Report 2

Jakub Domasik

17 11 2019

Q1

(a)

We want to find stationary point of:

$$f(u_{1:K}) = \sum_{i=1}^{n}\sum_{k=1}^{K}(\gamma_{ik} log(\prod_{j=1}^{n} u_{kj}^{x_{ij}}(1-u_{kj})^{1-x_{ij}}))$$

First, we expand the function:

$$f(u_{1:K}) = \sum_{i=1}^{n}\sum_{i=1}^{n}(\gamma_{ik} log(\prod_{j=1}^{n} \mu_{kj}^{x_{ij}}(1-\mu_{kj})^{1-x_{ij}})) = \sum_{i=1}^{n}\sum_{i=1}^{n}(\gamma_{ik}\sum_{j=1}^{P}(log(u_{kj}^{x_{ij}}(1-u_{kj})^{1-x_{ij}}))$$

$$= \sum_{i=1}^{n}\sum_{i=1}^{n}(\gamma_{ik}\sum_{j=1}^{P}(x_{ij} log u_{kj} + (1-x_{ij} log(1-u_{kj})))$$

Next, we differentiate $f(u_k)$ and set it equal to 0:

$$f(u_k)' = \sum_{i=1}^{n}\sum_{k=1}^{K}(\gamma_{ik}\sum_{j=1}^{P}(\frac{x_{ij}}{u_{kj}} - \frac{1-x_{ij}}{1-u_{kj}})) = 0$$

The sum is qeual to 0, if for every k we have:

$$\sum_{i=1}^{n}(\gamma_{ik}\sum_{j=1}^{P}(\frac{x_{ij}}{u_{kj}} - \frac{1-x_{ij}}{1-u_{kj}})) = 0$$

Now, $\frac{x_{ij}}{u_{kj}} - \frac{1-x_{ij}}{1-u_{kj}}$ are the entries of $\frac{x_i}{u_k} - \frac{1-x_i}{1-u_k}$ and $x_{ij} \in 0, 1$ so:

$$f(u_k)' = \sum_{i=1}^{n}\sum_{k=1}^{K}(\gamma_{ik}\sum_{j=1}^{P}(\frac{x_{ij}}{u_{kj}} - \frac{1-x_{ij}}{1-u_{kj}})) = 0 \iff \sum_{i=1}^{n}(\gamma_{ik}\sum_{j=1}^{P}(\frac{x_i}{u_k} - \frac{1-x_i}{1-u_k})) = 0$$

Next,

$$\sum_{i=1}^{n}(\gamma_{ik}\sum_{j=1}^{P}(\frac{x_i}{u_k} - \frac{1-x_i}{1-u_k})) = \sum_{i=1}^{n}\gamma_{ik}\frac{x_i}{u_k} - \sum_{i=1}^{n}\gamma_{ik}\frac{1-x_i}{1-u_k} = 0$$

So,

$$\sum_{i=1}^{n}\gamma_{ik}\frac{x_i}{u_k} = \sum_{i=1}^{n}\gamma_{ik}\frac{1-x_i}{1-u_k} = 0 \iff u_k\sum_{i=1}^{n}\gamma_{ik}(1-x_i) = (1-u_k)\sum_{i=1}^{n}\gamma_{ik}x_i \iff u_k(\sum_{i=1}^{n}\gamma_{ik}x_i + \sum_{i=1}^{n}\gamma_{ik}(1-x_i)) = \sum_{i=1}^{n}\gamma_{ik}x_i$$

$$\iff u_k = \frac{\sum_{i=1}^n \gamma_{ik} x_i}{(\sum_{i=1}^n \gamma_{ik} x_i + \sum_{i=1}^n \gamma_{ik}(1-x_i))} = \frac{\sum_{i=1}^n \gamma_{ik} x_i}{\sum_{i=1}^n \gamma_{ik}}$$

Therefore, we have found stationary point as intended. ### (b) (i)

```
## EM algorithm for a mixture of Bernoullis

## logsumexp(x) returns log(sum(exp(x))) but performs the
## computation in a more stable manner
logsumexp <- function(x) return(log(sum(exp(x - max(x)))) + max(x))

prob <- function(x,mu,return.log=FALSE) {
  l <- sum(log(mu[x==1]))+sum(log(1-mu[x==0]))
  if (return.log) {
    return(l)
  } else {
    return(exp(l))
  }
}

compute_ll <- function(xs,mus,lws,gammas) {
  ll <- 0
  n <- dim(xs)[1]
  K <- dim(mus)[1]
  for (i in 1:n) {
    for (k in 1:K) {
      if (gammas[i,k] > 0) {
        ll <- ll + gammas[i,k]*(lws[k]+prob(xs[i,],mus[k,],return.log=TRUE)-
                                log(gammas[i,k]))
      }
    }
  }
  return(ll)
}



em_mix_bernoulli <- function(xs,K,start=NULL,max.numit=Inf) {
  p <- dim(xs)[2]
  n <- dim(xs)[1]

  # lws is log(ws)
  # we work with logs to keep the numbers stable
  # start off with ws all equal
  lws <- rep(log(1/K),K)

  if (is.null(start)) {
    mus <- .2 + .6*xs[sample(n,K),]
  } else {
    mus <- start
  }
  gammas <- matrix(0,n,K)

  converged <- FALSE
```

```
  numit <- 0
  ll <- -Inf
  print("iteration : log-likelihood")
  while(!converged && numit < max.numit) {
    numit <- numit + 1
    mus.old <- mus
    ll.old <- ll

    ## E step - calculate gammas
    for (i in 1:n) {
      # the elements of lprs are log(w_k * p_k(x)) for each k in {1,...K}
      lprs <- rep(0,K)
      for (k in 1:K) {
        lprs[k] <- lws[k] + prob(xs[i,],mus[k,],return.log=TRUE)
      }
      # gammas[i,k] = w_k * p_k(x) / sum_j {w_j * p_j(x)}
      gammas[i,] <- exp(lprs - logsumexp(lprs))
    }

    ll <- compute_ll(xs,mus,lws,gammas)
    # we could also compute the log-likelihood directly below
    # ll <- compute_ll.direct(xs,mus,lws)

    # M step - update ws and mus
    Ns <- rep(0,K)
    for (k in 1:K) {
      Ns[k] <- sum(gammas[,k])
      lws[k] <- log(Ns[k])-log(n)

      mus[k,] <- rep(0,p)
      for (i in 1:n) {
        mus[k,] <- mus[k,]+gammas[i,k]/Ns[k]*xs[i,]
      }
    }
    # to avoid a numerical issue since each element of mus must be in [0,1]
    mus[which(mus > 1,arr.ind=TRUE)] <- 1 - 1e-15
    if (numit < 50) {
      print(paste(numit,": ",ll))}
    # we stop once the increase in the log-likelihood is "small enough"
    if (abs(ll-ll.old) < 1e-5) converged <- TRUE
  }
  return(list(lws=lws,mus=mus,gammas=gammas,ll=ll))
}
```

The algorithm is run on *newspapers* data, particulary on *documents* dataset with K = 4. The code prints likelihoods calculated during first 50 iterations.

```
load("20newsgroups.RData")
clusters <- em_mix_bernoulli(documents, 4)
```

```
## [1] "iteration : log-likelihood"
## [1] "1 :  -487561.731895519"
## [1] "2 :  -251505.404680153"
```

```
## [1] "3 :  -248036.467180821"
## [1] "4 :  -244502.020180398"
## [1] "5 :  -241079.239620968"
## [1] "6 :  -239423.86237813"
## [1] "7 :  -238863.683074978"
## [1] "8 :  -238623.493119204"
## [1] "9 :  -238413.701651017"
## [1] "10 :  -238212.141573239"
## [1] "11 :  -238049.146982644"
## [1] "12 :  -237877.145950541"
## [1] "13 :  -237677.546970367"
## [1] "14 :  -237518.901743327"
## [1] "15 :  -237425.928851203"
## [1] "16 :  -237358.431060805"
## [1] "17 :  -237318.987802273"
## [1] "18 :  -237290.843329535"
## [1] "19 :  -237262.907177979"
## [1] "20 :  -237240.597546274"
## [1] "21 :  -237223.283301544"
## [1] "22 :  -237205.560656241"
## [1] "23 :  -237188.975824124"
## [1] "24 :  -237175.178683399"
## [1] "25 :  -237161.360347137"
## [1] "26 :  -237148.136495211"
## [1] "27 :  -237136.557586264"
## [1] "28 :  -237121.220235708"
## [1] "29 :  -237112.716780044"
## [1] "30 :  -237105.293405395"
## [1] "31 :  -237094.685869164"
## [1] "32 :  -237076.895265173"
## [1] "33 :  -237063.963263939"
## [1] "34 :  -237054.020233722"
## [1] "35 :  -237044.811605212"
## [1] "36 :  -237035.876314661"
## [1] "37 :  -237027.05616315"
## [1] "38 :  -237018.261069431"
## [1] "39 :  -237009.287981659"
## [1] "40 :  -236999.890160895"
## [1] "41 :  -236989.868267209"
## [1] "42 :  -236978.962411731"
## [1] "43 :  -236966.641757889"
## [1] "44 :  -236953.371370661"
## [1] "45 :  -236939.947146408"
## [1] "46 :  -236926.362135534"
## [1] "47 :  -236913.036906618"
## [1] "48 :  -236900.232916924"
## [1] "49 :  -236887.795717112"
```

We check if the algorithm is working correctly by calculating success rate.

```
load("20newsgroups.RData")
preds <- matrix(0, nrow = dim(documents)[1], ncol = 4)
for (i in 1:dim(documents)[1]) {
  best <- 0
```

```
  for (j in 1:4) {
    if (clusters$gammas[i,j] > best ) {
      best <- clusters$gammas[i,j]
      c <- j
    }

  }
  preds[i,c] <- 1

}

preds2 <- matrix(0, nrow = dim(documents)[1], ncol = 4)
preds2[,1] <- preds[,4]
preds2[,2] <- preds[,1]
preds2[,3] <- preds[,3]
preds2[,4] <- preds[,2]
loss <- 0
for (i in 1:dim(documents)[1]) {
  for (j in 1:4) {
    if(preds2[i,j] == 1 && newsgroups.onehot[i,j] != preds2[i,j])
      loss <- loss + 1
    }

}
1-(loss/dim(documents)[1])
```

```
## [1] 0.6278168
```

I was forced to change the order of the clusters in *preds* data frame to match the clusters in *newsgroups.onehot2* file. It resulted with getting success rate 0.6 which is very promising and much better than just choosing the clusters at random. I conclude that the algorithm works as intended.

2.

(a)

I am asked to run the Thompson Sampling algorithm and ϵ-decreasing algorithm on two-armed bandits with probabilities of succes for each arm: 0.6 and 0.4.

Thompson Sampling

```
## thompson sampling
prob = c(0.6, 0.4)

this_arm <- function(plays, successes) {
  # vector of successes
  # we add a positive number to avoid zeros
  alpha_s <-  10 + successes
  # vector of failures
  alpha_f <-  10 + plays - successes

  # beta density for arm no 1
```

```
  m1 <- rbeta(1, alpha_s[1], alpha_f[1])
  # beta density for arm no 2
  m2 <- rbeta(1, alpha_s[2], alpha_f[2])

  #choosing which arm to play next
  if(m1>m2) {
    return(1)
  }
    else {
      return(2)
    }

}

# two-armed bandit is played using thompson sampling
thompson_bernoulli <- function(prob,n) {
  arm <- rep(0,n)
  reward <- rep(0,n)
  ## array consisting of number of plays for each arm
## array consiting of number of successes for each arm
  plays <- rep(0,2)
  successes <- rep(0,2)

  #thompson algorithm
  for (i in 1:n) {
    a <- this_arm(plays,successes)
    r <- runif(1) < prob[a]
     reward[i] <- r
    arm[i] <- a
    plays[a] <- plays[a] + 1
    successes[a] <- successes[a] + r

  }
  return(list(arm=arm,reward=reward))
}
```

We use the test developed in Lab 5 to check if the algorithm works correctly.

```
# testing
## computing thompson sampling for given probabilities and on 100000 trials

## checking the rate of success
# if algorithm works, the sum should be close to max(0.3, 0.6)
score_t <- thompson_bernoulli(prob = prob, n = 50000)
avg_reward_t <- c(0)
for (i in 1:length(score_t$reward)) {
  avg_reward_t[i] <- mean(c(score_t$reward[1:i]))
}
ratio_t = sum(score_t$reward) / length(score_t$reward)
ratio_t
```

```
## [1] 0.60026
```

Since $\max(0.6, 0.4) = 0.6$ and the ratio obtained is very close to this number, we conclude that the algorithm works correctly.

ϵ-decreasing algorithm

```
## e-decreasing algorithm

epsilon_decreasing_n <- function(prob,n) {
  arm <- rep(0,n)
  reward <- rep(0,n)
  ## initial number of plays and number of successes is 0 for each arm
  plays <- rep(0,2)
  successes <- rep(0,2)
  C = 5000
  ## at first, play each arm once
  for (i in 1:2) {
    a <- i
    r <- runif(1) < prob[a]
    plays[a] <- plays[a] + 1
    successes[a] <- successes[a] + r
    arm[i] <- a
    reward[i] <- r
  }
  ## now follow the epsilon decreasing strategy
  for (i in 3:n) {
    epsilon = min(1, C/n )
    # with probability epsilon, pick an arm uniformly at random
    if (runif(1) < epsilon) {
      a <- sample(2,1)
    } else { # otherwise, choose the "best arm so far".
      a <- which.max(successes/plays)
    }
    ## simulate the reward
    r <- runif(1) < prob[a]
    # update the number of plays, successes
    plays[a] <- plays[a] + 1
    successes[a] <- successes[a] + r
    # record the arm played and the reward received
    arm[i] <- a
    reward[i] <- r
  }
  return(list(arm=arm,reward=reward))
}
```

We use similar test as before to check if the algorithm is working correctly.

```
score_n <- epsilon_decreasing_n(prob = prob, n = 50000)

ratio_n = sum(score_n$reward) / length(score_n$reward)
ratio_n
```

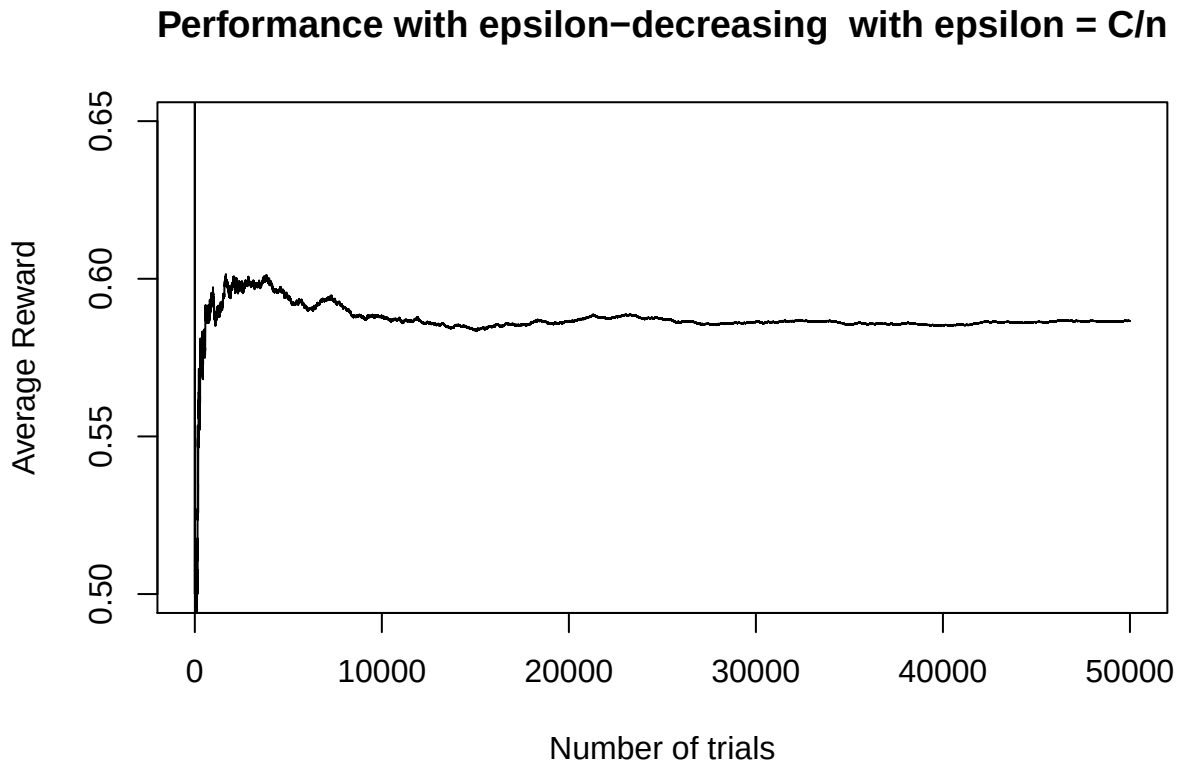
```
## [1] 0.58904
```

Again, the result is very close to 0.6, so the algorithm is working as intended.

(b)

We describe the behavior of ϵ-decreasing algorithm, where $\epsilon = min(1, Cn^{-1})$, where C is some positive constant.

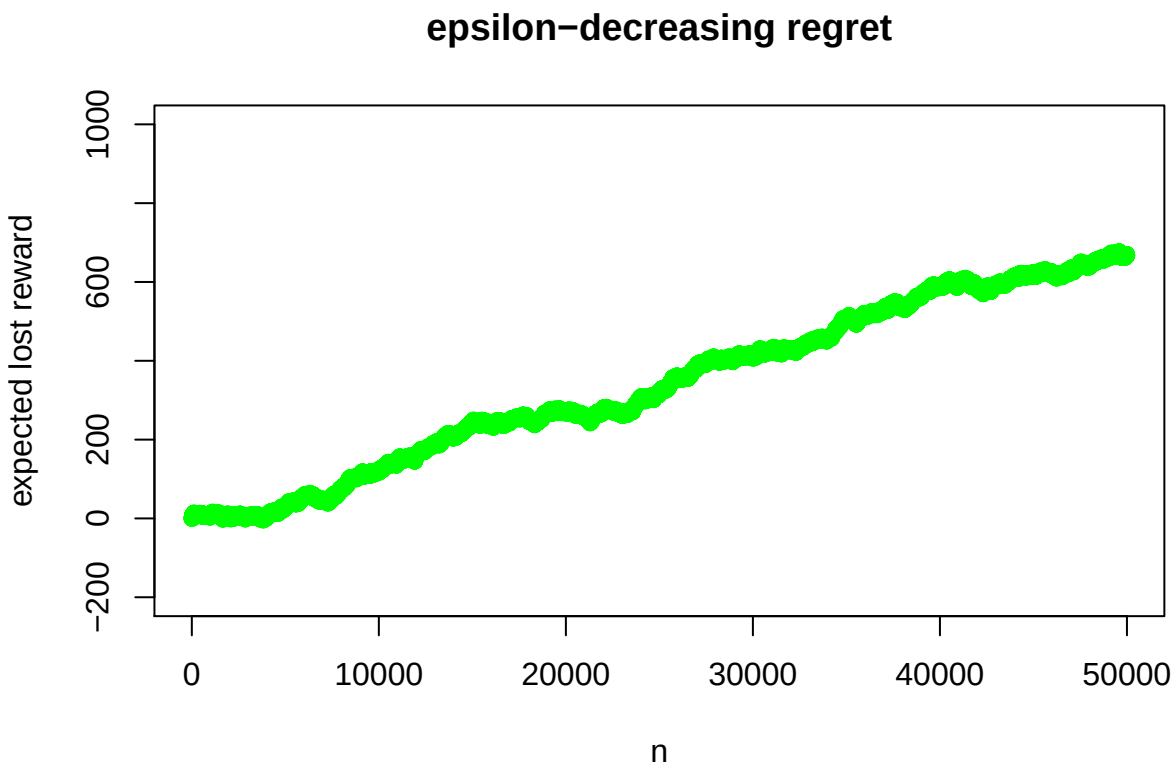
```
score_n <- epsilon_decreasing_n(prob = prob, n = 50000)
avg_reward_n <- c(0)
for (i in 1:length(score_n$reward)) {
  avg_reward_n[i] <- mean(c(score_n$reward[1:i]))
}

plot(avg_reward_n, ylab = "Average Reward",
     type = "l", xlab = "Number of trials",
     main = "Performance with epsilon-decreasing  with epsilon = C/n" ,ylim = c(0.5, 0.65))
```



The algorithm stabilizes very close to 0.6 and relatively quickly reaches its equilibrium.

Now, let us look at how the expected loss reward changes over time for this value of epsilon.

```
regret1 <- c(0)
for(i in 1:length(score_n$reward)) {
regret1[i] <- mean(sum(prob[1] - score_n$reward[1:i]))
}
plot(regret1, col = "green", ylim = c(-200,1000), xlab = "n",
     ylab = "expected lost reward", main = "epsilon-decreasing regret")
```



We may conclude that the algorithm behaves as expected and that it is consistent with my implementation of it.

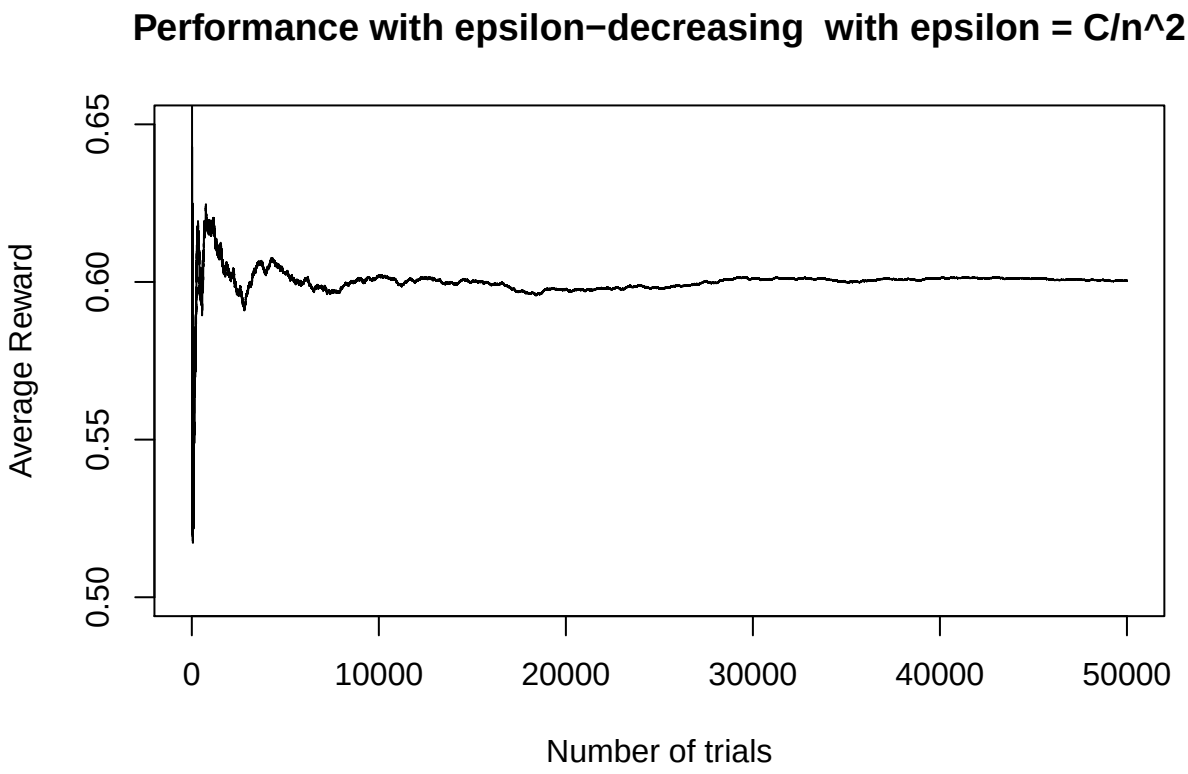
(c)

We perform similar operations for ϵ-decreasing algorithm using $\epsilon = Cn^{-2}$, where Cis some positive constant.

```
epsilon_decreasing_n2 <- function(prob,n) {
  arm <- rep(0,n)
  reward <- rep(0,n)
  ## initial number of plays and number of successes is 0 for each arm
  plays <- rep(0,2)
  successes <- rep(0,2)
  C = 5000
  ## at first, play each arm once
  for (i in 1:2) {
    a <- i
    r <- runif(1) < prob[a]
    plays[a] <- plays[a] + 1
    successes[a] <- successes[a] + r
    arm[i] <- a
    reward[i] <- r
  }
  ## now follow the epsilon decreasing strategy
  for (i in 3:n) {
```

```
    epsilon = min(1, C/(n^2) )
    # with probability epsilon, pick an arm uniformly at random
    if (runif(1) < epsilon) {
      a <- sample(2,1)
    } else { # otherwise, choose the "best arm so far".
      a <- which.max(successes/plays)
    }
    ## simulate the reward
    r <- runif(1) < prob[a]
    # update the number of plays, successes
    plays[a] <- plays[a] + 1
    successes[a] <- successes[a] + r
    # record the arm played and the reward received
    arm[i] <- a
    reward[i] <- r
  }
  return(list(arm=arm,reward=reward))
}
```

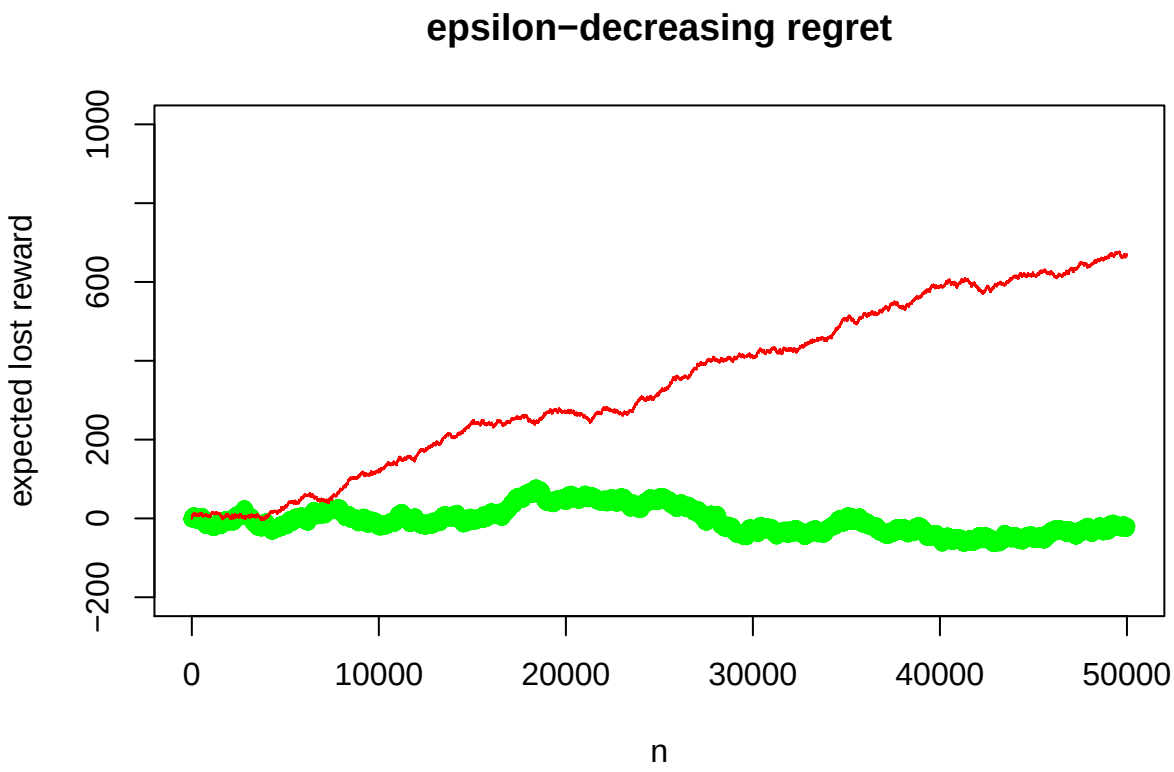
```
score_n2 <- epsilon_decreasing_n2(prob = prob, n = 50000)
avg_reward_n2 <- c(0)
for (i in 1:length(score_n2$reward)) {
  avg_reward_n2[i] <- mean(c(score_n2$reward[1:i]))
}

plot(avg_reward_n2, ylab = "Average Reward",
     type = "l", xlab = "Number of trials",
     main = "Performance with epsilon-decreasing  with epsilon = C/n^2" ,ylim = c(0.5, 0.65))
```



We see that the algorithm stabilizes after roughly 10000 repetitions and achieves average reward, which is very close to 0.6 so it converges eventually.

Now, we calculate the regret for the new algorithm.

```
regret2 <- c(0)
for(i in 1:length(score_n2$reward)) {
regret2[i] <- mean(sum(prob[1] - score_n2$reward[1:i]))
}
plot(regret2, col = "green", ylim = c(-200,1000), xlab = "n",
     ylab = "expected lost reward", main = "epsilon-decreasing regret")
lines(regret1, col = "red")
```



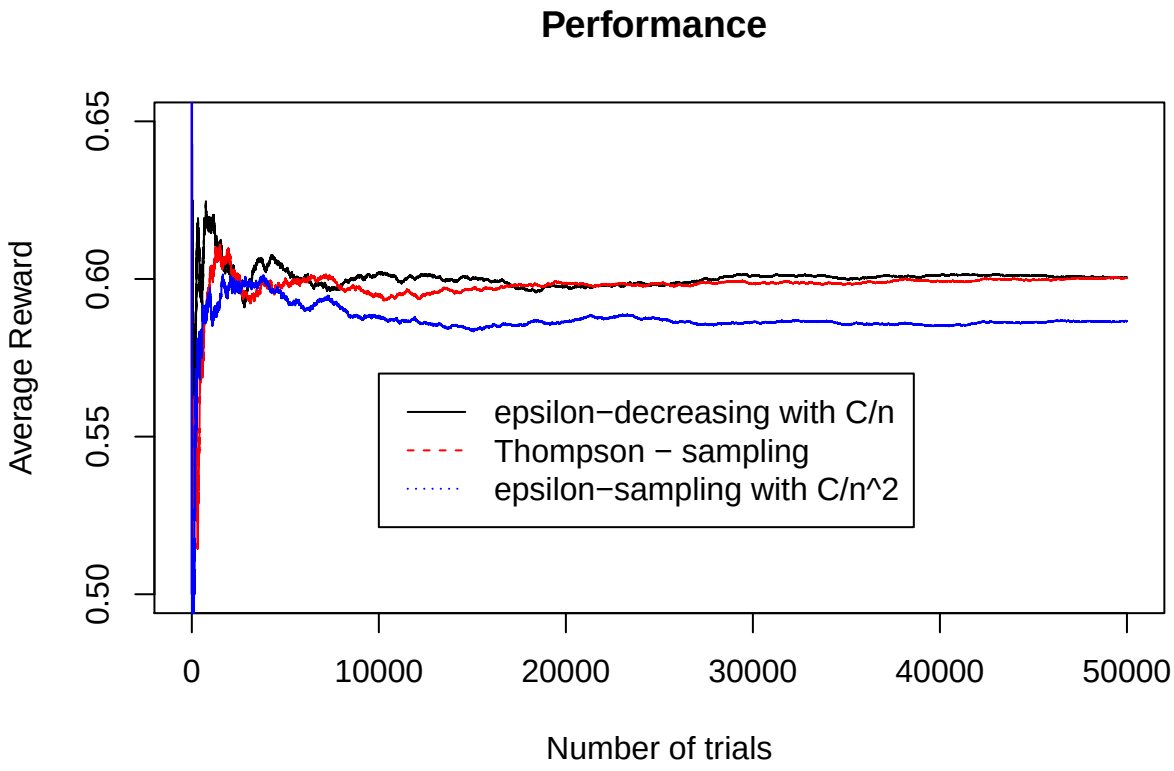
Looking at the graph above, it reveals that the second algorithm has a negative expected loss reward. It means, that the algorithm with $\epsilon = Cn^{-2}$ provides more rewards. We can now compare the number of the rewards.

```
sum(score_n2$reward) - sum(score_n$reward)
```

```
## [1] 688
```

No, we compare the performance of all our algorithms: Thompson sampling and both versions ϵ-decreasing algorithm.

```
plot(avg_reward_n2, ylab = "Average Reward",
     type = "l", xlab = "Number of trials",
     main = "Performance " ,ylim = c(0.5, 0.65), col = "black")
lines(avg_reward_t, col = "red")
lines(avg_reward_n, col = "blue")
legend(10000, 0.57, legend = c("epsilon-decreasing with C/n",
                               "Thompson - sampling", "epsilon-sampling with C/n^2"),
       col = c("black", "red", "blue"), lty = 1:3)
```



As we see, the Thompson sampling algorithm converges the most quickly. However, after roughly 30 000 repetitions epsilon-decreasing algorithm with $\epsilon = Cn^{-2}$) has very similar performance. All ofour algorithms converge eventually but the Thompson Sampling with $\epsilon = Cn^{-1}$ converges slightly below 0.6 # 3.
(a)

I am using funnction *distances.l2* to calculate the distances between observations.

```
distances.l1 <- function(x,y) {
  d <- 0
  for (i in 1:length(x)) {
    d <- d + abs(x[[i]]-y[[i]])
  }
  return(d)
}

distances.l2 <- function(x,y) {
  d <- 0
  for (i in 1:length(x)) {
    d <- d + (x[[i]]-y[[i]])^2
  }
  return(d)
}



knn.regression.test <- function(k,train.X,train.Y,test.X,test.Y,distances, bin = FALSE) {
```

```
  #algorithm can't run if k is larger than number of observations in training dataset
  if(k > nrow(train.X)) {
return("ERROR!!!")  }
      estimates <- numeric(0)

  # we need two loops
  # first over test data
  for(i in 1:nrow(test.X)) {
    indexes <- numeric(0)
    calculated_distances <- numeric(0)


    #second over train data
    for (j in 1: nrow(train.X)) {
      calculated_distances <- c(calculated_distances,
                                distances(c(test.X[i,], test.Y[i]),
                                          c(train.X[j,], train.Y[j])))
    indexes <- c(indexes, j)
    }

    #creating the matrix  with euclidian distances between labaled points and the new point
    classification <- data.frame(calculated_distances, indexes)

    # using order function to sort distances in order to find nearest neighbors
    classification <- classification[order(classification$calculated_distances),]

    #selecting k nearest neighbors
    classification <- classification[1:k, ]

    #calculating inverse-distance weights as asked
    inv_weighted_distances <- sum(train.Y[classification$indexes] *
                                    calculated_distances[classification$indexes] ^ (-1))/
      sum(calculated_distances[classification$indexes] ^ (-1))
      estimates <- c(estimates, inv_weighted_distances)
    #print(estimates)

  }


  loss <- sum((as.vector(test.Y) - estimates) ^ 2)
  if (bin) {
    return(list(loss, estimates))
  }
  else {
    return(loss)
  }
}
```

(b)

Toy dataset 1

```r
set.seed(3000)
n <- 100
train.X <- matrix(sort(rnorm(n)),n,1)
train.Y <- (train.X < -0.5) + train.X*(train.X>0)+rnorm(n,sd=0.03)
test.X <- matrix(sort(rnorm(n)),n,1)
test.Y <- (test.X < -0.5) + test.X*(test.X>0)+rnorm(n,sd=0.03)
```

Now, the code below is written to find the optimal value of k.

```r
least_loss <- Inf
best_k <-0

for (k in 1:nrow(test.X)) {
  loss <- knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2)
  if(least_loss > loss) {
    least_loss <- loss
    best_k <- k
  }

}
print(paste0("Best value of k is ", best_k, ". Least loss is ", least_loss, "."))
```

```
## [1] "Best value of k is 7. Least loss is 0.0493035081014927."
```

Toy dataset 2

```r
# toy dataset 2
train.X <- matrix(rnorm(200),100,2)
train.Y <- train.X[,1]
test.X <- matrix(rnorm(100),50,2)
test.Y <- test.X[,1]
k <- 3
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2)
```

```
## [1] 2.372337
```

```r
# FInd best values of k
least_loss <- Inf
best_k <-0

for (k in 1:nrow(test.X)) {
  loss <- knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2)
  if(least_loss > loss) {
    least_loss <- loss
    best_k <- k
  }

}
print(paste0("Best value of k is ", best_k, ". Least loss is ", least_loss, "."))
```

```
## [1] "Best value of k is 4. Least loss is 2.30302893186138."
```

(c)

```
library("lasso2")
library(ggplot2)
data(Iowa)
train.X=as.matrix(Iowa[seq(1,33,2),1:9])
train.Y=c(Iowa[seq(1,33,2),10])
test.X=as.matrix(Iowa[seq(2,32,2),1:9])
test.Y=c(Iowa[seq(2,32,2),10])
k <- 5
results <- knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2, bin = TRUE)
predictions <- data.frame()
predictions <- data.frame(Year = seq(1931, 1961, by = 2), actual = test.Y,
                          predicted = results[[2]])

ggplot(predictions, aes(Year, y = Yield, color = Key)) +
  geom_line(aes(y = actual, col = "Actual")) +
  geom_line(aes(y = predicted, col = "Predicted")) +
  ggtitle("kNN with k = 5")
```


```
print(paste0('loss for kNN with k = 5: ', sum((test.Y - predictions$predicted) ^ 2)))
```

```
## [1] "loss for kNN with k = 5: 316.739488249416"
```

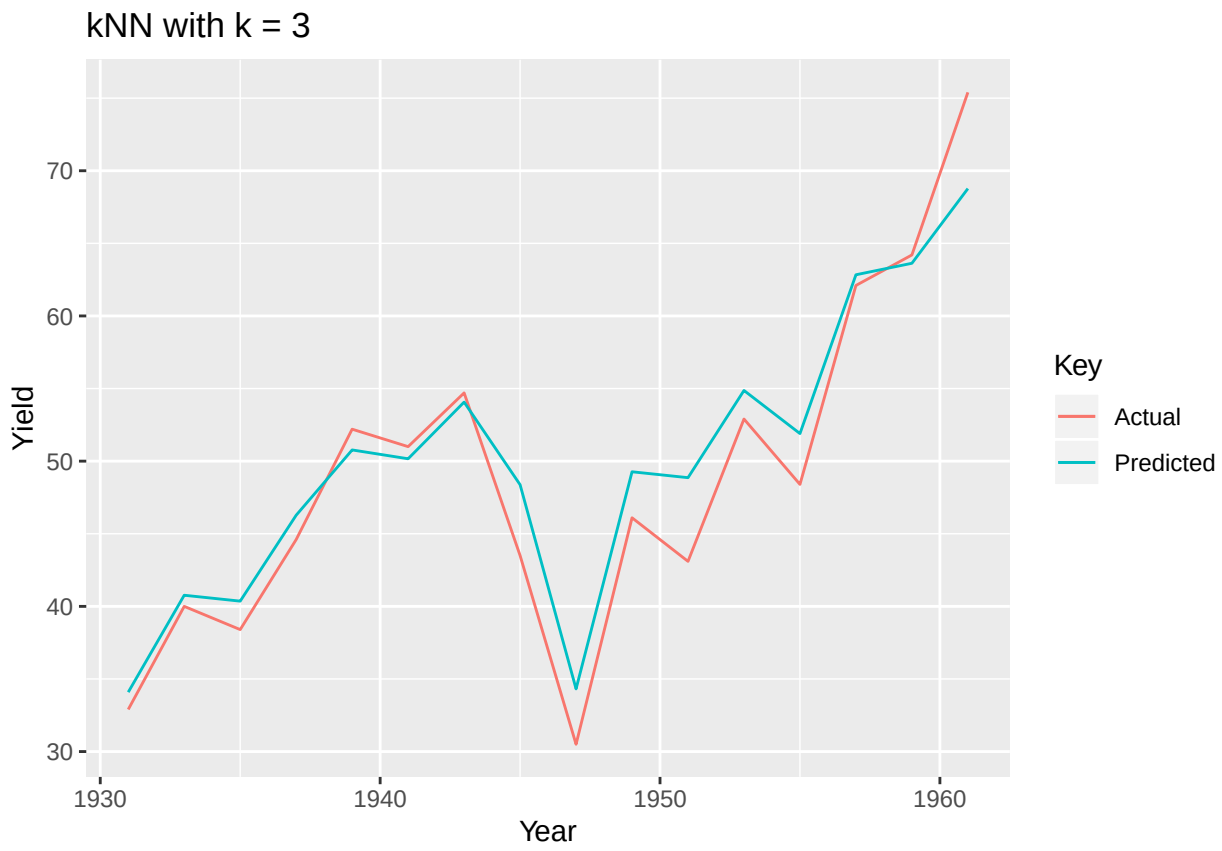
(d)

```
least_loss <- Inf
best_k <-0

for (k in 1:100) {
  loss <- knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2)
  if(least_loss > loss) {
    least_loss <- loss
    best_k <- k
  }

}
print(paste0("Best value of k is ", best_k, ". Least loss is ", least_loss, "."))
```

```
## [1] "Best value of k is 3. Least loss is 154.29718962952."
```

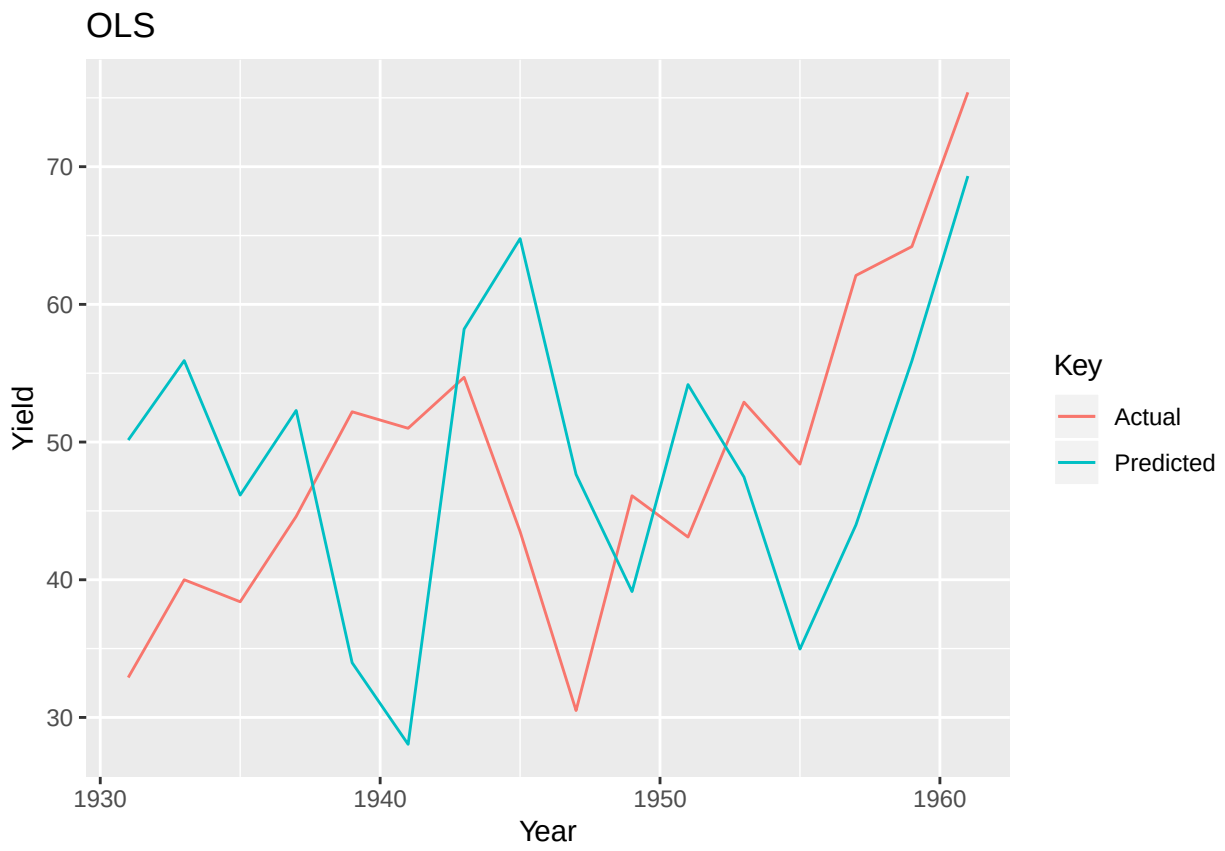
Therefore, we visualise the results with k = 3.

```
k <- 3
results <- knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2, bin = TRUE)
predictions <- data.frame()
predictions <- data.frame(Year = seq(1931, 1961, by = 2), actual = test.Y,
                          predicted = results[[2]])

ggplot(predictions, aes(Year, y = Yield, color = Key)) +
  geom_line(aes(y = actual, col = "Actual")) +
  geom_line(aes(y = predicted, col = "Predicted")) +
  ggtitle("kNN with k = 3")
```

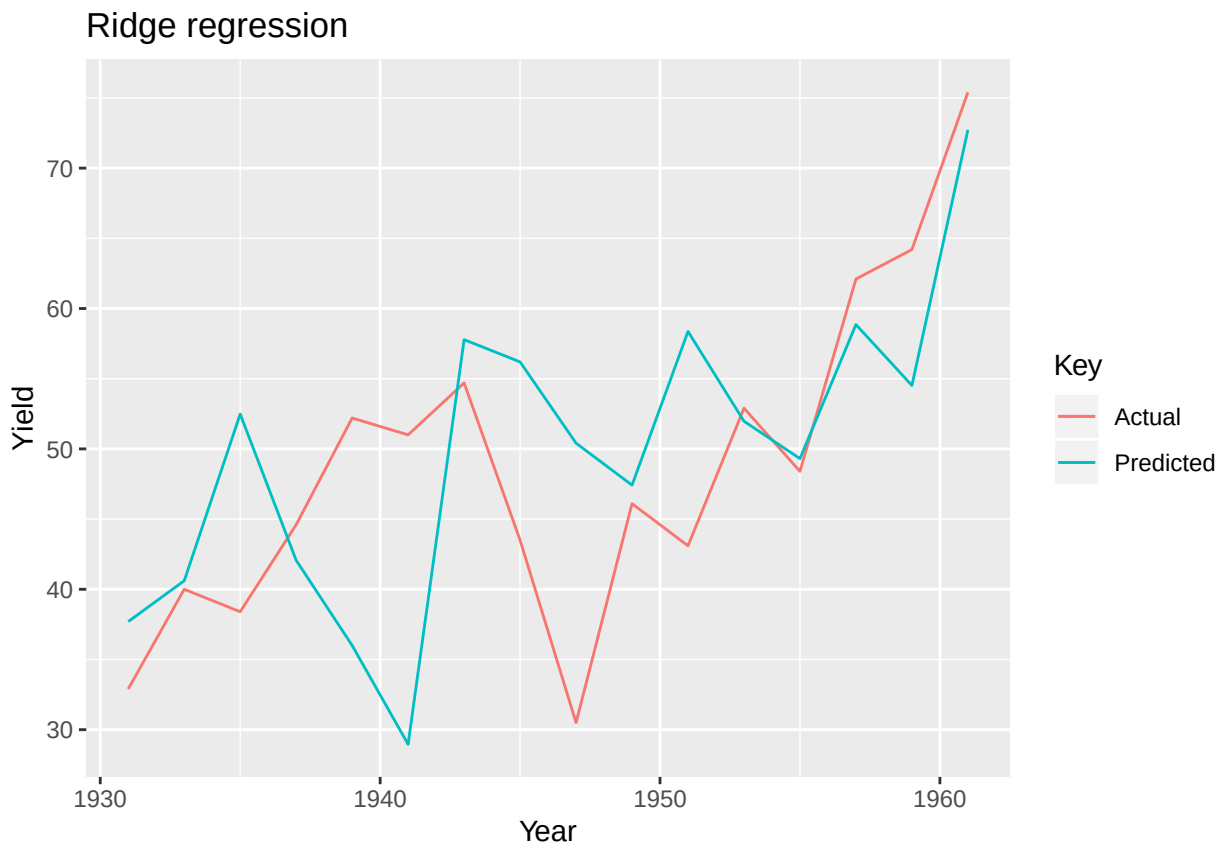


Now, we want to compare the results of kNN algorithm with OLS and ridge regression.

```
data <- data.frame(train.X, train.Y)
least_squares <- lm(train.Y ~ Rain0 + Rain1 + Rain2
                    + Rain3  + Temp1 + Temp2 + Temp3 + Temp4, data = data)
preds_ls <- predict(least_squares, as.data.frame(test.X))

library(MASS)
library(lmridge)
ridge <- lmridge(train.Y ~ Year + Rain0 + Rain1 + Rain2
                 + Rain3  + Temp1 + Temp2 + Temp3 + Temp4, data = data)
preds_ls <-  data.frame(Year = seq(1931, 1961, by = 2), actual = test.Y,
predicted = preds_ls)
preds_r <- predict.lmridge(ridge, as.data.frame(test.X))
preds_r <-  data.frame(Year = seq(1931, 1961, by = 2), actual = test.Y,
predicted = preds_r)

ggplot(preds_ls, aes(Year, y = Yield, color = Key)) +
  geom_line(aes(y = actual, col = "Actual")) +
  geom_line(aes(y = predicted, col = "Predicted")) +
  ggtitle("OLS")
```




```
ggplot(preds_r, aes(Year, y = Yield, color = Key)) +
  geom_line(aes(y = actual, col = "Actual")) +
  geom_line(aes(y = predicted, col = "Predicted")) +
  ggtitle("Ridge regression")
```




```
print(paste0('loss for OLS: ', sum((test.Y - preds_ls$predicted) ^ 2)))
```

```
## [1] "loss for OLS: 3103.35373258034"
```

```
print(paste0('loss for ridge regression: ', sum((test.Y - preds_r$predicted) ^ 2)))
```

```
## [1] "loss for ridge regression: 1891.33168243517"
```

Therefore, both OLS and ridge regression perform much worse than kNN with k = 3. However, OLS still performs much better than ridge regression. OLS and ridgemay be tricked to work better by manipualating the variables by eg. squaring them, taking roots etc.