QSPR with 'camb'
Chemically **A**ware **M**odel **B**uilder

Daniel Murrell*[1,3] and Isidro Cortes-Ciriano†[2,3]

[1] *Unilever Centre for Molecular Science Informatics, Department of Chemistry, University of Cambridge, Cambridge, United Kingdom.*
[2] *Unite de Bioinformatique Structurale, Institut Pasteur and CNRS UMR 3825, Structural Biology and Chemistry Department, 25-28, rue Dr. Roux, 75 724 Paris, France.*
[3] *Equal contributors*

June 20, 2014

In the following sections, we demonstrate the utility of the `camb` package by presenting a pipeline which generates various aqueous solubility models using 2D molecular descriptors calculated by the PaDEL-Descriptor package as input features. These models are then ensembled to create a single model with a greater predictive accuracy.

Firstly, the package needs to be loaded and the working directory set:

```
library(camb)
setwd(path_to_working_directory)
```

*dsmurrell@gmail.com
†isidrolauscher@gmail.com

1 Compounds

1.1 Reading and Preprocessing

The compounds are read in and standardised. Internally, the Indigo C API [1], which is incorporated into the `camb` package, is use to perform this task. Molecules are represented with implicit hydrogens, dearomatized, and passed through the InChI format to ensure that tautomers are represented by the same SMILES.

The `StandardiseMolecules` function enables the representation of molecular structures in a similarly processed form. The different arguments of this function allow control over the maximum number of (i) fluorines, (ii) chlorines, (iii) bromines, and (iv) iodines the molecule contains in order to be retained for training. Inorgnaic molecules (those containing atoms not in {H, C, N, O, P, S, F, Cl, Br, I}) are removed if the argument `remove.inorganic` is set to `TRUE`. This is the function's default behaviour. The upper and lower limits for the molecular mass can be set with the arguments `min.mass.limit` and `max.mass.limit`. The name of the file containing the chemical structures is input to the argument `structures.file`.

```
StandardiseMolecules(structures.file="solubility_2007_ref2.sdf",
                     standardised.file="standardised.sdf",
                     removed.file="removed.sdf",
                     properties.file = "properties.csv",
                     remove.inorganic=TRUE,
                     fluorine.limit=3,
                     chlorine.limit=3,
                     bromine.limit=3,
                     iodine.limit=3,
                     min.mass.limit=20,
                     max.mass.limit=900)
```

Molecules that Indigo manages to parse and that pass the filters are written to the file indicated in the argument "standardised.file". Molecules that are discarded for training purposes are written to the file indicated in the "removed.file" argument.

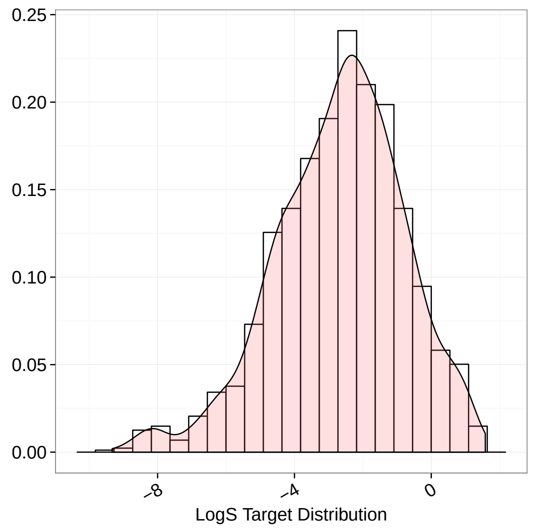

Figure 1: LogS Target Distribution

2 Target Visualisation

The properties specified in the structure file of all molecules and an index, in the column "kept", indicating which molecules were deleted (0) and kept (1), are written to the file indicated in the argument "properties.file" which is in CSV format. In this case the file is `"properties.csv"`.

```
properties <- read.table("properties.csv", header=TRUE, sep="\t")
properties <- properties[properties$Kept==1, ]
head(properties)
targets <- data.frame(Name = properties$NAME, target = properties$EXPT)
p <- DensityResponse(targets$target) + xlab("LogS Target Distribution")
p
```

2.1 PaDEL Descriptors

One and two-dimensional PaDEL[**padel**] descriptors and fingerprints can be calculated with the function ”GeneratePadelDescriptors”:

```
descriptors <- GeneratePadelDescriptors(standardised.file = "standardised.sdf",
    types = c("2D"), threads = 1)
descriptors <- RemoveStandardisedPrefix(descriptors)
saveRDS(descriptors, file = "descriptors.rds")
```

3 Statistical Pre-processing

Merge the calculated descriptors and the target values by name into a single data.frame. Check that the number of rows of the merged and original data.frames are the same. Split the data.frame into *ids*, x and y where *ids* are the molecule names, x are the descriptor values and y is the target values.

```
all <- merge(x = targets, y = descriptors, by = "Name")
ids <- all$Name
x <- all[3:ncol(all)]
y <- all$target
```

Sometimes, some descriptors are not calculated for all molecules, thus giving a ”NA” or ”Inf” as descriptor value. Instead of removing that descriptor for all molecules, the missing descriptor values can be imputed from the corresponding descriptor values of the rest of molecules. Descriptor values equal to ”Inf” are converted to ”NA”. For the imputation of missing descriptor values, the R package *impute* is required. Depending on the R version, it can be accessed from either *CRAN* or *Bioconductor*.

```
x.finite <- ReplaceInfinitesWithNA(x)
x.imputed <- ImputeFeatures(x.finite)
```

```
## Loading required package:  impute

## Cluster size 1606 broken into 375 1231
## Done cluster 375
## Done cluster 1231
```

Split the dataset into a training (80%) and a holdout (20%) set that will be used to assess the predictive ability of the models. Remove the following descriptors: (i) those with a variance close to zero (near-zero variance), and (ii) those highly correlated:

```
dataset <- SplitSet(ids, x.imputed, y, percentage = 20)
dataset <- RemoveNearZeroVarianceFeatures(dataset,
    frequencyCutoff = 30)

## 397 features removed with variance below cutoff

dataset <- RemoveHighlyCorrelatedFeatures(dataset,
    correlationCutoff = 0.95)

## 121 features removed with correlation above cutoff
```

Convert the descriptors to z-scores by centering them to have a mean of zero and scaling them to have unit variance:

```
dataset <- PreProcess(dataset)

## Error:  could not find function "PreProcess"
```

Given that cross-validation (CV) will be used to optimize the hyperparameters of the models, we divide the training set in 5 folds:

```
dataset <- GetCVTrainControl(dataset)

## Error:  could not find function "GetCVTrainControl"

saveRDS(dataset, file = "dataset_logS_preprocessed.rds")
```

All models are trained with the same CV options, *i.e.* the arguments of the function 'GetCV-TrainControl' to allow ensemble modeling (see below). It is important to mention that the functions presented in the previous code blocks depend on functions from the *caret* package, namely:

- RemoveNearZeroVarianceFeatures : nearZeroVar
- RemoveHighlyCorrelatedFeatures : findCorrelation
- PreProcess : preProcess
- GetCVTrainControl : trainControl

Experienced users might want to control more arguments of the underlying *caret* functions. This is certainly possible as the arguments given to the *camb* functions will be subsequently given to the *caret* counterparts. The default values of these function however permit the less experienced user to fo throught the statistical preprocessing steps with ease, though guaranteeing that the choice of the argument values is reasonable.

4 Model Training

In the following section we will present the different steps required to train a QSPR model with *camb*. It should be noted that the above steps can be run locally on a low powered computer such as a laptop and the preprocessed dataset saved to disk. This can then be copied to a high powered machine with multiple cores for model training and the resulting models saved back to the local machine. Pro tip: Dropbox can be used to sync this proceedure so that manual transfer is not required.

```
dataset <- readRDS("dataset_logS_preprocessed.rds")
# register the number of cores to use in training
registerDoMC(cores = 10)


## Error:  could not find function "registerDoMC"
```

4.1 Support Vector Machines (SVM)

Firstly, a SVM will be trained[**svmreview**]. We define an exponential grid (base 2) to optimize the hyperparameters. The `train` function from the `caret` package is used directly for model training.

```
method <- "svmRadial"
tune.grid <- expand.grid(.sigma = expGrid(-8, 4, 2,
    2), .C = c(1e-04, 0.001, 0.01, 0.1, 1, 10, 100))
model <- train(dataset$x.train, dataset$y.train, method,
    tuneGrid = tune.grid, trControl = dataset$trControl)
saveRDS(model, file = paste(method, ".rds", sep = ""))
```

4.2 Random Forest

We proceed similarly in the case of a random forest (RF) model[**rf**].

```
method <- "rf"
tune.grid <- expand.grid(.mtry = seq(5, 100, 5))
model <- train(dataset$x.train, dataset$y.train, method,
    tuneGrid = tune.grid, trControl = dataset$trControl)
saveRDS(model, file = paste(method, ".rds", sep = ""))
```

4.3 Gradient Boosting Machine

We proceed similarly in the case of a gradient boosting machine (GBM) model[**gbm**].

```
method <- "gbm"
tune.grid <- expand.grid(.n.trees = c(500, 1000), .interaction.depth = c(25),
    .shrinkage = c(0.01, 0.02, 0.04, 0.08))
model <- train(dataset$x.train, dataset$y.train, method,
    tuneGrid = tune.grid, trControl = dataset$trControl)
saveRDS(model, file = paste(method, ".rds", sep = ""))
```

Determine if your hyper-parameter search needs to be altered. In the following we focus on the RF model, though the same steps can be applied to the GBM and SVM models. If your hyper-parameters lead you to what looks like a global minimum then you can stop scanning the space of hyper-parameters, otherwise you need to adjust the grid and retrain your model.

```
model <- readRDS("rf.rds")
plot(model, metric = "RMSE")
```

5 Model Evaluation

Once the models are trained, the cross validated metrics can be calculated: We assume that the metric used for the choice of the best combination of hyperparameters is 'RMSE', which is normally considered as the aim of bioactivity modeling, *i.e.* how far (on average) are our predictions from the real the bioactivity values?.

```
print(RMSE_CV(model, digits = 3))


## Error:  could not find function "RMSE_CV"


print(Rsquared_CV(model, digits = 3))
```

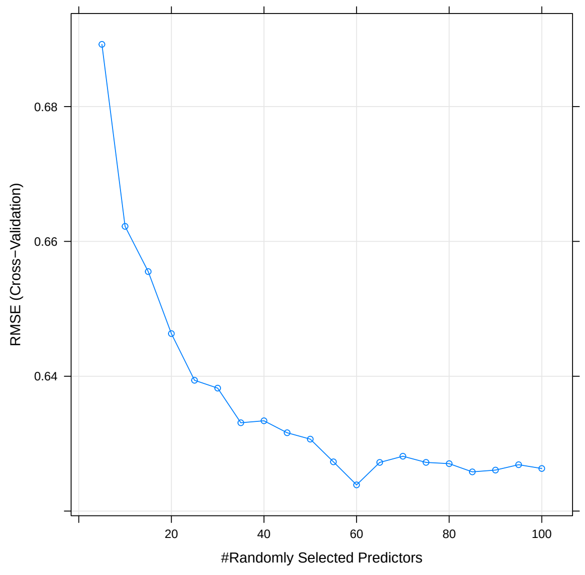

Figure 2: CV RMSE over the hyperparameters

```
## Error:  could not find function "Rsquared_CV"
```

6 Bibliography

References

[1] Indigo, "Indigo cheminformatics library," 2013. [Online]. Available: `http://ggasoftware.com/opensource/indigo/`.