Computational Drug Discovery S01E01

Nacho Garcia

12/09/2019

Libraries

We will use **Rcpi and rcdk** to transform the SMILES into Chemical descriptors, **keras/tensorflow** for the ML part and **ggplot2** for plotting.

```r
library(Rcpi)
library(rcdk)
library(keras)
library(ggplot2)
```

Declaration of functions

We create the function *scale.db* to standardize new data based on old data. This makes different datasets comparable.

```r
#Standarize based on previous data
scale.db<-function(df.new, df.old){
  if(ncol(df.new)!=ncol(df.old)) print("Error number of columns!")
  for (i in 1:ncol(df.new)) {
    df.new[,i]<- (df.new[,i]-mean(df.new[,i]))/sd(df.new[,i])
  }
  return(as.matrix(df.new))
}
```

Folder assignment

We define where our training data is (*SMILES_Activity.csv*) and our problem data is (ZINCDB folder).

```r
#Folders
path<-"/home/nacho/StatisticalCodingClub/SMILES_Activity.csv"
pathDB<-"/home/nacho/StatisticalCodingClub/ZINCDB/"
```

Descriptor calculation

We use the functions *parse.smiles* to create a list and we extract the descriptor of the elements of the list with *extractDrugAIO*

```r
#Loading and descriptors calculation
df <- read.csv(path)
mols <- sapply(as.character(df$SMILE), parse.smiles)
desc <- extractDrugAIO(mols,  silent=FALSE)
```

Checkpoint

It is possible to save time by saving and loading the descriptors of the training dataset

```r
#Checkpoint
#write.csv(desc, "/home/nacho/StatisticalCodingClub/parameters.csv")
desc<-read.csv("/home/nacho/StatisticalCodingClub/parameters.csv")
```

Data cleaning

We need to remove columns with NA values or with only one value

```r
#Automatic cleaning
desc$X<-NULL

findNA<-apply(desc, 2, anyNA)
desc<-desc[,which(findNA==FALSE)]

findIdentical<- apply(desc, 2, sd)
desc<-desc[,which(findIdentical!=0)]
```

and other "problematic" columns (You can find those out after loading the drugs to evaluate)

```r
#Manual cleaning
to.remove<-c("ATSc1",
             "ATSc2",
             "ATSc3",
             "ATSc4",
             "ATSc5",
             "BCUTw.1l",
             "BCUTw.1h",
             "BCUTc.1l",
             "BCUTc.1h",
             "BCUTp.1l",
             "BCUTp.1h",
             "khs.tCH",
             "C2SP1")

`%!in%` = Negate(`%in%`)
desc<-desc[,which(names(desc) %!in% to.remove)]
```

Data standarization (or normalization)

By standardizing the data we force the mean of the data to be 0 and the standard deviation to be 1 (aka as Z score). By normalizing the data we force the values to range between 0 and 1.

$$Z_i = \frac{X_i - \mu(X)}{\sigma(X)}$$

$$N_i = \frac{X_i - min(X)}{max(X) - min(X)}$$

```r
x<-scale(desc)
# normalize <- function(x) {
#   return ((x - min(x)) / (max(x) - min(x)))
# }
#
# x<-apply(desc,2,normalize)

#y<-scale(df$Activity)
y<-(df$Activity-min(df$Activity))/(max(df$Activity)-min(df$Activity))
```

Splitting the data into training and validating sets

To know how well the model generalizes, we use a validation set. The model has never *seen* the SMILES from the validation set. The better the model performs in the validation set, the better it is. Models that perform well in the training set and bad in the validation set might suffer from overfitting.

```
#Train/Test
val.ID<-sample(c(1:1452),140 )

x.val<-x[val.ID,]
y.val<-y[val.ID]
x<-x[-val.ID,]
y<-y[-val.ID]
```

Creating and training the model using Keras/TensorFlow

Here we declare the model using keras_model_sequential() and we add layers using the pipeline operator %>% (Shortcut: *Ctrl+M*)

```
model<-keras_model_sequential()

model %>% layer_dense(units = 80, activation = "relu",  input_shape = (ncol(x))) %>%
  layer_dense(units = 40, activation = "relu") %>%
  layer_dense(units = 10, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

We then need to compile the model defining the optimizer and the loss function

```
compile(model, optimizer = "adagrad", loss = "mean_absolute_error")
```

And we train the model

```
history<-model %>% fit(x=x,
                       y=y,
                       validation_data=list(x.val, y.val),
                       batch_size=5,
                       epochs=5)

plot(history)
```

To check the performance of the model we predict the activity from the validation set and we compare them with the real activity.

```
#Prediction over validation

y.hat<-predict(model, x.val)

ggplot()+
  geom_jitter(aes(x=y.val, y=y.hat), colour="red", size=2, alpha=0.5)+
  geom_smooth(aes(x=y.val, y=y.hat))+
  xlab("Activity")+
  ylab("Predicted Activity")+
  theme_minimal()
```

Model optimization

In order to optimize the model we search for hyperparameters to find the architecture/parameters that perform better. There are several ways to do it in Keras but we are going to use a loop that randomly takes certain parameters.

```
#Optimization
reset_states(model)
rounds<-100
models<-as.data.frame(matrix(data = NA, ncol =9 ,nrow = rounds))
colnames(models)<-c("UnitsL1", "UnitsL2", "UnitsL3", "act", "L1.norm", "L2.norm", "L1.norm.Rate", "L2.no
```

We create a dataframe (*models*) to store all the information about the different parameters tested.

```
for (i in 1:rounds) {

  #Hyperparams
  unitsL1<-round(runif(1, min=20, max = 100))
  unitsL2<-round(runif(1, min=10, max = 60))
  unitsL3<-round(runif(1, min=2, max = 20))
  actlist<-c("tanh","sigmoid","relu","elu","selu")
  act<-actlist[round(runif(1,min = 1, max=5))]
  L1.norm<-round(runif(1,min = 0, max = 1))
  L2.norm<-round(runif(1,min = 0, max = 1))
  L1.norm.Rate<-runif(1, min = 0.1, max = 0.5)
  L2.norm.Rate<-runif(1, min = 0.1, max = 0.5)

  setTxtProgressBar(pb, i)
  model.search<-keras_model_sequential()

  model.search %>% layer_dense(units = unitsL1, activation = act, input_shape = (ncol(x)))
  if(L1.norm==1)  model.search %>% layer_dropout(rate=L1.norm.Rate)
  model.search %>% layer_dense(units = unitsL2, activation = act)
  if(L2.norm==1)  model.search %>% layer_dropout(rate=L2.norm.Rate)
  model.search %>% layer_dense(units = unitsL3, activation = act) %>%
  layer_dense(units = 1, activation = "sigmoid")

  compile(model.search, optimizer = "adagrad", loss = "mean_squared_error")
  model.search %>% fit(x=x,
                       y=y,
                       verbose=0,
                       batch_size=10,
                       epochs=10
  )

  y.hat<-predict(model.search, x.val)
  MSE<-(sum((y.val-y.hat)^2))/length(y.val)

  #FIlling dataframe
  models$MSE[i]<-MSE
  models$act[i]<-act
  models$UnitsL1[i]<-unitsL1
  models$UnitsL2[i]<-unitsL2
  models$UnitsL3[i]<-unitsL3
  models$L1.norm[i]<-L1.norm
  models$L2.norm[i]<-L2.norm
  models$L1.norm.Rate[i]<-L1.norm.Rate
  models$L2.norm.Rate[i]<-L2.norm.Rate
  reset_states(model.search)
  }
```

Then, we select the best model to evaluate its performance on the validation dataset.

```
#Validation best performer
models<-models[complete.cases(models),]
unitsL1<- models$UnitsL1[models$MSE==min(models$MSE)]
unitsL2<- models$UnitsL2[models$MSE==min(models$MSE)]
unitsL3<- models$UnitsL3[models$MSE==min(models$MSE)]

act<-models$act[models$MSE==min(models$MSE)]
L1.norm<-models$L1.norm[models$MSE==min(models$MSE)]
L2.norm<-models$L2.norm[models$MSE==min(models$MSE)]
L1.norm.Rate<-models$L1.norm.Rate[models$MSE==min(models$MSE)]
L2.norm.Rate<-models$L2.norm.Rate[models$MSE==min(models$MSE)]

model.best<-keras_model_sequential()

model.best %>% layer_dense(units = unitsL1, activation = act, input_shape = (ncol(x)))
if(L1.norm==1)  model.best %>% layer_dropout(rate=L1.norm.Rate)
model.best %>% layer_dense(units = unitsL2, activation = act)
if(L2.norm==1)  model.best %>% layer_dropout(rate=L2.norm.Rate)
model.best %>% layer_dense(units = unitsL3, activation = act) %>%
  layer_dense(units = 1, activation = act)

compile(model.best, optimizer = "adam", loss = "mean_squared_error")
model.best %>% fit(x=x,
                     y=y,

                     batch_size=10,
                     epochs=10
)

y.hat<-predict(model.best, x.val)

ggplot()+
  geom_jitter(aes(x=y.val, y=y.hat), colour="red", size=2, alpha=0.5)+
  geom_smooth(aes(x=y.val, y=y.hat))+
  xlab("Activity")+
  ylab("Predicted Activity")+

  theme_minimal()
```

Virtual screening

Now that we have established a trained a better model we are going to find the activity values for some unknown compounds.

```
#Screening
db.completed<-vector()
db<-list.files(pathDB)
db.toprocess<-setdiff(db,db.completed)
```

```
if(length(db.toprocess>0)) continue=TRUE
```

We check and load the files in the *pathDB* one by one. Once a file is processed it is not going to be processed anymore because we add the name of the file to the db.completed folder. In this way, the model never stops while there are files to process and new files can be added *on the fly*

```
while (continue) {

  batchsize<-10000
  batch.vector<-c(1,batchsize)

  smi.db<-read.csv(paste(pathDB,db.toprocess[1],sep=""), sep = " ", head=FALSE)
  db.completed[length(db.completed)+1]<-db.toprocess[1]

while(nrow(smi.db)>batch.vector[2]){
  mols.db <- sapply(as.character(smi.db$V1[batch.vector[1]:batch.vector[2]]), parse.smiles)
  descs.db <- extractDrugAIO(mols.db,  silent=FALSE)
  findNA<-apply(descs.db, 2, anyNA)
  descs.db<-descs.db[,which(findNA==FALSE)]
  desc.in.model<-colnames(desc)
  descs.db<-descs.db[,which(names(descs.db) %in% desc.in.model)]

  descs.db<-scale.db(descs.db,desc)

  predicted.act<-predict(model.best, descs.db)

  if(!exists("activity.db"))
   {activity.db<- predicted.act}
  else{ activity.db<-c(activity.db, predicted.act)}

  batch.vector[1]<-sum(batch.vector)
  batch.vector[2]<-batch.vector[2]+batchsize
  if(batch.vector[2]>nrow(smi.db)) batch.vector[2]<-nrow(smi.db)

}

if(!exists("compound.db"))
{compound.db<- smi.db$V2}
else{ compound.db<-c(compound.db, smi.db$V2)}

db.toprocess<-setdiff(db,db.completed)
if(length(db.toprocess==0)) continue==FALSE
}

output<-data.frame(compound.db,activity.db)
```

This part loads the files in batches because the parse.smiles function collapses if very large sets of SMILES are processed.

The last part of the script stores the activities in a dataframe