SL_EX_4

roi hezkiyahu

24 12 2022

```
library(dplyr)
library(ggplot2)
library(purrr)
library(caret)
library(class)
library(tidymodels)
library(furrr)
library(rpart)
library(rpart)
library(keras)
```

Q1

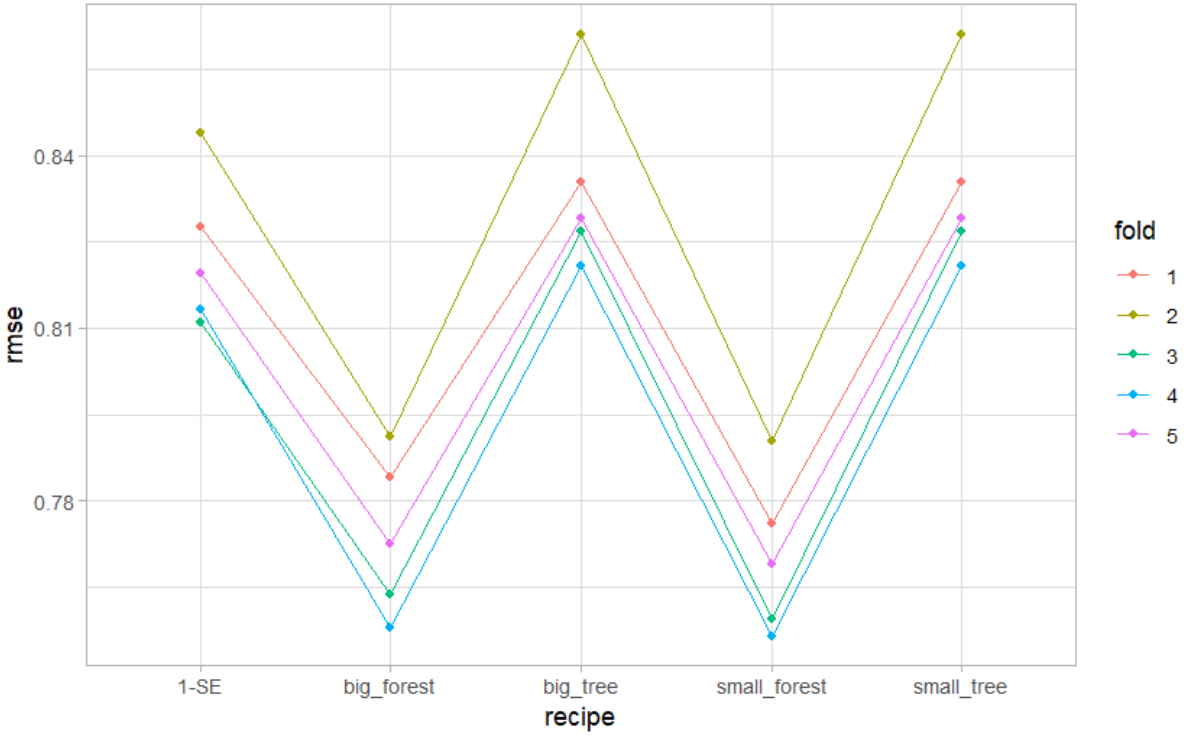
1. **Playing around with trees**
   Run a variety of tree-based algorithms on our competition data and show their performance. Compare:

   - Small tree without pruning
   - Large tree without pruning
   - Large tree after pruning with 1-SE rule
   - Bagging/RF on small trees (100 iterations)
   - Bagging/RF large trees (100 iterations)

   Do this under five-fold cross validation on our competition training set, and use the results of the five different folds to calculate confidence intervals for performance. Plot all the results in a reasonable way (e.g. using boxplot()) and comment on them. Explain your choices of "small" and "large".
   Hints: a. Start early since bagging may take a while to run. b. Use as a basis the code from class which implements much of this.

```
plot_cv_res <- function(cv_results){
  cv_results_longer <- cv_results %>%
  pivot_longer(cols = colnames(cv_results),names_to = "recipe",values_to = "rmse") %>% mutate(fold = ceiling(1:(length(colna
mes(cv_results))*cv_v)/length(colnames(cv_results))))

  cv_results_longer %>% group_by(recipe) %>% summarize(m = median(rmse)) %>% arrange(m)

  cv_results_longer %>%
    mutate(fold = factor(fold)) %>%
    ggplot(aes(recipe, rmse,group=fold,color = fold)) +
    geom_line(aes(group=fold)) +
    geom_point() +
    theme_light()
}

set.seed(123)
con <- url("http://www.tau.ac.il/~saharon/StatsLearn2022/train_ratings_all.dat")
X <- tibble(read.table(con))
con <- url("http://www.tau.ac.il/~saharon/StatsLearn2022/train_y_rating.dat")
y <- read.table(con)

cv_model <- function(df_split, model, prune_tree = FALSE){
  train_df <- training(df_split)
  test_df <- testing(df_split)

  if (prune_tree){
    tree.mod= rpart (y~.,data=train_df,cp=0.0001)
    tree.res = printcp(tree.mod)
    #1-SE rule
    chosen.prune = min ((1:dim(tree.res)[1]) [tree.res[,"xerror"] < min(tree.res[,"xerror"]+tree.res[,"xstd"])])
    model = prune(tree.mod, cp=tree.res[chosen.prune,"CP"])
    model_pred <- predict(model, test_df %>% select(-y))
    rmse_vec(test_df$y, model_pred)
  }
  else{
    model <- model %>% fit(formula = y~., data = train_df)
    model_pred <- predict(model, test_df %>% select(-y))
    rmse_vec(test_df$y, model_pred$.pred)
  }
}

train <- X %>% mutate(y = y %>% pull)

small_tree <- decision_tree(mode = "regression", tree_depth = 5) %>%
  set_engine("rpart")

big_tree <- decision_tree(mode = "regression") %>%
  set_engine("rpart")

big_forest <- rand_forest(mode = "regression", trees = 100) %>%
  set_engine("randomForest")

small_forest <- rand_forest(mode = "regression", trees = 100, min_n = 50) %>%
  set_engine("randomForest")


cv_v <- 5
cv_splits<- vfold_cv(train, v = cv_v)
cv_results <- tibble(
        "1-SE" = map_dbl(cv_splits$splits,cv_model,tree_spec,TRUE),
        "small_tree" = map_dbl(cv_splits$splits,cv_model,small_tree),
        "big_tree" = map_dbl(cv_splits$splits,cv_model,big_tree),
        "big_forest" = map_dbl(cv_splits$splits,cv_model,big_forest),
        "small_forest" = map_dbl(cv_splits$splits,cv_model,small_forest))
```

```
plot_cv_res(cv_results)
```



- in this plot we can see the behavior of each fold, alltough it is not the confidence interval it is indicative to where are test results should be (between the top and buttom folds)
- we can see that the small forest achieved the best results, the big forest probably overfits and the single trees probably underfits
- for the small tree i used he tree_depth parameter and choose arbitrarily 5, for the large tree i just didnt specify anything which by default leads to the larges tree possible
- for the 1 - SE tree i used what we saw in class
- for the random forest i used the min_n argument for the small trees which prevents the tree from being very deep, for the large forest i didnt specify anything meaning each tree can be as large as possible

Q2

2. **AdaBoost and ϵ-Adaboost**
This problem refers to the AdaBoost algorithm (Freund and Schapire, 1997), which is used for binary classification with labels $y_i \in \{\pm 1\}$. Adaboost initializes $\hat{f}_0 = 0, w_i \equiv 1$, then for $t = 1...T$ updates:

   (a) Fit a classification tree with response y and weights w on the observations, getting tree h_t
   (b) Denote by Err_t the (weighted) misclassification error of h_t
   (c) Set $\alpha_t = 0.5 \log\left((1 - Err_t)/Err_t\right)$
   (d) Update weights: $w_i \leftarrow w_i \exp\left(-\alpha_t(y_i h_t(x_i))\right)$

   The model after step t is $f_t(x) = \sum_{u=1}^{t} \alpha_u h_u(x)$, the final model is f_T, and classification if according to the sign of $f_T(x)$.
   As we said in class, from the coordinate descent perspective of boosting, AdaBoost can be viewed as gradient boosting with loss function $L(y, \hat{y}) = \exp(-y\hat{y})$ and line-search steps, explicitly:

   - Fitting a classification tree is minimizing $\sum_i w_i y_i h_k(x_i) = \langle wy, h_k(X) \rangle$ (so $w_i y_i$ is the actual gradient)
   - The calculated α_t is the solution to the line search problem: $\alpha_t = \arg\min_\alpha \sum_i L\left(y_i, (f_{T-1} + \alpha h_t)(x_i)\right)$
   - The updated w_i is indeed (proportional to) the absolute value of the gradient:

$$w_i \propto \left| \frac{dL\left(y_i, l\right)}{dl}\Big|_{l=\hat{f}_t(x_i)} \right|$$

   (a) Choose one of the three properties above and prove explicitly that it holds (for example, if you choose the first one, show how fitting h_k classification tree minimizing weighted miclassification error is equivalent to choosing a coordinate descent direction in the exponential loss function).

(b) The code in www.tau.ac.il/~saharon/StatsLearn2022/AdaBoost.r implements AdaBoost on the competition data for the problem of whether $y > 3$ or $y \leq 3$. Read the code carefully to make sure you understand the details. Note especially the parameters "method" and "weights" that rpart takes.

i. With 8000-2000 training-test division as in the code, run the algorithm for 1000 iterations and draw a plot of training and test misclassification as a function of iterations. Explain its form.

ii. Change the algorithm from line-search boosting to ϵ-boosting with $\epsilon = 0.01$, not changing the other parameters. Run this version for 1000 iterations and draw the same plot. Discuss the results and compare to the line-search version.

iii. Now change the loss function from the exponential loss to squared error loss, and the approach to the regular gradient boosting with regression tree. Explain briefly in writing what you did, and run with the same parameters ($\epsilon = 0.01, T = 1000$). Classify according to sign of $\hat{y}$ and repeat the same analysis again. Discuss the relative results.

iv. (* +5) Play with the parameters in one (or all) of the algorithms (tree depth or other stopping criteria, ϵ, T) to change the results. Show how you can guarantee much better training results. Can you find settings that give much better testing results?

a

i'll prove the 2nd property:

$$\alpha_t = argmin_\alpha \sum_i L(y_i, F^{(t-1)} + \alpha h_t) := argmin_\alpha L_\alpha$$

$$\frac{\partial L_\alpha}{\partial \alpha} = \sum_i \frac{\partial exp\{y_i(F^{(t-1)} + \alpha h_t)\}}{\partial \alpha} = \sum_i \frac{\partial exp\{y_i F^{(t-1)}\} exp\{y_i \alpha h_t\}}{\partial \alpha} = \sum_i [y_i h_t exp\{y_i F^{(t-1)}\} exp\{y_i \alpha h_t\}] =$$

$$= \sum_{y_i = h_t} e^{y_i F^{(t-1)}} e^{-\alpha} - \sum_{y_i \neq h_t} e^{y_i F^{(t-1)}} e^{\alpha} := 0 \Rightarrow$$

$$\Rightarrow \sum_{y_i = h_t} e^{y_i F^{(t-1)}} e^{-\alpha} = \sum_{y_i \neq h_t} e^{y_i F^{(t-1)}} e^{\alpha} \Rightarrow$$

$$\Rightarrow \sum_{y_i = h_t} e^{y_i F^{(t-1)}} = \sum_{y_i \neq h_t} e^{y_i F^{(t-1)}} e^{2\alpha} \Rightarrow$$

$$\text{denote } w_i = w_i^{(t)}, \quad err = err^{(t)}$$

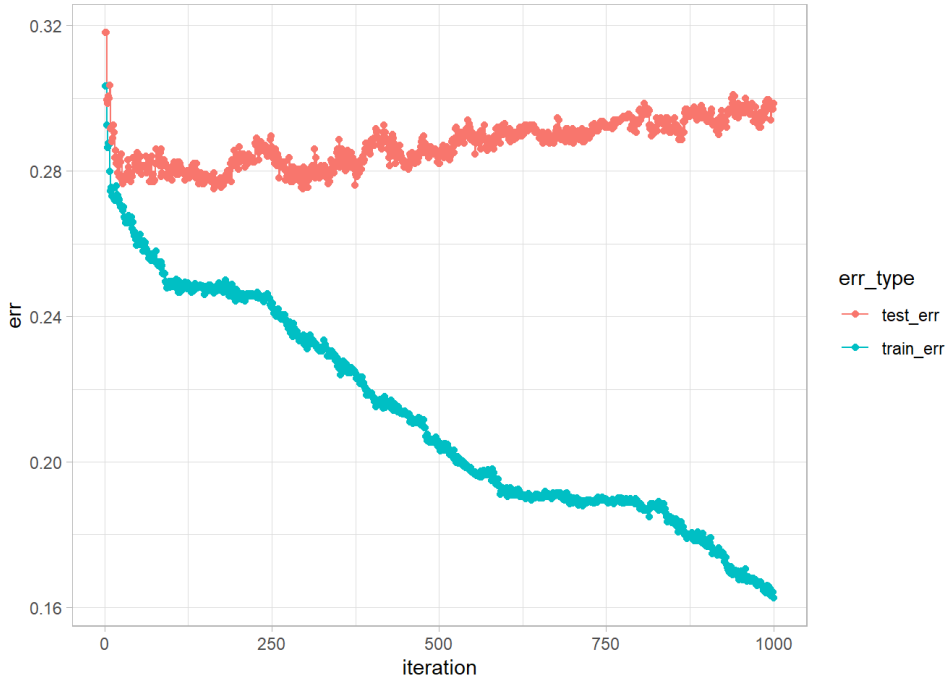
$$\Rightarrow e^{2\alpha} = \frac{\sum_{y_i = h_t} e^{y_i F^{(t-1)}}}{\sum_{y_i \neq h_t} e^{y_i F^{(t-1)}}} = \frac{\sum_{y_i = h_t} w_i}{\sum_{y_i \neq h_t} w_i} = \frac{\sum_i w_i I(y_i = h_t)}{\sum_{y_i \neq h_t} w_i} = \frac{\sum_i w_i I(y_i = h_t)}{\sum_i w_i I(y_i \neq h_t)} =$$

$$= \frac{1 - err}{err} \Rightarrow \alpha_t = \frac{1}{2} ln(\frac{1 - err}{err})$$

b

i

```
res1 = train_val_ada_boost()

res1 %>%
  pivot_longer(cols = !iteration,names_to = "err_type",values_to = "err") %>%
    ggplot(aes(iteration, err,group=err_type,color = err_type)) +
    geom_line(aes(group=err_type)) +
    geom_point() +
    theme_light()
```



- as we increase the number of iterations the training error and test error decreases until ~ iteration 200, this is where the test error increases.
- this is due to the fact that the adaboost algorithm overfits the training data, this is why the training error keeps decreasing while the test error increases

ii

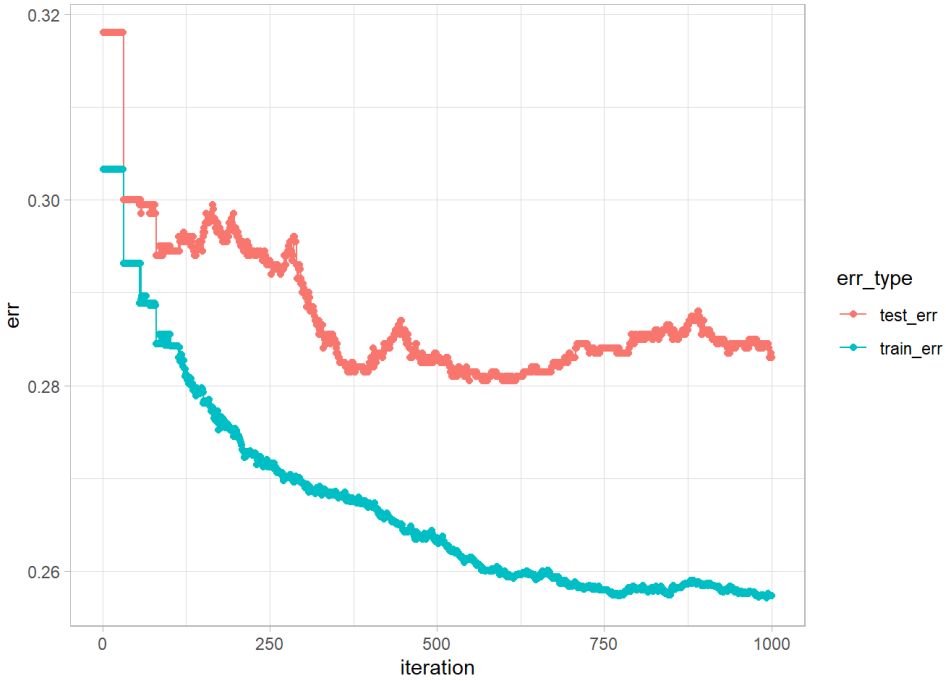
```
res2 <- train_val_ada_boost(epsilon = 0.01)

res2 %>%
  pivot_longer(cols = !iteration,names_to = "err_type",values_to = "err") %>%
    ggplot(aes(iteration, err,group=err_type,color = err_type)) +
    geom_line(aes(group=err_type)) +
    geom_point() +
    theme_light()
```

- we can see that the train error decrease much slower, also the test error does increase as much as in the line search, this is due to the fact that we are taking a fixed step and not the best step as in line search.

iii

- i changed the objective of the trees from the original y to the gradients

```
train_val_ada_boost_mse <- function(maxdepth = 2, cp = 0.00001, epsilon = 0.01, n_iter = 1000){
  gbm_tr = trtr
  train_miss_calssification <- c()
  test_miss_calssification <- c()
  w.now = rep (1, dim(trtr)[1]) # initialize w=1
  err.boost=err.tr.boost=NULL
  pred.boost = numeric(dim(va)[1])
  tr.boost = numeric(dim(trtr)[1])
  alpha = epsilon
  for (i in 1:n_iter){
      tree.mod= rpart (y~.,data=gbm_tr,method="anova",maxdepth=maxdepth,cp=cp)
      yhat.now = predict(tree.mod)
      pred.boost = pred.boost + alpha*predict(tree.mod, newdata=va)
      tr.boost = tr.boost + alpha*yhat.now
      gbm_tr$y = trtr$y-tr.boost
      train_err = mean (sign(tr.boost)!=trtr$y)
      test_err = mean (sign(pred.boost)!=va$y)
      # cat (i, "train:", train_err, " test:", test_err,"\n")
      train_miss_calssification = c(train_miss_calssification,train_err)
      test_miss_calssification = c(test_miss_calssification,test_err)

  }
  out_tbl <- tibble(train_err = train_miss_calssification,
         test_err = test_miss_calssification,
         iteration = seq(1,n_iter))
  return(out_tbl)
}

res3 <- train_val_ada_boost_mse()

res3 %>%
  pivot_longer(cols = !iteration,names_to = "err_type",values_to = "err") %>%
    ggplot(aes(iteration, err,group=err_type,color = err_type)) +
    geom_line(aes(group=err_type)) +
    geom_point() +
    theme_light()
```

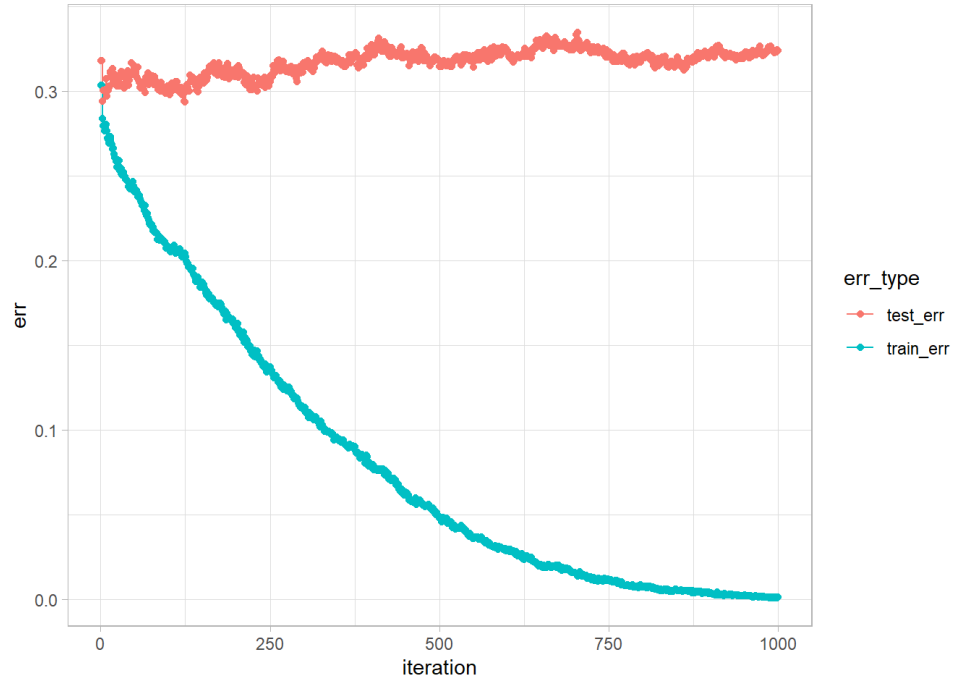


- the results are not that good, firstly we achive a worse train and test error than before, second we get stuck rather quickly, changing the learning rate might help

iv

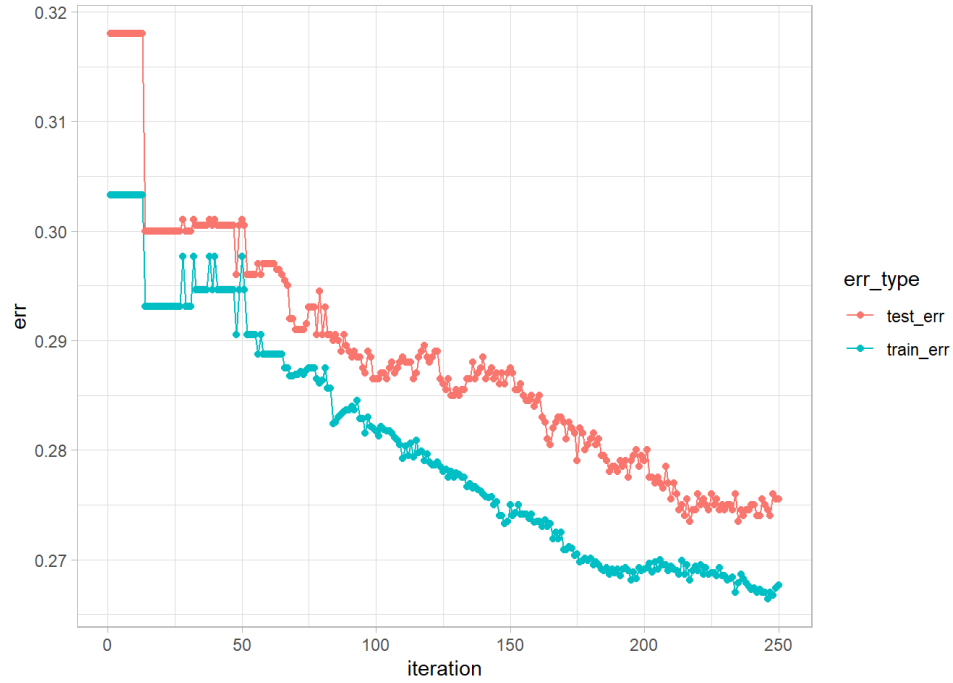
training results

```
res4 <- train_val_ada_boost(cp = 0, maxdepth = 3)
res4 %>%
  pivot_longer(cols = !iteration,names_to = "err_type",values_to = "err") %>%
    ggplot(aes(iteration, err,group=err_type,color = err_type)) +
    geom_line(aes(group=err_type)) +
    geom_point() +
    theme_light()
```



testing results

```
res2 <- train_val_ada_boost(epsilon = 0.005, n_iter = 250, cp = 0.0001)

res2 %>%
  pivot_longer(cols = !iteration,names_to = "err_type",values_to = "err") %>%
    ggplot(aes(iteration, err,group=err_type,color = err_type)) +
    geom_line(aes(group=err_type)) +
    geom_point() +
    theme_light()
```



- we can see that we get a better test error than before, this is due to stopping before the model starts overfitting, and chossing a better learning rate

Q3

3. **ESL 7.6 (7.5 in first edition): Degrees of freedom of Nearest Neighbors**
Prove that in the standard i.i.d error model (which the book calls "additive error"), the effective degrees of freedom of k-NN with N observations is N/k.

lets formalate KNN as a linear estimator:
denote: $N_k(x)$ the k nearest niebors of x

$$\hat{y}_i = \frac{1}{k} \sum_{x_j \in N_k(x_i)} y_j = \frac{1}{k} \sum_j y_j I(x_j \in N_k(x_i))$$

difine $S \in \mathbb{R}^{n \times n}$

$$S_{ij} = \begin{cases} 1/k & x_i \in N_k(x_j) \\ 0 & x_i \notin N_k(x_j) \end{cases}$$

thus the dof that the k-nn model has is: $trace(S) = \frac{N}{k}$

Q4

4. **Neural networks:**

   (a) Assume we are given a modeling problem with $x \in \mathbb{R}^p$ and $y \in \{0, 1\}$, which are can treat as a regression or classification problem (but prediction is always by comparing predictions to 0.5 and predicting either 0 or 1). For the following popular models, describe a neural network that implements them:

   - Standard linear regression
   - Logistic regression

   Explain in what sense the network implements them. Specifically, do we expect to get the same fitted model from the network as from the regular model when applied to data? Why yes or why not?

   (b) The code nn.r reads the South African heart dataset, divides it into training and test sets, and uses Keras to apply a NN with one hidden neuron and logit (=sigmoid) activation. It also applies and tests logistic regression. Use this skeleton to:

   i. Implement all four models described in the previous part
   ii. Prepare 2*2 confusion tables of predicted vs. actual labels
   iii. Briefly discuss the results compared to your expectations from the previous section

   (c) Implement a more complex architectures (e.g., a hidden layer with three nodes, and then an output layer, see commented code in the file) and apply it to the data. You may play with some of the parameters if necessary. Discuss its test-set performance.

   (d) Implement a network with a hidden layer with three nodes and an output layer, where all activations are linear. What form does the final model have? What functions of the original x variables are being fitted?

   Resources for this problem:
   Keras help
   Keras in R

   Note: You are of course welcome to solve and submit this problem in Python or any other environment, all Keras/Tensorflow models in the example R code should be easily transportable.

a

linear activation function: $f(x^t w) = \sum_i x_i w_i$

for both models i will use a neural net with an input layer and output layer with no hidden layers:
linear regression:
we can use a linear activation function and a L_2 loss function and our model will be:
$I(Xw > 0.5)$, where X is the input matrix and w is the vector of our weights
thus the neural nets optimizes the following problem via GD: $w = argmin_w ||Xw - y||$
due to the convexity of the problem if we choose a small enouth learning rate we will achive the same solution as linear regression
logistic regression:
we can use a sigmoid activation function and a cross entropy loss function and our model will be:

$$I(\sigma(Xw) > 0.5)$$

thus the neural nets optimizes the following problem via GD: $w = argmin_w - \sum_i y_i ln(\sigma(Xw)_i) + (1 - y_i) ln(1 - \sigma(Xw)_i)$

we can expect here different solutions beacuse the optimization algorithms differ, but the solutions should be rather similar
if we choose a small enough tolerance for the algorithm we should reach the global minima - thus the same solution
both networks implements the regression models in the sense that they seek to solve the same problem but via a different algorithm
note that if we want to enclude bias in our model it needs to be in the input layer as an input of 1 (similar to what we do in OLS mat

b

logistic

```
heart = read.csv("https://web.stanford.edu/~hastie/ElemStatLearn/datasets/SAheart.data",row.names=1)
heart$famhist= as.numeric(heart$famhist == "Present") # need only numbers
heart=array(unlist(heart),dim=c(462,10)) # move from data frame to array for keras

n = dim(heart)[1]
p = dim(heart)[2]
test.id = sample(n,n/3)

x_train = heart[-test.id,-p]
y_train = heart[-test.id,p]
x_test = heart[test.id,-p]
y_test = heart[test.id,p]


batch_size <- 32
epochs <- 1000
model_logistic <- keras_model_sequential()

model_logistic %>%
  layer_dense(units = 1, activation = 'sigmoid', input_shape = c(dim(x_train)[2]))


summary(model_logistic)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense (Dense)                (None, 1)                 10
## =========================================================================
## Total params: 10
## Trainable params: 10
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model_logistic %>% compile(loss = 'binary_crossentropy',optimizer = optimizer_adam(),metrics = c('accuracy'))

model_logistic %>% fit(x_train, y_train, validation_data = list(x_test, y_test),
  batch_size = batch_size,epochs = epochs,verbose = 1)
```

```
print("neural net")
```

```
## [1] "neural net"
```

```
phat_NN_1_sig = predict(model_logistic,x_test, batch_size = NULL, verbose = 0, steps = NULL)
summary(phat_NN_1_sig)
```

```
##        V1
##  Min.   :0.001943
##  1st Qu.:0.147885
##  Median :0.302227
##  Mean   :0.339609
##  3rd Qu.:0.512195
##  Max.   :0.824366
```

```
tbl = table(phat_NN_1_sig>0.5, y_test) # 2*2 table
print(tbl)
```

```
##        y_test
##          0  1
##   FALSE 83 30
##   TRUE  13 28
```

```
paste("the test set preformance is:" , round(sum(diag(tbl))/sum(tbl),4))
```

```
## [1] "the test set preformance is: 0.7208"
```

```
# logistic regression
print("logistic regression")
```

```
## [1] "logistic regression"
```

```
mod.lr = glm(y~., data=data.frame(x=x_train,y=y_train), family=binomial)
phat_logit = predict.glm(mod.lr, newdata = data.frame(x=x_test,y=y_test),type="response")
table(phat_logit>0.5, y_test) # 2*2 table
```

```
##       y_test
##         0  1
##   FALSE 80 27
##   TRUE  16 31
```

- the results seems as we would expect, they are close but not the same

linear

```
batch_size <- 32
epochs <- 1000
model_lr <- keras_model_sequential()
model_lr %>%
  layer_dense(units = 1, activation = 'linear', input_shape = c(dim(x_train)[2]))


summary(model_lr)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_1 (Dense)              (None, 1)                 10
## =========================================================================
## Total params: 10
## Trainable params: 10
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model_lr %>% compile(loss = 'mse',optimizer = optimizer_adam(),metrics = c('accuracy'))

model_lr %>% fit(x_train, y_train, validation_data = list(x_test, y_test),
  batch_size = batch_size,epochs = epochs,verbose = 1)
```

```
print("neural net")
```

```
## [1] "neural net"
```

```
phat_NN_1_sig = predict(model_lr,x_test, batch_size = NULL, verbose = 0, steps = NULL)
summary(phat_NN_1_sig)
```

```
##       V1
##  Min.   :-2.35497
##  1st Qu.: 0.06143
##  Median : 0.25823
##  Mean   : 0.27359
##  3rd Qu.: 0.50915
##  Max.   : 1.75110
```

```
tbl = table(phat_NN_1_sig>0.5, y_test) # 2*2 table
print(tbl)
```

```
##       y_test
##         0  1
##   FALSE 81 31
##   TRUE  15 27
```

```
paste("the test set preformance is:" , round(sum(diag(tbl))/sum(tbl),4))
```

```
## [1] "the test set preformance is: 0.7013"
```

```
print("linear regression")
```

```
## [1] "linear regression"
```

```
mod.lr = lm(y~., data=data.frame(x=x_train,y=y_train))
phat_logit = predict(mod.lr, newdata = data.frame(x=x_test))
table(phat_logit>0.5, y_test) # 2*2 table
```

```
##        y_test
##          0  1
##   FALSE 81 30
##   TRUE  15 28
```

- we would expect the results to be exactly the same, we are not getting the same results thus we can tweak the parameters of the optimizer or use a different one, (this results changes with knitting the notebook, i sometimes get very close results [even the same ones], and sometimes very different ones, usually its the first case)

C

```
model_3_layers <- keras_model_sequential()
model_3_layers %>%
  layer_dense(units = 3, activation = 'relu', input_shape = c(dim(x_train)[2]))%>%
  layer_dense(units = 2, activation = 'relu')%>%
  layer_dense(units = 1, activation = 'sigmoid')


summary(model_3_layers)
```

```
## Model: "sequential_2"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_4 (Dense)              (None, 3)                 30
## _________________________________________________________________________
## dense_3 (Dense)              (None, 2)                 8
## _________________________________________________________________________
## dense_2 (Dense)              (None, 1)                 3
## =========================================================================
## Total params: 41
## Trainable params: 41
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model_3_layers %>% compile(loss = 'binary_crossentropy',optimizer = optimizer_adam(),metrics = c('accuracy'))

model_3_layers %>% fit(x_train, y_train, validation_data = list(x_test, y_test),
  batch_size = batch_size,epochs = epochs,verbose = 1)
```

```
phat_NN_1_sig = predict(model_3_layers,x_test, batch_size = NULL, verbose = 0, steps = NULL)
summary(phat_NN_1_sig)
```

```
##        V1
##  Min.   :0.09614
##  1st Qu.:0.15378
##  Median :0.29337
##  Mean   :0.34364
##  3rd Qu.:0.50869
##  Max.   :0.78833
```

```
tbl = table(phat_NN_1_sig>0.5, y_test) # 2*2 table
print(tbl)
```

```
##        y_test
##          0  1
##   FALSE 81 28
##   TRUE  15 30
```

```
paste("the test set preformance is:" , round(sum(diag(tbl))/sum(tbl),4))
```

```
## [1] "the test set preformance is: 0.7208"
```

- i am not sure what will come up with this run, but sometimes we can see that the models has improved, and sometimes that it doesn't, i tried playing with the number of neuron at each layer, didnt came up with a stable result. this could be due to the fact that a larger network has higher flexibility so it can fit the data better, but it might also over fit, we are dealing with more than 40 parameters in the shallowest networks which might tend to overfit if we have a small dataset like in our case.

d

```
rm(model)
```

```
## Warning in rm(model): object 'model' not found
```

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 3, activation = 'linear', input_shape = c(dim(x_train)[2]))%>%
  layer_dense(units = 1, activation = 'linear')


summary(model)
```

```
## Model: "sequential_3"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_6 (Dense)              (None, 3)                 30
## _________________________________________________________________________
## dense_5 (Dense)              (None, 1)                 4
## =========================================================================
## Total params: 34
## Trainable params: 34
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model %>% compile(loss = 'mse',optimizer = optimizer_adam(),metrics = c('accuracy'))

model %>% fit(x_train, y_train, validation_data = list(x_test, y_test),
  batch_size = batch_size,epochs = epochs,verbose = 1)
```

$$\text{let } w_{ij}^{(k)} \text{ be the weights of the kth hidden layer between the ith node in layer k and jth node in layer k-1}$$

$$\text{the fist layer outputs: } o_i^{(1)} = x^t w_i^{(1)}, \text{ where } w_i \text{ is the vector of weights for the edges to node i}$$

$$\text{the 2nd layer (output layer) outputs: } \hat{y}_i = \sum_{k=1}^{3} w_k^{(2)} \sum_{j=1}^{n} x_j w_{kj}^{(1)} = w^{(2)t} W^{(1)t} X_i^t$$

- it is a linear function of the original x