Final Project Report

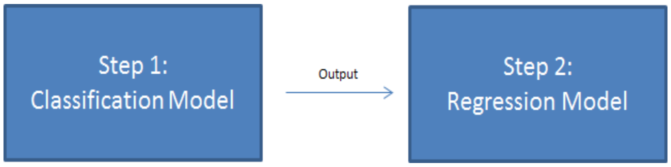
STAT 640: STATISTICAL LEARNING AND DATA MINING

Team: Probability1

FAYZAN TALPUR AND ZIWEI ZHOU

1. Overview

We have a two-step approach to this problem as shown in the following graph.



For Classification Model, The Idea is to find the most cross-correlated sentences and then use these sentences as either subsets or voting rule(s) to be used in the final regression model. For Example, sentence 1: "William Prefers Nineteen Dark Mails" is highly cross-correlated with Sentence 6:"William Prefers Nineteen Dark Nails". When we use the cross-correlation on Y Spectrum data, the results are as accurate as 25% to 60%. When the same approach is applied towards X, then the results are as high as 15% to 25%. These numbers were optimized for a static window that checks for max cross-correlation between test sentence and training sentence for a moving percentile. However, the correct way to optimize the classification model would be introduce moving windows with different percentiles between test and training sentence. In other words, the model can be improved by properly dividing the sentences into words and then classifying words instead of finding the most cross-correlated sentences.

For Regression Model, we applied a consistent approach here. We take the sentence to be predicted and divided it into 5 parts (corresponding to 5 words) and then fit a local model to each part. From the cross-correlation that came from step 1, we use correlation numbers as weights in re-sampling.

Our Final Model has an RMSE of 14.31 on private board and we ranked 18th in the final standing.

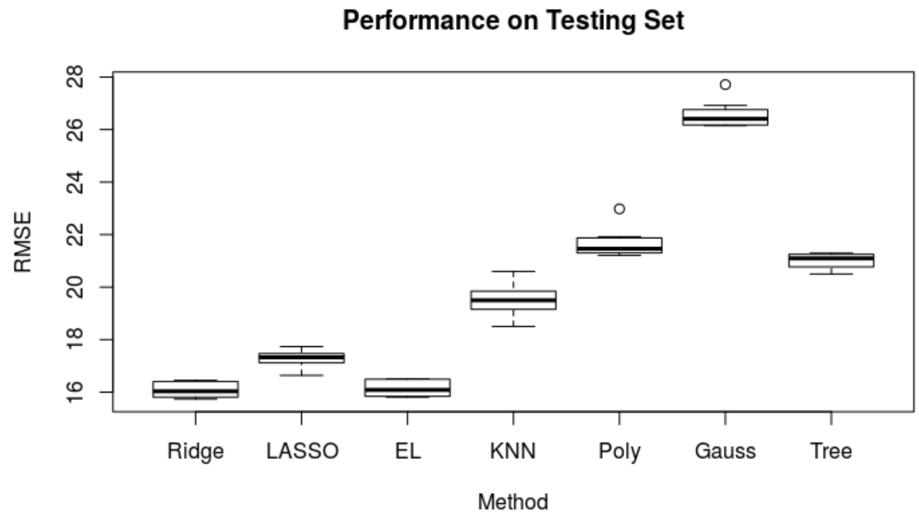
2. Base Learners

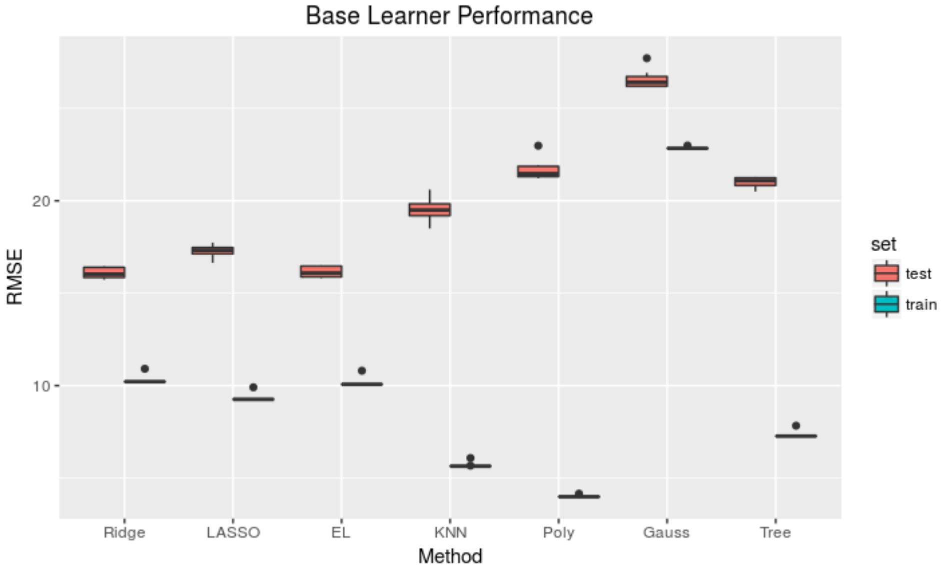
The base model we have tried:

- Ridge
- Lasso
- Elastic Net
- Kernel ridge (Polynomial Kernel and Gaussian Kernel)
- K-nearest neighbor (for regression)
- Trees

We tune each method using 10-fold CV method and report error on test set. We compared each base learner based on a subsetted dataset instead of whole dataset since some of the method require a lot computational resources.

As we expected method like kernel regression with polynomial kernel and k-nearest neighbor all suffer from overfitting problem. Gaussian kernel regression, however, underfit data, meaning Gaussian kernel doesn't describe the data very well. Polynomial kernel, K-NN and Tree have the lowest training error which indicate they can describe response very well, if combined with other method we may be able to reduce the variance and have better prediction. But kernel method and K-NN is very computational expensive. Performance also an important factor we should concern. Among all base learners Ridge have the best performance (low on prediction error and computational easy). We also selected Tree in addition to ridge as our base learner since variance of tree can be reduced via boosting.





3. Ensembles

To perform ensemble learning method, we need to further split the data:

Data Set	Function	Proportion
Training/ Validation Set	Training and validation	70%
Query Set	Calculate ensemble weight	20%
Test Set	Report final error	10%

We use two models: Ridge, Ridge (with feature expansion) and Tree. For each model we tune it in the training set separately and then calculate the ensemble weight from prediction error on query set:

$$w_i = \frac{\frac{1}{\epsilon_i}}{\sum_{i=1}^{n} \frac{1}{\epsilon_i}}$$

The procedure of ensemble method can be described as follows:

1. Repeat several times:
   a. Shuffle sentences
   b. Split the data (training/validation, query, test)
   c. Tuning each model under training/validation set
   d. Calculate ensemble weight on query set
2. Take mean value of ensemble weight
3. Predicting on test set using two models and combines the prediction together.

The resulting ensemble weight is as follows:

Base Learner	Weight	Individual RMSE
Ridge	0.36	15.78
Tree	0.27	20.71
Ridge (with feature expansion)	0.36	15.87

Using this method for unseen test data set we reduced MSE from 15.78 to 15.73

Individually, these base learners don't have good result, but when stacked together the prediction is better.

4. Model Validation & Assessment

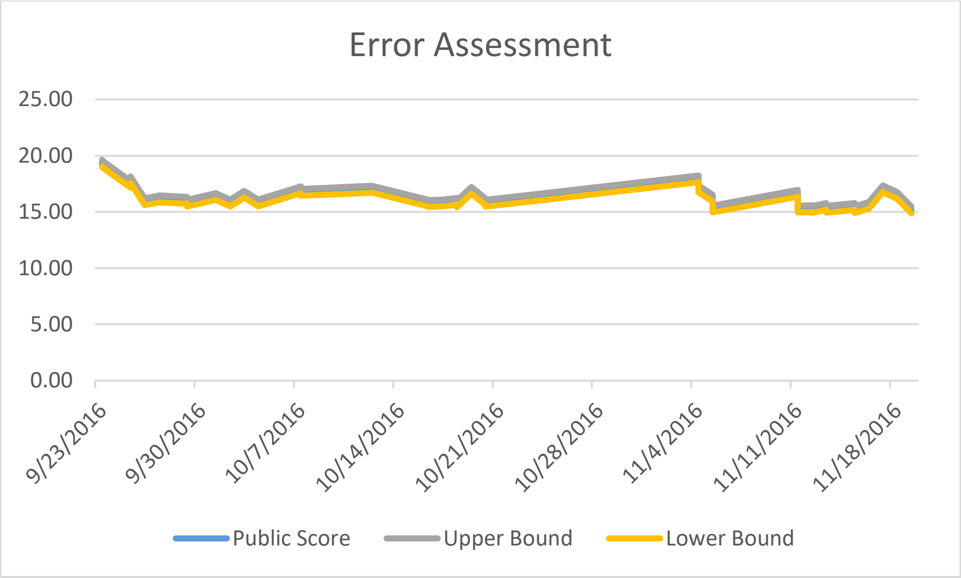
We split the dataset into 2 part: 112 sentences for training and validation, 28 sentences for testing. (Later we will further split the data for more advanced method such as ensemble learning). The split is randomized. For each method, we repeat the splitting several times to make sure we get the most accurate error rate. The process can be described as follows:

Repeat several times:

1. Shuffle sentences
2. Split the data (80% training/validation, 20% testing)
3. Parameter tuning on training/validation set
4. Assessing error test set

After each cycle we compare the mean value of testing error.

The error assessment shows on average; we were +/- 0.3 of our public RMSE submission. We have plotted the graph below to show this.



5. Best Scoring Model

Our Best Scoring Model uses the following steps

1) Classification

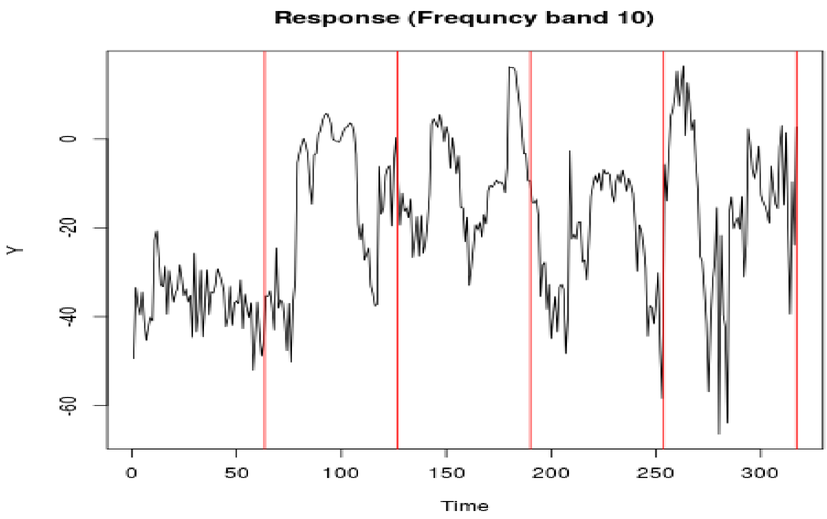
The classification model can be summarized into following steps

1. Use cross-correlation to screen X. Anything that has max cross-correlation of 10% higher was selected.
2. To use cross-correlation, we took the mean of these X's to get one-time series. (Average state of brain activity)
3. When comparing two sentences, truncate the larger one and perform the cross-correlation. Do this against all training sentences.
4. Use the cross-correlation numbers as weights for regression mode

By weighting training sentences with cross-correlation numbers, we reduced MSE from 15.30 to 15.17

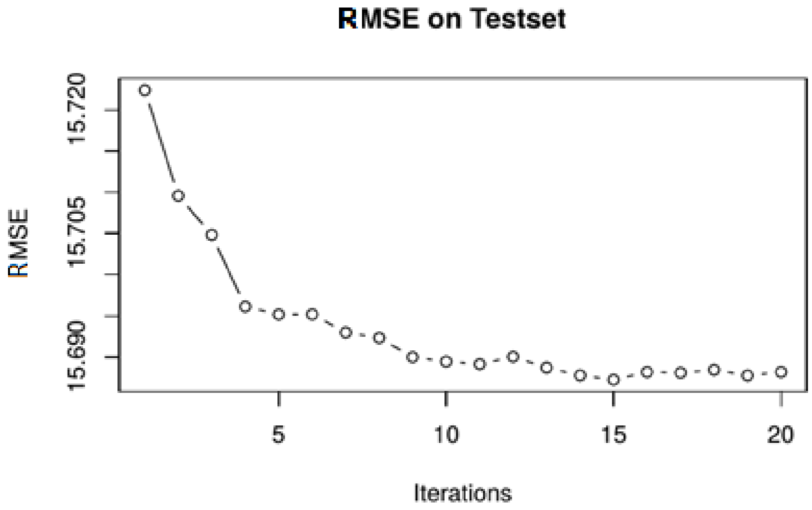
2) Local Models

We divided each sentence into 5 part (20% each part, representing 5 individual word). Ideally, we want to have separate model for each word. But in this way the parameter we need to tune will be five times than single model, and it's very hard to tune that many parameters. Besides, with more model we suffer overfitting problem. That's why we decide to reduce it two 3 parts instead (combined middle 3 into 1).



3) Bagging

We generate dataset for training from original dataset by bootstrapping with weight, which is calculated from autocorrelations between sentences. We tested bagging method on smaller training set and we can see that the prediction error is lowered in each iteration. By using bagging method RMSE is reduced from 15.28 to 15.18 (on public leaderboard)



4) Training Pipeline

The training and prediction process is carried out sentence by sentence. For each testing sentence, we calculate auto-correlation with training set and bootstrapping training set, repeat the process several times and take mean values.

The pipeline of our method is as follows:

1. Data Pretreatment (sentence wise scaling)

2. Variable Selection
3. Split data into training/set (80%) and testing data (20%)
4. Repeat b times:
    a. For testing sentences, calculate auto-correlation with training sentences
    b. Assign weight to each training sentence
    c. Generate training data for models by bootstrapping with weight
    d. Train and tuning models
    e. Make predictions
    f. Proceed to next sentences
5. Take average value of predictions

5) Performance

By using local models, we are able to reduce RMSE from 15.78 to 15.28, using bagging method further reduced to 15.18 and classification step reduced to 15.13.

The model finally gets RMSE 15.13 on public leaderboard.

6. Statistical Innovation

We think that our innovation comes from using cross-correlation to reduce the training set. Intuitively, the more training data that we use, the better our predictions become. However, here the problem is that more data is adding noise to our predictions. Using cross-correlation help us zone in set of training observation that was more useful for our predictions.

7. Interesting Neuroscience Findings

The most interesting findings we discovered are as following

1) There is lag between ECOG Data and sound being played. The lag is not that much but it is enough for us to use cross-correlation.
2) We Used univariate screening to search for the most correlated variables against Y. The Idea here is that brain activity should go up as the sound as being processed by certain parts of brain. We found the max cross-correlation for each X against each Y and then saved the results in a final correlation matrix. For any X, that has max cross-correlation greater than 10%, must be responding to stimuli from Y or sound. The 10% threshold was arbitrary and was chosen this number to limit the number of final X's that will be selected. In the end, we the final variables were the most useful variables.

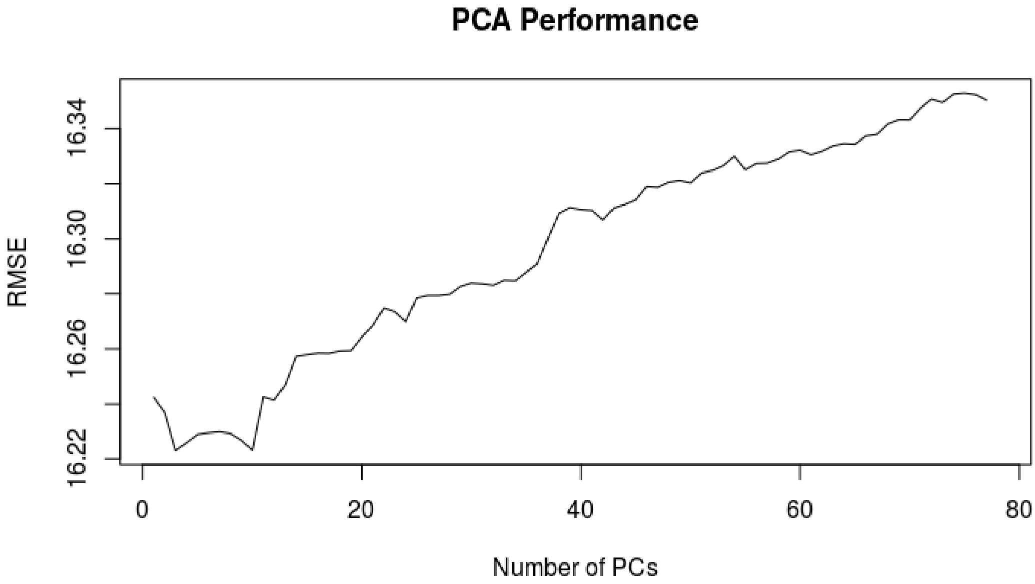
X Name	Node	Frequency
X14_G14_Delta	G14	Delta
X84_G14_Theta	G14	Theta
X154_G14_Alpha	G14	Alpha
X180_LMIT4_Alpha	LMIT4	Alpha
X223_G13_Beta	G12	Beta
X278_S4_Beta	S4	Beta
X293_G13_Low_Gamma	G13	Low Gamma
X347_S3_Low_Gamma	S3	Low Gamma

X Name	Node	Frequency
X362_G12_High_Gamma	G12	High Gamma
X363_G13_High_Gamma	G13	High Gamma
X364_G14_High_Gamma	G14	High Gamma
X372_G22_High_Gamma	G22	High Gamma
X373_G23_High_Gamma	G23	High Gamma
X405_LLF5_High_Gamma	LLF5	High Gamma

From what we can see in the cross-correlation matrix, two things are very clear to us

- High Gamma is the most useful frequency for decoding speech. It responded heavily to sound stimuli.
- The most useful Nodes are G12, G13 and G14. We believe that these nodes were placed on the audio cortex of the brain or they are pretty close to that region. This result is also consistent. We can see that these same nodes report the highest cross-correlation (in the 20%'s) and when reducing the threshold from 10%, also shows that more of this area is correlated against Y. Thus, there is stability as well.

3) The number of nodes that are useful for decoding speech is probably less than 15. A good way to think about is to use # of PCA components and measure RMSE as more PC's are added. We used 10-fold cross-validation to measure RMSE and got the following graph.



8. What We Learned

This project was much harder to approach and solve. The initial problem was how are we going to approach the problem. We tried dimension reduction, variable selection, cleaning data, different kernels, and then different methods just to improve the RMSE score. The end conclusion is that we need to understand data before applying any algorithm method. Understanding the structure of data or having an intuition of data helps a lot.

Also, we were steering our decisions based on lowering RMSE. However, measuring RMSE on time series is a very strict measure of error. When we created the two step model of classification and regression, the initial model had an RMSE of 23. However, when we plotted the prediction against the actual sentence. The result was that it was matching some responses or words really well. We used phase data to re-create the sound than it was an actual sentence. This was the first time that we were able to create sound instead of static. The lesson learned was that applying an algorithm blindly does not work. You need to have some sense of underlying data before making any model(s).

Appendix: R Code Used

```
################################################
## Multiple Model with Resampling (20%-60%-20%)
################################################


load("/scratch/zz38/org.RData")
source("function.R")
CVSwitch=1
lam <- rep(30,5)

## library(doMC)
## registerDoMC(3)
library(stringr)
library(parallel)

nocores <- detectCores() - 1
cl <- makeCluster(nocores,type='FORK')

PreProcess <- function(inData,mode="Delayed",pcaN=600)
{
    inData <- as.matrix(inData)
    if (mode=="Delayed" | mode=="DelayedPCA")
    {

    ## Train
        trans <- inData


        ## Future 1
        ##trans0 <- rbind(trans,0)
        ##trans0 <- trans0[-1,]

        ## Combine ExpSineSquared
        trans1 <- cbind(exp(sin( trans )^2),trans)
    ## trans1 <- cbind(sin(trans),cos(trans),sin(trans/2),cos(2*trans),sin(1/(trans+0.001)),trans1)
        ## trans1 <- cbind(trans1,trans^2,trans^3)
        outData0 <- trans1

        if (mode=="DelayedPCA")
        {
            ## PCA
            require(caret)
            outData0.df <- createDataframe(outData0,mode=1)
            PCAProc <- preProcess(outData0.df,method='pca',pcaComp=pcaN)
            outPC <- predict(PCAProc,outData0.df)
            outData <- cbind(1,outPC)
        }
        else
        {
            outData <- cbind(1,outData0)
        }
    }
```

```r
    return(outData)
}



RidgeFit <-  function(indf,lambda)
{
    Xtr <- as.matrix(PreProcess(indf[,33:ncol(indf)]))
    Ytr <- as.matrix(indf[,1:32])

    ## Fitting
    XTX <- t(Xtr) %*% Xtr
    beta <- solve(XTX + diag(rep(lambda/2*nrow(Xtr)),ncol(Xtr)))%*% t(Xtr)
%*% Ytr
    return(beta)
}


RidgePredict <- function(indf,betar,startpt=33)
{
    ## startpt is used to adapt for dataset including Y
    Xtr <- as.matrix(PreProcess(indf[,startpt:ncol(indf)]))
    return(Xtr %*% betar)
}


TreeFit <- function(indf,m,complex=0.01)
{
    indf <- as.data.frame(indf)
    require(rpart)
    TreeForm <- as.formula(paste("Y",m,"~.",sep=""))
    klist <- c(1:32)[-m]
    subdf <- indf[,-klist]

return(rpart(TreeForm,data=subdf,method="anova",control=rpart.control(cp=comp
lex)))
}

TreeFitP <- function(indf,cp=0.01,clst=cl)
{
    indf <- as.data.frame(indf)
    require(rpart)
    Tfit <- parLapply(clst,1:32,TreeFit,complex=cp,indf=indf)
    return(Tfit)
}

TreePredict <- function(indf,inmodelList,m)
{
    indf <- as.data.frame(indf)
    require(rpart)
    Tpredict <- predict(inmodelList[[m]],newdata=indf)
    return(Tpredict)
}

TreePredictP <- function(indf,inmodelList,clst=cl)
```

```
{
    indf <- as.data.frame(indf)
    require(rpart)
    TPredict <- lapply(1:32,TreePredict,indf=indf,inmodelList=inmodelList)
    TPredict.mtx <- matrix(unlist(TPredict),byrow=F,ncol=32)
    return(TPredict.mtx)
}

NFold <- function(fold=5,fold_id,inmatrix)
{
    nobs <- nrow(inmatrix)
    cut <- floor(nobs/fold)
    if ( fold_id != fold )
    {
        id_low <- cut * (fold_id -1) +1
        id_high <- cut * fold_id
    } else
    {
        id_low <- cut * (fold_id -1) +1
        id_high <- nobs
    }
    cvset_id <- id_low:id_high
    cvset <- inmatrix[cvset_id,]
    training <- inmatrix[-cvset_id,]
    return(list(training.idx=training,cvset.idx=cvset))
}



DataPartition <- function(breakpoints)
{
    ## Create index accroding to breakpoints
    ## Output list, with each member being a data chunk (sentence)
    nchunck <- nrow(breakpoints)

    breakpoints0 <- rbind(0,breakpoints)
    idx.partition <<- list()

    ## Cut word (5 WORD 20%)
    Wid <- matrix(nrow=nchunck,ncol=2*5)

    ## Structure of Wid
    ##
    ## -----------------------------------------------
    ## 1 | Word1 start | Word 1 end | Word 2 start |...
    ## -----------------------------------------------

    ## Generate cut point
    for ( i in 1:nchunck)
    {
        distance0 <-  abs( breakpoints0[i+1,1] - breakpoints0[i,1] )
        distance <- c()
        distance[1] <- 0
        distance[2] <- floor(distance0*0.20)
        distance[3] <- floor(distance0*0.40)
        distance[4] <- floor(distance0*0.60)
        distance[5] <- floor(distance0*0.80)
        distance[6] <- floor(distance0*1)


        for ( j in 1:5)
        {

            Wid[i,2*j-1] <- breakpoints0[i,1] + distance[j] +1
            Wid[i,2*j] <- breakpoints0[i,1] + distance[j+1]
        }


    }

    ## Adjust for end point
    Wid[,5*2] <- breakpoints[,1]
    return(Wid)
}


ObsJoin <- function(inid,indf)
{
    ## Assume its nx2 dimension
    ## to avoid condition statement
    outid <- inid[1,1]:inid[1,2]
    for ( i in 2:nrow(inid) )
    {
        ## print(i)
        temp <- inid[i,1]:inid[i,2]
        outid <- c(outid,temp)
    }
    outdf <- indf[outid,]
    return(outdf)
}


######################################################

keep <-
c(3,4,6,9,10,11,12,14,15,16,21,22,24,25,26,29,33,34,36,40,41,48,55,56,57,59,7
0,72,73,76,78,79,80,81,82,84,85,86,92,93,94,97,99,101,104,105,106,109

,111,122,123,125,129,137,138,139,143,144,148,149,150,151,153,154,155,156,162,
163,164,169,171,173,174,175,176,179,180,195,199,205,206,207,208,209

,210,211,212,213,214,215,216,217,218,219,221,223,224,226,231,232,233,234,239,
240,241,243,244,246,249,250,265,275,276,277,278,279,280,289,291,293

,296,302,307,313,314,318,319,328,335,341,345,346,347,348,349,350,354,355,358,
362,363,364,365,371,372,373,381,385,390,401,403,404,405,407,409,410
               ,411,413)

## Training Scale per sentence
breakpoints.tr <- read.table('train_breakpoints.txt',header=F)

perX.tr <- PerScaleX(X.tr.org[,keep],breakpoints.tr)
colnames(perX.tr) <- str_replace(colnames(perX.tr),"[A-Z]|[a-z]","X")
training <- createDataframe(perX.tr,Y.tr.org,mode=0)
```

```
colnames(training)[33:ncol(training)] <- colnames(perX.tr)

## Testing Scale per sentece
breakpoints.tst <- read.table('test_breakpoints.txt',header=F)
testingX <- PerScaleX(X.tst.org[,keep], breakpoints.tst)
colnames(testingX) <- str_replace(colnames(testingX),"[A-Z]|[a-z]","X")

## Reference data ( kaggle score 15.73 ) DO NOT ABUSE!!
Yref <- matrix(read.csv('/scratch/zz38/Prediction3.csv',header=T)[,2],ncol=1)



msestst <- msetst <- msecv <- c()

## Sampling weights
train_sentences <- read.table("/scratch/zz38/train_sentences.txt",header=F)
wordtable <- apply(train_sentences,2,table)
## small1 <- names(which(wordtable$V1<10))
small2 <- names(which(wordtable$V2<11))
small3 <- names(which(wordtable$V3<10))
small4 <- names(which(wordtable$V4<10))
small5 <- names(which(wordtable$V5<5))


weights <- 0.1
## weights <- ifelse(train_sentences$V1 %in% small1, weights+0.05 ,weights )
## weights <- ifelse(train_sentences$V2 %in% small2, weights+0.2 ,weights )
weights <- ifelse(train_sentences$V3 %in% small3, weights+0.2 ,weights )
weights <- ifelse(train_sentences$V4 %in% small4, weights+0.2 ,weights )
weights <- ifelse(train_sentences$V5 %in% small5, weights+0.6 ,weights )
weights <- matrix(weights,ncol=1)

CorResultTrain <- readRDS('/scratch/zz38/CorResultTrain.rds')

if (CVSwitch == 1)
{

#### Cross Validation

    ## Gerate index Matrix (sentences cut point) including word cut
    idpartition <- DataPartition(breakpoints.tr)

    ## Generate N fold cutpoint
    fold <- 10
    for ( f in 1:fold)
    {
        ## seprate training and cvset
        trainID <- NFold(fold,f,idpartition)$training.idx
        trainID.sen <- NFold(fold,f,matrix(c(1:140),ncol=1))$training.idx

        cvID <- NFold(fold,f,idpartition)$cvset.idx
        cvID.sen <- NFold(fold,f,matrix(c(1:140),ncol=1))$cvset.idx

        ## seprate cv dataset
        ## training set don't need this step, because non-continous
breakpoints
```

```
        ## reconstruct  cvset breakpoints (mimic application on test set)
        breakpoints.cv <- as.matrix( (cvID - (cvID[1]-1))[,6] )

        cvset.fold <- training[cvID[1]:cvID[length(cvID)],]
        mse.word <- matrix(nrow=nrow(breakpoints.cv),ncol=5)

        counter=1
        for (s in cvID.sen)
        {
            ## print(paste("Sentence:",s))

            weight <- as.matrix(CorResultTrain[[s]][,2])
            trainID.weight <- NFold(fold,f,weight)$training.idx

            ## Parallelized  model fitting
            ## print(paste("Fold:",f," Training..."))
            Iter=10
            for (it in 1:Iter)
            {
                ## 5 word separate training


print(paste("Sentence:",counter,"/",length(cvID.sen),"Iteration:",it))
                obs <-
sample(1:nrow(trainID),replace=F,size=100,prob=trainID.weight)

                fitting <- function(w)
                {
                    tr <- ObsJoin(trainID[obs,(2*w-1):(2*w)],training)
                    ## TreeFitP(tr,cp=5)
                    RidgeFit(tr,lam[w])
                }


clusterExport(cl,varlist=c("ObsJoin","RidgeFit","trainID.sen","trainID","trai
ning","lam","obs","PreProcess","TreeFitP"))
                betar <- parLapply(cl,1:5,fitting)

                ## ---------------------------------------
                ## Fitting Tree
                ## betar <- list()

                ## for (w in 1:3)
                ## {
                ##     tr <- ObsJoin(trainID[obs,(2*w-1):(2*w)],training)
                ##     betar[[w]] <- TreeFitP(tr,cp=5,cl)
                ## }


                ## Prediction

                breakpoints.cv0 <- rbind(0,breakpoints.cv)

                ## print(paste("Prediction...Sentence:",counter))
                cvset <- cvset.fold[
(breakpoints.cv0[counter,1]+1):breakpoints.cv0[counter+1,1],]
```

```
                cvsetX <- cvset[,33:ncol(cvset)]
                cvsetY <- cvset[,1:32]

                distance <- floor(nrow(cvset)/5)
                ## generate id first
                cv.wid <- matrix(nrow=5,ncol=2)
                for (w in 1:5)
                {
                    cv.wid[w,1] <- distance*(w-1)+1
                    cv.wid[w,2] <- distance*w
                }
                cv.wid[5,2] <- nrow(cvsetX)

                ## Apply model
                for (w in 1:5)
                {

                    cv.w <- cvsetX[cv.wid[w,1]:cv.wid[w,2],]
                    cv.w.y <- cvsetY[cv.wid[w,1]:cv.wid[w,2],]
                    ## prdcvw <- TreePredictP(cv.w,betar[[w]])
                    prdcvw <- RidgePredict(cv.w,betar[[w]],startpt=1)

                    if (w==1)
                    {
                        prdcv  <- prdcvw
                    } else
                    {
                        prdcv <-  rbind(prdcv,prdcvw)
                    }
                    mse.word[counter,w] <- mse(prdcvw,cv.w.y)
                }

                if (it == 1)
                {
                    prdtemp <- prdcv
                } else
                {
                    prdtemp <- prdtemp + prdcv
                }

                prdcv2 <- prdtemp/it
            }

            if (counter==1)
            {
                prdcv.result <- prdcv
            } else
            {
                prdcv.result <- rbind(prdcv.result,prdcv)
            }
            counter=counter+1

        }
        print(paste("Fold:",f))
        print("Word by Word MSE")
        print(apply(mse.word,2,mean,na.rm=T))
        print("Overall MSE")
        msecv[f] <- mse(prdcv.result,cvset.fold[,1:32])
        print(msecv[f])

    }


} else
{



#### Full Model Prediction

    ## Generate id Matrix for training set
    idpart <- DataPartition(breakpoints.tr)

    ## Load correlations
    CorResultTest <- readRDS('/scratch/zz38/CorResultTest.rds')

    ## Shuffle sentences ( no cv set)
    trainSentence <- sample(1:140,replace=F,prob=weights)
    trainID <- idpart[trainSentence,]
    trainID.sen <- c(1:140)

    ## Generate id matrix for testing set
    idpart.tst <- DataPartition(breakpoints.tst)
    ## testing dataset: testingX

    breakpoints.tst0 <- rbind(0,breakpoints.tst)
    for (i in 1:nrow(breakpoints.tst))
    {
        weights.ts <- as.matrix(CorResultTest[[i]][,2])
        tst <- testingX[ (breakpoints.tst0[i,1]+1):breakpoints.tst0[i+1,1], ]

        distance0 <- nrow(tst)

        distance <- c()
        distance[1] <- 0
        distance[2] <- floor(distance0*0.20)
        distance[3] <- floor(distance0*0.80)
        distance[4] <- floor(distance0*1)

        ## generate id first
        tst.wid <- matrix(nrow=3,ncol=2)

        ## Model Training
        Iter=10
        for (it in 1:Iter)
        {
            print(paste("Sentence:",i,"/70","Iteration:",it))
            obs <- sample(1:140,replace=F,size=100,prob=weights.ts)

            fitting <- function(w)
            {
                tr <- ObsJoin(trainID[obs,(2*w-1):(2*w)],training)
                RidgeFit(tr,lam[w])
            }
```

```
clusterExport(cl,varlist=c("ObsJoin","RidgeFit","trainID.sen","trainID","training","lam","obs","PreProcess","TreeFitP"))
            betar <- parLapply(cl,1:3,fitting)

            ## Fitting Tree Model
            ## betar <- list()

            ## for (w in 1:3)
            ## {
            ##     tr <- ObsJoin(trainID[obs,(2*w-1):(2*w)],training)
            ##     betar[[w]] <- TreeFitP(tr,cp=5,cl)
            ## }

            ## Prediction

            for (w in 1:3)
            {
                tst.wid[w,1] <- distance[w] + 1
                tst.wid[w,2] <- distance[w+1]
            }
            tst.wid[3,2] <- nrow(tst)

            ## Apply model
            for (w in 1:3)
            {
                tst.w <- tst[tst.wid[w,1]:tst.wid[w,2],]
                ## prdtstw <- TreePredictP(tst.w,betar[[w]])
                prdtstw <- RidgePredict(tst.w,betar[[w]],startpt=1)
                if (w==1)
                {
                    prdtst  <- prdtstw
                } else
                {
                    prdtst <-  rbind(prdtst,prdtstw)
                }

            }

            if (it==1)
            {
                prdtemp <- prdtst
            } else
            {
                prdtemp <- prdtemp + prdtst
            }
            prdtst2 <- prdtemp/it
        }

        ## Combine sentences back
        if (i==1)
        {
            prdtst.result <- prdtst
        } else
        {
            prdtst.result <- rbind(prdtst.result,prdtst)
        }

     }

## ## Visualize prediction
## library(plotly)
## plot_ly(z=prdtst.result,type="surface")

## flat and output
yhat.tst <- flatY(prdtst.result)
KaggleOutput(yhat.tst)
}



####################################
####################################
## Generic functions
####################################
####################################

createDataframe <- function(inDataX,inDataY,mode)
{
    ## Mode 0 return concate both X and Y together
    ## Mode 1 return only X
    outDataX <- data.frame(inDataX)
    px <- ncol(inDataX)
    py <- ncol(inDataY)
    for (i in 1:px)
    {
        colnames(outDataX)[i] <- paste("X",i,sep='')
    }
    if (mode==0)
    {
        outData <- data.frame(inDataY,outDataX)
        for (i in 1:py)
        {
            colnames(outData)[i] <- paste("Y",i,sep='')
        }
        return(outData)
    }

    if (mode==1) return(outDataX)

}


## Function to calculate MSE
mse <- function(prd,tru)
{
    m <- mean( (prd-tru)^2 )
    return(sqrt(m))
}

## Scale step by step
PerScaleX <- function(indata,breakpoints)
    {
        breakpoints0 <- rbind(0,breakpoints)
        outdata <- indata
        for (i in 1:dim(breakpoints)[1])
            {
                nl <- breakpoints0[i,1]
                nh <- breakpoints0[i+1,1]
                chunck <- indata[(nl+1):nh,]
                outdata[(nl+1):nh,] <- (scale(chunck))
                print(paste(i,nl,nh))
            }
        return(outdata)
        }


sMA <- function(inMatrix,alpha=0.8)
    {
        X1 <- rbind(inMatrix,0) ; X1 <- X1[-1,]
        X2 <- rbind(0,inMatrix) ; X2 <- X2[-nrow(X2),]
        X <- cbind(X1,X2)
        Y <- alpha*X1+(1-alpha)*X2
        return(Y)
        }

##########################################
## Randomized Cross Validation


DataPartition <- function(breakpoints)
{
    ## Create index accroding to breakpoints
    ## Output list, with each member being a data chunk (sentence)
    nchunck <- nrow(breakpoints)

    breakpoints0 <- rbind(0,breakpoints)
    idx.partition <<- list()

    ## Cut word (5 WORD 20%)
    Wid <- matrix(nrow=nchunck,ncol=2*3)

    ## Structure of Wid
    ##
    ## -----------------------------------------------
    ## 1 | Word1 start | Word 1 end | Word 2 start |...
    ## -----------------------------------------------

    ## Generate cut point
    for ( i in 1:nchunck)
    {
        distance <- floor( abs( breakpoints0[i+1,1] - breakpoints0[i,1] )/5 )

        for ( j in 1:3)
            {
                Wid[i,2*j-1] <- breakpoints0[i,1] + distance*(j-1) +1
                Wid[i,2*j] <- breakpoints0[i,1] + distance*j
            }

    }

    ## Adjust for end point
    Wid[,6] <- breakpoints[,1]
    return(Wid)
}

ObsJoin <- function(inid,indf)
{
    ## Assume its nx2 dimension
    for ( i in 1:nrow(inid) )
    {

        temp <- indf[inid[i,1]:inid[i,2],]
        if ( i == 1)
        {
            outdf <- temp
        } else
        {
            outdf <- rbind(outdf,temp)
        }
    }
    return(outdf)
}


reconstruct <- function(inlist,indf,nw=5)
{
    for (w in 1:nw)
    {
        temp <- ObsJoin(inlist[,(2*w-1):(2*w)],training)
        if (w==1)
        {
            outdf <- temp
        } else
        {
            outdf <-  rbind(outdf,temp)
        }

    }
    return(outdf)

}



Reshuffle <- function(inlist,indf,portion=0.9)
{
    ## Shuffle sentences,
    ## Input: Index chunck
    ## Output: portion% training set and 1-portion cvset
    n <- length(inlist)
    ## Generating sequence
    id <- sample(1:n)
    id.train <- id[1:floor(portion*n)]
    trainingset.idx <- inlist[id.train]
    cvset.idx <- inlist[-id.train]
    training.shuf <- reconstruct(trainingset.idx)
    cv.shuf <- reconstruct(cvset.idx)
    training.df <- indf[training.shuf,]
    cvset.df <- indf[cv.shuf,]
    return(list(training=training.df,cv=cvset.df))
}

flatY <- function(inmatrix)
{
    ## Flattern matrix to 1 column, refer to competition upload requirement
    outmatrix <- matrix(inmatrix,ncol=1)
}

KaggleOutput <- function(inmatrix)
{
    ## Output CSV file per competition requirement
    Ycsv <-
data.frame(Id=c(1:(dim(inmatrix)[1])),Prediction=as.vector(inmatrix))
    write.csv(Ycsv,file="Prediction.csv",row.names=F)
}
```