given by

$$g(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi\nu\sigma^2}\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(x-\mu)^2}{\nu\sigma^2}\right)^{-\frac{\nu+1}{2}} \tag{1}$$

with parameters $\mu \in \mathbf{R}$, $\nu > 0$, $\sigma^2 > 0$. Here, Γ is the gamma function.

For independent observations $X_1, X_2, \ldots, X_n$ with density as in (1) there exist no nice analytic solutions to the MLE — even with ν known.

There is another way

A joint approach with latent variables

Consider (X, W) with density

$$f(x, w) = \frac{1}{\sqrt{\pi\nu\sigma^2}2^{(\nu+1)/2}\Gamma(\nu/2)} w^{\frac{\nu-1}{2}} e^{-\frac{w}{2}\left(1+\frac{(x-\mu)^2}{\nu\sigma^2}\right)} \tag{2}$$

Notice that $W \mid X = x \sim \Gamma(\alpha, \beta)$ — the gamma distribution — with parameters

$$\alpha = \frac{\nu+1}{2} \qquad \beta = \frac{1}{2}\left(1 + \frac{(x-\mu)^2}{\nu\sigma^2}\right). \tag{3}$$

This can be used to show that X indeed has the marginal density of the t-distribution with parameters μ, σ^2, ν.

The Full Maximum Likelihood Estimate

MLE with fixed ν

Assume $\nu > 0$ known. I.i.d. $(X_1, W_1), (X_2, W_2), \ldots, (X_n, W_n)$ have log-likelihood

$$\ell(\theta) \simeq -\frac{n}{2} \log \sigma^2 - \frac{1}{2\nu\sigma^2} \sum_{i=1}^{n} W_i (X_i - \mu)^2. \tag{4}$$

The solution is that of the weighted least squares:

$$\hat{\mu} = \frac{\sum_{i=1}^{n} W_i X_i}{\sum_{i=1}^{n} W_i}, \qquad \hat{\sigma}^2 = \frac{1}{n\nu} \sum_{i=1}^{n} W_i (X_i - \hat{\mu})^2. \tag{5}$$

EM algorithm

MLE of the marginal likelihood for fixed ν

Only X is observed and so the full MLE cannot be computed. Using the EM algorithm we iteratively maximize the quantity

$$Q(\theta \mid \theta') = E_{\theta'}(\log f_\theta(X, W) \mid X) \tag{6}$$

where $\theta = (\mu, \sigma^2)$. Using the log-likelihood from earlier

$$Q(\theta \mid \theta') \simeq -\frac{n}{2} \log \sigma^2 - \frac{1}{2\nu\sigma^2} \sum_{i=1}^{n} E_{\theta'}(W_i \mid X_i)(X_i - \mu)^2. \tag{7}$$

The maximizer is then

$$\hat{\mu}_{\theta'} = \frac{\sum_{i=1}^n E_{\theta'}(W_i \mid X_i) X_i}{\sum_{i=1}^n E_{\theta'}(W_i \mid X_i)}, \quad \hat{\sigma}^2_{\theta'} = \frac{1}{n\nu} \sum_{i=1}^{n} E_{\theta'}(W_i \mid X_i)(X_i - \hat{\mu}_{\theta'})^2.$$

EM algorithm

E-step and M-step

The E-step boils down to computing $E_{\theta'}(W_i \mid X_i)$

$$E_{\theta'}(W_i \mid X_i) = \frac{\alpha}{\beta} = \frac{\nu' + 1}{2} \frac{1}{\frac{1}{2}\left(1 + \frac{(X_i - \mu')^2}{\nu' \sigma'^2}\right)} = \frac{\nu' + 1}{1 + \frac{(X_i - \mu')^2}{\nu' \sigma'^2}} \tag{8}$$

and doing the M-step by computing

$$\hat{\mu}_{\theta'} = \frac{\sum_{i=1}^{n} E_{\theta'}(W_i \mid X_i) X_i}{\sum_{i=1}^{n} E_{\theta'}(W_i \mid X_i)}, \quad \hat{\sigma}^2_{\theta'} = \frac{1}{n\nu} \sum_{i=1}^{n} E_{\theta'}(W_i \mid X_i)(X_i - \hat{\mu}_{\theta'})^2.$$

Implementation of EM-algorithm

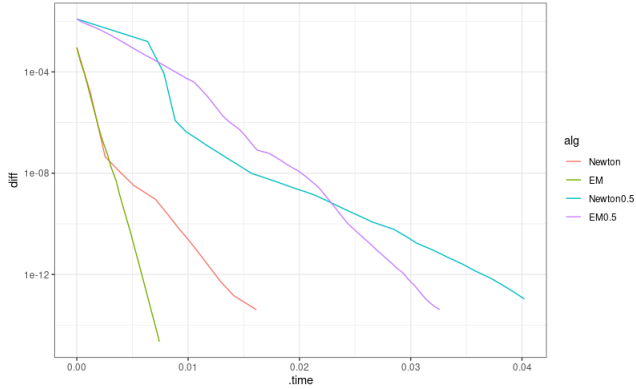
```
EM <- function(x, nu, cb = NULL, maxit = 500L, min.eps = 1e-7, par = NULL) {
  if(is.null(par)) par <- c(median(x), IQR(x))
  par1 <- numeric(2)
  n <- length(x)
  EW <- numeric(n)
  for(i in 1:maxit) {
    EW <- (nu + 1) / (1 + ((x - par[1])^2) / (nu * par[2]))
    par1[1] <- sum(EW * x) / sum(EW)
    par1[2] <- sum(EW * (x - par[1])^2) / (n * nu)
    if(!is.null(cb)) cb()
    if(norm(par - par1, "2") < min.eps * (norm(par1, "2") + min.eps)) break
    par <- par1
  }
  if(i == maxit) warning("Maximum number of itertaions ", maxit, " reached.")
  names(par1) <- c("mu", "sigma")
  list(par = par1, iterations = i, nu = nu)
}
```

Implementation of the Newton Algorithm

```
Newton <- function(par, H, gr, hess,
                   d = 0.8, c = 0.2, gamma0 = 1,
                   min.eps = 1e-7, maxit = 500L, cb = NULL) {
  for(i in 1:maxit) {
    value <- H(par)
    grad <- gr(par)
    Hessian <- hess(par)
    rho <- - drop(solve(Hessian, grad))
    gamma <- gamma0
    par1 <- par + gamma * rho
    h_prime <- crossprod(grad, rho)
    while(min(H(par1), Inf, na.rm = TRUE) > value +  c * gamma * h_prime) {
      gamma <- d * gamma
      par1 <- par + gamma * rho
    }
    if(!is.null(cb)) cb()
    if(norm(par - par1, "2") < min.eps * (norm(par1, "2") + min.eps)) break
    par <- par1
  }
  if(i == maxit) warning("Maximal number, ", maxit, ", of iterations reached")
  list(par = par1, iterations = i)
}
```

Comparison

Simulate 10.000 samples from a t-distribution with parameters $\mu = 5$, $\sigma^2 = 3$ and $\nu = 2.5$ and 10.000 samples with $\nu = 0.5$. IQR for $\nu = 2.5$ is 2.685394 and 5.211938 for $\nu = 0.5$. Median is 4.997973 and 4.98833 respectively.



Comparison

Benchmarks Using bench Package

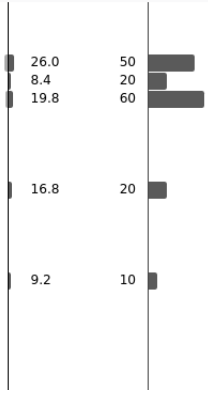
$(4.999499, 2.937087)$ for $\nu = 2.5$ and $(4.996215, 2.913278)$ for $\nu = 0.5$ estimated by the EM-algorithm. $(4.999498, 2.937088)$ for $\nu = 2.5$ and $(4.996214, 2.913275)$ for $\nu = 0.5$ estimated by the Newton algorithm.

expression	min	median	`itr/sec`	mem_alloc
<bch:expr>	*<bch:tm>*	*<bch:tm>*	*<dbl>*	*<bch:byt>*
EM	5.81ms	6.41ms	151.	5.41MB
NEWTON	11.82ms	12.58ms	74.6	12.19MB
EM0.5	20.94ms	21.41ms	45.8	19.91MB
NEWTON0.5	27.92ms	28.57ms	34.8	26.25MB

Profiling of the Newton Algorithm Implementation

Using 100.000 samples.

```
Newton <- function(par, H, gr, hess, d = 0.8, c = 0.2, gamma0 = 1, min.eps =
1e-7, maxit = 500, cb = NULL) {
  for(i in 1:maxit) {
    value <- H(par)                                                    -32.4   26.0   50
    grad <- gr(par)                                                            8.4   20
    Hessian <- hess(par)                                               -19.8   19.8   60
    rho <- - drop(solve(Hessian, grad))
    gamma <- gamma0
    par1 <- par + gamma * rho
    h_prime <- crossprod(grad, rho)
    while(min(H(par1), Inf, na.rm = TRUE) > value +  c * gamma * h_prime) {          16.8   20
      gamma <- d * gamma
      par1 <- par + gamma * rho
    }
    if(!is.null(cb)) cb()
    if(norm(par - par1, "2") < min.eps * (norm(par1, "2") + min.eps)) break          9.2    10
    par <- par1
  }
  if(i == maxit) warning("Maximal number, ", maxit, ", of iterations reached")
  list(par = par1, iterations = i)
}
```



Profiling of the Newton Algorithm

Implementation of Log-likelihood and Gradient

```
get_like <- function(x, nu) {
  n <- length(x); force(nu)
  function(par) {
    mu <- par[1]; sigma <- par[2]
    K <- sum(log(1 + (x - mu)^2 / (nu * sigma)))
    log(sigma) / 2 + (nu + 1) * K / (2 * n)
  }
}

get_grad <- function(x, nu) {
  n <- length(x); force(nu)
  function(par) {
    mu <- par[1]; sigma <- par[2]
    C1 <- (x - mu) / (1 + (x - mu)^2 / (nu * sigma))
    K_mu <- sum(C1)
    K_sigma <- sum(C1 * (x - mu))
    grad_mu <- -(nu + 1) * K_mu / (n * nu * sigma)
    grad_sigma <- 1 / (2 * sigma) - (nu + 1) * K_sigma / (2 * n * nu * sigma^2)
    c(grad_mu, grad_sigma)
  }
}
```

Implementation of Log-likelihood, Gradient and Hessian Using Rcpp

```
// [[Rcpp::export]]
double CPP_likelihood(NumericVector par, NumericVector x, double nu) {
  double K = sum(log(1 + (x - par[0]) * (x - par[0]) / (nu * par[1])));
  return log(par[1]) / 2 + (nu + 1) * K / (2 * x.size());
}

// [[Rcpp::export]]
NumericVector CPP_gradient(NumericVector par, NumericVector x, double nu) {
  int n = x.size();
  double K_mu = 0, K_sigma = 0;
  for(int i = 0; i < n; ++i) {
    double C1 = (x[i] - par[0]) /
      (1 + (x[i] - par[0]) * (x[i] - par[0]) / (nu * par[1]));
    K_mu += C1;
    K_sigma += C1 * (x[i] - par[0]);
  }
  NumericVector grad(2);
  grad[0] = -(nu + 1) * K_mu / (n * nu * par[1]);
  grad[1] = 1 / (2 * par[1]) -
    (nu + 1) * K_sigma / (2 * n * nu * par[1] * par[1]);
  return grad;
}
```

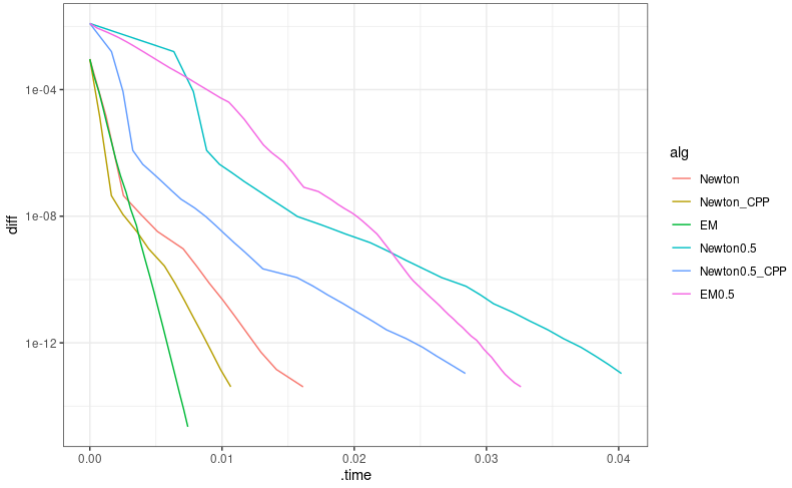
Implementation of Log-likelihood, Gradient and Hessian Using Rcpp

```
// [[Rcpp::export]]
NumericMatrix CPP_hessian(NumericVector par, NumericVector x, double nu) {
  int n = x.size();
  double K1 = 0, K2 = 0, K3 = 0, K4 = 0, K5 = 0, K6 = 0;
  for(int i = 0; i < n; ++i) {
    double C0 = 1 / (1 + (x[i] - par[0]) * (x[i] - par[0]) / (nu * par[1]));
    double C1 = C0 * (x[i] - par[0]);
    double C2 = C1 * (x[i] - par[0]);
    K1 += C0;
    K2 += C1;
    K3 += C1 * C1;
    K4 += C2;
    K5 += C2 * C2;
    K6 += C1 * C1 * (x[i] - par[0]);
  }
  NumericMatrix hess(2, 2);
  hess(0, 0) = (nu + 1) * K1 / (n * nu * par[1]) +
    2 * (nu + 1) * K3 / (n * nu * nu * par[1] * par[1]);
  hess(0, 1) = (nu + 1) * K2 / (n * nu * par[1] * par[1]) -
    (nu + 1) * K6 / (n * nu * par[1] * par[1] * par[1]);
  hess(1, 0) = hess(0, 1);
  hess(1, 1) = -1 / (2 * par[1] * par[1]) +
    (nu + 1) * K4 / (n * nu * par[1] * par[1] * par[1]) -
    (nu + 1) * K5 / (2 * n * nu * nu * par[1] * par[1] * par[1] * par[1]);
  return hess;
}
```

Benchmarks of New Log-likelihood, Gradient and Hessian

expression	min	median	`itr/sec`	mem_alloc
<bch:expr>	*<bch:tm>*	*<bch:tm>*	*<dbl>*	*<bch:byt>*
LIKE	152µs	155.9µs	5904.	78.17KB
LIKE_CPP	126µs	129µs	7153.	2.49KB
GRAD	157µs	164.1µs	5727.	234.52KB
GRAD_CPP	94µs	94.6µs	10274.	2.49KB
HESS	250µs	260.3µs	3637.	547.2KB
HESS_CPP	104µs	105.1µs	9157.	2.49KB

Benchmarks of New Log-likelihood, Gradient and Hessian



Benchmarks of New Log-likelihood, Gradient and Hessian

expression	min	median	`itr/sec`	mem_alloc
<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:byt>
EM	6.09ms	6.54ms	148.	5.41MB
NEWTON	11.85ms	13.37ms	73.9	12.19MB
NEWTON_CPP	8.2ms	8.64ms	114.	418.3KB
EM0.5	20.84ms	21.44ms	46.4	19.91MB
NEWTON0.5	28.07ms	28.68ms	34.6	26.25MB
NEWTON0.5_CPP	17.03ms	17.3ms	57.3	595.14KB

Rcpp Implementation of EM-algorithm

```
double dist(double x1, double x2, double y1, double y2) {
  double x_sq = x1 * x1 + x2 * x2;
  double y_sq = y1 * y1 + y2 * y2;
  double xy = x1 * y1 + x2 * y2;
  return std::sqrt(x_sq + y_sq - 2 * xy);
}

// [[Rcpp::export]]
List CPP_EM(NumericVector par0,
                    NumericVector x,
                    double nu,
                    int maxit,
                    double eps) {
  int n = x.size();
  NumericVector par = clone(par0);
  NumericVector par1(2);
  int i;
  for(i = 0; i < maxit; ++i) {
    NumericVector EW = (nu + 1) /
      (1 + (x - par[0]) * (x - par[0]) / (nu * par[1]));
    par1[0] = sum(EW * x) / sum(EW);
    par1[1] = sum(EW * (x - par[0]) * (x - par[0])) / (n * nu);
    double norm_new = dist(par[0], par[1], par1[0], par1[1]);
    double norm_old = std::sqrt(par1[0] * par1[0] + par1[1] * par1[1]);
    if(norm_new < eps * (norm_old + eps)) break;
    par[0] = par1[0];
    par[1] = par1[1];
  }
  if(i != maxit) i += 1;
  return List::create(par1, i);
}
```

Implementation of Newton Algorithm Using R's C Interface

```
SEXP C_Newton(SEXP par0, SEXP H, SEXP gr, SEXP hess,
              SEXP d, SEXP c, SEXP gamma0,
              SEXP eps, SEXP maxit,
              SEXP env) {
  int n = length(par0), info;
  SEXP H_call = PROTECT(lang2(H, R_NilValue));
  SEXP gr_call = PROTECT(lang2(gr, R_NilValue));
  SEXP hess_call = PROTECT(lang2(hess, R_NilValue));
  SEXP value, grad, hessian, Hpar;
  SEXP par = PROTECT(allocVector(REALSXP, n));
  SEXP par1 = PROTECT(allocVector(REALSXP, n));
  double value_, *grad_, *hessian_;
  double *par1_ = REAL(par1), *par_ = REAL(par);
  double gamma0_ = REAL(gamma0)[0], d_ = REAL(d)[0], c_ = REAL(c)[0];
  double eps_ = REAL(eps)[0], maxit_ = INTEGER(maxit)[0];
  double *rho = (double*)malloc(sizeof(double) * n);

  for(int i = 0; i < n; ++i)
    REAL(par)[i] = REAL(par0)[i];
  int k;

  for(k = 0; k < maxit_; ++k) {
    double gamma = gamma0_;
    double h_prime;
    SETCADR(H_call, par);
    value = PROTECT(eval(H_call, env));
    SETCADR(gr_call, par);
    grad = PROTECT(eval(gr_call, env));
    SETCADR(hess_call, par);
    hessian = PROTECT(eval(hess_call, env));
    hessian_ = REAL(hessian);
    grad_ = REAL(grad);
    for(int j = 0; j < n; ++j)
      grad_[j] = -grad_[j];
    value_ = REAL(value)[0];
    solve(n, hessian_, grad_, rho, &info);
    if(info != 0) break;
    for(int j = 0; j < n; ++j) {
      par1_[j] = par_[j] + gamma * rho[j];
      h_prime += grad_[j] * rho [j];
    }
    while(1 == 1) {
      SETCADR(H_call, par1);
      Hpar = eval(H_call, env);
      if(ISNAN(REAL(Hpar)[0]) ||
          REAL(Hpar)[0] > value_ + c_ * gamma * h_prime) {
        gamma = d_ * gamma;
        for(int j = 0; j < n; ++j)
          par1_[j] = par_[j] + gamma * rho[j];
      } else break;
    }
    UNPROTECT(3);
    double norm = norm2(par1_, n);
    for(int j = 0; j < n; ++j)
      par_[j] -= par1_[j];
    double norm_new = norm2(par_, n);
    if(norm_new < eps_ * (norm + eps_)) break;
    for(int j = 0; j < n; ++j) {
      par_[j] = par1_[j];
    }
  }
```

Implementation of Newton Algorithm Using R's C Interface

Continued

```
  SEXP result = PROTECT(allocVector(VECSXP, 2));
  SET_VECTOR_ELT(result, 0, par1);
  if(k != maxit_) k += 1;
  if(info != 0) {
    SET_VECTOR_ELT(result, 1, ScalarInteger(-1));
  } else {
    SET_VECTOR_ELT(result, 1, ScalarInteger(k));
  }
  UNPROTECT(6);
  free(rho);
  return result;
}
```

Implementation of Newton Algorithm Using R's C Interface

```
double norm2(double *x, int m) {
  char jobu = 'N', jobvt = 'N';
  int n = 1, info, lwork;
  if(m > 3) {
    lwork = m + 2;
  } else {
    lwork = 5;
  }
  double U, VT, s;
  double *A = (double*)malloc(sizeof(double) * m);
  double *work = (double*)malloc(sizeof(double) * lwork);
  for(int i = 0; i < m; ++i)
    A[i] = x[i];
  dgesvd_(&jobu, &jobvt, &m, &n, A, &m, &s,
          &U, &m, &VT, &m, work, &lwork, &info);
  free(A);
  free(work);
  return s;
}

void solve(int n, double *a, double *b, double *result, int *info) {
  for(int i = 0; i < n; ++i)
    result[i] = b[i];
  int nrhs = 1;
  int *ipiv = (int*)malloc(sizeof(int) * n);
  dgesv_(&n, &nrhs, a, &n, ipiv, result, &n, info);
  free(ipiv);
}
```

Wrappers

```
Newton_C <- function(par, H, gr, hess,
                     d = 0.8, c = 0.2, gamma0 = 1,
                     min.eps = 1e-7, maxit = 500) {
  result <- .Call("C_Newton",
                  par,
                  H,
                  gr,
                  hess,
                  d,
                  c,
                  gamma0,
                  min.eps,
                  as.integer(maxit),
                  environment())
  if(result[[2]] == -1L)
    warning("Matrix solve went wrong.")
  if(result[[2]] == maxit)
    warning("Maximal number, ", maxit, ", of iterations reached")
  list(par = result[[1]], iterations = result[[2]])
}
```

Wrappers

```
EM_cpp <- function(par = NULL, x, nu, cb = NULL, maxit = 500, min.eps = 1e-7) {
  if(is.null(par)) par <- c(median(x), IQR(x))
  par1 <- numeric(2)
  n <- length(x)
  EW <- numeric(n)
  result <- CPP_EM(par, x, nu, maxit, min.eps)
  if(result[[2]] == maxit) warning("Maximum number of itertaions ", maxit, " reached.")
  names(result[[1]]) <- c("mu", "sigma")
  list(par = c(result[[1]]),
       iterations = result[[2]],
       nu = nu)
}
```

Final Benchmarks

	expression	min	median	itr/s…[1]	mem_al…[2]
	<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:by>
1	EM	5.82ms	6.36ms	148.	5.41MB
2	EM_CPP	3.75ms	3.87ms	255.	2.1MB
3	NEWTON	11.85ms	12.49ms	75.6	12.19MB
4	NEWTON_CPP	8.19ms	8.6ms	112.	418.3KB
5	NEWTON_C	10.8ms	11.21ms	88.7	12.18MB
6	NEWTON_C_CPP	7.13ms	7.38ms	134.	403.27KB
7	EM0.5	21.02ms	21.46ms	46.0	19.91MB
8	EM0.5_CPP	12.92ms	13.1ms	76.1	6.84MB
9	NEWTON0.5	28.01ms	28.53ms	34.5	26.25MB
10	NEWTON0.5_CPP	16.98ms	17.29ms	57.4	595.14KB
11	NEWTON0.5_C	25.08ms	25.61ms	38.8	26.15MB
12	NEWTON0.5_C_CPP	14.3ms	14.74ms	67.8	560.27KB

Appendix

Final Implementation

```
EM_cpp <- function(par = NULL, x, nu, cb = NULL, maxit = 500, min.eps = 1e-7) {
  if(is.null(par)) par <- c(median(x), IQR(x))
  par1 <- numeric(2)
  n <- length(x)
  EW <- numeric(n)
  result <- CPP_EM(par, x, nu, maxit, min.eps)
  if(result[[2]] == maxit) warning("Maximum number of itertaions ", maxit, " reached.")
  names(result[[1]]) <- c("mu", "sigma_sq")
  structure(
    list(par = c(result[[1]]),
         iterations = result[[2]],
         nu = nu,
         x = x),
    class = "em_estimate"
    )
}
```

Appendix

Final Implementation

```
EM_fisher <- function(mle, x, nu) {
  Q <- Q_func(x, nu, mle)
  phi <- phi_func(x, nu)
  IY <- numDeriv::hessian(Q, mle)
  IX <- (diag(1, 2) - t(numDeriv::jacobian(phi, mle))) %*% IY
  solve(IX)
}

Q_func <- function(x, nu, mle) {
  force(x); force(nu); force(par);
  function(par) Q_cpp(par, mle, x, nu)
}

phi_func <- function(x, nu) {
  force(x); force(nu)
  function(par) phi_cpp(par, x, nu)
}

confint.em_estimate <- function(x, level = 0.95) {
  qq <- level + (1 - level) / 2
  qq <- qnorm(qq)
  invf <- EM_fisher(x$par, x$x, x$nu)
  mu <- x$par[1] + c(-1, 1) * qq * sqrt(invf[1, 1])
  sigma <- x$par[2] + c(-1, 1) * qq * sqrt(invf[2, 2])
  names(mu) <- c("lwr", "upr")
  names(sigma) <- c("lwr", "upr")
  list(mu = mu, sigma_sq = sigma)
}
```

Appendix

Final Implementation

```
// [[Rcpp::export]]
NumericVector phi_cpp(NumericVector par0,
                      NumericVector x,
                      double nu)  {
  int n = x.size();
  NumericVector par(2);
  NumericVector EW = (nu + 1) /
    (1 + (x - par0[0]) * (x - par0[0]) / (nu * par0[1]));

  par[0] = sum(EW * x) / sum(EW);
  par[1] = sum(EW * (x - par[0]) * (x - par[0])) / (n * nu);

  return par;
}

// [[Rcpp::export]]
double Q_cpp(NumericVector par,
             NumericVector par1,
             NumericVector x,
             double nu) {

  int n = x.size();
  double value = 0.5 * log(par[1]) * n;
  NumericVector C2 = (nu + 1) /
    (1 + (x - par1[0]) * (x - par1[0]) / (nu * par1[1]));
  return value + sum(C2 * (x - par[0]) * (x - par[0]) / (2 * nu * par[1]));
}
```

Appendix

Full RcppArmadillo Implementation of Newton Algorithm

```
// [[Rcpp::export]]
List FitT(NumericVector par0, NumericVector x, double nu, double c,
          double d, double gamma0, int maxit, double eps) {

  arma::mat::fixed<2, 2> hess;
  arma::vec::fixed<2> grad;
  arma::vec::fixed<2> p1;
  arma::vec::fixed<2> p2;
  NumericVector par1(2), par = clone(par0);
  int n = x.size();
  int i;
  for(i = 0; i < maxit; ++i) {
    double value = CPP_likelihood(par, x, nu);
    NumericVector gr = CPP_gradient(par, x, nu);
    NumericMatrix hessian = CPP_hessian(par, x, nu);
    grad[0] = gr[0], grad[1] = gr[1];
    hess(0, 0) = hessian(0, 0), hess(0, 1) = hessian(0, 1);
    hess(1, 0) = hessian(1, 0), hess(1, 1) = hessian(1, 1);
    arma::vec::fixed<2> rho = -arma::solve(hess, grad);
    double gamma = gamma0;
    par1[0] = par[0] + gamma * rho[0];
    par1[1] = par[0] + gamma * rho[1];
    double h_prime = rho[0] * gr[0] + rho[1] * gr[1];
    while(CPP_likelihood(par1, x, nu) > value + c * gamma * h_prime) {
      gamma *= d;
      par1[0] = par[0] + gamma * rho[0];
      par1[1] = par[1] + gamma * rho[1];
    }
    p1[0] = par1[0], p1[1] = par1[1];
    p2[0] = par1[0] - par[0], p2[0] = par1[1] - par[1];
    double norm = arma::norm(p1, 2);
    double norm_new = arma::norm(p2, 2);
    if(norm_new < eps * (norm + eps)) break;
    par[0] = par1[0], par[1] = par1[1];
  }
  if(i != maxit) i -= 1;
  NumericVector iter(1);
  iter[0] = i;
  return List::create(par1, iter);
}
```