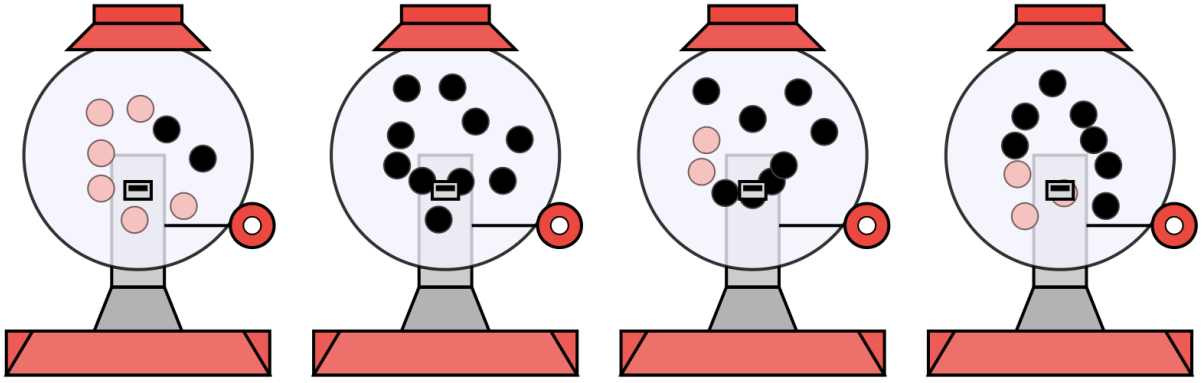
Computational Cognitive Science 2025-2026

Tutorial 6: Multi-armed bandits



Bandits

Today's tutorial is a tour of a classic multi-arm Bernoulli bandit task and some of the models we discussed in lecture.

First, let's define a function that will try a policy with a particular multi-armed bandit problem, and return the mean reward as well as a dataframe containing all choices and rewards.

```
banditTask <- function(policy,horizon,armProbs) {
  nArms <- length(armProbs)
  choices <- c()
  rewards <- c()
  for(t in 1:horizon) {
    nextChoice <- policy(choices,rewards,nArms) # Use policy ot make a choice
    nextReward <- rbinom(1,1,armProbs[nextChoice]) # Get reward given the actual arm probabilities
    choices <- append(choices,nextChoice) # Record choice
    rewards <- append(rewards,nextReward) # Record Reward
  }
  c(mean(rewards),data.frame(choices=choices,rewards=rewards)) # Mean reweards and reward history
}
```

Question 1: What is the `horizon` argument? Our bandit policies are not given the value of horizon. Would we expect this to be the case for all bandit policies we discussed? If so, why? If not, which one(s) need to be aware of the horizon?

Next, we will define three policies: random, win stay loose shift (WSLS), and greedy.

A policy $\pi(a|s)$ is a mapping from a state s to an action a. The state in the case of a bandit task is the record of all previous actions and obtained rewards.

```
randomPolicy <- function(choices,rewards,nArms) {
  # sample a random arm
```

```
  sample(1:nArms,1,replace=TRUE,prob=rep(1/nArms,nArms))
}

wslsPolicy <- function(choices,rewards,nArms) {
  if(length(choices) == 0) {
    # if there was no prior choice, sample a random arm
    sample(1:nArms,1,replace=TRUE,prob=rep(1/nArms,nArms))
  } else {
    # if there was a prior choice stay if it involved a reward,
    # ... sample another option randomly otherwise
    lastReward <- rewards[length(rewards)]
    lastChoice <- choices[length(choices)]
    if(lastReward==1) {
      lastChoice
    } else {
      options <- (1:nArms)[-lastChoice]
      sample(options,1,replace=TRUE,prob=rep(1/(nArms-1),nArms-1))
    }
  }
}

# - A lazy implementation; recomputing counts is inefficient
# - This is a policy generator, which returns a policy with particular
#   hyperparameters alpha, beta of the Beta prior over each arm.
greedyPolicy <- function(alpha,beta) {
  function(choices,rewards,nArms) {
    ratios <- rep(0,nArms)
    for(i in 1:nArms) {
      num <- sum(rewards[choices==i])+alpha
      den <- sum(choices==i)+alpha+beta
      ratios[i] <- num/den
    }

    validArms <- which(ratios == max(ratios))
    nV <- length(validArms)
    if(nV == 1) {
      validArms
    } else {
      sample(validArms,1,prob=rep(1/nV,nV))
    }
  }
}
```

Question 2: The greedy policy has hyperparameters alpha and beta. Why can't we just initialize all ratios with 0 before any choices were made?

Question 3: What does the ratio parameter correspond to in the greedy policy (hint: have a look at the Beta distribution)?

Question 4: What are the memory requirements for each policy? How does the cost of computing the next choice change with the number of trials?

Let's take a Bernoulli bandit task with reward probabilities of 0, 1/4, 1/2, and 3/4, and see how WSLS compares to a greedy policy. Here's a single run, with a random policy for comparison (though we can straightforwardly compute the expected reward of a random policy).

```
exampleArms <- c(0,.25,.5,.75)

randRes <- banditTask(randomPolicy,2000,exampleArms)[[1]]
wslsRes <- banditTask(wslsPolicy,2000,exampleArms)[[1]]
greedyRes <- banditTask(greedyPolicy(1,1),2000,exampleArms)[[1]]

randRes
```

```
## [1] 0.374
```

```
wslsRes
```

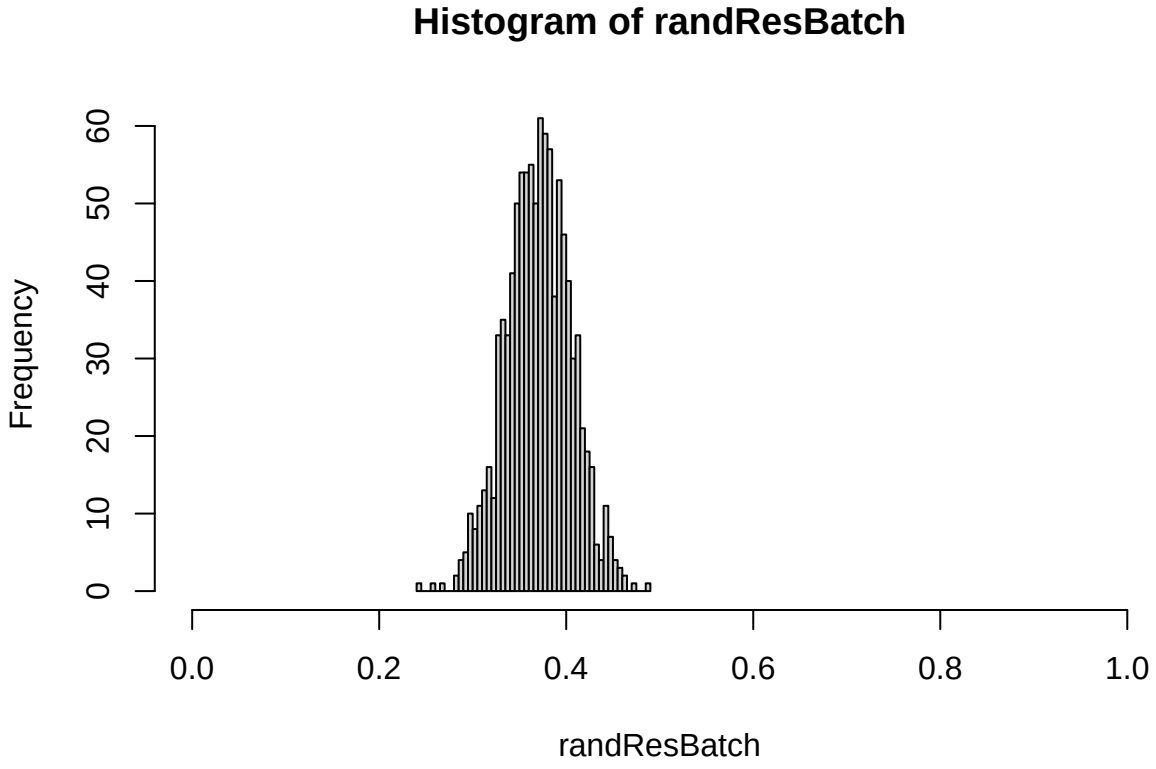
```
## [1] 0.521
```

```
greedyRes
```

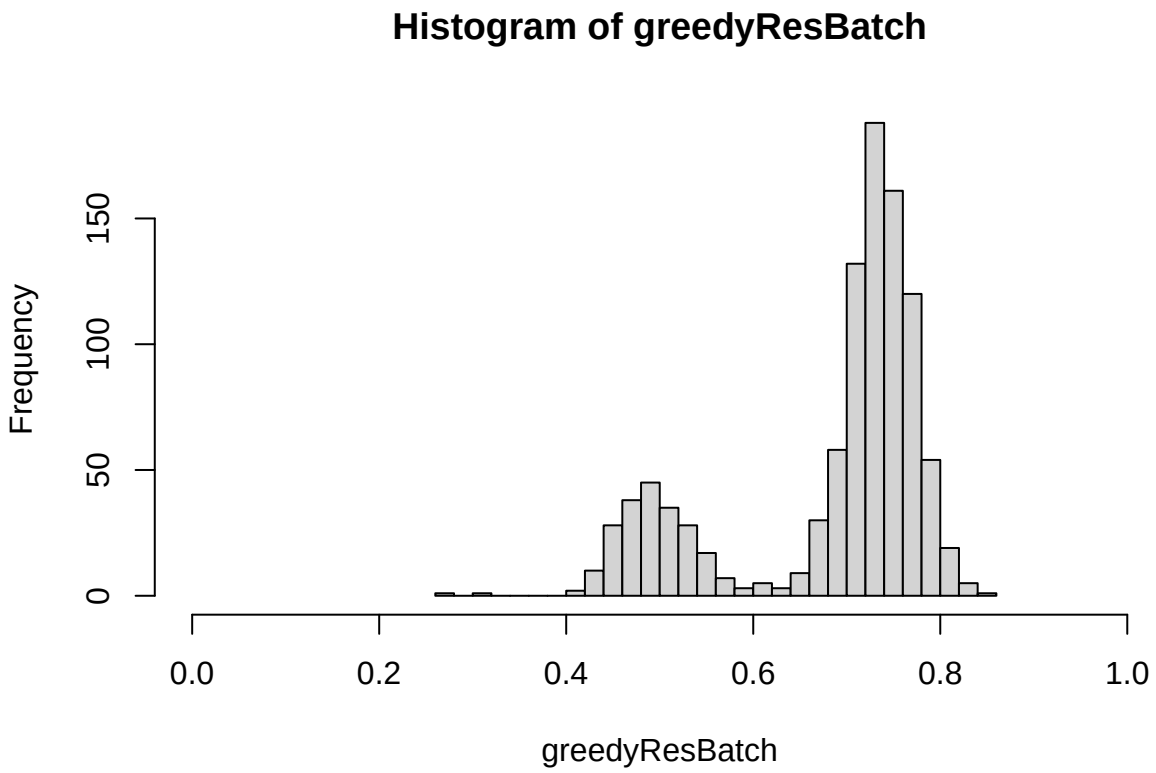
```
## [1] 0.753
```

Now let's look at distributions of reward over several runs of the task, with 200 trials each.

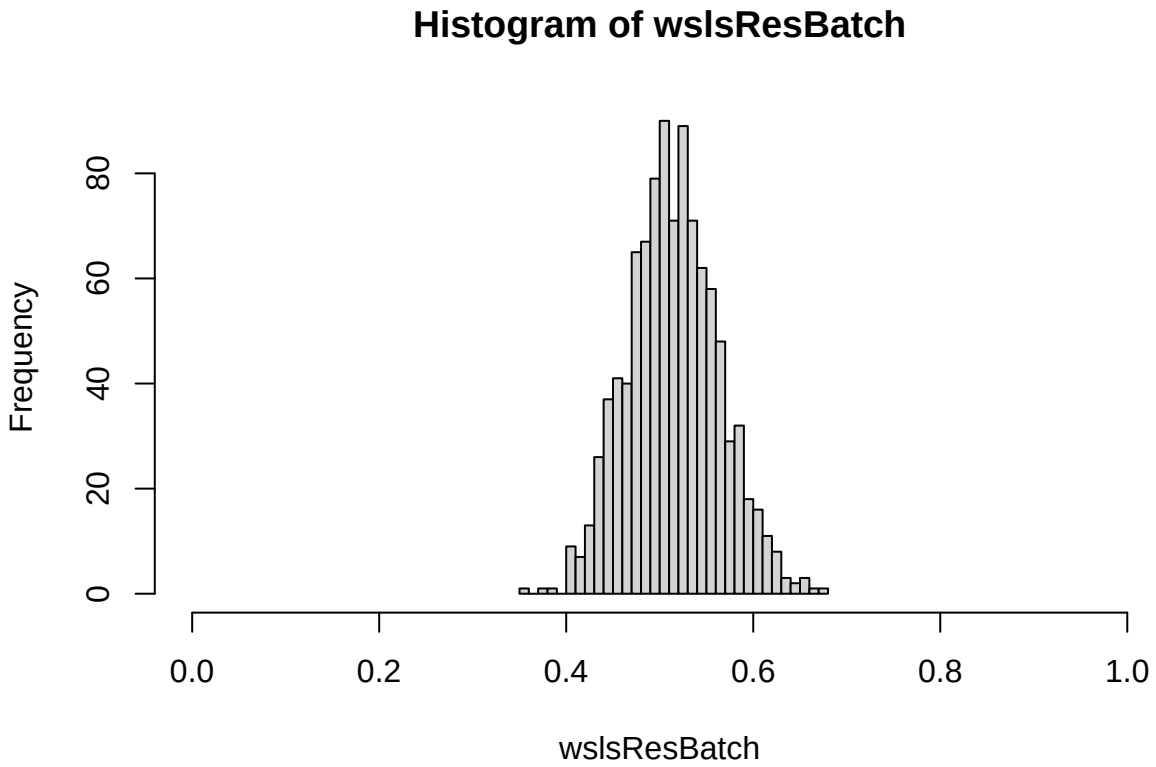
```
nRuns <- 1000
randResBatch <- numeric(nRuns)
greedyResBatch <- numeric(nRuns)
wslsResBatch <- numeric(nRuns)

for(i in 1:nRuns) {
  randResBatch[i] <- banditTask(randomPolicy,200,exampleArms)[[1]]
  greedyResBatch[i] <- banditTask(greedyPolicy(1,1),200,exampleArms)[[1]]
  wslsResBatch[i] <- banditTask(wslsPolicy,200,exampleArms)[[1]]
}

hist(randResBatch,breaks=40, xlim=c(0,1))
```



```
hist(greedyResBatch,breaks=40, xlim=c(0,1))
```




```
hist(wslsResBatch,breaks=40, xlim=c(0,1))
```

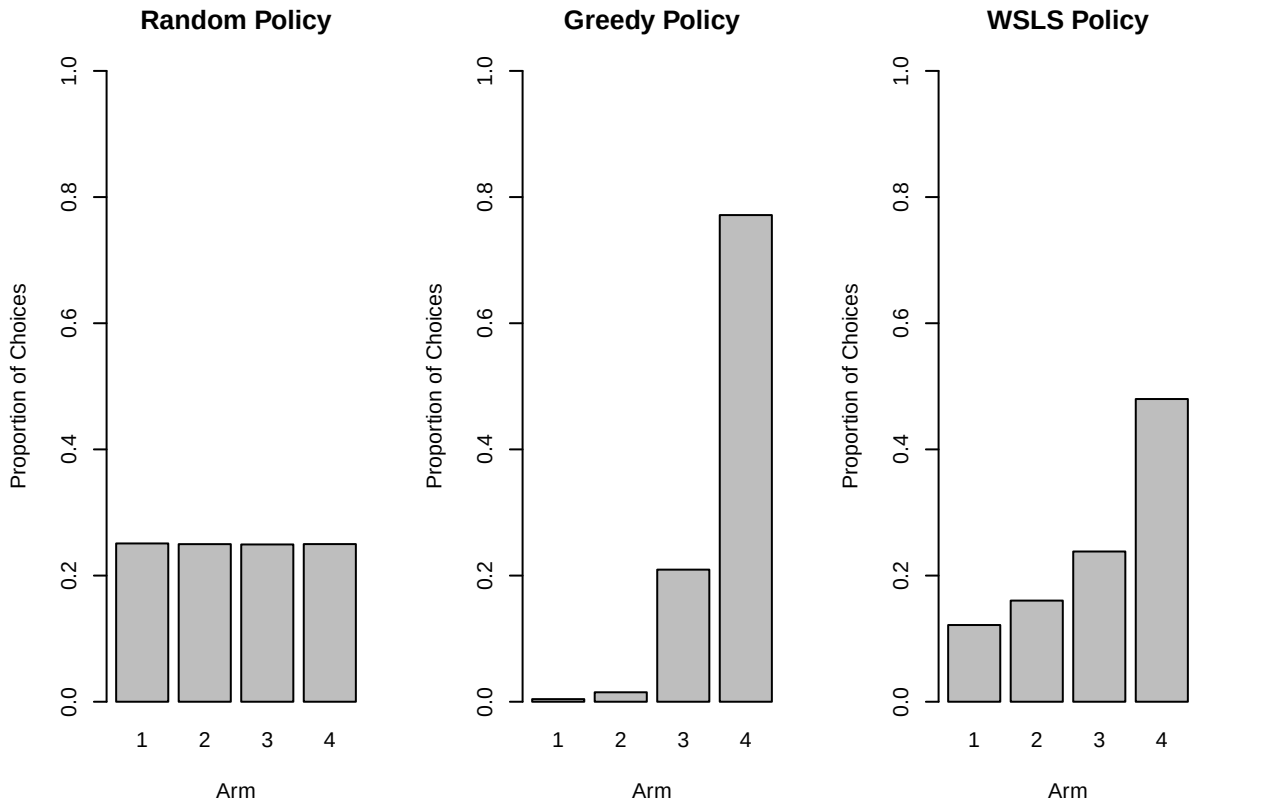


Question 5: Is it surprising that greedy policy outperformed WSLS? Give an example to illustrate why or

why not.

Plot the choice distribution:

```
  # Store full results for each run
  randResBatchFull <- list()
  greedyResBatchFull <- list()
  wslsResBatchFull <- list()

  for(i in 1:nRuns) {
    randResBatchFull[[i]] <- banditTask(randomPolicy,200,exampleArms)
    greedyResBatchFull[[i]] <- banditTask(greedyPolicy(1,1),200,exampleArms)
    wslsResBatchFull[[i]] <- banditTask(wslsPolicy,200,exampleArms)
  }


# Sum the total choices for each arm across all runs
# Beware the error indexing the list elements: Error in x[[2]]$choices : $ operator is invalid for atom
randResChoices <- lapply(randResBatchFull, function(x) x$choices)
greedyResChoices <- lapply(greedyResBatchFull, function(x) x$choices)
wslsResChoices <- lapply(wslsResBatchFull, function(x) x$choices)
randCounts <- table(factor(unlist(randResChoices), levels=1:4))
greedyCounts <- table(factor(unlist(greedyResChoices), levels=1:4))
wslsCounts <- table(factor(unlist(wslsResChoices), levels=1:4))


# Plot as bar plots showing proportion of choices for each arm
par(mfrow=c(1,3))
barplot(randCounts / sum(randCounts), names.arg=1:4, ylim=c(0,1),
  main="Random Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(greedyCounts / sum(greedyCounts), names.arg=1:4, ylim=c(0,1),
  main="Greedy Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(wslsCounts / sum(wslsCounts), names.arg=1:4, ylim=c(0,1),
  main="WSLS Policy", xlab="Arm", ylab="Proportion of Choices")
```



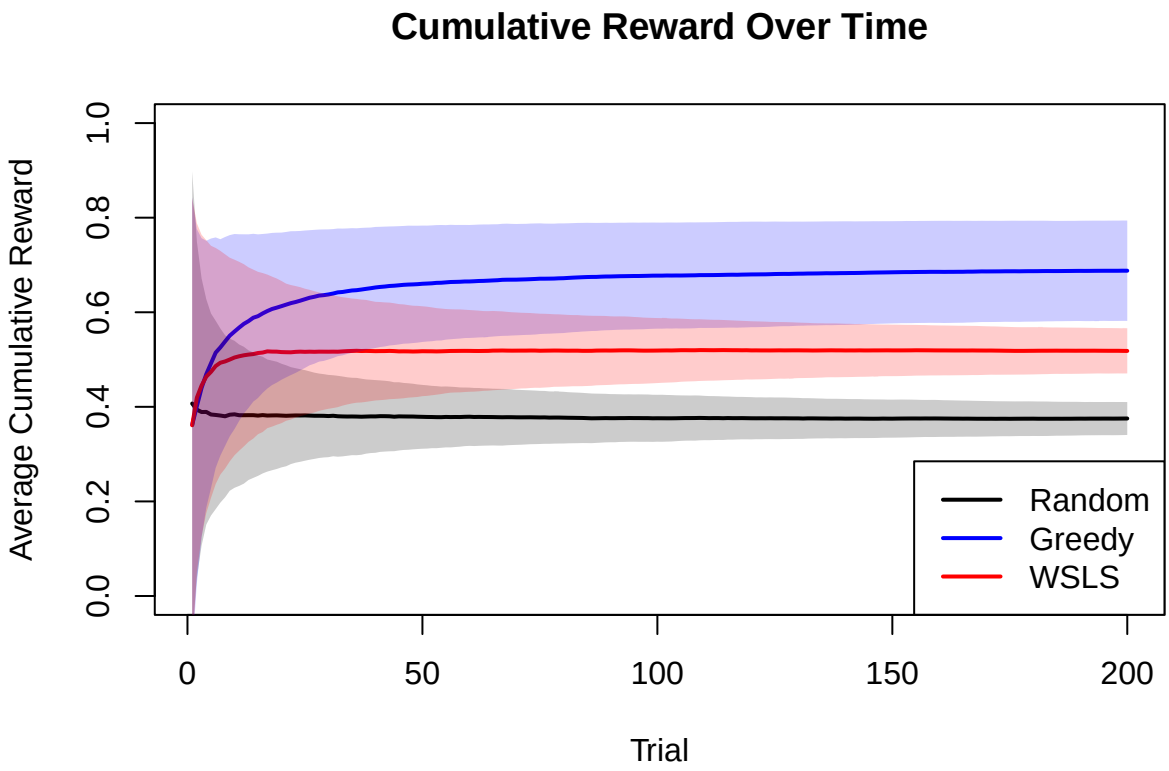

```
par(mfrow=c(1,1))
```

Lets plot the full reward trajectory

```
# Cululative reward of each policy over time
randCumRewards <- matrix(0, nrow=nRuns, ncol=200)
greedyCumRewards <- matrix(0, nrow=nRuns, ncol=200)
wslsCumRewards <- matrix(0, nrow=nRuns, ncol=200)
for(i in 1:nRuns) {
  randRewards <- randResBatchFull[[i]]$rewards
  greedyRewards <- greedyResBatchFull[[i]]$rewards
  wslsRewards <- wslsResBatchFull[[i]]$rewards
  randCumRewards[i, ] <- cumsum(randRewards) / (1:200)
  greedyCumRewards[i, ] <- cumsum(greedyRewards) / (1:200)
  wslsCumRewards[i, ] <- cumsum(wslsRewards) / (1:200)
}
```

```
# Calculate mean and standard error for each time point
randMean <- colMeans(randCumRewards)
greedyMean <- colMeans(greedyCumRewards)
wslsMean <- colMeans(wslsCumRewards)

randSD <- apply(randCumRewards, 2, sd)# / sqrt(nRuns)
greedySD <- apply(greedyCumRewards, 2, sd)# / sqrt(nRuns)
wslsSD <- apply(wslsCumRewards, 2, sd) #/ sqrt(nRuns)

# Create the plot
plot(1:200, randMean, type='l', col='black', lwd=2, ylim=c(0, 1),
  xlab='Trial', ylab='Average Cumulative Reward',
  main='Cumulative Reward Over Time')
```

```
lines(1:200, greedyMean, col='blue', lwd=2)
lines(1:200, wslsMean, col='red', lwd=2)

# Add shaded regions for standard dev
polygon(c(1:200, 200:1), c(randMean + randSD, rev(randMean - randSD)),
     col=rgb(0,0,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(greedyMean + greedySD, rev(greedyMean - greedySD)),
     col=rgb(0,0,1,0.2), border=NA)
polygon(c(1:200, 200:1), c(wslsMean + wslsSD, rev(wslsMean - wslsSD)),
     col=rgb(1,0,0,0.2), border=NA)

# Add legend
legend('bottomright', legend=c('Random', 'Greedy', 'WSLS'),
    col=c('black', 'blue', 'red'), lwd=2)
```

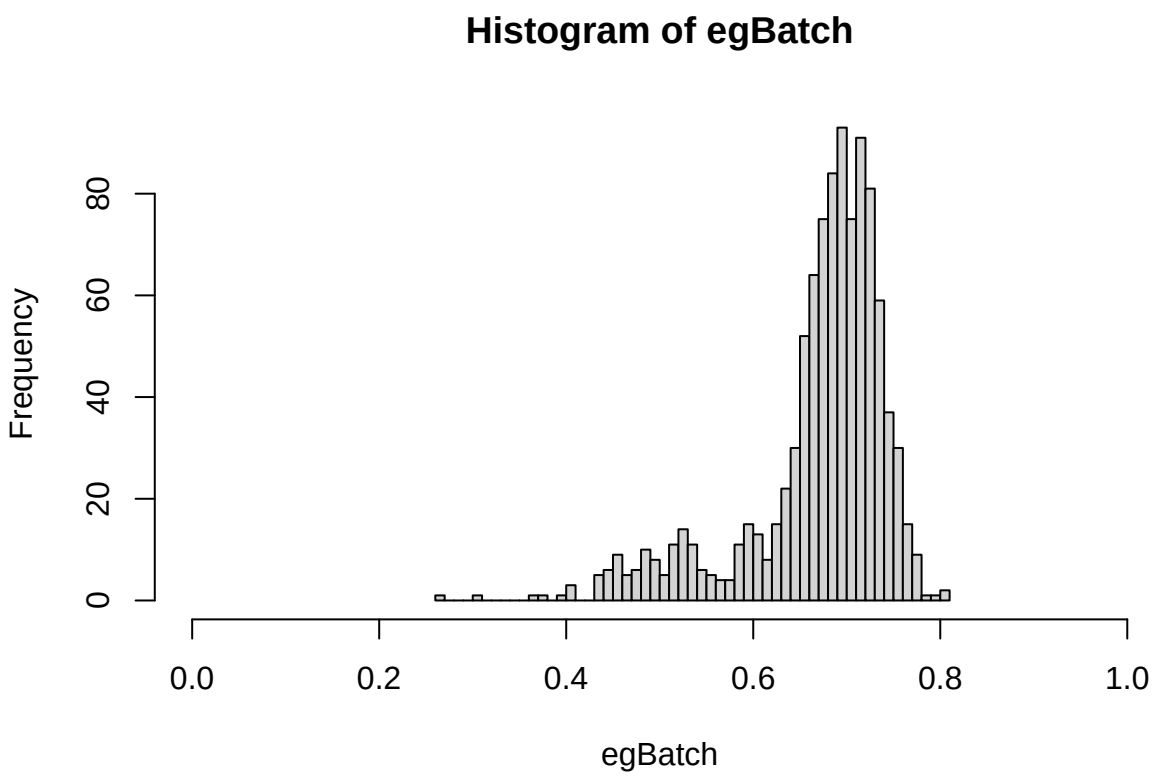


Question 6: Why are there multiple modes in the histogram for the greedy strategy? What might we do to make the worse modes disappear?

Question 7: How would we implement an ϵ-greedy model without writing much new code?

```
epsilonGreedyPolicy <- function(innerGreedy,epsilon) {
  function(choices,rewards,nArms) {
    if(runif(1) < epsilon) {
      randomPolicy(choices,rewards,nArms)
    } else {
      innerGreedy(choices,rewards,nArms)
    }
  }
}
myEpsGreedy <- epsilonGreedyPolicy(greedyPolicy(0.001,0.001),.1)
```

```r
egBatch <- numeric(nRuns)

for(i in 1:nRuns) {
  egBatch[i] <- banditTask(myEpsGreedy,200,exampleArms)[[1]]
}

hist(egBatch,breaks=40, xlim=c(0,1))
```




```r
mean(egBatch)
```

```
## [1] 0.67263
```

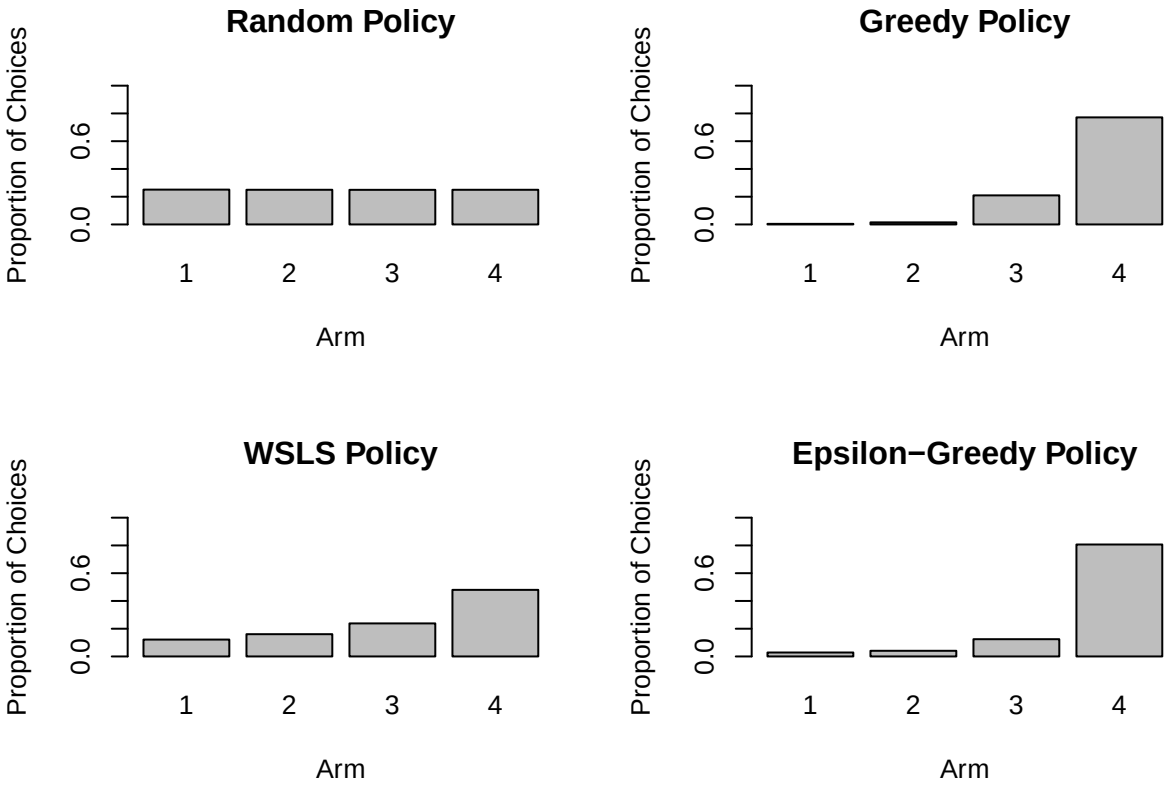
```r
mean(greedyResBatch)
```

```
## [1] 0.686595
```

Adding epsilon greedy to the comparisons:

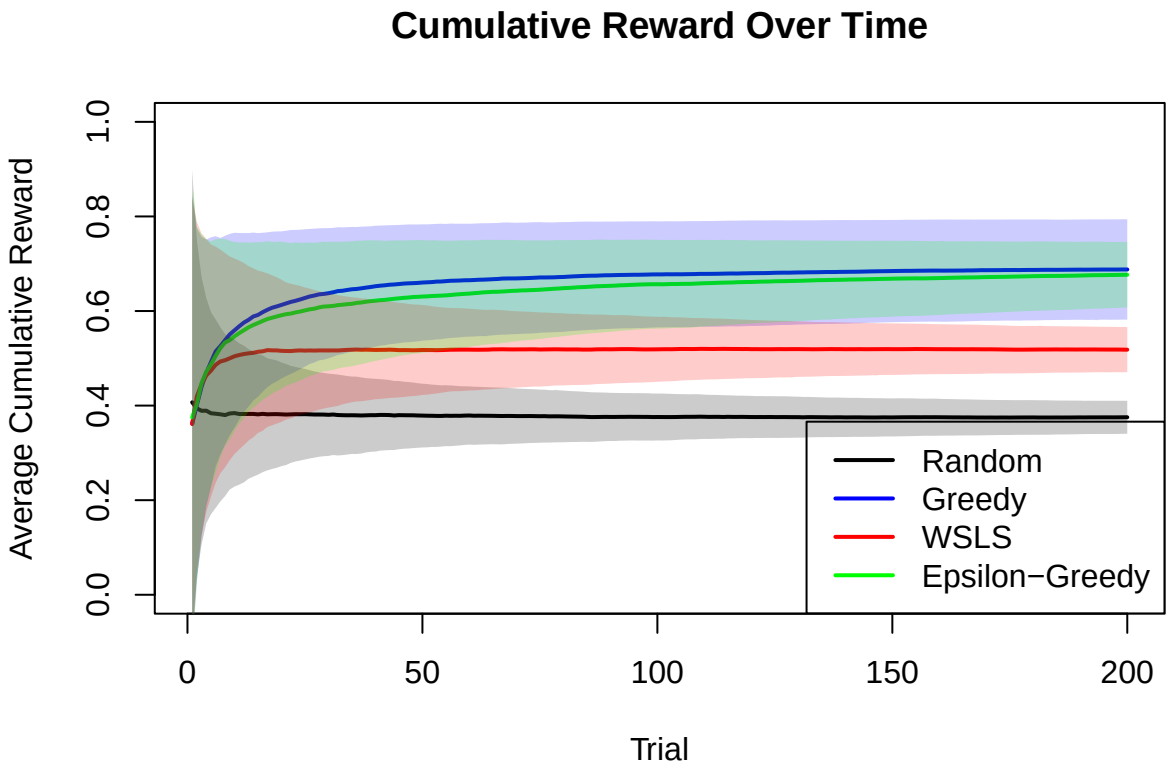
```r
# Store full results for epsilon-greedy runs
egResBatchFull <- list()

for(i in 1:nRuns) {
  egResBatchFull[[i]] <- banditTask(myEpsGreedy,200,exampleArms)
}

# Get choices for epsilon-greedy
egResChoices <- lapply(egResBatchFull, function(x) x$choices)
egCounts <- table(factor(unlist(egResChoices), levels=1:4))

# Plot choice distributions including epsilon-greedy
par(mfrow=c(2,2))
```

```r
barplot(randCounts / sum(randCounts), names.arg=1:4, ylim=c(0,1),
  main="Random Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(greedyCounts / sum(greedyCounts), names.arg=1:4, ylim=c(0,1),
  main="Greedy Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(wslsCounts / sum(wslsCounts), names.arg=1:4, ylim=c(0,1),
  main="WSLS Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(egCounts / sum(egCounts), names.arg=1:4, ylim=c(0,1),
  main="Epsilon-Greedy Policy", xlab="Arm", ylab="Proportion of Choices")
```




```r
par(mfrow=c(1,1))

# Calculate cumulative rewards for epsilon-greedy
egCumRewards <- matrix(0, nrow=nRuns, ncol=200)
for(i in 1:nRuns) {
  egRewards <- egResBatchFull[[i]]$rewards
  egCumRewards[i, ] <- cumsum(egRewards) / (1:200)
}

# Calculate mean and standard deviation
egMean <- colMeans(egCumRewards)
egSD <- apply(egCumRewards, 2, sd)

# Create updated cumulative reward plot
plot(1:200, randMean, type='l', col='black', lwd=2, ylim=c(0, 1),
  xlab='Trial', ylab='Average Cumulative Reward',
  main='Cumulative Reward Over Time')
lines(1:200, greedyMean, col='blue', lwd=2)
lines(1:200, wslsMean, col='red', lwd=2)
lines(1:200, egMean, col='green', lwd=2)
```

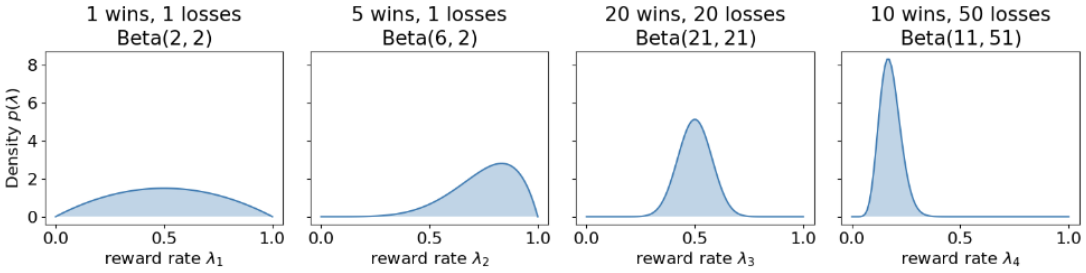
```r
# Add shaded regions
polygon(c(1:200, 200:1), c(randMean + randSD, rev(randMean - randSD)),
     col=rgb(0,0,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(greedyMean + greedySD, rev(greedyMean - greedySD)),
     col=rgb(0,0,1,0.2), border=NA)
polygon(c(1:200, 200:1), c(wslsMean + wslsSD, rev(wslsMean - wslsSD)),
     col=rgb(1,0,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(egMean + egSD, rev(egMean - egSD)),
     col=rgb(0,1,0,0.2), border=NA)

# Add legend
legend('bottomright', legend=c('Random', 'Greedy', 'WSLS', 'Epsilon-Greedy'),
    col=c('black', 'blue', 'red', 'green'), lwd=2)
```



Question 8: In the lecture, we discussed Thompson sampling as a viable alternative. How would you modify the code below to implement Thomposon sampling for our case of Bernoulli bandits?

Approach 5: Thompson sampling (Bernoulli bandits)

- The reward distribution for each option i is a Bernoulli distribution $p(r|\lambda_i)$ with reward rate λ_i
- For each option, we maintain a belief over the reward rate given our observations so far $p(\lambda_i|D_{1:t-1})$. We use a Beta distribution.



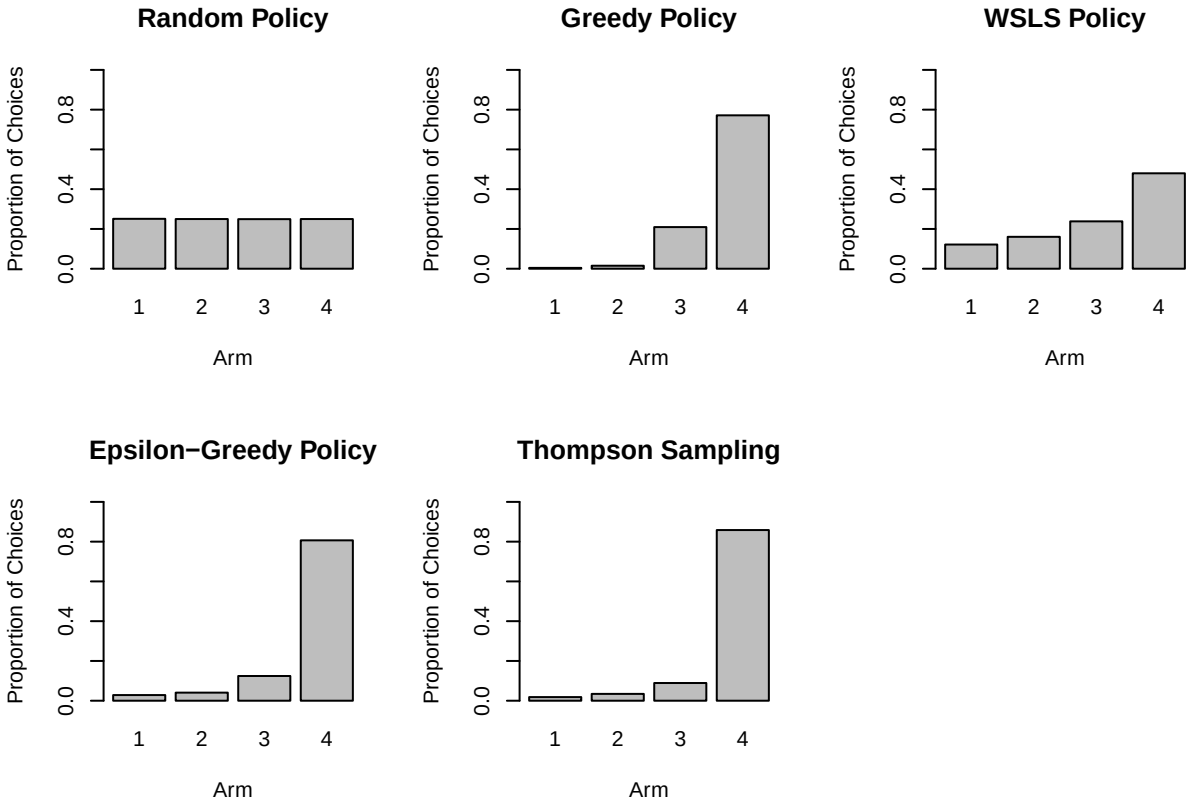
- To make a choice at time t, we sample reward distribution parameters $\lambda_i^{(t)} \sim p(\lambda_i|D_{1:t-1})$.
- We now pick that action that maximizes our expected reward. In the Bernoulli case, this is simply the choice with the highest $\lambda_i^{(t)}$

```
# - A lazy implementation; recomputing counts is inefficient
# - This is a policy generator, which returns a policy with particular
#   hyperparameters alpha, beta of the Beta prior over each arm.
thompsonSampling <- function(alpha,beta) {
  function(choices,rewards,nArms) {
    successes <- rep(0,nArms)
    failures <- rep(0,nArms)
    for(i in 1:nArms) {
      successes[i] <- sum(rewards[choices==i])+alpha
      failures[i] <- sum(rewards[choices==i]==0)+beta
    }

    # YOUR CODE GOES HERE. Remember: you can sample from a beta distribution with rbeta
    lambdas <- rep(0, nArms)
    for (i in 1:nArms) {
      lambdas[i] <- rbeta(1, successes[i], failures[i])
    }
    which(lambdas == max(lambdas))
  }
}
```

Now add Thompson sampling to the comparisons:

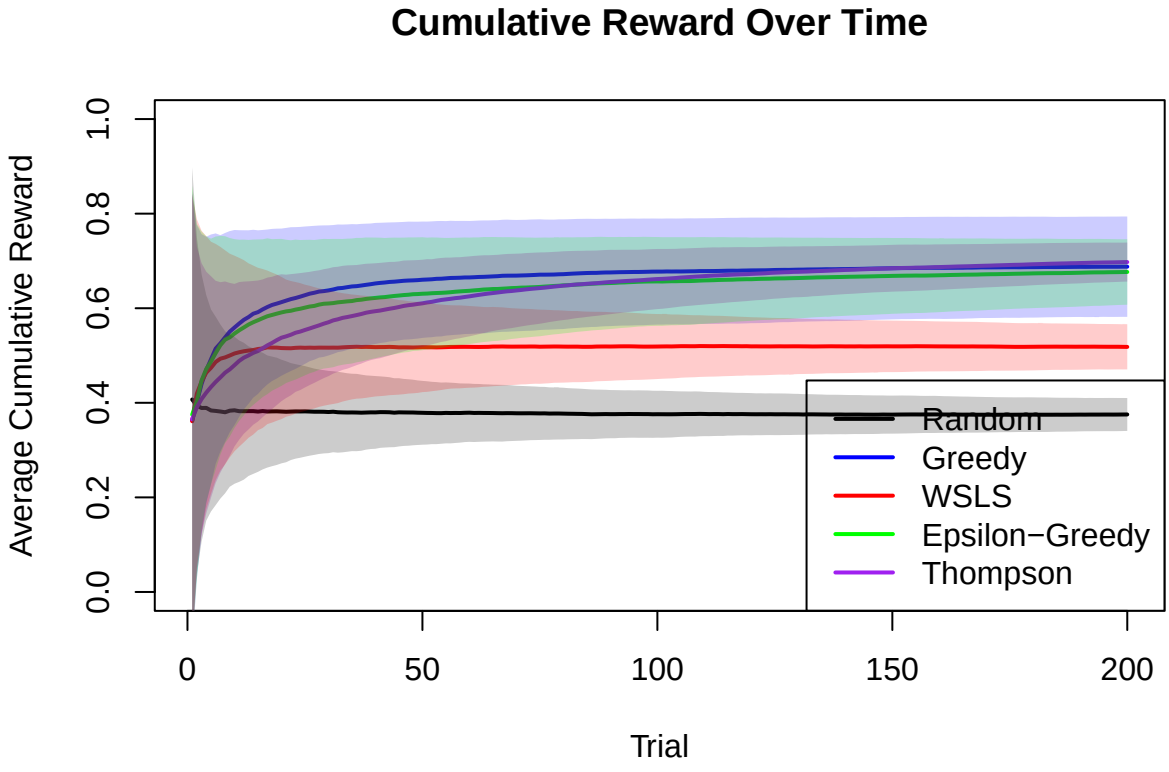
```
# Store full results for Thompson sampling runs
thompsonResBatchFull <- list()
```

```r
for(i in 1:nRuns) {
  thompsonResBatchFull[[i]] <- banditTask(thompsonSampling(1,1),200,exampleArms)
}

# Get choices for Thompson sampling
thompsonResChoices <- lapply(thompsonResBatchFull, function(x) x$choices)
thompsonCounts <- table(factor(unlist(thompsonResChoices), levels=1:4))

# Plot choice distributions including Thompson sampling
par(mfrow=c(2,3))
barplot(randCounts / sum(randCounts), names.arg=1:4, ylim=c(0,1),
  main="Random Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(greedyCounts / sum(greedyCounts), names.arg=1:4, ylim=c(0,1),
  main="Greedy Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(wslsCounts / sum(wslsCounts), names.arg=1:4, ylim=c(0,1),
  main="WSLS Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(egCounts / sum(egCounts), names.arg=1:4, ylim=c(0,1),
  main="Epsilon-Greedy Policy", xlab="Arm", ylab="Proportion of Choices")
barplot(thompsonCounts / sum(thompsonCounts), names.arg=1:4, ylim=c(0,1),
  main="Thompson Sampling", xlab="Arm", ylab="Proportion of Choices")
par(mfrow=c(1,1))
```



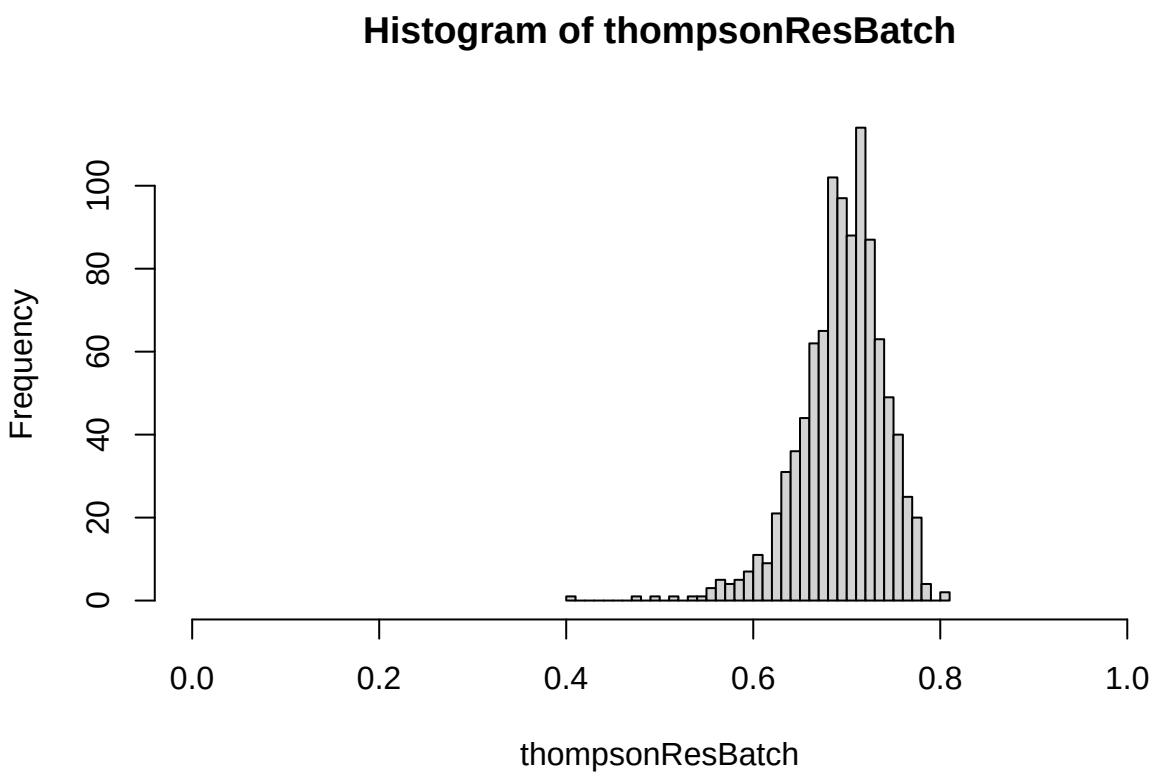

```r
# Calculate cumulative rewards for Thompson sampling
thompsonCumRewards <- matrix(0, nrow=nRuns, ncol=200)
for(i in 1:nRuns) {
  thompsonRewards <- thompsonResBatchFull[[i]]$rewards
  thompsonCumRewards[i, ] <- cumsum(thompsonRewards) / (1:200)
}
```

```r
# Calculate mean and standard deviation
thompsonMean <- colMeans(thompsonCumRewards)
thompsonSD <- apply(thompsonCumRewards, 2, sd)

# Create updated cumulative reward plot
plot(1:200, randMean, type='l', col='black', lwd=2, ylim=c(0, 1),
  xlab='Trial', ylab='Average Cumulative Reward',
  main='Cumulative Reward Over Time')
lines(1:200, greedyMean, col='blue', lwd=2)
lines(1:200, wslsMean, col='red', lwd=2)
lines(1:200, egMean, col='green', lwd=2)
lines(1:200, thompsonMean, col='purple', lwd=2)

# Add shaded regions
polygon(c(1:200, 200:1), c(randMean + randSD, rev(randMean - randSD)),
     col=rgb(0,0,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(greedyMean + greedySD, rev(greedyMean - greedySD)),
     col=rgb(0,0,1,0.2), border=NA)
polygon(c(1:200, 200:1), c(wslsMean + wslsSD, rev(wslsMean - wslsSD)),
     col=rgb(1,0,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(egMean + egSD, rev(egMean - egSD)),
     col=rgb(0,1,0,0.2), border=NA)
polygon(c(1:200, 200:1), c(thompsonMean + thompsonSD, rev(thompsonMean - thompsonSD)),
     col=rgb(0.5,0,0.5,0.2), border=NA)

# Add legend
legend('bottomright', legend=c('Random', 'Greedy', 'WSLS', 'Epsilon-Greedy', 'Thompson'),
    col=c('black', 'blue', 'red', 'green', 'purple'), lwd=2)
```

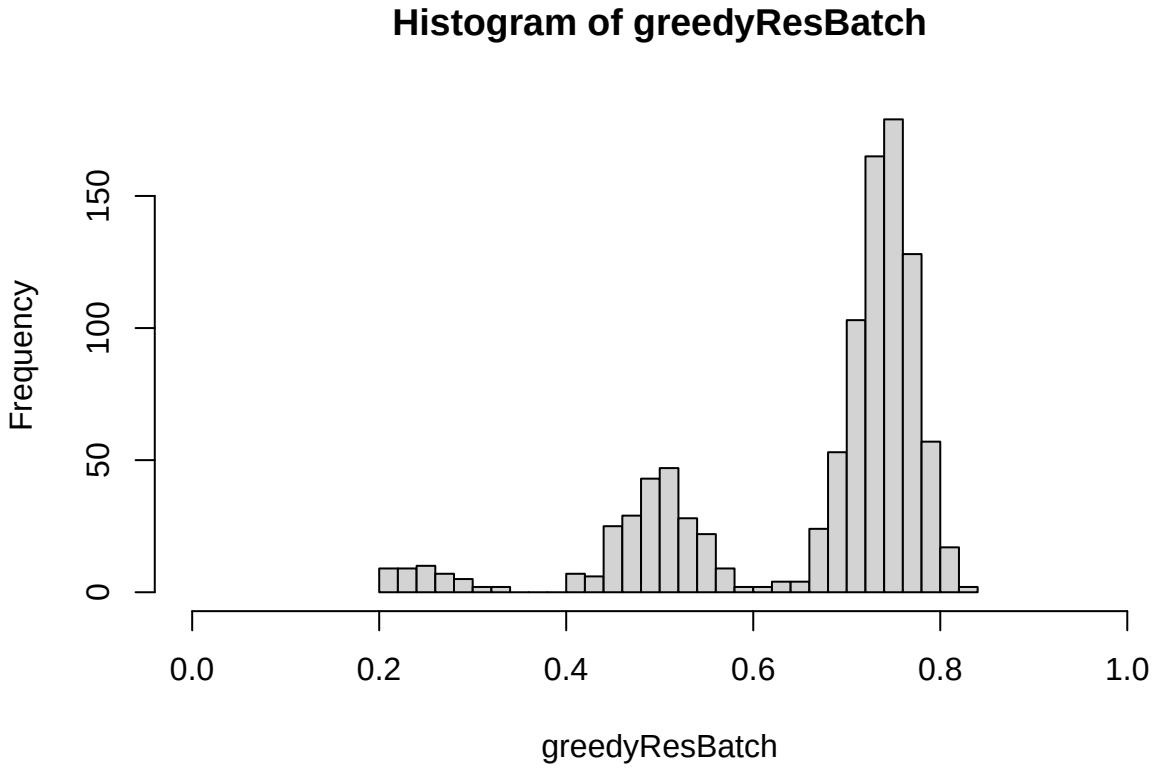


Question 9: Let's compare Thompson sampling to the greedy policy. What can we say?

```
nRuns <- 1000
randResBatch <- numeric(nRuns)
thompsonResBatch <- numeric(nRuns)
greedyResBatch <- numeric(nRuns)
wslsResBatch <- numeric(nRuns)

for(i in 1:nRuns) {
  randResBatch[i] <- banditTask(randomPolicy,200,exampleArms)[[1]]
  thompsonResBatch[i] <- banditTask(thompsonSampling(1,1),200,exampleArms)[[1]]
  greedyResBatch[i] <- banditTask(greedyPolicy(0.001,0.001),200,exampleArms)[[1]]
  wslsResBatch[i] <- banditTask(wslsPolicy,200,exampleArms)[[1]]
}

hist(thompsonResBatch,breaks=40, xlim=c(0,1))
```




```
hist(greedyResBatch,breaks=40, xlim=c(0,1))
```



Question 10: What do you think is important to consider in a model of decision making in bandit tasks, e.g., phenomena it should be able to capture? Discuss.

Question 11: What might you do to make bandit tasks more realistic or representative of analogous tasks in everyday life? Discuss.