

Stan超初心者講習

清水裕士

自己紹介

- 清水裕士
 - 関西学院大学社会学部
- 専門
 - 社会心理学, グループ・ダイナミックス
- 趣味
 - 心理統計学
 - 統計ソフトウェアの開発
- Web
 - Twitter: @simizu706
 - Webサイト: <http://norimune.net>



本発表のねらい

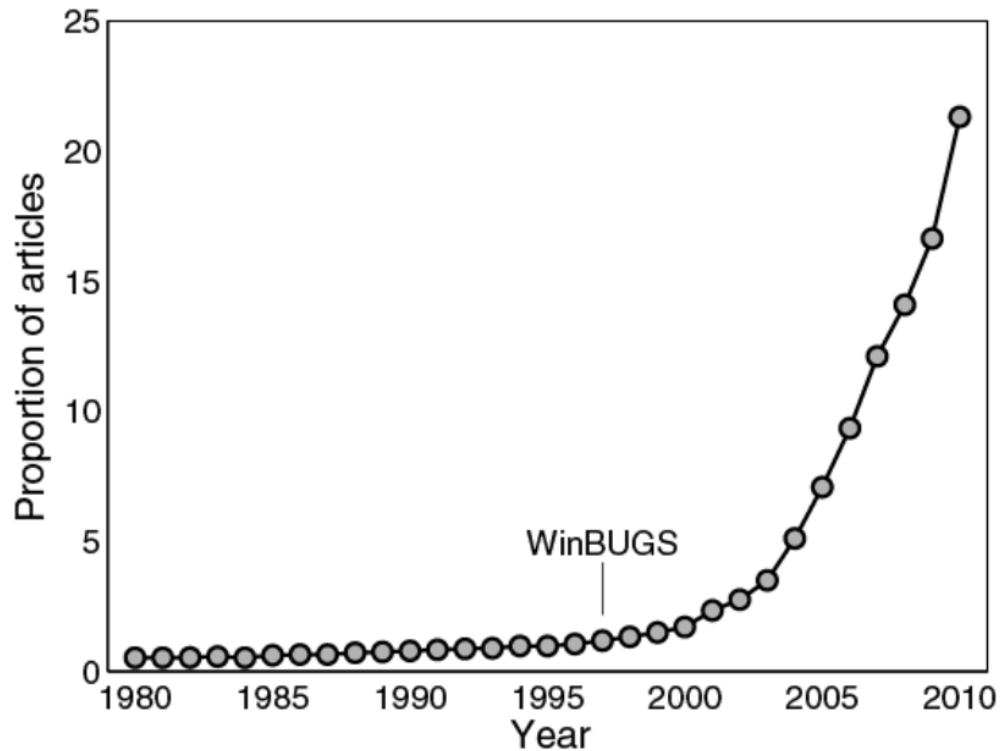
- 超初級Stan入門
 - Stanの基礎中の基礎的なモデルを実行
 - 回帰分析まで行けたらいいな, ぐらい
- プログラミングを知らない人向け
 - なので, プログラミングを知ってる人にとっては遠回りに見えたりするかも

(超)初級編のメニュー

- 二項分布
- 正規分布
- 二群の平均値の差の推論
- 相関係数
- 回帰分析
- 重回帰分析
- ロジスティック回帰分析

ベイズ統計モデリング

- ベイズ推定を用いた論文は急増中



Lee & Wagenmakers, 2014)

今日の主役は

- 今、もっとも開発が熱いMCMC用フリーソフト



Stan

Stanとは

- 統計モデルを書くだけでベイズ推定ができる
 - 統計モデル・・・尤度関数と事前分布
 - 推定したいもの・・・確率分布のパラメータ
- 尤度関数
 - 確率分布のパラメータとデータの関係性を表す
- 事前分布
 - パラメータの事前知識

同様のソフトはいろいろある

- BUGS
 - 初代MCMC用ソフト
- JAGS
 - BUGSに似た文法のソフト
- 僕はStanしか知らないから, Stanを使う(おい

Stanの特徴

- MCMCでベイズ推定
 - ハミルトニアン・モンテカルロ法
 - それを応用したNo-U-Turn Sampler(NUTS)
- 変分ベイズでベイズ推定
 - 自動微分変分推論(ADVI)
 - あくまで近似なのでMCMCより不正確
 - ただし, MCMCより圧倒的に速い

統計パッケージとStanの違い

- SPSSやRの関数
 - 特定の分布, 特定のモデルのパラメータのみを推定
 - データの入力形式がわかれば簡単に推定できる
- StanやBUGSなどのソフトウェア
 - 分布やモデルなどを全部指定する必要がある
 - 幅広い分布, モデル, 細やかな設定

Stanで統計モデリング

Rでstanを使うための準備

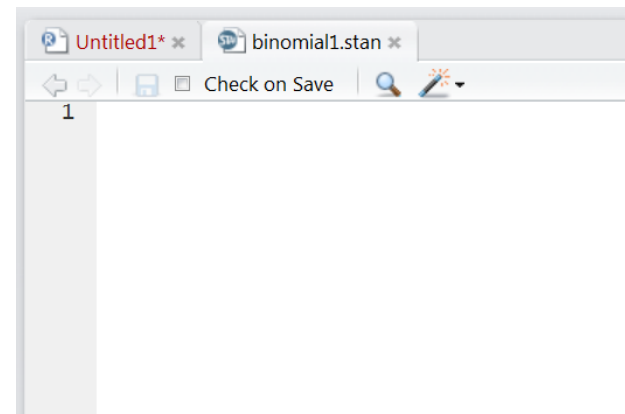
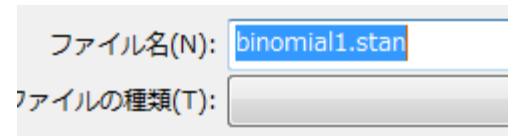
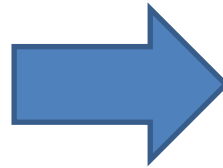
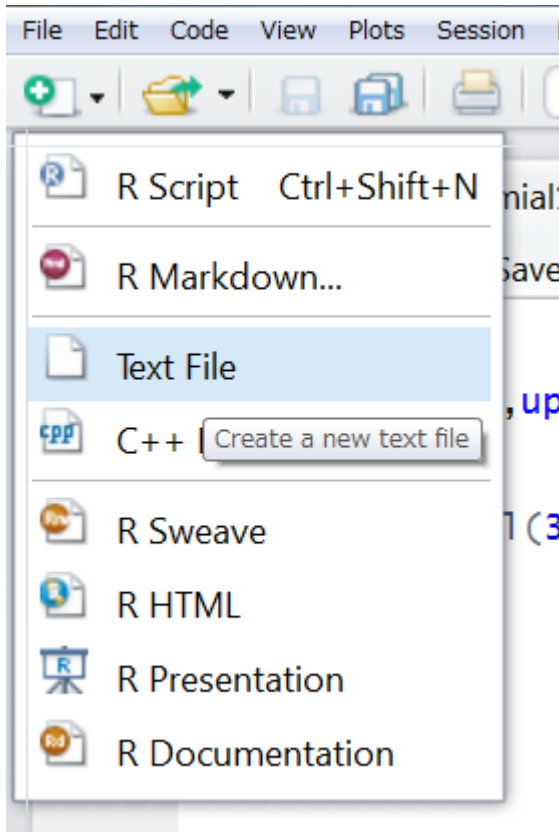
- Rをインストール
 - 必須
- Rstudioをインストール
 - 必須ではないが必須だと思ってもらって構わない
- rstanパッケージをインストール
 - OSによって必要なツールが違うのでリンク先を見て環境に合わせて準備してね
 - <https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started-%28Japanese%29>

Stanを使う流れ

- Stanコードを書く
 - 幅広いモデルを記述できる
- C++でコンパイル
 - 最初にコンパイルしたら, それ以降は不要
- MCMCでサンプリング
 - rstanからStanを実行してMCMC推定
- 結果を出力
 - R上で推定値やグラフを出力

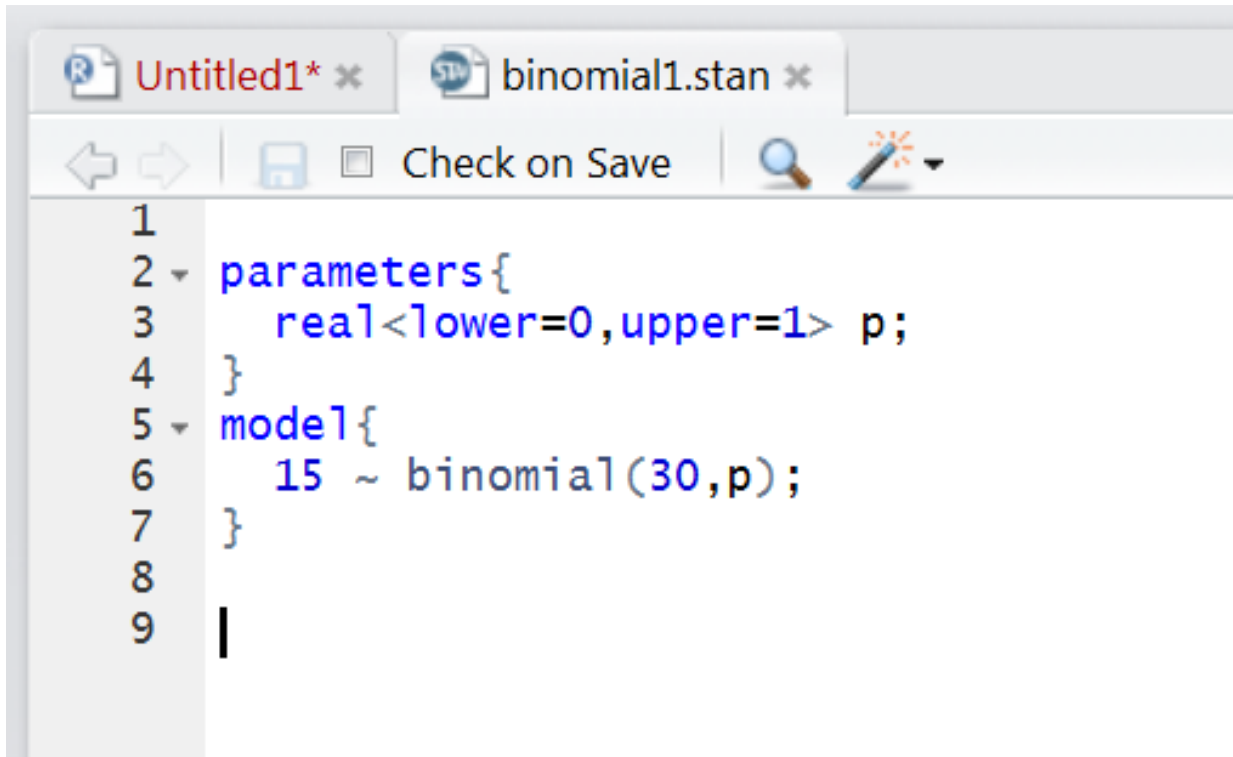
Rstudioを使うと捗る

- Textファイルで新規作成→.stanで保存



Rstudioを使うと捗る

- stanの予約語がハイライトされる



The screenshot shows the RStudio interface with a file named 'binomial1.stan' open. The code is written in the editor and features syntax highlighting for Stan keywords. The keywords 'parameters', 'model', 'real', and '~' are highlighted in blue. The code defines a parameter 'p' as a real number between 0 and 1, and a model where a variable '15' follows a binomial distribution with 30 trials and probability 'p'.

```
1
2 parameters{
3   real<lower=0,upper=1> p;
4 }
5 model{
6   15 ~ binomial(30,p);
7 }
8
9 |
```

Stanでのプログラミング

- Rによく似た言語
 - Rユーザーは比較的扱いやすい
- ブロックごとに記述
 - データを宣言するブロック, パラメータを宣言するブロック, モデルを書くブロック・・・など
 - ブロックの順番は固定
 - これはレベルが上がるごとに説明を追加していく

Stanのブロックの種類

- `function{}`
- `data{}`
- `transformed data{}`
- `parameters{}`
- `transformed parameters{}`
- `model{}`
- `generated quantities{}`

`data`, `parameters`, `model`の3つはほぼ必ず使うと考えていい

他のブロックも使うと便利だが、いつも使うわけではない

このスライドでは、レベルが上がるごとに使うブロックが増えていく(順番に説明します)

二項分布のモデリング

Stanコードの書き方 レベル1

- `parameters{}`
 - 推定するパラメータを宣言
 - 変数の型
 - `real`, `int`, `vector`, `matrix`, `simplex`, `cov_matrix`...
 - 下限と上限を指定可能
 - `lower=` で下限 `upper=` で上限
- `model{}`
 - 確率モデルを指定
 - たくさんの確率分布を使用可能
 - `normal`, `binomial`, `poisson`, `gamma`, `beta`, `lognormal`...

parameters

- 成功率 p を推定したい
 - p は実数なのでrealと宣言する
 - p は0～1の範囲をとるパラメータなので、範囲の指定もしたい

```
real<lower=0,upper=1> p;
```

- 文末には「必ず」セミコロン”;”を書くこと

model

- 確率モデルとして二項分布を使う
 - `binomial()`を使う
 - 30回の試行で成功率 p の二項分布
 - 15回成功した, というデータが得られた

$15 \sim \text{binomial}(30, p);$

- チルダ“ \sim ”はその分布に従うということ
- データが1つで, パラメータが1つなので推定可能

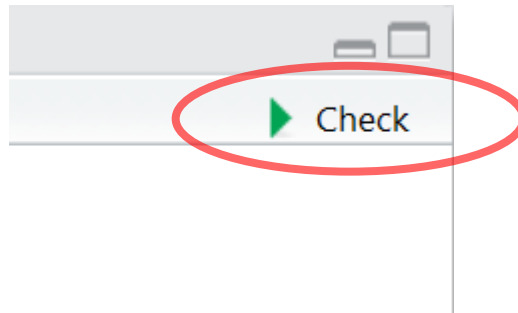
超短いstanコード

binomial1.stan

```
parameters{  
  real<lower=0,upper=1> p;  
}  
model{  
  15 ~ binomial(30,p);  
}
```

Rstudioを使うと捗る

- 文法チェック＋上書き保存をしてくれる



```
> rstan::rstudio_stanc("binomial1.stan")  
binomial1.stan is syntactically correct.
```

最後の行は何も書かない

```
2 parameters{  
3   real<lower=0,upper=1> p;  
4 }  
5 model{  
6   15 ~ binomial(30,p);  
7 }
```

7行目は}が入っていて、それが最終行になっている
すると...

```
> rstan::rstudio_stanc("binomial1.stan")  
binomial1.stan is syntactically correct.  
Warning message:  
In readLines(file) : incomplete final line found on 'binomial1.stan'
```

警告が出る
この場合は、7行目でもう一度改行するとよい

Rからstanを走らせる

- Rで以下のコードを書く

```
fit1 <- stan("binomial1.stan")
```

```
SAMPLING FOR MODEL 'binomial1' NOW (CHAIN 1).
```

```
Chain 1, Iteration:    1 / 2000 [  0%] (warmup)
Chain 1, Iteration:   200 / 2000 [ 10%] (warmup)
Chain 1, Iteration:   400 / 2000 [ 20%] (warmup)
Chain 1, Iteration:   600 / 2000 [ 30%] (warmup)
Chain 1, Iteration:   800 / 2000 [ 40%] (warmup)
Chain 1, Iteration:  1000 / 2000 [ 50%] (warmup)
Chain 1, Iteration: 1001 / 2000 [ 50%] (Sampling)
Chain 1, Iteration: 1200 / 2000 [ 60%] (Sampling)
Chain 1, Iteration: 1400 / 2000 [ 70%] (Sampling)
Chain 1, Iteration: 1600 / 2000 [ 80%] (Sampling)
Chain 1, Iteration: 1800 / 2000 [ 90%] (Sampling)
Chain 1, Iteration: 2000 / 2000 [100%] (Sampling)#
# Elapsed Time: 0.011 seconds (Warm-up)
#                  0.01 seconds (Sampling)
#                  0.021 seconds (Total)
#
```

Rからstanを走らせる

- 結果を確認

```
> fit1
```

```
Inference for Stan model: binomial1.
```

```
4 chains, each with iter=2000; warmup=1000; thin=1;
```

```
post-warmup draws per chain=1000, total post-warmup draws=4000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
p	0.50	0.00	0.09	0.34	0.44	0.5	0.56	0.67	1476	1
lp__	-22.67	0.02	0.70	-24.61	-22.82	-22.4	-22.23	-22.18	1498	1

```
Samples were drawn using NUTS(diag_e) at Wed May 11 14:44:38 2016.
```

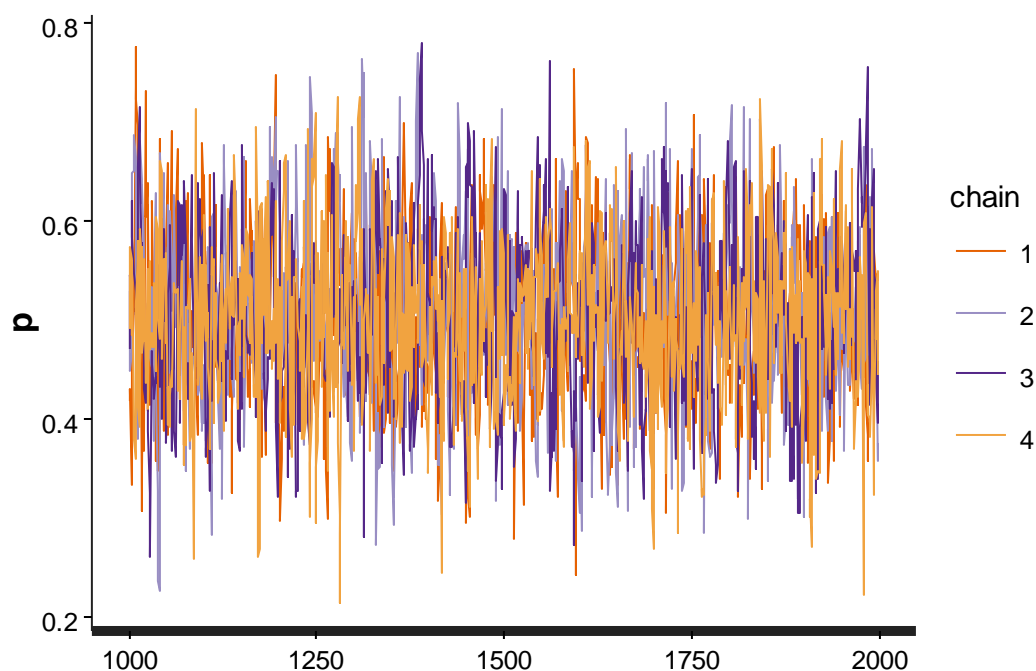
```
For each parameter, n_eff is a crude measure of effective sample size,  
and Rhat is the potential scale reduction factor on split chains (at  
convergence, Rhat=1).
```

二項分布のパラメータpは、平均が0.5, 95%CIが0.34~0.67
Rhatがすべて1.05以下なので収束していると判断できる

トレースプロット

- パラメータのサンプリングの軌跡

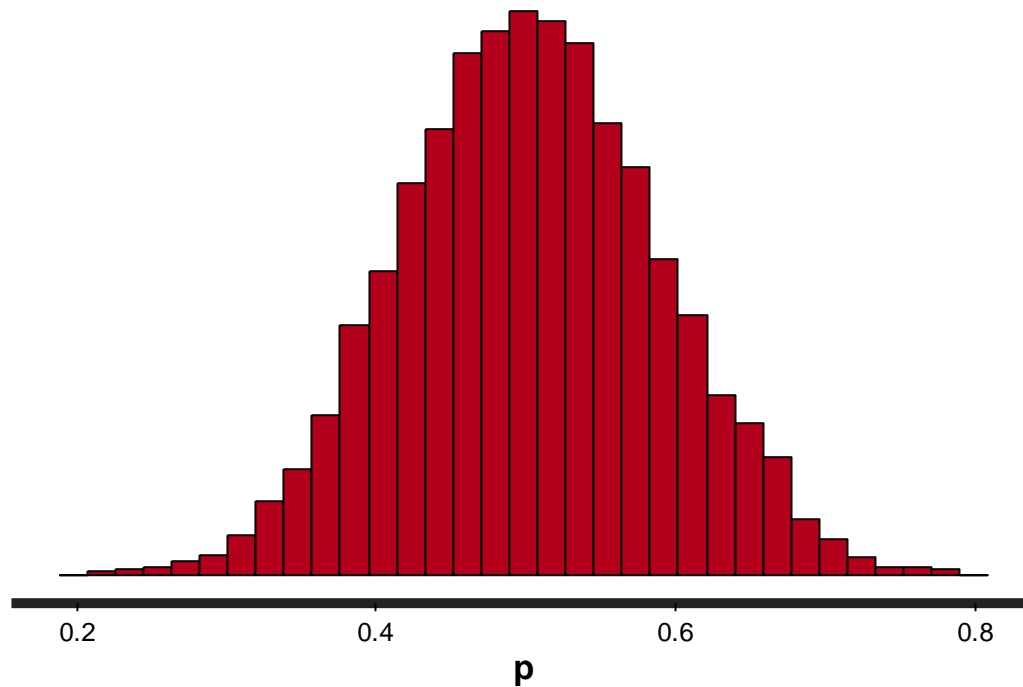
`stan_trace(fit1, pars="p")`



ヒストグラム

- パラメータの事後分布

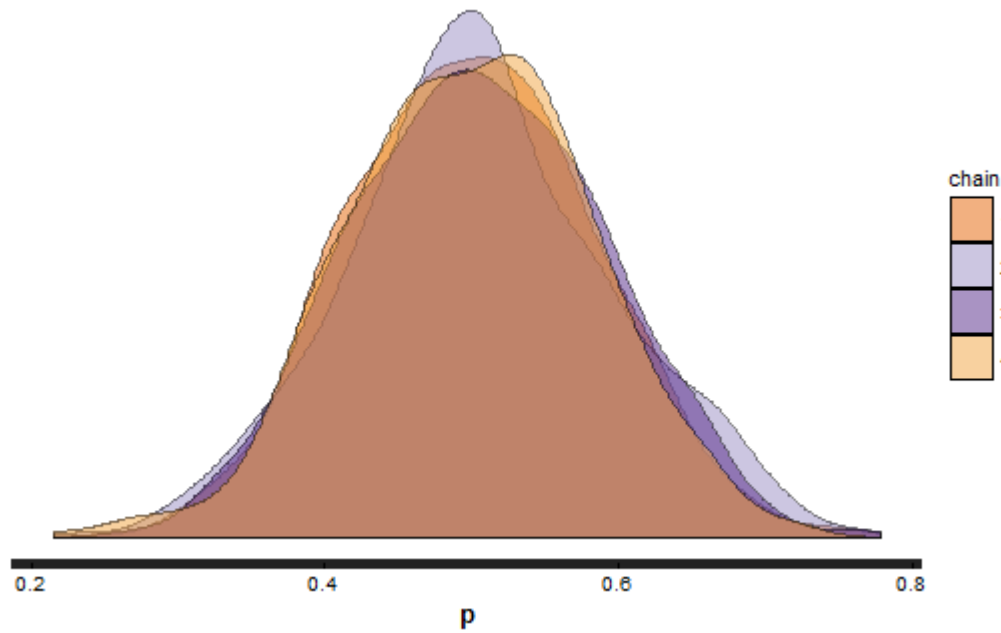
`stan_hist(fit1, pars="p")`



カーネル密度

- パラメータの事後分布の密度推定

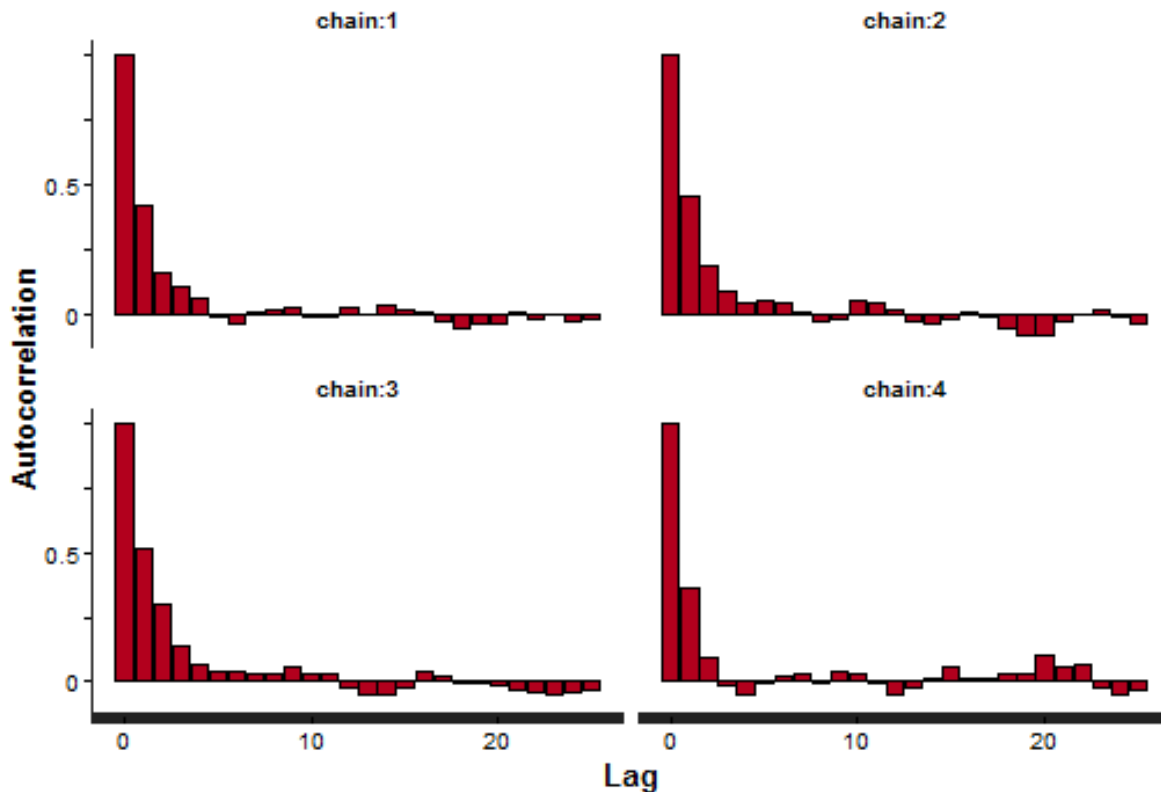
`stan_dens(fit1, pars="p", separate_chains = TRUE)`



自己相関

- 自己相関が高いと収束してない

`stan_ac(fit1, pars="p", separate_chains = TRUE)`



収束の判断

- Rhat
 - 全部のパラメータが1.05以下ならOKと判断
- チェインごとのカーネル密度
 - 収束しているなら, すべてのチェインできれいに重なっているはず
- 自己相関
 - 定常分布に収束したら, ひとつ前のサンプルとは相関が生じなくなる
 - 自己相関が高いなら, まだ収束していない可能性

要素をRに取り出す

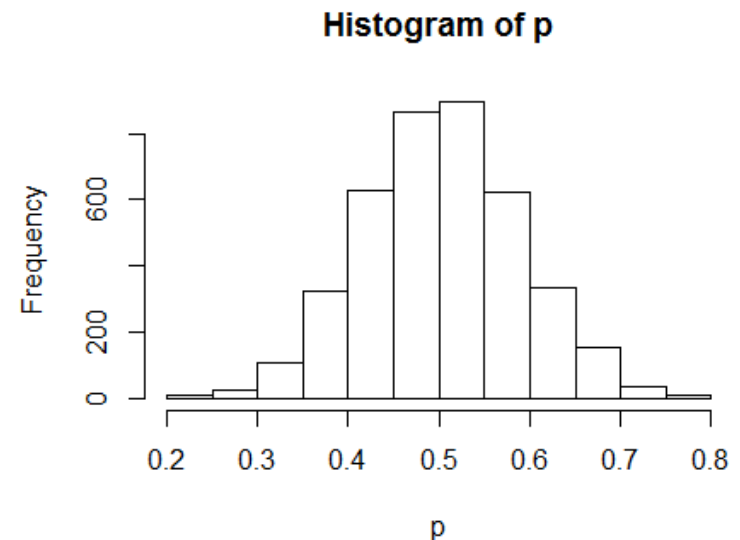
- `extract()`を使う
 - ただし, 同名の関数が多いので, `rstan::extract()`とするのが無難

```
p <- rstan::extract(fit1)$p
```

```
hist(p)
```

```
mean(p)
```

```
> mean(p)  
[1] 0.5027928
```



データの読み込み

データが変わるたびにコンパイル？

```
parameters{  
  real<lower=0,upper=1> p;  
}  
model{  
  15 ~ binomial(30,p);  
}
```

それは面倒くさい

Stanコードの書き方 レベル2

- data {}
 - RからデータをStanに読み込ませる
 - Stanコードを変えずに、データだけを変えることができる

binomial2.stan

```
data{  
  int D; // 生起数  
  int N; // 試行数  
}  
parameters{  
  real<lower=0,upper=1> p; //成功率  
}  
model{  
  D ~ binomial(N,p);  
}
```

int型は整数の型
離散分布のデータは, int型
である必要がある

Rからデータを入力する

- data=にリスト型で指定
 - 成功数20, 試行数30を入力する場合,
 - `fit2 <- stan("binomial2.stan", data = list(D=20,N=30))`
 - 順番は入れ替わっても問題ない
- 直接入れてもいいが・・・
`datastan <- list(D=20, N=30)`
`fit2 <- stan("binomial2.stan",data=datastan)`
としたほうが可読性が高い(と思う)

Rコードから実行

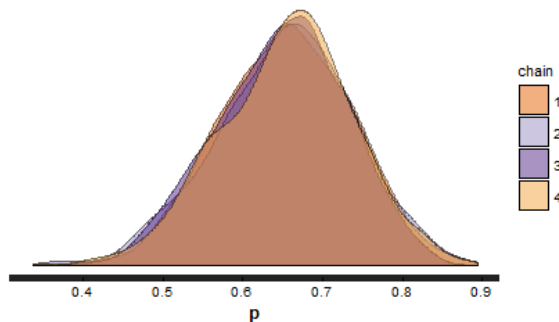
- データを指定する

```
datastan <- list(D=20, N=30)
```

```
fit2 <- stan("binomial2.stan", data=datastan)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%
p	0.65	0.00	0.08	0.48	0.60	0.66	0.71	0.81
lp__	-21.10	0.02	0.73	-23.12	-21.27	-20.82	-20.64	-20.59

```
stan_dens(fit2, pars="p", separate_chains=T)
```



rstanの使い方あれこれ

最初にコンパイルしてしまう方法

- `stan_model()`でモデルを先に作る
`model2 <- stan_model("binomial2.stan")`
- `sampling()`で作ったモデルを実行する
`fit2 <- sampling(model2, data=list(D=20, N=30))`

`stan()`でもOKだけど、あとあとのことを考えると、`stan_model()`と`sampling()`を分けて使うほうが融通がきいて良い

モデルを保存する

- コンパイルしたモデルをRDSファイルとして保存
`saveRDS(model2, "model2.rds")`
- 保存したRDSファイルをモデルとして使う
`model2 <- readRDS("model2.rds")`

これで毎回コンパイルせずともすぐに実行可能

rstanのオプションをいじる

- rstanのオプション `rstan_option()`
 - 設定を変えることができる
- Stanモデルの自動保存をオンにする
`rstan_options(auto_write = TRUE)`
- 自動的にマルチコアでチェーンを走らせる
`options(mc.cores = parallel::detectCores())`
 - ただし、ファイヤーウォールの設定によってはrstanからのマルチコアのコードが弾かれてエラーも出ないという事態がありうる。
その場合は`mc.cores=1`にする

samplingのオプション

- サンプルング数
 - iter =
 - デフォルトは2000 複雑なモデルの場合は2000じゃ足りない場合もある
- バーンイン期間
 - warmup =
 - デフォルトはiterの半分 iterを大きくしたら, warmupは相対的に小さくてもいける場合もある
 - 別にいじらなくてもいい

samplingのオプション

- 間引き間隔

- thin =
- デフォルトは1 1の場合はすべての推定値を使うが, 2にすると1つとばしで推定値を使う
- iterを増やすとMCMCサンプルも多くなって重くなったりもするので, thinを大きくして事後分布のサイズを抑えることも有効
- 自己相関を抑えたいときもthinを大きめにするといい

samplingのオプション

- マルコフ連鎖の数
 - chains =
 - デフォルトは4 最低でも2は走らせないと, 収束したかどうかちゃんとわからない
 - chains=1でもrhatは出力されるが, かなり怪しい(と思う)
 - モデルをチェックするときは2でもいいが, 最終的な解を計算するときは4ぐらいあったほうがいいかも
- 使うコアの数
 - cores =
 - デフォルトは1
 - コアの数チェーンの数より多くても意味がない
 - 並列化が可能なPCならチェーンと同じ数指定すると, 推定時間が短くなる

rstanコードの例

- 設定
 - サンプルング5000回
 - バーンイン(warmup)期間1000回
 - マルコフ連鎖の数を3つ
 - コアも3つ

```
data <- list(D=20,N=30)
model2 <-
stan_model("binomial2.stan")
fit2 <- sampling(model2,
                  data=datastan,
                  iter=5000,
                  warmup=1000,
                  chains=3,
                  cores=3)
```

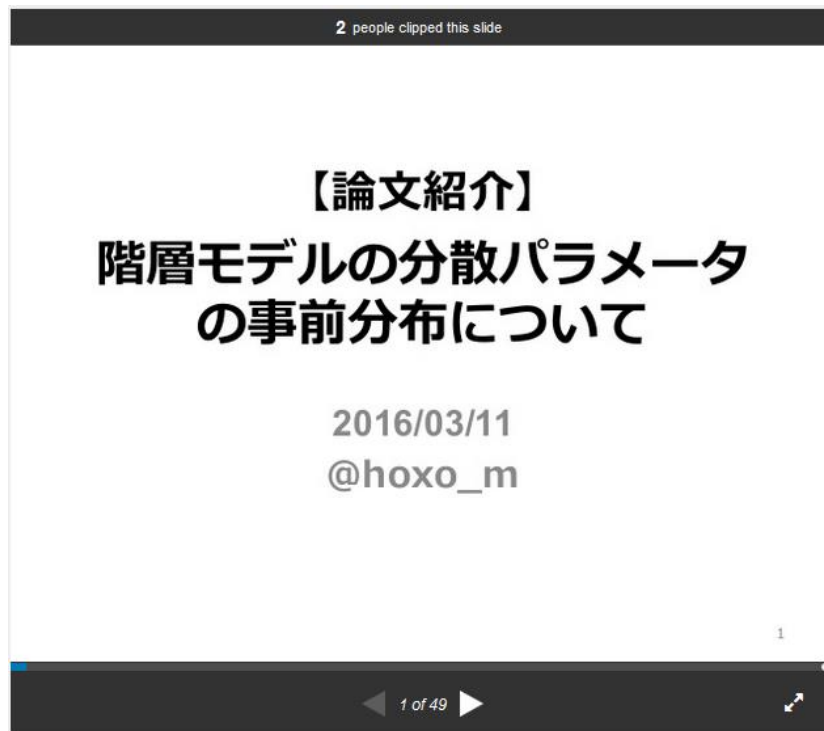
事前分布の指定

Stanコードの書き方 レベル3

- 事前分布を指定する
 - 事前分布は, `model`に記述する
 - 何も書かないと, $-\infty \sim \infty$ の一様分布となる
 - ただし, 今回は $0 \sim 1$ の一様分布となる
 - 何も書かなくてもそれほど問題はない
 - ただし, サンプルサイズが小さい場合は, 適切な事前分布を設定した方が良い推定ができる
- 特定の事前分布を指定してみる
 - 確率 p はベータ分布と相性が良い

事前分布について

- @hoxo_mさんのスライドが役立つ
 - http://www.slideshare.net/hoxo_m/ss-59418886



初心者はとりあえず..

- 選択肢1: 何も指定しない
 - つまり, 無情報一様分布
 - ただし, 非正則な分布になる
 - サンプルサイズが小さい場合は不適切になることもある
- 選択肢2: パラメータに合った弱情報事前分布
 - 平均値, 回帰係数: 正規分布 (or コーシー分布)
 - $\sim \text{normal}(0, 100)$; SDはパラメータのスケールに合わせる
 - 標準偏差, 分散: 半コーシー分布
 - $\sim \text{cauchy}(0, 5)$; 加えて, パラメータの下限を0にしておく
 - 確率, 比率: ベータ分布か一様分布
 - $\sim \text{beta}(1, 1)$;
 - $\sim \text{uniform}(0,1)$; この二つは同じ分布になる
 - 相関行列
 - `lkj_corr(1)`あるいは`lkj_corr(2)` パラメータが1は無情報, 2は弱情報
 - stanマニュアルは2を推奨しているが, 最尤法と一致させるには1を使う

とりあえず無情報事前分布を指定

binomial3.stan

```
data{  
  int D; // 生起数  
  int N; // 試行数  
}  
parameters{  
  real<lower=0,upper=1> p; //成功率  
}  
model{  
  p ~ beta(1,1);  
  D ~ binomial(N,p);  
}
```

stanではコメントアウトは//を使う

← パラメータが1,1のベータ分布
0~1の範囲で当確率になる分布

データを読み込む2

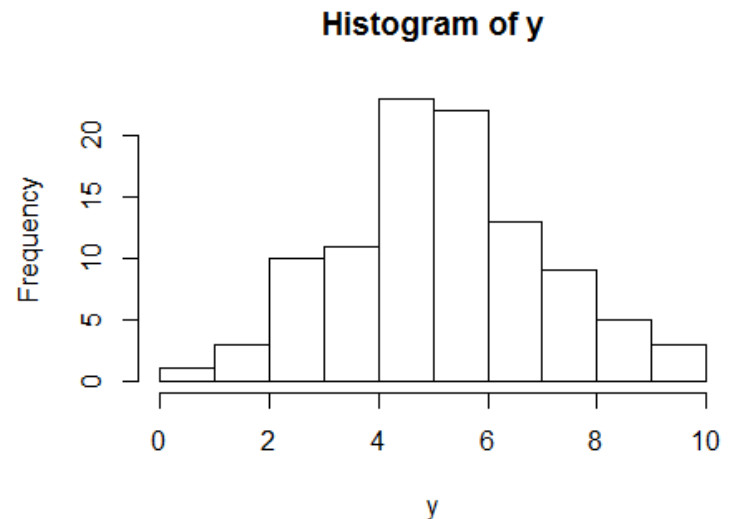
正規分布のパラメータの推定

- 100人のデータが正規分布から生成
 - Rを使って乱数を生成してみる
 - 平均=5, 標準偏差=2の正規分布

```
set.seed(123)
```

```
y <- rnorm(100,5,2)
```

```
hist(y)
```



これを逆にStanで推定する！

Stanコードの書き方 レベル4

- データを配列として指定する
 - 配列: 複数の値が入る箱のようなもの
 - `real y[100];`
 - `y`は実数で, サイズ100の配列であると宣言
 - 今回は, $N=100$ のデータ
- 正規分布の確率モデル
 - パラメータは`mu`と`sigma`で宣言
 - `~ normal(mu, sigma)`と書いて, 統計モデルを表現
 - Stanの正規分布の尺度パラメータは分散ではなく, SD
 - BUGSやJAGSは分散の逆数をいれるが, Stanは違うので注意

FOR文

- 繰り返しを記述する文法
 - ほとんどのプログラミング言語で登場
 - Rと基本的には同じ書き方
- `for(){}`
 - `for`のあとの`()`には繰り返しの設定, `{}`には繰り返したい処理を書く
 - 今回は100個のデータに統計モデルを指定したい

FOR文

- 繰り返しの設定
 - `for(i in 1:100)`と書くと、`i`という変数が1～100代入される間処理が繰り返されることを意味する
 - つまり、100回処理を繰り返す、ということ
- 繰り返したい処理
 - `{}`内に繰り返したい処理を書くが、大抵は

```
for(n in 1:100){
  繰り返したい処理
}
```

というように、改行することが多い。

正規分布のStanコード

normal0.stan

```
data{
  real y[100]; //データ
}
parameters{
  real mu; //平均値
  real<lower=0> sigma; //標準偏差
}
model{
  mu ~ normal(0,100); //平均値の事前分布
  sigma ~ cauchy(0,5); //標準偏差の事前分布
  for(n in 1:100){
    y[n] ~ normal(mu,sigma);    100人分, Forで繰り返す
  }
}
```

y[n]は, nが1番目から100番目まで変化するので, yの要素の全部について平均mu, SDがsigmaの正規分布に従うことを指定していることになる

Rコード

```
model4 <- stan_model("normal0.stan")  
datastan <- list(y=y)  
fit4 <- sampling(model4,data=datastan)  
fit4
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu	5.18	0.00	0.18	4.83	5.06	5.19	5.31	5.52	2436	1
sigma	1.84	0.00	0.13	1.60	1.75	1.83	1.93	2.11	2544	1
lp__	-110.19	0.03	0.98	-112.72	-110.58	-109.90	-109.49	-109.24	1367	1

配列の要素も変数にする

- データサイズも一般化する
 - `real y[100];`だと、データサイズが変わるとコードも変えないといけない
 - そこで、データサイズをNという変数で宣言する

```
int N;
```

```
real y[N];
```

- このとき、要素数を指定するNは必ずint型
- For文もNを使って回す
 - `for(n in 1:N)`

レベル4.1

normal1.stan

```
data{  
  int N; //サンプルサイズ  
  real y[N]; //データ  
}  
parameters{  
  real mu; //平均値  
  real<lower=0> sigma; //標準偏差  
}  
model{  
  mu ~ normal(0,100); //平均値の事前分布  
  sigma ~ cauchy(0,5); //標準偏差の事前分布  
  for(n in 1:N){  
    y[n] ~ normal(mu,sigma);  
  }  
}
```

Nはint型で宣言する必要あり

Stanコードのエラー

Stanでの2種類のエラー

- 文法エラー
 - Stanコードが間違えている場合
 - RstudioのCheckボタンで確認できる
 - `stan()`で走らせる前にチェックしておくといい
- サンプルング中のエラー
 - 文法的には合ってるけど、上手く計算出来ない場合
 - 配列の要素数が指定された値を超えた
 - SDは非負しか認められないが、負の値が入った
 - 共分散行列が対称になってない などなど

初心者が多い文法エラー

- セミコロンを入れ忘れる
 - たぶん最初の80%はこれが原因
 - エラーが出たらセミコロン！
- スペルミス
 - これはRでもよくある
 - Stanでは大文字小文字も区別される
- 関数の引数の型が、変数の型に合っていない
 - 関数には、引数にいれる変数の型が決まっている
 - 変数の型については次のレベルで解説
- 行列演算が正しく記述されてない
 - 列と行を間違えてる
 - 計算された結果とそれを入れる変数の型や要素数があってない

初心者が多い計算のエラー

- パラメータの範囲が指定されていない
 - 分散やSDは非負でないといけない
 - `<lower=0>`がちゃんと付いているか確認
- 入力データと変数の型・要素数があってない
 - int型なのにデータに少数が含まれている
 - `x[100]`で宣言してるのに、データは200個ある・・・など
- 要素数を超える
 - For文の繰り返しと、配列の要素数があってない

```
real x[50];  
for(i in 1:100){  
    x[i] <- hogehoge;  
}
```

xは50しか要素がないのに、iは51以上の値が入りうるのでエラーが出る

宣言したが使っていないパラメータ

- Stanは何も言ってくれない
 - 消し忘れがあったとか, そういうのが合ってもStanは気にせずサンプリングする
- あっていいことは何もない
 - 収束が悪くなったり, 計算時間が増えたり, エラーが出たりする
 - モデルを変更したら, 余計なパラメータが宣言されて残っていないかを確認しよう

ベクトル化したコードの書き方と 変数の型について

レベル5 ベクトル化

- For文をベクトル処理することもできる

normal2.stan

```
model{  
  mu ~ normal(0,100); //平均値の事前分布  
  sigma ~ cauchy(0,5); //標準偏差の事前分布  
  y ~ normal(mu,sigma);  
}
```

- N次元のyに対して、一度にnormal()の分布に従うという処理を行っている
- こうすると、正規分布の尤度を計算する処理が一部共有できるので、計算が早くなる

ベクトル化

- For文を使うとき

```
# Elapsed Time: 0.041 seconds (Warm-up)
#               0.038 seconds (Sampling)
#               0.079 seconds (Total)
#
```

- ベクトルとして書くとき

```
# Elapsed Time: 0.015 seconds (Warm-up)
#               0.013 seconds (Sampling)
#               0.028 seconds (Total)
#
```

StanにおけるFor文

- Rと違って遅いわけではない
 - C++にコンパイルしてから走らせているので、Rのように極端にFor文が遅くなるということはない
 - なので、あえてFor文を避ける必要はあまりない
- 関数処理をベクトル化することのメリット
 - たとえば分散をlogするとか、共分散行列の逆行列を計算するとか、そういう処理が1回で済む
 - For文を使うと、それを毎回行うので、その分計算量が増えてしまう

Stanにおける変数の型 1

- 基本的な変数の型
 - real: 実数型 連続値の場合はこれを使う
 - int: 整数型 離散データの場合はこれを使う
- 分布によって, 対応する型が違う
 - 連続分布: 正規, ガンマ, ベータ, 対数正規・・・
 - real型を使わないといけない
 - 離散分布: 二項, ポアソン, 負の二項・・・
 - int型を使わないといけない

Stanにおける変数の型 2

- 行列演算を使える変数型
 - vector: (列)ベクトル
 - 数値が複数個縦に並んだもの
 - row_vector: 行ベクトル
 - 数値が横に並んだもの
 - matrix: 行列(マトリックス)
 - ベクトルが複数並んだもの

(列)ベクトル

1
2
3
4
5

行ベクトル

1	2	3	4	5
---	---	---	---	---

マトリックス

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

Stanにおける変数の型 2.1

- 行列型の宣言の仕方
 - `vector[2] x;`
 - `matrix[4,5] x;`
 - ということに, 変数名の前に要素数を指定
- 上限, 下限がある場合
 - `vector<lower=0>[5] sigma;`
 - ということに, 要素数の前に範囲を指定

Stanにおける変数の型 3

- 配列型

- `x[10]`: `x`は10個の変数が入った箱
- 複数の変数を規則正しく配置したもの
- 1次元配列, 2次元配列・・・というように次元を増やすことができる

- 配列の宣言

- `real x[5]` : 実数が5個の1次元配列
- `int x[5,5]` : 整数が 5×5 に配置された2次元配列
- `vector[2] x[5]` : 要素数2のベクトルが5個入った配列

Stanにおける変数の型 3.1

- 配列型と行列型の違い
 - どちらも複数の値が入れられる点は同じ
- ベクトルは行列演算が可能
 - 線形代数の演算が使える
 - 繰り返し処理をせず, 簡単・高速に計算可能
 - しかし, real型しか扱えない
 - 行列演算を行う独立変数はベクトルで指定したほうがいい
- 配列は行列演算ができない
 - しかし, int型を含むことが可能
 - 離散分布を仮定する従属変数のデータでは, int型の配列を利用する必要がある

vector型の配列の注意

- 要素を指定したい場合

- `vector[M] x[N]` で宣言したとき,

- 二重For文での処理の仕方は,

- `for(n in 1:N){`

- `for(m in 1:M){`

- `x[n][m] <- ほげほげ;`

- `}`

- `}`

配列で指定した要素が先, ベクトルや行列
の要素が後

`x[n,m]`と書くこともできるが, 上のほうがわかりやすいかも

関数に合わせて宣言

- 関数は引数の型が決まっている
 - たとえば正規分布の平均パラメータはrealで宣言しないといけない
 - int型は使えない
 - Stanでは離散型のパラメータは(現状)許されていない
- 多変量正規分布はvectorとmatrixが引数
 - データと平均はvector型
 - 共分散行列はmatrix型
 - というように、使う分布・関数に合わせて変数を宣言する必要がある

ややこしい...

- 覚えておくこと
 - パラメータはreal型, vector型, matrix型で宣言
 - データは分布の性質に合わせる
 - 連続ならreal型 ただし, 多変量正規分布はvector型
 - 離散ならint型

生成量の計算

情報量規準を計算したい

- WAIC
 - 広く使える情報量規準
 - AICのベイズ版
- Stanでも簡単に計算可能
 - データそれぞれについての対数尤度をはきだしてやり, それをlooパッケージの`waic()`にいれる
 - 自分で計算用の関数を作ることでもある

生成量の計算

- generated quantities{
 - Stanで推定したパラメータやデータを使って, 好きな値を生成する
 - 今回は対数尤度を出力する
- ブロック内で変数を宣言する必要がある
 - 出力したい対数尤度を変数として宣言
 - `real log_lik[N];` とする。N個の対数尤度を格納

対数尤度の計算

- `normal_log()`を使う
 - 確率分布`_log()`は、対数尤度(確率)を計算する関数
 - `normal_log(データ, 平均, 標準偏差)`という順に引数を入れる
 - 平均とSDはMCMCで推定されたものを代入する
 - 今回は正規分布だが、Stanに入っているすべての確率分布で`_log`をつければ同じように使える

※`hoge_log()`の関数では、引数にベクトルとスカラーが混ざっているのでベクトル化はできない

Stanコード レベル6

```
data{
  int N; //サンプルサイズ
  real y[N]; //データ
}
parameters{
  real mu; //平均値
  real<lower=0> sigma; //標準偏差
}
model{
  mu ~ normal(0,100); //平均値の事前分布
  sigma ~ cauchy(0,5); //標準偏差の事前分布
  y ~ normal(mu,sigma);
}
generated quantities{
  real log_lik[N];
  for(n in 1:N){
    log_lik[n] <- normal_log(y[n],mu,sigma);
  }
}
```

normal3.stan

データを固定した場合のパラメータの尤度が対数で返される

現バージョンでは代入は"<-"を使っているが、
今後は"<-"は非推奨になり、"="を使うらしい

WAICを計算するためのコード

- 以下のコードを読み込む

```
WAIC <- function(fit){  
  log_lik <- rstan::extract(fit)$log_lik  
  lppd <- sum(log(colMeans(exp(log_lik))))  
  p_waic <- sum(apply(log_lik,2,var))  
  waic <- 2*(-lppd+p_waic)  
  return(list(waic=waic,lppd=lppd,p_waic=p_waic))  
}
```

WAICの計算

- 手作りWAIC()を使ってWAICを計算する

```
fit6 <- stan("normal3.stan",data=list(N=100,y=y))
```

```
WAIC(fit6)
```

```
> WAIC(fit6)
```

```
$waic
```

```
[1] 406.9578
```

WAIC

```
$lppd
```

```
[1] -201.6729
```

モデルの対数尤度

```
$p_waic
```

```
[1] 1.806
```

実効パラメータ数

looパッケージのwaic()もある

```
library(loo)  
loo::waic(rstan::extract(fit6)$log_lik)
```

Computed from 4000 by 100 log-likelihood matrix

	Estimate	SE
elpd_waic	-203.5	6.8
p_waic	1.8	0.3
waic	407.0	13.7

looのwaic()は引数がStanフィットではなく、対数尤度のMCMCサンプルである点が違うので注意

AICと比較

- Rでlm()を使って計算

– AIC(lm(y~1))

```
> AIC(lm(y~1))  
[1] 407.1679
```

まあ、だいたい同じ

```
> WAIC(fit6)  
$waic  
[1] 406.9578
```

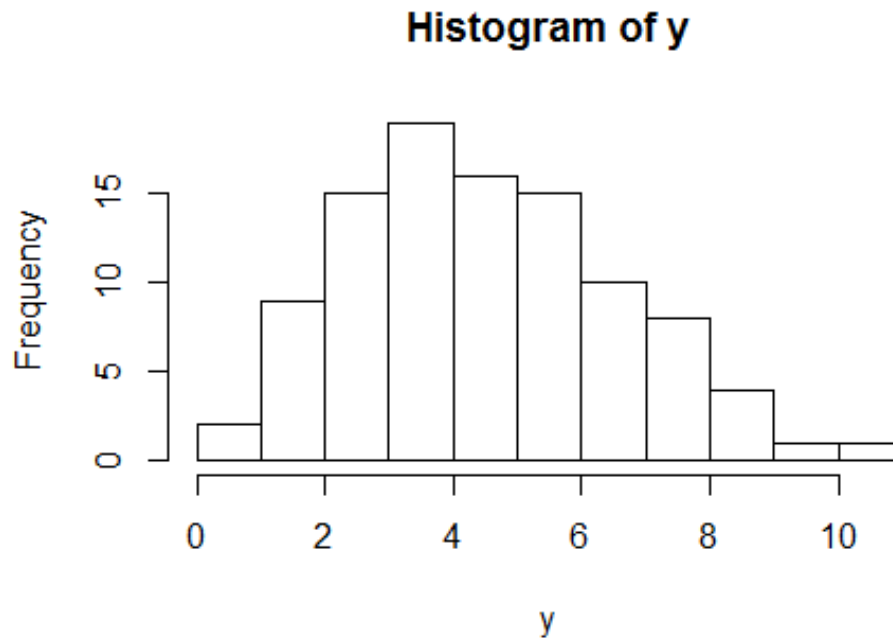
違う確率分布からデータを作ると・・・

- 平均5のポアソン分布

```
set.seed(123)
```

```
y <- rpois(100,5)
```

```
hist(y)
```



Stanで正規分布を仮定した推定

- さっきと同じStanのモデルを走らせる

```
fit6_2 <- stan("normal3.stan", data=list(N=100, y=y))  
WAIC(fit6_2)
```

```
> WAIC(fit6_2)
```

```
$waic  
[1] 438.6802
```

```
$lppd  
[1] -217.5488
```

```
$p_waic  
[1] 1.791301
```

ポアソン分布から生成

```
> WAIC(fit6)
```

```
$waic  
[1] 406.9578
```

```
$lppd  
[1] -201.6729
```

```
$p_waic  
[1] 1.806
```

正規分布から生成

当たり前だが、
正規分布から生成
したデータのほうが
WAICが小さい

特定のパラメータだけ出力する

- RでStanの結果を出力する場合
 - 今回のように, log_likは100個分計算されているので, 出力が膨大になる

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu	5.01	0.00	0.22	4.58	4.85	5.01	5.16	5.44	2387	1
sigma	2.16	0.00	0.15	1.89	2.05	2.16	2.27	2.48	2448	1
log_lik[1]	-1.80	0.00	0.07	-1.95	-1.85	-1.80	-1.75	-1.67	2487	1
log_lik[2]	-2.12	0.00	0.10	-2.32	-2.19	-2.12	-2.06	-1.95	2420	1
log_lik[3]	-1.80	0.00	0.07	-1.95	-1.85	-1.80	-1.75	-1.67	2487	1
log_lik[4]	-2.66	0.00	0.16	-2.99	-2.77	-2.65	-2.56	-2.39	2512	1
log_lik[5]	-3.42	0.01	0.25	-3.95	-3.58	-3.40	-3.24	-2.98	2545	1
log_lik[6]	-2.67	0.00	0.16	-3.02	-2.78	-2.67	-2.56	-2.39	2397	1
log_lik[7]	-1.69	0.00	0.07	-1.83	-1.74	-1.69	-1.64	-1.56	2387	1
log_lik[8]	-2.66	0.00	0.16	-2.99	-2.77	-2.65	-2.56	-2.39	2512	1
log_lik[9]	-1.69	0.00	0.07	-1.83	-1.74	-1.69	-1.64	-1.56	2387	1
log_lik[10]	-1.69	0.00	0.07	-1.83	-1.74	-1.69	-1.64	-1.56	2387	1
log_lik[11]	-3.42	0.01	0.25	-3.95	-3.58	-3.40	-3.24	-2.98	2545	1
log_lik[12]	-1.69	0.00	0.07	-1.83	-1.74	-1.69	-1.64	-1.56	2387	1
log_lik[13]	-1.80	0.00	0.07	-1.95	-1.85	-1.80	-1.75	-1.66	2340	1
log_lik[14]	-1.69	0.00	0.07	-1.83	-1.74	-1.69	-1.64	-1.56	2387	1
log_lik[15]	-2.67	0.00	0.16	-3.02	-2.78	-2.67	-2.56	-2.39	2397	1
log_lik[16]	-2.66	0.00	0.16	-2.99	-2.77	-2.65	-2.56	-2.39	2512	1

特定のパラメータだけ出力する

- 特定のパラメータだけ出力したい場合

```
print(fit6,pars=c("mu","sigma"))
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu	5.01	0	0.22	4.58	4.85	5.01	5.16	5.44	2387	1
sigma	2.16	0	0.15	1.89	2.05	2.16	2.27	2.48	2448	1

```
print(fit6,pars=c("mu","sigma"),digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu	5.007	0.005	0.222	4.576	4.854	5.007	5.158	5.438	2387	1.000
sigma	2.164	0.003	0.154	1.888	2.054	2.157	2.267	2.483	2448	1.001

出力桁数を変えることも可

宣言したが何も値が入らない生成量

- generated quantities{}で宣言した変数
 - すべてのサンプリングが終わってから生成される
 - そのとき, 何も値が入らない変数があると, 最後の最後でエラーが出る
 - 長いサンプリングが終わってワクワクしていても, そこで躓くと脱力感が半端ないので注意しよう

平均値の差の推論

t 検定みたいなことしたい

- 二つの群の平均に差があるのか？
 - t 検定のように、データから同じ平均値の分布から生成されているのか、そうでないのかについて推論したい
- ベイズでは帰無仮説検定は使わない
 - 信用区間を使った推論
 - 生成量を使って差がある確率を推論

信用区間を使った推論

- 推定した平均パラメータの差の信用区間
 - 差の95%信用区間に0を含んでいるなら, 頻度主義の帰無仮説検定と同様, 有意な差ではないと解釈することができる
- 二つのデータに異なる分布を仮定
 - 平均, 標準偏差が違うと仮定
 - 平均値の差はどの程度大きいだろうか

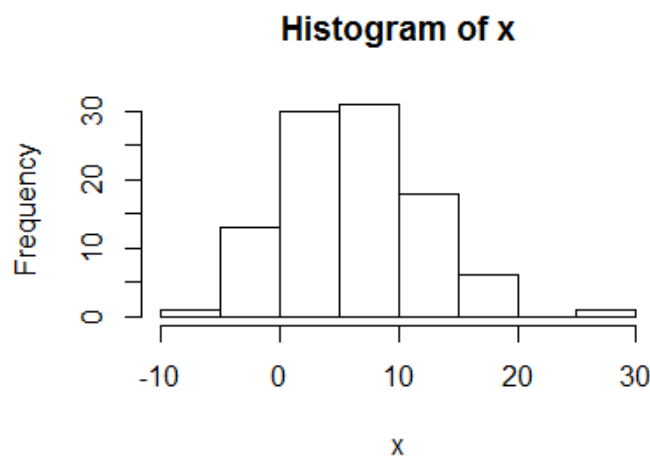
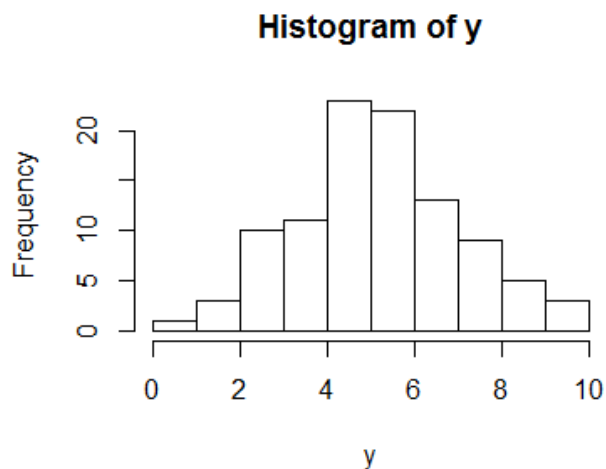
二つのデータを生成

- 異なるパラメータの正規分布から生成

```
set.seed(123)
```

```
y <- rnorm(100,5,2)
```

```
x <- rnorm(100,7,6)
```



普通に t 検定を試みる

`t.test(x,y)`

```
Welch Two Sample t-test
```

```
data: x and y
```

```
t = 1.93, df = 118.41, p-value = 0.056
```

```
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
```

```
-0.03052349  2.37833828
```

```
sample estimates:
```

```
mean of x mean of y
```

```
6.354719  5.180812
```

– x のほうが y より大きいが、有意ではない...

Stanコードの書き方 レベル7

normal4.stan

```
data{
  int N; //サンプルサイズ
  real y[N]; //データ1
  real x[N]; //データ2
}
parameters{
  real mu_y; //yの平均値
  real<lower=0> sigma_y; //yの標準偏差
  real mu_x; //xの平均値
  real<lower=0> sigma_x; //xの標準偏差
}
model{
  mu_y ~ normal(0,100);
  sigma_y ~ cauchy(0,5);
  mu_x ~ normal(0,100);
  sigma_x ~ cauchy(0,5);

  y ~ normal(mu_y,sigma_y);
  x ~ normal(mu_x,sigma_x);
}
generated quantities{
  real diff;
  diff <- mu_x-mu_y;
}
```

- 別々のSDを推定
 - 異分散を仮定しているので、Welch検定と同様の分析
 - 等分散にすることも可能
- 差を生成する
 - 生成量として二つの平均の差を計算している

結果

- Rコード

```
model7 <- stan_model("normal4.stan")
```

```
fit7 <- sampling(model7, data=list(N=100, y=y, x=x))
```

- diffに注目

- 差は1.17で95%CI = -0.04 ~ 2.38

- 有意な差があるとはいえない

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu_y	5.18	0.00	0.19	4.82	5.06	5.18	5.30	5.56	2656	1
sigma_y	1.84	0.00	0.13	1.61	1.75	1.84	1.93	2.12	2720	1
mu_x	6.35	0.01	0.59	5.20	5.96	6.35	6.76	7.50	3200	1
sigma_x	5.82	0.01	0.42	5.06	5.53	5.80	6.08	6.71	2691	1
diff	1.17	0.01	0.61	-0.04	0.75	1.17	1.59	2.38	3074	1
lp__	-335.67	0.04	1.44	-339.38	-336.37	-335.35	-334.60	-333.89	1193	1

$\mu_x > \mu_y$ である確率を求めたい

- diffは差の事後分布
 - つまり確率分布
 - diffの事後分布のうち、0より大きな値を取る確率が、 $\mu_x > \mu_y$ である確率といえる

- R上で簡単に計算可能

```
diff <- rstan::extract(fit7)$diff
```

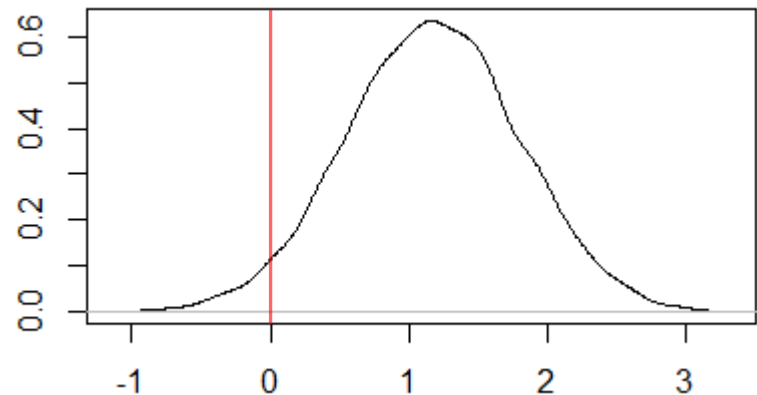
```
p <- sum(ifelse(diff>0,1,0)) / length(diff)
```

```
> p  
[1] 0.97
```

0より大きな値を1に変換してその総和をサンプリング数で割れば、0より大きな値をとった割合を計算できる

差の事後分布

- 差が0より高い = $\mu_x > \mu_y$ である確率
 - 97%
 - 97.5%が帰無仮説検定の5%の基準 → 弱い主張
- 事後分布を視覚的にも確認
`plot(density(diff))`



相関係数

相関係数のベイズ推定

- 相関係数は多変量正規分布のパラメータ
 - 心理統計では、相関係数は標準得点の共分散として定義されることが多いが...
 - ベイズの枠組みでは確率モデルとして表現する必要がある
- 多変量正規分布で2変数をモデリングする
 - $\vec{y} \sim \text{multi_normal}(\vec{\mu}, \Sigma)$
 - $\vec{\mu}$ は平均ベクトル, Σ は共分散行列

多変量正規乱数の生成

- MASSパッケージのmvrnorm()を使う

```
library(MASS)
```

```
mu <- c(0,0)
```

```
Sigma <- matrix(c(1,0.7,0.7,1),2,2)
```

```
set.seed(123)
```

```
y <- mvrnorm(25,mu,Sigma)
```

– 相関が0.7の標準多変量正規分布

普通に相関を計算

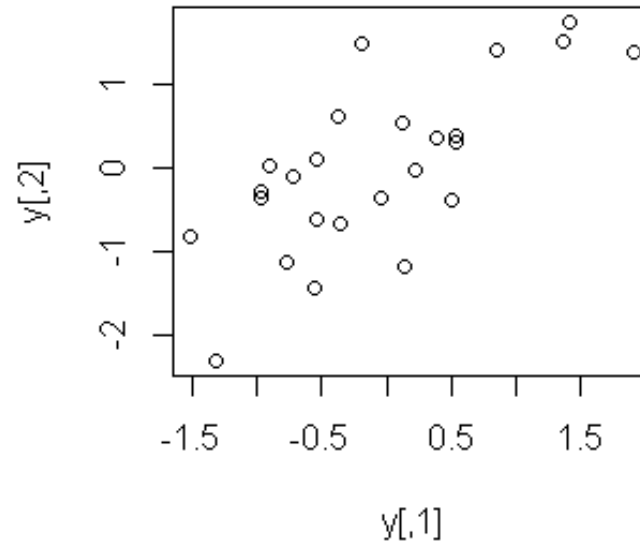
- `cor()`を使う

`cor(y)`

`plot(y)`

	[,1]	[,2]
[1,]	1.0000000	0.7220626
[2,]	0.7220626	1.0000000

$r = 0.722$



Stanコードレベル8

- transformed parameters{
 - パラメータがモデルに合うように変形する
 - parameters{で指定するパラメータは事前分布が設定しやすいかたちで
 - transformed parameters{は確率分布に合う形で
- 多変量正規分布のパラメータは共分散行列
 - 相関係数をパラメータにして, transformed parameters{内で共分散行列に変換しよう

ちょっと長い

corr1.stan

```
data{
  int N; //サンプルサイズ
  vector[2] y[N];
}
parameters{
  vector[2] mu; //平均ベクトル
  vector<lower=0>[2] sigma; //標準偏差ベクトル
  real<lower=-1,upper=1> rho; //相関係数
}
transformed parameters{
  cov_matrix[2] Sig; //共分散行列
  Sig[1,1] <- sigma[1]^2;
  Sig[2,2] <- sigma[2]^2;
  Sig[1,2] <- rho*sigma[1]*sigma[2];
  Sig[2,1] <- rho*sigma[1]*sigma[2];
}
model{
  mu ~ normal(0,100);
  sigma ~ cauchy(0,5);
  rho ~ uniform(-1,1); //-1~1の 一様分布
  y ~ multi_normal(mu,Sig);
}
```

← 2変数なのでvectorの要素数は2

多変量正規分布のパラメータである共分散行列Sigは, cov_matrix型で宣言する

yをvector型配列で宣言

```
data{  
  int N; //サンプルサイズ  
  vector[2] y[N];  
}
```

- yの確率モデルは多変量正規分布
 - 多変量正規分布はStanではmulti_normal()
 - multi_normal()はデータがvector型である必要
- vectorの配列
 - 2変数なのでyは2次元ベクトルを宣言
 - 2次元ベクトルがN人分あるので, y[N]となる
 - このような宣言の仕方はよく使うので覚えよう

解説

- パラメータは平均と標準偏差と相関係数
 - 平均の事前分布は分散が大きい正規分布
 - 標準偏差は半コーシー
 - 相関係数は-1~1の一様分布
- 共分散行列を作る
 - `cov_matrix[]`は共分散行列型 変数の数を指定
 - 対角項(分散)は標準偏差の2乗
 - 共分散は相関 * $SD_x * SD_y$
 - それを`multi_normal()`で推定

やってみる

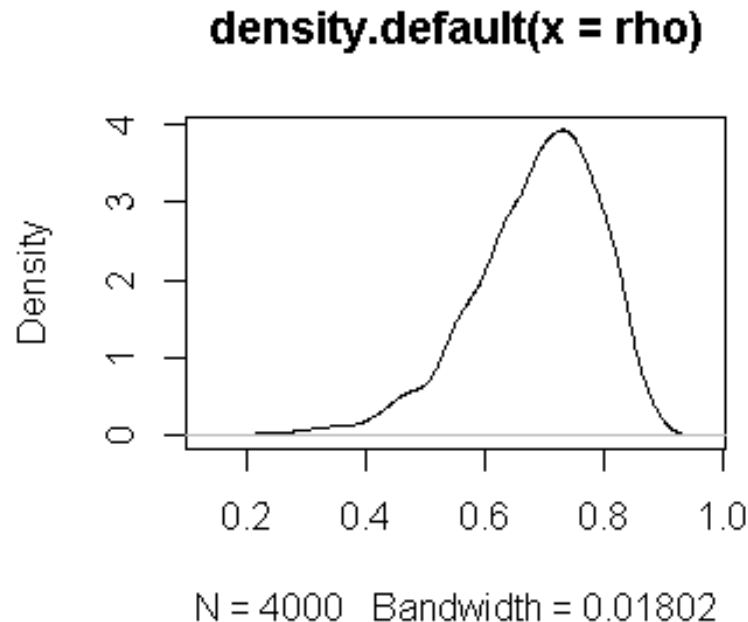
```
model8 <- stan_model("corr1.stan")  
fit8 <- sampling(model8, data=list(N=25, y=y))  
print(fit8, pars="rho", digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
rho	0.69	0	0.11	0.43	0.62	0.7	0.76	0.85	1486	1

- 相関の事後分布の平均値0.69は、普通に計算した0.72と違いがある
- 中央値=0.70は比較的近い

事後分布をちゃんと見てみる

```
rho <- rstan::extract(fit8)$rho  
plot(density(rho))
```



分布がゆがんでる

相関係数は-1~1の範囲をとるので、-1や1に近づくと分布が左右対称ではなくなる

つまり、平均値は最尤法の結果と一致しなくなる(最尤推定値は分布の最頻値)

最頻値を計算してみる

- 事後確率最大値 (MAP推定値)
 - カーネル密度推定から「なんとなく」計算可能

```
map_mcmc <- function(z){  
  density(z)$x[which.max(density(z)$y)]  
}  
  
map_mcmc(rho)
```

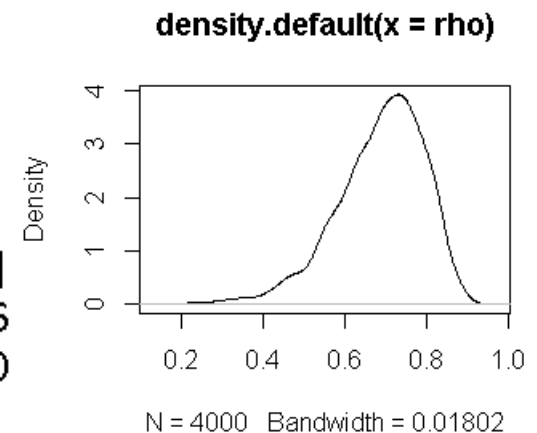
```
> map_mcmc(rho)
```

```
[1] 0.7302759
```

```
           [,1]      [,2]  
[1,] 1.0000000 0.7220626  
[2,] 0.7220626 1.0000000
```

普通の相関係数と近くなった
※ただしあくまで近似です

cor()で計算した相関係数



でもこの方法, ちょっと遅い

- Stanは多変量正規分布の処理が遅い
 - 2変数の相関ならいいが, 多変数だとかなり遅い
 - 別の分布を使って推定する方法を考える
- ウィッシュャート分布を使う
 - 正規分布に従う値の2乗和の分布・・・ χ^2 乗分布
 - χ^2 乗分布の多変量版・・・ウィッシュャート分布
 - 言い換えれば, 偏差積和行列を生成するような確率分布
 - よって, データも偏差積和行列を入力する

Stanコード レベル9

corr2.stan

```
data{
  int N; //サンプルサイズ
  matrix[2,2] spd; //偏差積和行列
}
parameters{
  vector<lower=0>[2] sigma; //標準偏差ベクトル
  real<lower=-1,upper=1> rho; //相関係数
}
transformed parameters{
  cov_matrix[2] Sig; //共分散行列
  Sig[1,1] <- sigma[1]^2;
  Sig[2,2] <- sigma[2]^2;
  Sig[1,2] <- rho*sigma[1]*sigma[2];
  Sig[2,1] <- rho*sigma[1]*sigma[2];
}
model{
  sigma ~ cauchy(0,5);
  rho ~ uniform(-1,1);
  spd ~ wishart(N-1,Sig); //N-1は自由度
}
```

平均値はパラメータからなくなっているので注意

偏差積和行列にたいして、ウィッシュャート分布を仮定

Rコード

```
model9 <- stan_model("corr2.stan")  
N <- nrow(y)  
data <- list(N=N,spd=cov(y)*(N-1))  
fit9 <- sampling(model9,data=data)  
print(fit9,pars="rho",digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
rho	0.682	0.003	0.111	0.431	0.617	0.696	0.764	0.853	1447	1.002

model8とほぼ同じ結果

多変量に対応したい

- 行列演算を使えば多変量の拡張も簡単
 - ちょっと, コードを理解するのが大変だが...
 - 初心者はとりま「そういうもんだ」と思ってほしい
- `corr_matrix`型を使う
 - 相関行列用の型
 - 対角項が1で, 非対角項は-1~1の範囲
 - 事前分布は, `lkj_corr()`という分布を使う
 - パラメータは1で無情報, 2で弱情報事前分布になる
 - 今回は無情報事前分布を使う

quad_form_diag()

- 相関行列とSDベクトルを共分散行列にする
 - `Sig <- quad_form_diag(rho, sigma);`
 - 行列演算を簡略化した関数

`quad_form_diag(rho, sigma)`

`= diag_matrix(sigma) * rho * diag_matrix(sigma)`

※`diag_matrix()`: 引数を対角項に並べた対角行列

Stanコード レベル10

corr3.stan

```
data{
  int N; //サンプルサイズ
  int M; //変数の数
  matrix[M,M] spd; //偏差積和行列
}
parameters{
  vector<lower=0>[M] sigma; //標準偏差ベクトル
  corr_matrix[M] rho; //相関行列
}
transformed parameters{
  cov_matrix[M] Sig; //共分散行列
  Sig <- quad_form_diag(rho,sigma); //相関行列とSDを共分散行列に変換
}
model{
  sigma ~ cauchy(0,5);
  rho ~ lkj_corr(1); //相関行列の無情報事前分布
  spd ~ wishart(N-1,Sig);
}
```

Rコード

```
model10 <- stan_model("corr3.stan")
N <- nrow(y)
data <- list(N=N,M=2,spd=cov(y)*(N-1))
fit10 <- sampling(model10,data=data)
print(fit10,pars="rho",digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
rho[1,1]	1.000	0.000	0.000	1.000	1.000	1.000	1.000	1.00	4000	NaN
rho[1,2]	0.688	0.003	0.107	0.445	0.624	0.701	0.763	0.86	1306	1.001
rho[2,1]	0.688	0.003	0.107	0.445	0.624	0.701	0.763	0.86	1306	1.001
rho[2,2]	1.000	0.000	0.000	1.000	1.000	1.000	1.000	1.00	3242	0.999

結果はこれまでと一致している

応用的な変数の型 1

- simplex型
 - 総和が1になるvector型の特殊版
 - 主に混合分布モデルの混合率パラメータに使う
- unit_vector型
 - ノルムが(二乗和が)1になるvector型の特殊版
 - ただし, 使う要素+1の要素数を宣言しないといけない
- ordered型
 - 要素が昇順になるように並ぶvector型の特殊版
 - 順序ロジスティック回帰の切片やIRTの閾値パラメータで使う
- positive_ordered型
 - ordered型で, かつ, 正の値をとるような制約を持ったvector型

応用的な変数の型 2

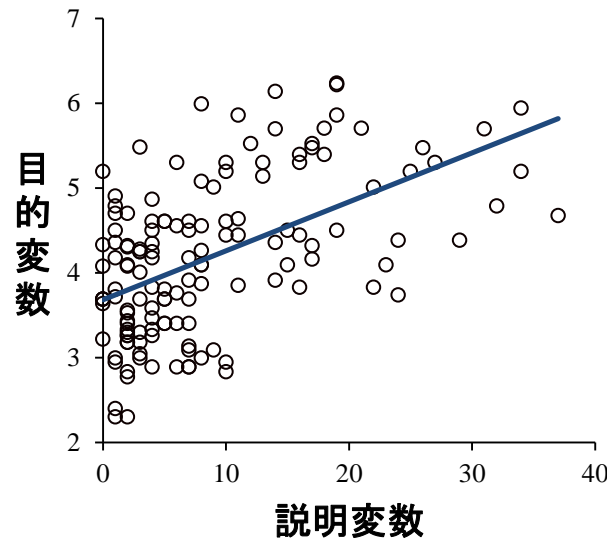
- `cov_matrix` , `corr_matrix`型
 - 共分散, 相関行列の変数型
 - `matrix`型の特殊版 対称正定置行列を仮定
 - 相関行列は対角項がすべて1になっている
- `cholesky_factor_cov`, `cholesky_factor_corr`型
 - 共分散(相関)行列をコレスキー分解したもの
 - 行列演算の高速化のためや, 因子分析の因子負荷量パラメータのために使われる

回歸分析

回帰分析のおさらい

- 目的変数を説明変数で予測

$$- Y_i = \alpha + \beta X_i + e_i$$



- 回帰係数

– 説明変数が1点増加したときの、目的変数の平均的な増加量の推定値

回帰分析の仮定

- データは独立に生成されている
 - 独立性(正確には無相関の仮定)
 - データ内に局所的な相関がない
- データは同じ(分散の)分布から生成されている
 - 同一性(均一分散の仮定)
 - データはすべて同じ(分散の)分布に従っている
- データは正規分布から生成されている
 - 正規性

回帰分析のモデリング

- 単回帰分析

$$Y_i = \alpha + \beta X_i + e_i$$

$$e_i \sim \text{normal}(0, \sigma)$$

- e は Y と予測値 \hat{Y} との差 つまり残差
- 残差が平均=0, SD= σ の正規分布に従う

– パラメータは, 切片 α , 係数 β , 残差SD σ の3つ

- モデリングは・・・

– 残差 e を切片, 係数, データから計算して, それが平均0, SD= σ の正規分布に従うことを記述すればOK

– $e = Y - \hat{Y} = Y - (\alpha + \beta X)$

Stanコード レベル11

reg1.stan

```
data{
  int N; //サンプルサイズ
  vector[N] y; //目的変数
  vector[N] x; //説明変数
}
parameters{
  real alpha; //切片
  real beta; //回帰係数
  real<lower=0> sigma; //残差SD
}
model{
  vector[N] e; //残差をmodel{}内で宣言
  //事前分布
  alpha ~ normal(0,100);
  beta ~ normal(0,100);
  sigma ~ cauchy(0,5);
  //回帰モデル
  for(n in 1:N){
    e[n] <- y[n] - (alpha + beta*x[n]); //残差を計算
  }
  e ~ normal(0,sigma); //残差が正規分布に従う
}
```

今回は、データをvector型で宣言
その理由はレベル13で...

データを作る

- 切片=2, 係数=5, 残差SD=3のyを作る

```
set.seed(123)
```

```
x <- rnorm(100,0,1) #xは標準正規分布から生成
```

```
y <- 2 + 5*x + rnorm(100,0,3) #残差は平均0, SD=3
```

- lm()で確認

```
summary(lm(y~x))
```

```
Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	1.6916	0.2927	5.78	8.87e-08	***
x	4.8426	0.3206	15.10	< 2e-16	***

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 2.912 on 98 degrees of freedom
```

Rコード

```
model11 <- stan_model("reg1.stan")  
data <- list(N=100,y=y,x=x)  
fit11 <- sampling(model11,data=data)  
fit11
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
alpha	1.70	0.01	0.30	1.12	1.49	1.70	1.90	2.28	2715	1
beta	4.83	0.01	0.33	4.19	4.61	4.83	5.05	5.47	2816	1
sigma	2.95	0.00	0.22	2.57	2.79	2.93	3.08	3.42	2704	1
lp__	-156.66	0.03	1.21	-159.79	-157.23	-156.34	-155.75	-155.23	1444	1

パラメータの真値は, $\alpha=2$, $\beta=5$, $\sigma=3$
うまく推定できている

ただし警告が出る

```
reg1.stan is syntactically correct.  
> rstan::rstudio_stanc("reg1.stan")  
DIAGNOSTIC(S) FROM PARSER:  
Warning (non-fatal):  
Left-hand side of sampling statement (~) may contain a non-linear transform  
of a parameter or local variable.  
If so, you need to call increment_log_prob() with the log absolute determinan  
nt of the Jacobian of the transform.  
Left-hand-side of sampling statement:  
    e ~ normal(...)  
  
reg1.stan is syntactically correct.
```

サンプリングステートメント(~)は、データに対して直接指定する分には大丈夫だけど、変換した値(今回は残差e)に対して適用すると、警告がでる

※increment_log_prob()については中級編で解説する

モデリングをちょっと変える

- 単回帰分析(再掲)

$$Y_i = \alpha + \beta X_i + e_i$$

$$e_i \sim \text{normal}(0, \sigma)$$

– パラメータは, 切片 α , 係数 β , 残差SD σ の3つ

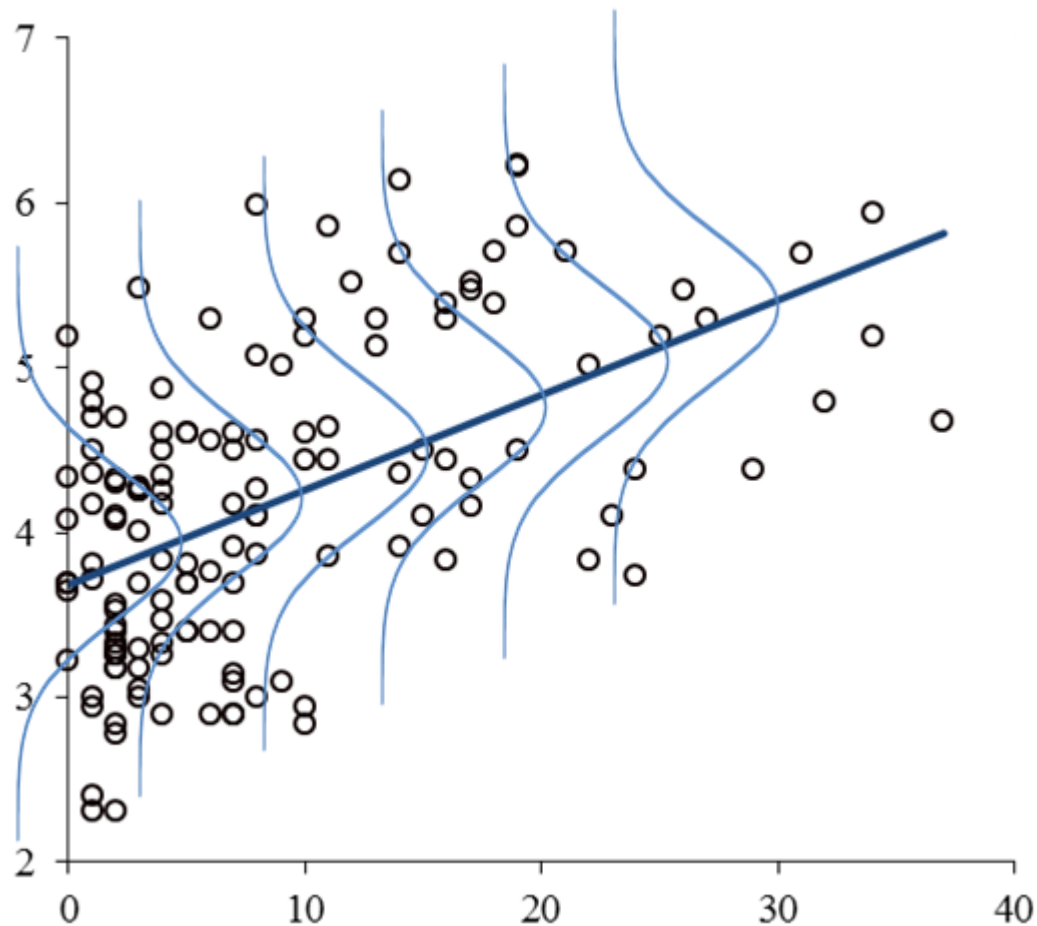
- モデリングを次のようにしてみる

$$\rightarrow Y_i - \hat{Y}_i \sim \text{normal}(0, \sigma)$$

$$\rightarrow Y_i \sim \text{normal}(\hat{Y}_i, \sigma)$$

つまり, 目的変数 Y は, 予測値 \hat{Y} を平均とした正規分布に従う
こうすると, データ Y に対して確率分布を直接当てはめられる

こんなイメージ



各データは、予測値を
平均とした正規分布に
従う

平均が説明変数の値
によって変わる, 条件
付き正規分布

Stanコード レベル12

reg2.stan

```
data{
  int N; //サンプルサイズ
  vector[N] y; //目的変数
  vector[N] x; //説明変数
}
parameters{
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model{
  vector[N] yhat; //予測値をmodel{}内で宣言
  alpha ~ normal(0,100);
  beta ~ normal(0,100);
  sigma ~ cauchy(0,5);
  for(n in 1:N){
    yhat[n] <- alpha+beta*x[n]; //予測値を計算
  }
  y ~ normal(yhat,sigma);
}
```

Rコードと結果

```
model12 <- stan_model("reg2.stan")  
fit12 <- sampling(model12,data=list(N=100,x=x,y=y))  
fit12
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
alpha	1.69	0.01	0.30	1.09	1.50	1.70	1.90	2.26	2773	1
beta	4.85	0.01	0.33	4.19	4.62	4.84	5.07	5.50	2727	1
sigma	2.94	0.00	0.21	2.57	2.79	2.93	3.07	3.38	2269	1
lp__	-156.64	0.04	1.23	-159.85	-157.21	-156.32	-155.72	-155.22	1092	1

コンパイル時にエラーが出ずに, さっき
と同じ結果が得られる

行列演算で予測値を計算したい

- 予測値
 - $\hat{Y} = \alpha + \beta X$
 - \hat{Y} と X は 100 人の値が入ったベクトル
 - これを For 文を使わずにベクトル化して計算
 - ただし, この場合, x は配列で宣言してはいけない
- 説明変数は vector もしくは matrix で宣言
 - 予測値を計算するときに行列演算が必要だから
 - 目的変数はどちらでもいい 今回は vector で宣言

Stanコード レベル13

reg3.stan

```
data{
  int N; //サンプルサイズ
  vector[N] y; //目的変数
  vector[N] x; //説明変数
}
parameters{
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model{
  vector[N] yhat; //予測値をmodel{}内で宣言 ※vectorで
  //事前分布
  alpha ~ normal(0,100);
  beta ~ normal(0,100);
  sigma ~ cauchy(0,5);
  //回帰モデル
  yhat <- alpha+beta*x; //予測値をベクトル計算
  y ~ normal(yhat,sigma);
}
```

xはvector型で宣言
yは今回はrealの配列でもいい
が, xと揃えている

xをvectorではなく配列にすると・・

```
data{  
  int N; //サンプルサイズ  
  real y[N]; //データ1  
  real x[N]; //データ2  
}
```

SYNTAX ERROR, MESSAGE(S) FROM PARSER:

No matches for:

```
  real * real[]
```

ERROR at line 19

```
17:      sigma ~ cauchy(0,5);  
18:  
19:      yhat <- alpha+beta*x;  
20:      y ~ normal(yhat,sigma);
```

怒られる・・

これは, real型であるbetaとrealの配列型のxでは掛け算ができないから

レベル12のようにfor文を使えば可能だが, その分遅くなるし, 長くなる

重回歸分析

重回帰分析

- 説明変数が複数の回帰分析
 - 基本的には, `reg3.stan`を拡張すればいくつでも増やすことができる
- 行列演算を使ってすっきりしたコードにしたい
 - この辺りは線形代数の知識がいるが...
 - 「そういうもんだ」というのでもいいので真似て書いて走らせてみてください

説明変数をmatrixでまとめる

- 切片は全て1のベクトル
 - 切片は、値が全て1をいれた説明変数の係数として推定することができる
 - よって、 α と β という区別はなくなり、 β だけでよい
 - $Y_i = \beta X_i + e_i$
 - X に切片を推定するための、1が並んだベクトルが含まれている
- matrix型で宣言する
 - `matrix[N,M] x;`とする ただし、 M は変数の数+1
 - 切片の分を忘れないように
- 予測値の計算
 - 行列の掛け算はかける向きが重要

データを作る

```
set.seed(123)
```

```
x1 <- rnorm(100,0,1)
```

```
x2 <- rnorm(100,0,1)
```

```
y <- 2 + 5*x1 + 7*x2 + rnorm(100,0,3)
```

```
summary(lm(y~x1+x2))
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	2.4052	0.2884	8.339	5.08e-13	***
x1	4.6005	0.3146	14.623	< 2e-16	***
x2	7.0714	0.2970	23.811	< 2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.854 on 97 degrees of freedom

切片を含んだxを作る

- すべてが1のベクトルを説明変数に加える
intercept <- rep(1,100) #1が100個のベクトル
x <- data.frame(intercept,x1,x2)
head(x)

```
> head(x)
  intercept      x1      x2
1          1 -0.56047565 -0.71040656
2          1 -0.23017749  0.25688371
3          1  1.55870831 -0.24669188
4          1  0.07050839 -0.34754260
5          1  0.12928774 -0.95161857
6          1  1.71506499 -0.04502772
```

Stanコード レベル14

reg4.stan

```
data{
  int N; //サンプルサイズ
  int M; //変数の数(切片含む)
  vector[N] y; //目的変数
  matrix[N,M] x; //説明変数 matrixで宣言
}
parameters{
  vector[M] beta; //vectorで宣言
  real<lower=0> sigma;
}
model{
  beta ~ normal(0,100);
  sigma ~ cauchy(0,5);
  y ~ normal(x*beta,sigma);
}
```

ここでは, yhatを変数として宣言せず, 直接normal()の引数として式を入れている

xとbetaの順番がさっきと逆になっているのに注意

$N \times M$ の行列xに右からM次元ベクトル β をかけると, N次元ベクトルの予測値になる
それはyのベクトルの次元と一致している

Rコード

```
data <- list(N=100, M=3, y=y, x=x) #M=3に注意
model14 <- stan_model("reg4.stan")
fit14 <- sampling(model12, data=data)
print(fit12, probs=c(0.025, 0.5, 0.975))
```

	mean	se_mean	sd	2.5%	50%	97.5%	n_eff	Rhat
beta[1]	2.41	0.01	0.29	1.84	2.41	2.99	2543	1
beta[2]	4.60	0.01	0.32	3.98	4.59	5.24	2937	1
beta[3]	7.07	0.01	0.30	6.49	7.07	7.66	2418	1
sigma	2.89	0.00	0.21	2.51	2.87	3.34	2251	1
lp__	-154.65	0.04	1.47	-158.49	-154.28	-152.85	1529	1

パラメータの真値は、切片=2, 係数=5と7, 残差SDは3
上手く推定できている

ロジスティック回帰

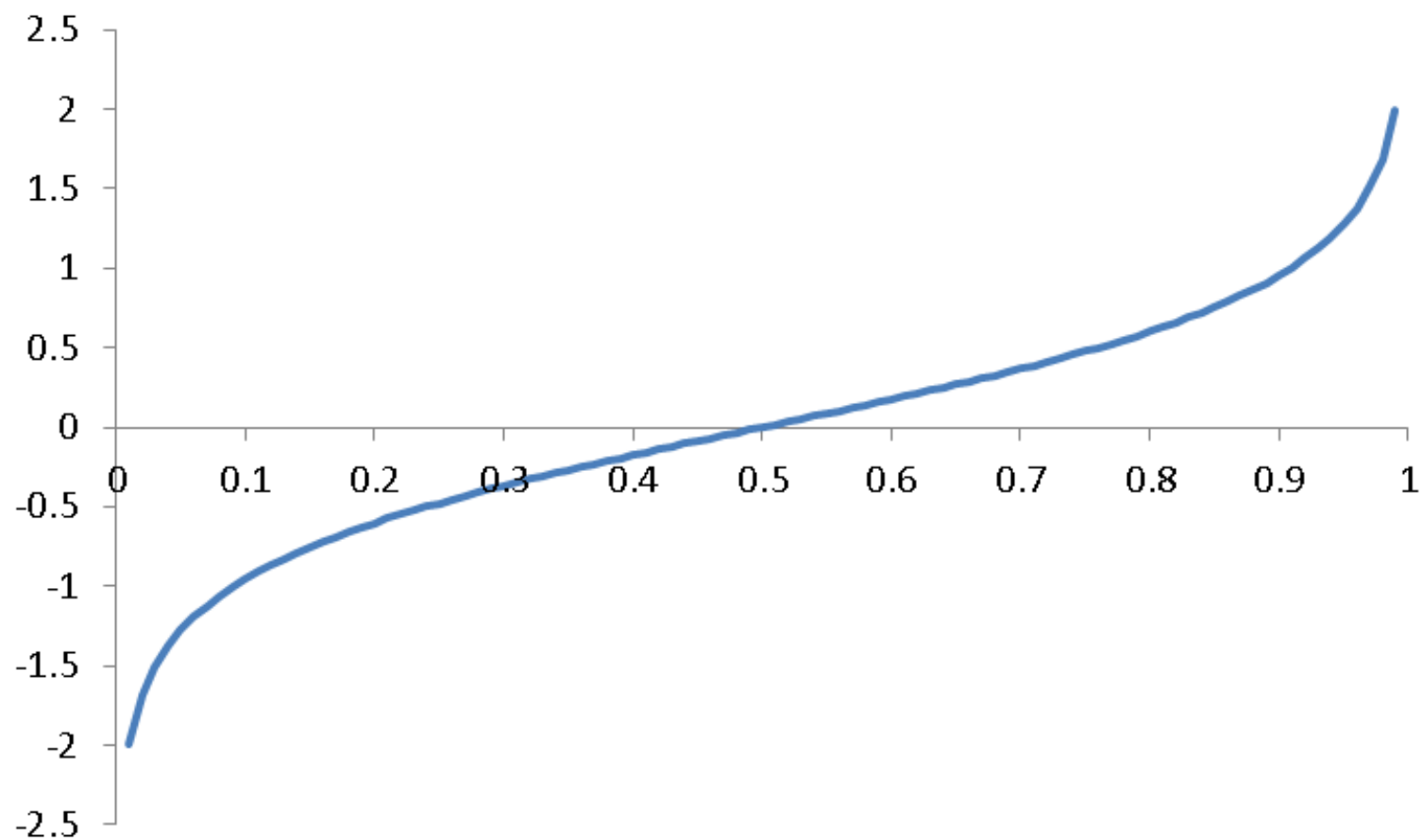
2値データを予測

- 0と1, どちらの確率が高いかを予測
 - 予測したいのは1になる確率 p
 - 確率分布はベルヌーイ分布を使う
- ベルヌーイ分布
 - 試行数が1回だけの二項分布
 - パラメータは成功率 p のみ

pを予測したいが・・・

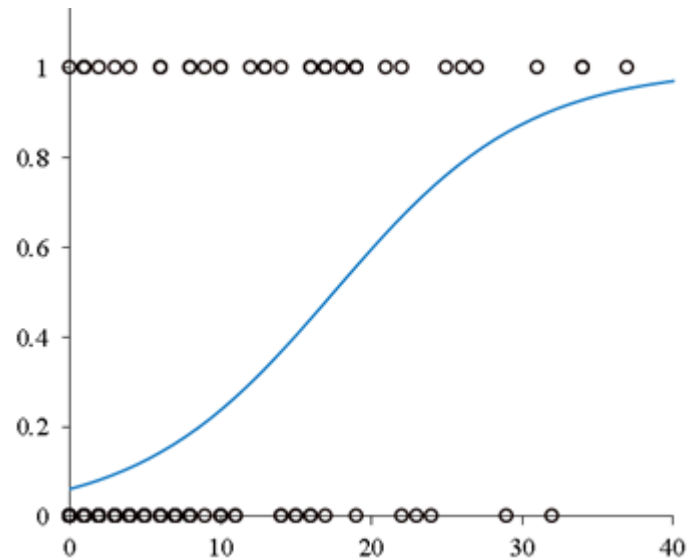
- 確率pは0~1の範囲しかとらない
 - 線形回帰をした場合, 予測値が0~1に収まらないと尤度が計算出来ない
 - 確率pを直接予測するのではなく, ロジット変換をした $\text{logit}(p)$ を予測する
 - $\text{logit}(p_i) = \log(p_i / (1-p_i)) = \alpha + \beta X_i$
- ロジットリンク
 - 0~1の値を, $-\infty \sim \infty$ に変換する非線形変換関数

ロジット変換



ロジスティック回帰

- 確率 p をロジット変換する線形モデル
 - 予測が直線ではなく, 曲線になる
 - この曲線をロジスティック曲線という
 - これをStanで表現する



ロジスティック回帰のモデリング

- 実際にモデリングするときは・・・
 - 確率 p をロジット変換するのではなく
 - 予測値をロジット変換の逆変換を行う
 - つまり, 予測値が0～1の範囲になるようにする
- ロジスティック回帰のモデル式は・・・
 - $p_i = \text{logit}^{-1}(\alpha + \beta X_i)$ ※ $\text{logit}^{-1}()$ は $\text{logit}()$ の逆関数
 - $Y_i \sim \text{bernoulli}(p_i)$
- Stanには`inv_logit()`関数がある
 - これをつかって予測値を変換し, ベルヌーイ分布のパラメータとする
 - $\text{logit}^{-1}(x) = \text{inv_logit}(x) = 1/(1+\exp(-x))$

Stanコード レベル15

logistic1.stan

```
data{
  int N; //サンプルサイズ
  int M; //変数の数(切片含む)
  int y[N]; //目的変数
  matrix[N,M] x; //説明変数 matrixで宣言
}
parameters{
  vector[M] beta; //vectorで宣言
}
transformed parameters{
  vector[N] p; //予測値pを宣言
  for(n in 1:N)
    p[n] <- inv_logit(x[n]*beta); //ロジットの逆変換
}
model{
  beta ~ normal(0,100);
  y ~ bernoulli(p); //ベルヌーイ分布
}
```

For文は処理が1行だけの場合は
{ }を省略することができる

inv_logit()はベクトル化に対応して
いない(たぶん)

データを作る

```
inv_logit <- function(x){  
  z <- 1/(1+exp(-x))  
  return(z)  
}
```

手作り逆ロジット変換の関数
なお, {psych}にlogistic()というものもある

```
set.seed(123)  
x1 <- rnorm(100,0,1)  
x2 <- rnorm(100,0,1)  
p <- inv_logit(1 + 2*x1 + 3*x2)  
y <- rbinom(100,1,p)
```

Rコード

```
intercept <- rep(1,100) #1が100個のベクトル  
x <- data.frame(intercept,x1,x2)  
data <- list(N=100,M=3,y=y,x=x)
```

```
model15 <- stan_model("logistic1.stan")  
fit15 <- sampling(model15,data=data)  
print(fit15,pars="beta",probs=c(0.025,0.5,0.975))
```

	mean	se_mean	sd	2.5%	50%	97.5%	n_eff	Rhat
beta[1]	1.72	0.01	0.51	0.81	1.68	2.82	1175	1
beta[2]	2.55	0.02	0.60	1.49	2.52	3.85	1025	1
beta[3]	3.81	0.03	0.85	2.37	3.75	5.72	959	1

bernoulli_logit()を使う

- Stanには変数変換を含んだ確率分布がある
 - パラメータが確率 p ではなく, ロジット変換された $\text{logit}(p)$ であるようなベルヌーイ分布
 - 他にも `poisson_log()` や `binomial_logit()` などがある
 - これらの分布を使ったほうが, ベクトル化もできて, 処理が早くなる

Stanコード レベル16

logistic2.stan

```
data{
  int N; //サンプルサイズ
  int M; //変数の数(切片含む)
  int y[N]; //目的変数
  matrix[N,M] x; //説明変数 matrixで宣言
}
parameters{
  vector[M] beta; //vectorで宣言
}
model{
  beta ~ normal(0,100);
  y ~ bernoulli_logit(x*beta);
}
```

こちらのほうがコードが短くなり、
また推定も速い

まとめ

Stanで統計モデリング

- 今回は初心者向けということで…
 - 他の統計パッケージで可能な分析を扱った
- Stanの真骨頂は，自由なモデリング
 - 他の統計パッケージではまだ対応していないモデルや，細やかなパラメータ制約などが魅力
 - 多様な確率分布が扱えるのもポイント
- 中級編ではそのあたりをやります

中級編(予定)

- 一般化線形混合モデル(GLMM)
 - 階層事前分布・階層ベイズモデル
- 項目反応理論
 - 潜在変数を含んだモデル
- 因子分析
 - 潜在変数が多変数の場合のモデル
- クラスター分析・混合分布モデル
 - `increment_log_pro()`を使うコードの記述
- その他要望があれば@simizu706まで

Enjoy, Stan!

@simizu706