

Project 3 - Baseline(GBM) and Improved(XGBoost with Canny feature) models

Grp2

Load library

```
if(!require("EBImage")){
  source("https://bioconductor.org/biocLite.R")
  biocLite("EBImage")
}

## Loading required package: EBImage
## Warning: package 'EBImage' was built under R version 3.5.1
if(!require("gbm")){
  install.packages("gbm")
}

## Loading required package: gbm
## Loaded gbm 2.1.4
library("EBImage")
library("gbm")
```

Step 0: specify directories.

Set the working directory to the image folder. Specify the training and the testing set. For data without an independent test/validation set, you need to create your own testing data by random subsampling. In order to obtain reproducible results, `set.seed()` whenever randomization is used.

```
set.seed(2018)
setwd("../")
```

Provide directories for training images. Low-resolution (LR) image set and High-resolution (HR) image set will be in different subfolders.

```
train_dir <- "../data/train_set/" # This will be modified for different data sets.
train_LR_dir <- paste(train_dir, "LR/", sep="")
train_HR_dir <- paste(train_dir, "HR/", sep="")
train_label_path <- paste(train_dir, "label.csv", sep="")
```

Step 1: set up controls for evaluation experiments.

In this chunk, we have a set of controls for the evaluation experiments.

- (T/F) cross-validation on the training set
- (number) K, the number of CV folds
- (T/F) process features for training set
- (T/F) run evaluation on an independent test set
- (T/F) process features for test set

```
run.cv=FALSE # run cross-validation on the training set
K <- 5 # number of CV folds
run.feature.train=FALSE # process features for training set
run.test=FALSE # run evaluation on an independent test set
run.feature.test=TRUE # process features for test set
run.train=FALSE
```

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications. In this example, we use GBM with different `depth`. In the following chunk, we list, in a vector, setups (in this case, `depth`) corresponding to models that we will compare. In your project, you might compare very different classifiers. You can assign them numerical IDs and labels specific to your project.

```
model_values <- seq(3, 11, 2)
model_labels = paste("GBM with depth =", model_values)
```

Step 2: import training images class labels.

We provide extra information of image label: car (0), flower (1), market (2). These labels are not necessary for your model.

```
#extra_label <- read.csv(train_label_path, colClasses=c("NULL", NA, NA))
```

Step 3: construct features and responses

`feature.R` should be the wrapper for all your feature engineering functions and options. The function `feature()` should have options that correspond to different scenarios for your project and produces an R object that contains features and responses that are required by all the models you are going to evaluate later. + `feature.R` + Input: a path for low-resolution images. + Input: a path for high-resolution images. + Output: an RData file that contains extracted features and corresponding responses

```
source("../lib/feature2.R")
tm_feature_train <- NA
if(run.feature.train){
  tm_feature_train <- system.time(dat_train <- feature(train_LR_dir, train_HR_dir))
  feat_train <- dat_train$feature
  label_train <- dat_train$label
  save(dat_train, file="../output/feature_train.RData")
}
```

Step 4: Train a classification model with training images

Call the train model and test model from library.

`train.R` and `test.R` should be wrappers for all your model training steps and your classification/prediction steps. + `train.R` + Input: a path that points to the training set features and responses. + Output: an RData file that contains trained classifiers in the forms of R objects: models/settings/links to external trained configurations. + `test.R` + Input: a path that points to the test set features. + Input: an R object that contains a trained classifier. + Output: an R object of response predictions on the test set. If there are multiple classifiers under evaluation, there should be multiple sets of label predictions.

```
source("../lib/train.R")
source("../lib/test.R")
```

Model selection with cross-validation

- Do model selection by choosing among different values of training model parameters, that is, the interaction depth for GBM in this example.

```
source("../lib/cross_validation.R")

if(run.cv){
  err_cv <- array(dim=c(length(model_values), 2))
  for(k in 1:length(model_values)){
    cat("k=", k, "\n")
    err_cv[k,] <- cv.function(feet_train, label_train, model_values[k], K)
  }
  save(err_cv, file="../output/err_cv.RData")
}
```

Visualize cross-validation results.

```
if(run.cv){
  load("../output/err_cv.RData")
  plot(model_values, err_cv[,1], xlab="Interaction Depth", ylab="CV Error",
       main="Cross Validation Error", type="n", ylim=c(0, 0.25))
  points(model_values, err_cv[,1], col="blue", pch=16)
  lines(model_values, err_cv[,1], col="blue")
  arrows(model_values, err_cv[,1]-err_cv[,2], model_values, err_cv[,1]+err_cv[,2],
        length=0.1, angle=90, code=3)
}
```

- Choose the “best” parameter value

```
model_best=model_values[1]
if(run.cv){
  model_best <- model_values[which.min(err_cv[,1])]
}

par_best <- list(depth=model_best)
```

GBM

- Train the model with the entire training set using the selected model (model parameter) via cross-validation.

```
tm_train=NA
if(run.train){
  tm_train_gbm <- system.time(fit_train_gbm <- train(feet_train, label_train, par_best))
  #tm_train <- system.time(fit_train <- train(feet_train, label_train))
  save(fit_train_gbm, file="../output/fit_train_gbm.RData")
}
```

XGBOOST

```
if(run.train){
  tm_train_xgboost <- system.time(fit_train_xgboost <- xgb_train(feet_train, label_train))
  save(fit_train_xgboost, file="../output/fit_train_xgboost.RData")
}
```

Step 5: Super-resolution for test images

Feed the final training model with the completely holdout testing data. + `superResolution.R` + Input: a path that points to the folder of low-resolution test images. + Input: a path that points to the folder (empty) of high-resolution test images. + Input: an R object that contains tuned predictors. + Output: construct high-resolution versions for each low-resolution test image.

```
source("../lib/superResolution.R")
test_dir <- "../data/test_set/" # This will be modified for different data sets.
test_LR_dir <- paste(test_dir, "LR/", sep="")
test_HR_dir <- paste(test_dir, "HR/", sep="")
###GBM test
if(run.test){
  load(file="../output/fit_train_gbm.RData")
  tm_test_gbm <- system.time(PSNR_gbm <- superResolution_gbm(test_LR_dir, test_HR_dir, fit_
})
#XGBoost test
if(run.test){
  load(file="../output/fit_train_gbm.RData")
  tm_test_xgboost <- system.time(PSNR_xgboost <- superResolution_xgboost(test_LR_dir, test_HR_dir, fit_
})
```

Summarize Running Time

Prediction performance matters, so does the running times for constructing features and for training the model, especially when the computation resource is limited.

```
cat("Time for constructing training features=", tm_feature_train, "s \n")
```

```
## Time for constructing training features= 10.936 s
```

```
cat("Time for training gbm model=", tm_train_gbm, "s \n")
```

```
## Time for training gbm model= 243.353 s
```

```
cat("Time for gbm super-resolution=", tm_test_gbm, "s \n")
```

```
## Time for gbm super-resolution= 2039.34 s
```

```
cat("Time for training xgboost model=", tm_train_xgboost, "s \n")
```

```
## Time for training xgboost model= 10.783 s
```

```
cat("Time for xgboost super-resolution=", tm_test_xgboost, "s \n")
```

```
## Time for xgboost super-resolution= 483.39 s
```

```
print(paste("PSNR_gbm:", PSNR_gbm, sep=" "))
```

```
## [1] "PSNR_gbm: 22.9720517131078"
```

```
print(paste("PSNR_xgboost:", PSNR_xgboost, sep=" "))
```

```
## [1] "PSNR_xgboost: 25.1509871835004"
```