Flowers Recognition

P.C.

12/11/2020

Dear reader,

This project aim to develop a machine learning algorithm able to recognize five types of flowers on pictures after training: daisies, dandelions, roses, sunflowers and tulips. The dataset used for this purpose is a subset of the second version of *Flower Recognition* uploaded by Alexander Mamaev on Kaggle and available at https://www.kaggle.com/alxmamaev/flowers-recognition (https://www.kaggle.com/alxmamaev/flowers-recognition).

This full dataset includes a couple of hundreds small pictures of those five different types of flower in individual folders (in total around 4000 flowers pictures). However by checking carefully the pictures, the full data set appears to be of average quality, we can notice for around a fourth of pictures one of the following issues:

- The flower is too small, not the main object of the picture, even for a few there is no flower at all on the picture;
- The picture is strongly overexposed, fading colors;
- The picture is a close up, *e.g.* a zoom on a single petal with shallow depth of field;
- The flower is wrongly classified, *e.g.* we find many dahlias classified as roses or dandelions, chamomilles and purple anemones classified as daisies, lavender, hortensias and hyacinths classified as tulips, etc;
- The flower is at a very late stage of its life and has no more (or very few withered) petals.

So I finally have decided to make my own subset of this dataset with 500 pictures of each flower type manually selected for their good quality. This data set is stored on my git hub and can be downloaded with the piece of code below:

```
# Downloading the different zip files from my GitHub. The data set is stored in 6 zip files, Flowers1.zip to Flowers6.zip.

for(n in c(1:6)) {

dl <- tempfile()
download.file(paste("https://github.com/cordierpc/FlowersRecognition/raw/main/Flowers", n, ".zip", sep = ""), dl)
unzip(dl)

}

rm(dl, n)
```

This project is organized as follow:

- Part 1: Data exploration
- Part 2: Description of the methodology
- Part 3: Application of the basic method
- Part 4: Enhancement of the method
- Part 5: Result on the validation set
- Part 6: Conclusion and perspectives

A small warning before starting: due to the volume of the data set (2500 pictures) and the number of iterations performed during the search of optimal values of parameters in section 3 and 4, the generation of the full report takes a long time, depending of the machine possibly a full day or so.

Part 1: Data exploration

After download, the folder *OriginalPictures* contains five folders (one per type of flowers), each folder storing the 500 small pictures of the corresponding flower type.

Let's have a quick look with some examples of the different pictures:

Random examples of (from left to right) daisy, dandelion, rose, sunflower and tulip pictures

Some pictures show a single flower, where others show groups of flowers of the same kind:

Examples of daisy pictures, solo (left) and multiple (right)

Examples of rose pictures, solo (left) and multiple (right)

Now, let's have a look to inividual type of flower:

Daisies are quite similar to each other, a yellow center with long and thin white petals. However some pictures are taken form the top, and other from the bottom. In the second case, the yellow core cannot be seen:

Examples of daisy pictures taken from the top (left) and from bottom (right)

Dandelions pictures are of two kinds, pictures of the flower (bright yellow long and thin petals), and pictures of the the seedhead (grey to white fluffy sphere). We also sometimes can have both on a same picture:

Examples of dandelion pictures with a flower (left), a seedhead (middle) and both (right)

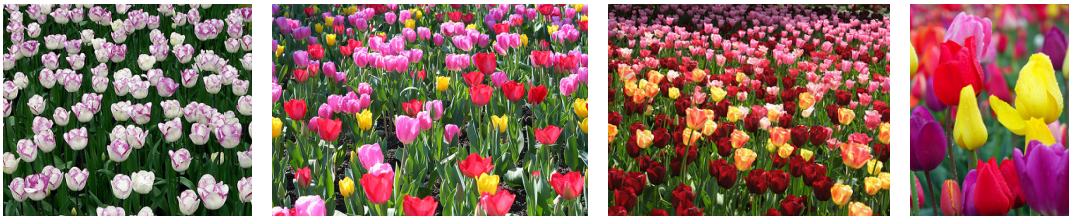
Roses pictures are more different, they reflect the different colors of roses we can find in nature, usually with shades from white to red, but also yellow or purple:

Examples of roses pictures of the different colors (from left to right): white, pink, red, yellow

Sunflowers are much more similar to each other, they have short yellow petals with a big dark center. Color of this center can however fluctuate from yellow to dark brown. Also pictures can show a single sunflower as well as a full sunflower field:

Examples of sunflower pictures with yellow center (left), dark center (middle) and a full field (right)

Tulips pictures are like roses, they have a large spectrum of colors, pale or dark and often mixed. But unlike roses, they tend to be more by groups in parterres, even if some pictures of single flowers can be found:



Examples of tulips pictures with their varieties of colors

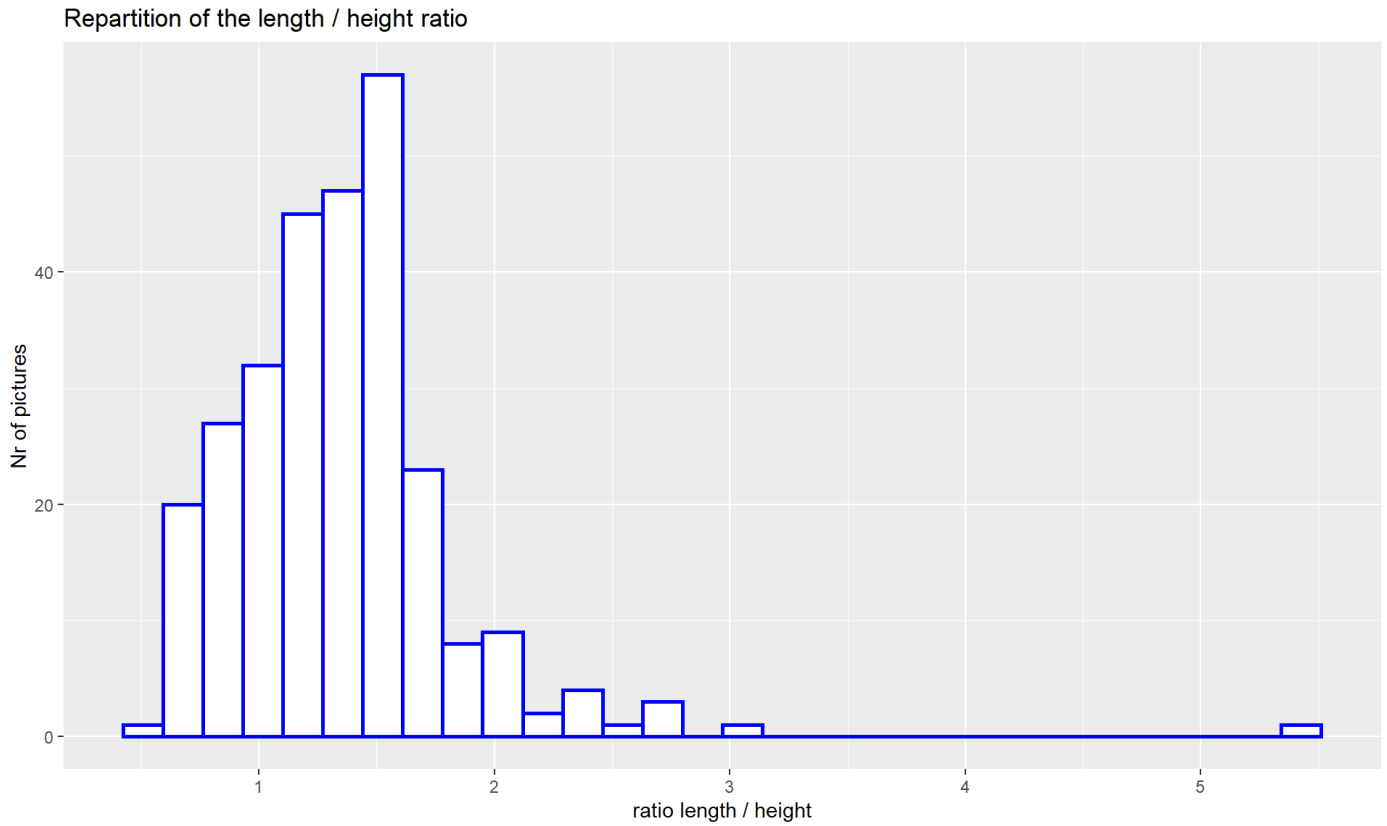
We can notice a large variety of backgrounds: black, full grass, full sky, landscapes, etc:

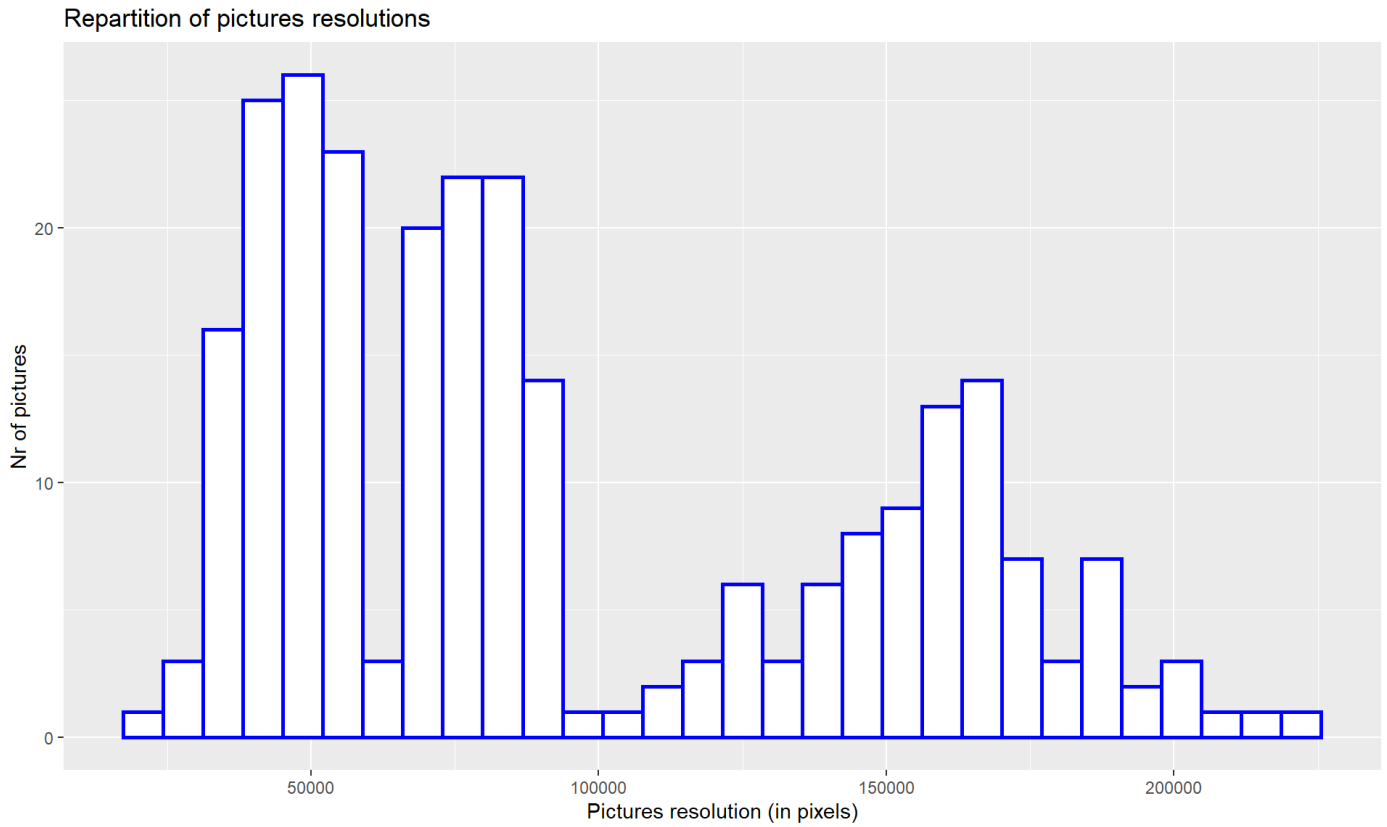
Example of backgrounds (from left to right): black, grass, sky, stones

There are no metadata in the pictures, so no tag embedded to specify which flower type is on each file. Also the name seems to be some kind of unique identifier, that give no information. So the pictures will be kept in their folder and the kind of flower will be deduced from the folder name and stored in a dataframe:

```
# Get all flower pictures from the 5 folders of the dataset, store them in a single dataframe and assign a type for each of th
em depending of the folder they are located.

flowersFullList <- data.frame(name = list.files(paste(getwd(), "/OriginalPictures/daisy", sep = "")),
                              address = paste(getwd(), "/OriginalPictures/daisy/", sep = ""), flowerType = "daisy")

flowersFullList <- rbind(flowersFullList,
                  data.frame(name = list.files(paste(getwd(), "/OriginalPictures/dandelion", sep = "")),
                              address = paste(getwd(), "/OriginalPictures/dandelion/", sep = ""), flowerType = "dandelion"))

flowersFullList <- rbind(flowersFullList,
                  data.frame(name = list.files(paste(getwd(), "/OriginalPictures/rose/", sep = "")),
                              address = paste(getwd(), "/OriginalPictures/rose/", sep = ""), flowerType = "rose"))

flowersFullList <- rbind(flowersFullList,
                  data.frame(name = list.files(paste(getwd(), "/OriginalPictures/tulip/", sep = "")),
                              address = paste(getwd(), "/OriginalPictures/tulip/", sep = ""), flowerType = "tulip"))

flowersFullList <- rbind(flowersFullList,
                  data.frame(name = list.files(paste(getwd(), "/OriginalPictures/sunflower/", sep = "")),
                              address = paste(getwd(), "/OriginalPictures/sunflower/", sep = ""), flowerType = "sunflower"))


# The original pictures are stored in the folder OriginalPictures, pictures after processing will be stored in folder Processe
dPictures, temporary pictures will be stored in folder TempPictures created before.

flowersFullList <- flowersFullList %>% mutate(source = paste(address, name, sep = "")) %>%
                                       mutate(original = source,
                                              temp = str_replace(source, "Original", "Temp"),
                                              processed = str_replace(source, "Original", "Processed")) %>%
                                       select(-address, -name)
```

Pictures are of small resolution, in most of cases retangle, landscape-oriented, with a length and width under 500 pixels. By curiosity, we can have a look on the repartition of sizes:





Flowers can be described mainly by their shape and their color. Considering that the five types we have here are quite distinct in terms of colors, except roses and tulips that may be overlapping, I will try to make a machine-learning algorithm able to recognize flower types from their color only.

Part 2: Description of the methodology

The 2500 pictures data set will be divided in a training set and a validation set. The validation set will be used to check the algorithm performance at the very end.

The following function *CreateUniformPartition* will create a balanced validation set, with the same ratio dispatched between training and validation sets for each type of flower (use of the function *createdatapartition* could not ensure this homogeneity). In our case, as we have in the full data set the same number of the different flower types, it will generate a validation set with an equal number of each flower type.

```
# This function generate a random test set with the same ratio of each flower Type.

CreateUniformPartition <- function(list, ratio, manualSeed) {

  set.seed(manualSeed, sample.kind="Rounding")

  testIndex <- integer()

  distinctFlowerType <- unique(list$flowerType)

  for(u in 1:length(distinctFlowerType)) {

    firstValue <- min(which(list$flowerType == distinctFlowerType[u])) - 1

    fullSamplePerType <- list %>% filter(flowerType == distinctFlowerType[u])

    testIndexPerType <- firstValue + sample(nrow(fullSamplePerType), nrow(fullSamplePerType)*ratio, replace = FALSE)

    testIndex <- c(testIndex, testIndexPerType)

  }

  testIndex

}


# Make the validation and test sets with 10%.

ratio <- 0.1
```

This function is used to generate the validation set with 10% of the original data set. In a second step, the remaining training set (90% of the original data set) is split in a training set and a test set (respectively 90% and 10% of the remaining data set), the training set will be used to train the model and the test set to finetune the parameters. Those values are chosen quite low to keep a sufficient size of the training set (and have an efficient training of the model), the full set not being so big. The drawback is that the test set will be quite small (45 pictures per flower type), so fluctuation of performance may not be very smooth and overtraining may be a risk (because finetuning of parameters would be done on a sample not enough representative).

```
# The validation set is the one we will use at the very end to check the algorithm efficiency.

validationIndex <- CreateUniformPartition(flowersFullList, ratio, 1)

trainAndTestSet <- flowersFullList[-validationIndex,]
validationSet <- flowersFullList[validationIndex,]


# Now make the test set with 10% of the remaining flowers.

testIndex <- CreateUniformPartition(trainAndTestSet, ratio, 2)

trainSet <- trainAndTestSet[-testIndex,]
testSet <- trainAndTestSet[testIndex,]

rm(CreateUniformPartition, trainAndTestSet, flowersFullList, testIndex, validationIndex, ratio, picProperties)
```

The algorithm will have to identify the type of flower from the picture colors. So the first step is to extract colors from the pictures. Transformation of a picture to a matrix of colors can easily be done using the R package *jpeg* and its function *readJPEG*. This function takes the picture address and returns a 3-dimensions matrix with length and width corresponding to the picture resolution, and three layers for red, green and blue (a 480 x 360 pictures will lead to a matrix of size 480 x 360 x 3), each layer storing respectively the value of the red, green and blue values in base 1. Later in this project we use the verbose *matrix form* to specify a picture transformed into a matrix using this function.

Also the opposite operation can be performed with function *writeJPEG*, which transforms a matrix into a JPEG. This will be very useful to visualize and store pictures after processing.

Identification of color shades of a list of pictures (either training set or test set) will be done using the following sequence:

- A loop will take one by one each picture of the list and transform it into its matrix form;
- This matrix form is simplified to transform each pixel in a code corresponding to its color (red, light red, dark red, etc.);
- A list of individual colors of the matrix is extracted with the number of pixels corresponding to this color;
- The black pixels are discarded from the list as black is not a relevant color to distinguish flowers;
- The number of pixels of each color is transformed in a ratio of remaining pixels (so black excluded) to have comparable results accross pictures, whatever the resolution.

We end up with a list like: *full red 5%, dark red 6%, light red 10%, white 15%, etc.*. Those figures will be the predictors of each flower. Note that at this stage, the list of colors is not well defined, we will come back to this a bit later.

By spreading each color as a column, we get a line per flower with a predictor by column. Lines of each flower can be stacked in a dataframe to get data ready for training using the *caret* package.

We can now start to design required functions:

The function *ListColorShades* receives a picture under its matrix form and divides it in a range of $(n+1)^3$ colors (each layer green, blue and red is divided in n+1 levels) where n is specified as input. It returns a list of colors under an integer code and the ratio of pixel of this color found in the matrix, black excluded:

```
# This function returns the number of each pixel of each color ((n+1)^3-colors scale) in the input picture.

ListColorShades <- function(matrixPic, nrLevels) {

# This simple function simplifies the picture color range in n+1 shades of red (0 to n), n+1 shades of green, n+1 shades of bl
ue and returns for each combination the ratio of this color in the picture.

  colorMatrix <- matrix(0, nrow = dim(matrixPic)[1], ncol = dim(matrixPic)[2])

  colorMatrix <- 100 * round(nrLevels*matrixPic[,,1], 0) + 10 * round(nrLevels*matrixPic[,,2], 0) + round(nrLevels*matrixPic
[,,3], 0)

  colorList <- data.frame(color = 0)

  for(i in 1:nrow(colorMatrix)) {

    colorList <- rbind(colorList, data.frame(color = colorMatrix[i,]))

  }

  # Count the number of non-black pixels.

  totalPoints <- colorList %>% filter(color > 0) %>% summarize(nr = n()) %>% pull(nr)

  # Compute the ratio of pixels of each color.

  colorList %>% filter(color > 0) %>% group_by(color) %>% summarize(nr = round(100 * n() / totalPoints, 0))
}
```

The number of levels has to be between 1 and 9. For example, if n = 3, each layer is rounded to the nearest integer between 0 and 3 (so n+1 possibilities), values are concatenated so that the three layers are merged into a unique integer (with n = 3: 000, 001, 002, 003, 100, etc. until 333). 000 is the code for black, 001 a dark blue, 003 a pure blue, 300 a pure red, 333 white, etc. The largest n is, the more accurate the range of color will be. Considering that pictures have different exposures, different contrasts depending of conditions, the camera, the light, a same flower may have different colors on different pictures, so having a too wide range of colors may not be good. The optimal number of colors will have to be determined experimentally.

The function discards black pixels and returns for each color the percentage of pixel of this color among the remaining (not-black) pixels. We consider the percentage of pixels and not an absolute number because not all pictures have the same size, the number of pixels is by design uneven from the beginning. Also not all flowers have the same size on all pictures.

The simple function *ComputeThePredictors* will list all possible colors (defined by the input *nrLevels*) and the ratio of each retrieved by function *ListColorShades*. It completes the color range by adding 0 for colors not found in the picture:

```
# This function receives a picture and the number of color levels and returns the ratio of each of the (3(n+1)) colors.

ComputeThePredictors <- function(matrixPic, nrLevels) {

  colorRef <- expand.grid(a = c(0:nrLevels), b = c(0:nrLevels), c = c(0:nrLevels))

  colorRef <- colorRef %>% mutate(color = as.numeric(paste(a, b, c, sep = ""))) %>%
                           select(color)

  finalColors <- ListColorShades(matrixPic, nrLevels)

  finalPred <- colorRef %>% left_join(finalColors, by = c("color")) %>%
                            mutate(nr = ifelse(is.na(nr), 0, nr), color = paste("col", color, sep = ""))

  finalPred

}
```

Finally, the function *GetThePredictors* takes as input a list of pictures (so the training set, test set or validation set), compute on each picture the ratio of each color, spread each picture on a line and stack predictors of all pictures of the list it in the dataframe *finalPredictors* which host at the end the list of predictors for the full list of pictures:

```
# This function receives a list of pictures and the number of color levels and returns a single line per picture with ((n + 1)
^3 columns, each column storing the ratio of this color in this picture. At the end we have the list of pictures and the predi
ctors for each of them.

GetThePredictors <- function(picturesList, nrLevels, isTraining) {

  # Create the dataframe to host the results.

  colorRef <- expand.grid(a = c(0:nrLevels), b = c(0:nrLevels), c = c(0:nrLevels))

  colorRef <- colorRef %>% mutate(color = as.numeric(paste(a, b, c, sep = ""))) %>%
                           select(color)

  finalPredictors <- rbind(data.frame(c = "flowerType", nr = 0, stringsAsFactors = FALSE),
                           data.frame(c = "name", nr = 0, stringsAsFactors = FALSE),
                           data.frame(c = paste("col", colorRef$c, sep = ""), nr = 0, stringsAsFactors = FALSE))

  finalPredictors <- finalPredictors %>% spread(c, nr) %>%
                                         filter(flowerType != "0")


# Load each picture, compute the predictors for each picture and add the line to the finalPredictors dataframe hosting predict
ors of all pictures of the list.

  for(i in 1:nrow(picturesList)) {

    processedMatrixFlower <- readJPEG(picturesList$source[i])

    finalPred <- ComputeThePredictors(processedMatrixFlower, nrLevels)

    finalPredictors <- rbind(finalPredictors, data.frame(finalPred %>%
                                                           mutate(flowerType = picturesList$flowerType[i], name = picturesList$s
ource[i]) %>%
                                                          spread(color, nr)))

  }


  # Keep colums having at least 1 non null value. Columns corresponding to colors not found in any picture are discarded.

  predictorsValues <- finalPredictors %>% select(-name, -flowerType)


  # If the list of pictures is the training set, it returns all predictors (colors having at least a value) and the flower typ
e, otherwise only the predictors.

  if(isTraining) {filteredValues <- data.frame(flowerType = as.factor(finalPredictors$flowerType), predictorsValues[colSums(pr
edictorsValues) > 0])}
                else {filteredValues <- data.frame(predictorsValues) }

  filteredValues

}
```

Please note the required input *isTraining*. This one is a switch, if the picture list is a training set it will returns the type of flower and predictors after filter color shades not found in any picture (to avoid irrelevant predictors). If *isTraining* is false, it returns predictors only, not the flower type.

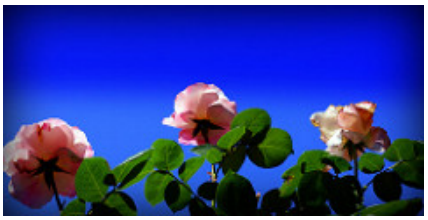
At the end of the process, colors not found in any pictures of the list are discarded to avoid having irrelevant predictors.

To visualise better, let's make a try on a single random picture:

```
# Example of predictors computation for a random rose picture.

exampleList <- data.frame(source = paste(getwd(), "/OriginalPictures/rose/18464075576_4e496e7d42_n.jpg", sep = ""), flowerType
= "rose", stringsAsFactors = FALSE)

nrLevels <- 2

examplePred <- GetThePredictors(exampleList, nrLevels, TRUE)

as.data.frame(examplePred)
```

```
##   flowerType col1 col10 col100 col11 col110 col111 col112 col12 col2 col211
## 1       rose   21     9      5     2      2      4      1    17   35      2
##   col222
## 1      1
```

The following rose picture:



is divided in:

- 35% of full blue (col2)
- 21% of dark blue (col1)
- 17% of light blue (col12)
- 9% of dark green (col10)
- 5% of dark red (col100)
- 4% of grey (col111)
- 2% of light red (col211)
- 2% of dark green-blue (col11)
- 2% of dark yellow (col110)
- 1% of light purple (col112)
- 1% of white (col222)

We already notice in this example that the flower itself can be a minority of pixels of the full picture, and dispatched on different colors (in this case dark red, light red, light purple, white, and green for the leaves). In this case most of pixels are blue, due to the dominence of the sky background. This already shows that a picture processing will be required to isolate flowers from the background to focus on relevant pixels.

We now have an efficient way to compute predictors for a list of pictures. We can now start the training and finetuning of parameters.

Part 3: Application of the basic method

As first attempt, the basic model consists in training the algorithm using the raw training pictures. We need to determine the best method and the number of color levels (the number of possible colors used for predictions) giving the best results. We will try seven different training methods, the ones we have learned during the previous modules (K-nearest Neighbors (knn), Random Forest (rf)) and some others I have chosen after searching a bit on internet (Gradient Boosting Machine (gbm), Generalized Linear Model (glmnet), Recursive Partitioning And Regression Trees (rpart), Neural Networks (nnet) and C5.0 Classification Model (C5.0)). LDA and QDA have also been tried but are failing for nrLevels > 3 and are not giving very good results at 3 and below.

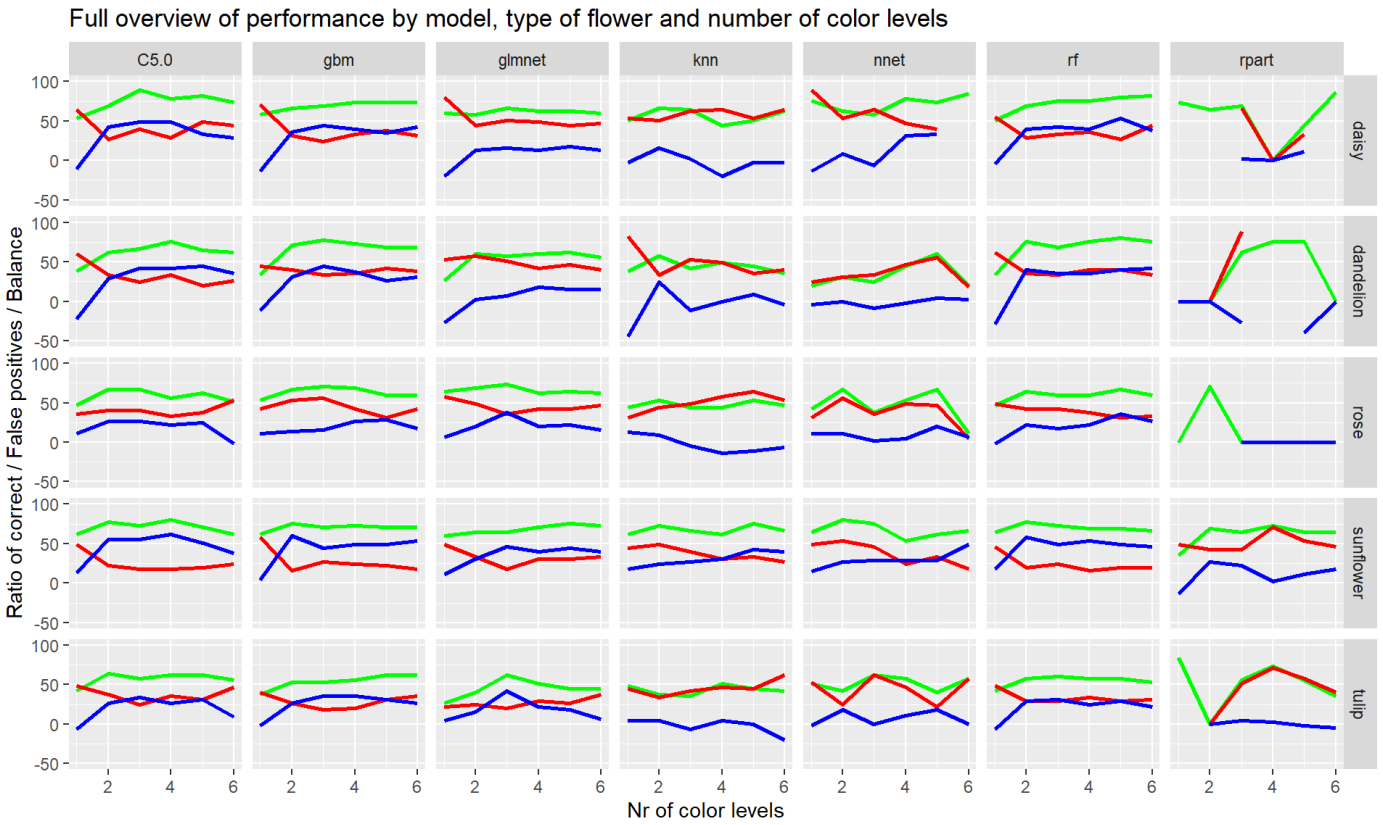
We will try from 1 to 6 color levels. 1 color level makes a range of 8 possible colors, 6 color levels makes a range of 343 possible colors (so 343 predictors, which starts to make a long computation time). The code below proceed in two steps, first it computes the predictors for training and test sets for a specific number of colors, secondly it performs the training and predictions for each of the seven methods. Predictions are stored at each step in the dataframe *compareTrainModels*.

```
trainModels <- c("knn", "rf", "gbm", "glmnet", "rpart", "nnet", "C5.0")


compareTrainModels <- data.frame(model = character(), flowerType = character(), name = character(), prediction = character(),
 nrLevels = numeric())


# The training and test sets are processed to get predictors, then each of the 7 models is tested, results are stored in dataf
rame compareTrainModels.
# The same operation is repeated with a different number of color levels.

  for (nrLevels in 1:6) {

  filteredValuesTraining <- GetThePredictors(trainSet, nrLevels, TRUE)

  filteredValuesTest <- GetThePredictors(testSet, nrLevels, FALSE)


# Perform training and predictions for the different models.

  for (m in 1:length(trainModels)) {

  set.seed(1, sample.kind="Rounding")

  trainingResults <- train(flowerType ~ ., filteredValuesTraining, method = trainModels[m])

  prediction <- predict(trainingResults, filteredValuesTest)

  compareTrainModels <- rbind(compareTrainModels, data.frame(model = trainModels[m], name = testSet$source, flowerType = testS
et$flowerType, prediction, nrLevels))

  }
  }


rm(nrLevels, m, prediction, trainingResults, filteredValuesTraining, filteredValuesTest)
```

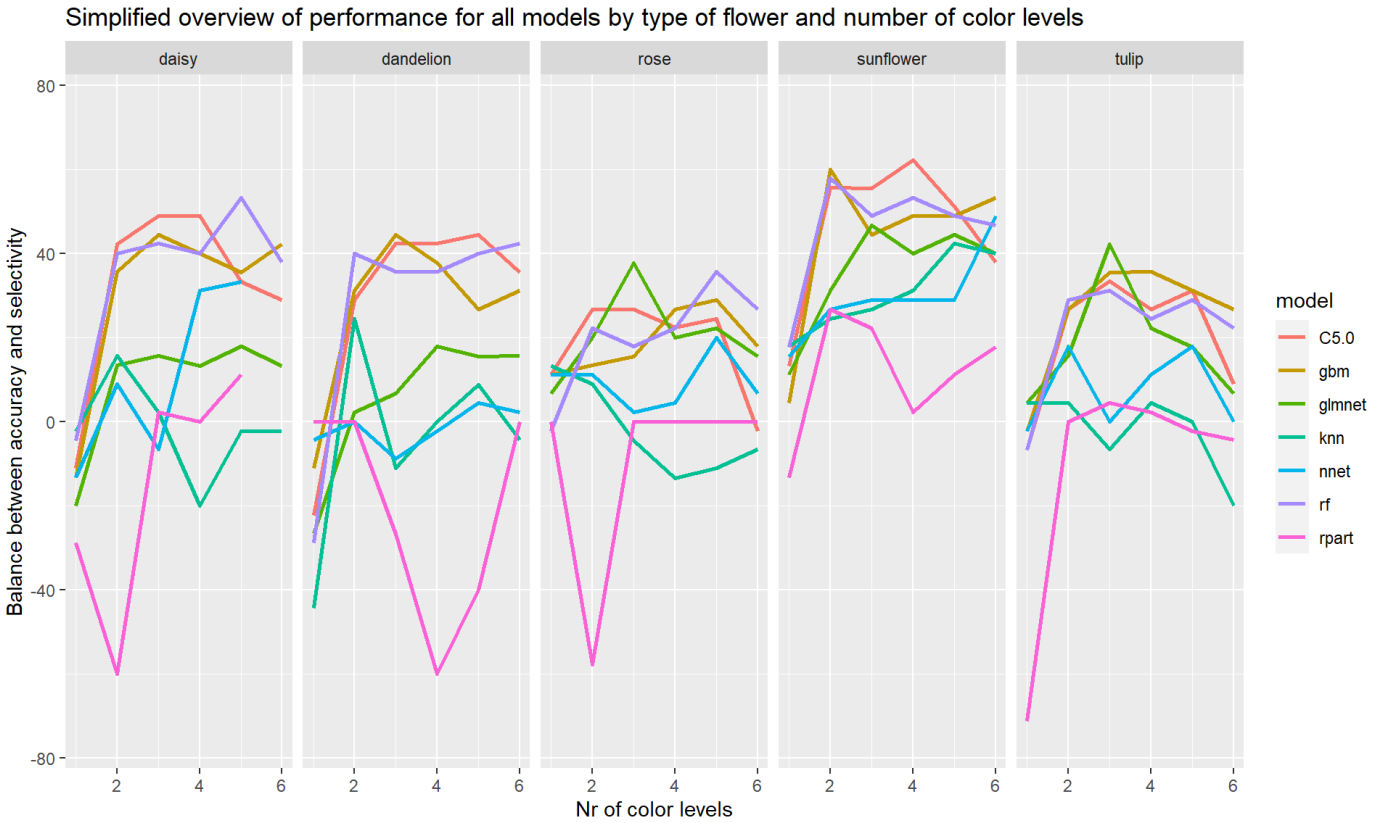
To assess the results, we will consider each type of flower individually and report individually accuracy and selectivity of each model and each number of color levels on a graph. The green line is the ratio of good prediction, the red line is the ratio of false positives (flower wrongly identified as this flower type), the blue line is the difference, so finally the balance between accuracy and selectivity. I consider the best model as the one with highest difference between the green and the red line, so finally with the best accuracy and selectivity balance (to not privilege for example a model that would identify everything as daisies, and then would have very high accuracies for daisies). This balance can be negative, in case of a higher number of false positive than flowers correctly identified.

```
# Results are grouped by flower type and model to get ratio of correct predictions and number of false positive for each model
and number of colors.

 results <- compareTrainModels %>%
            mutate(correct = flowerType == prediction) %>%
            group_by(nrLevels, flowerType, model) %>%
            summarize(correct = sum(correct), nr = n())


 falsePositives <- compareTrainModels %>%
                   mutate(correct = flowerType == prediction) %>%
                   filter(correct == FALSE) %>%
                   group_by(nrLevels, prediction, model) %>%
                   summarize(falsePositive = n())


 finalResults <- results %>% left_join(falsePositives, by = c("model", "nrLevels", "flowerType" = "prediction")) %>%
                         mutate(falsePositive = ifelse(is.na(falsePositive), 0, falsePositive)) %>%
                         mutate(correct = round(100 * correct/nr, 1),
                                falsePositive = round(100 * falsePositive/nr, 1),
                                balance = correct - falsePositive) %>%
                         select(- nr)


 rm(results, falsePositives)
```

Results are visualized on the graph below:



This graph gives a full overview, but is not easy to read. To get a better comparison, we can display only for each type of flower the balance accuracy / selectivity and show all models on a single graph:



At first sight we can see that rf, gbm and C5.0 seems to give the best results for all flowers. At least for daisies, dandelions and sunflowers, those two models seems to be far ahead. It is also mostly true for tulips and roses, but results are a bit more mitigated. for roses and tulips, glmnet also seems to give good results for nrLevel = 3.

To compute the optimal number *optimalNrColors*, we can't just take results of all models and all type of flowers and compare the average values, because some models work better than others, We will keep some, and discard others. So the optimal number of color levels should be checked for the best models only.

To do that, we keep for each couple "nr of color level / type of flowers" the best model and see for all remaining models what the optimal value of color levels is:

```
# Get the nr of colors giving the best results and store results with this value in resultsEvolution to compare the performanc
e of the model accross the different steps.

optimalNrColors <- finalResults %>% group_by(nrLevels, flowerType) %>%
                                    summarize(bestModel = max(balance)) %>%
                                    group_by(nrLevels) %>%
                                    summarize(overall = mean(bestModel)) %>%
                                    arrange(desc(overall)) %>%
                                    top_n(1) %>%
                                    select(nrLevels) %>%      # In case of equality, keep the lowest.
                                    arrange(nrLevels) %>%
                                    pull(nrLevels)
```

We get the best results at *optimalNrColors* = **3**, we keep this value for future use.

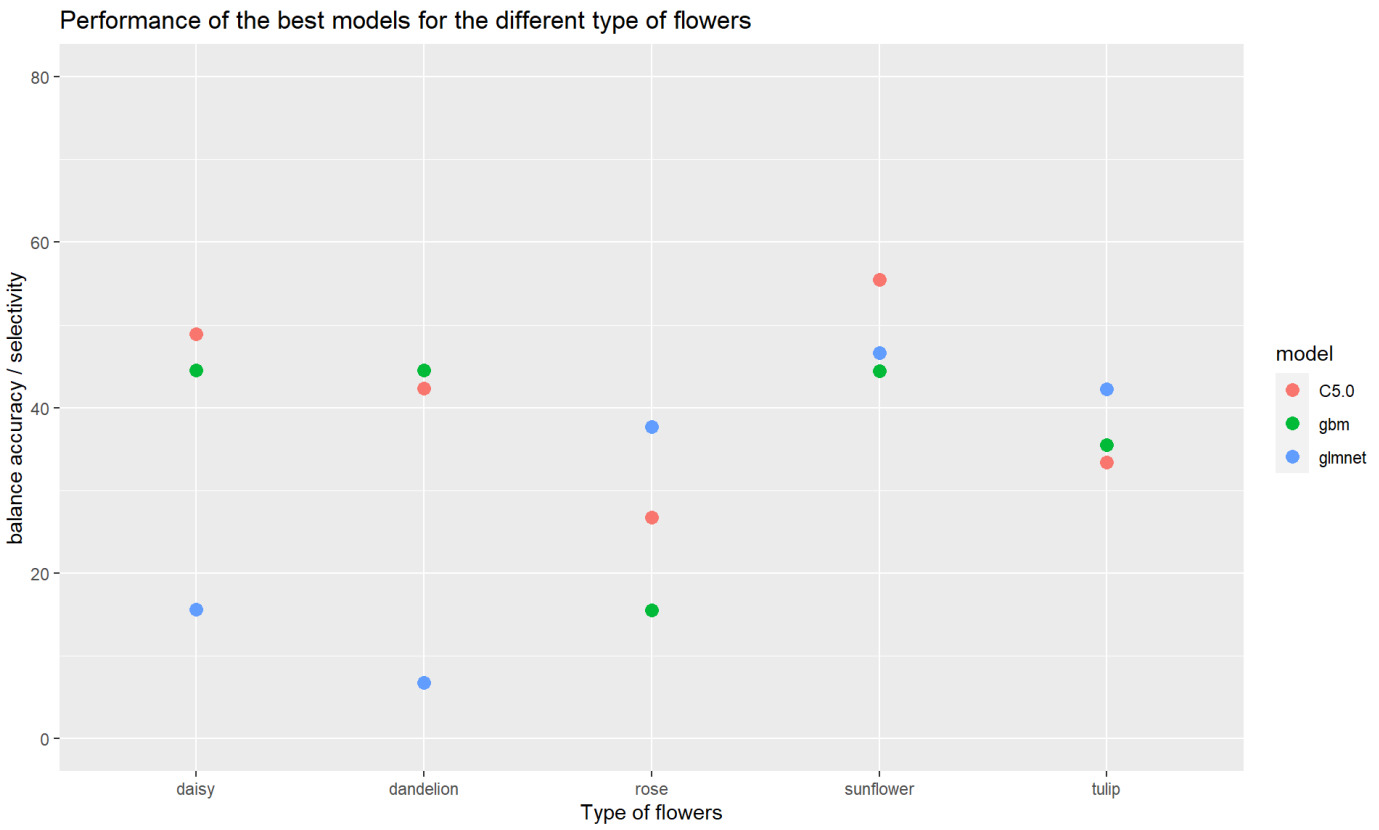
Now when we come to select the best model, we don't have to select only one, we can keep working with multiple models when we have our predictors (whereas working with different numbers of color levels would require to re-compute the predictors multiple times). If a model gives better results for tulips and another better results ofr daisies, we can keep both a consider one for identification of tuip and the other one for identification of daisies.

So the best models kept will be the models giving the best results for each type of flower:

```
# Get the models giving the best results for this number of Levels for each type of Flower.

bestModels <- finalResults %>% filter(nrLevels == optimalNrColors) %>%
                               group_by(flowerType) %>%
                               filter(balance == max(balance)) %>%
                               pull(model) %>%
                               unique()
```

We finally keep the following list: **C5.0, gbm, glmnet**.

We can now have a look more precisely on the four best models identified for the optimal number of colors:



Let's summarise results in a table. When considering the overall accuracy for all type of flowers, we can use the average ratio of correct prediction. However when checking individually performance on each type of flower, we need to consider the balance accuracy / selectivity:

```
##          step  model daisy dandelion rose sunflower tulip
## 1 Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 2 Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 3 Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

The best model at this stage give the following overall acuracy (percentage of flowers properly identified):

```
##           step model overallAccuracy
## 1 Raw Pictures  C5.0          70.68
```

At this stage, results are already not so bad, the best model gives an accuracy of **70.68**, even if the current algorithm takes not only the color of the flowers, but also the background, and other objects that would be on the picture, grass, sky, etc. We will now try to enhance the algorithm by adding pre-processing of the flower pictures to isolate more efficiently the flower from the other parts of the picture and tehn increase the accuracy.

Part 4: Enhancement of the method

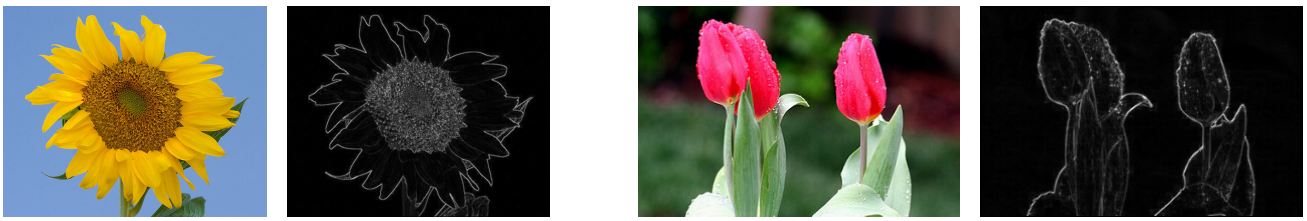
The main challenge here is to make the algorithm able to distinguish the flower from the background. It is obvious for a human but being able to explain rationnally to a machine what is the flower and what is the background is not so obvious.

An observation we can do is that a flower is a surface of light or bright color and the background is either a smooth surface (for pictures with a narrow depth of field), grass or sky. So finally the contour of the flower (as well as the contour of the different parts of the flower) can be identified as a line of high contrast, in best case as the zone of highest contrast of the picture.

The contrast level on a zone of a picture can be assimilated to the distance between the color of a pixel and the color of its adjacent pixels. The higher the distance is, the higher the contrast is between the two points. Hence I propose to identify the contrasts levels on a picture by computing for each pixel the sum of the distances between itself and the four (upper, lower, left and right) pixels, using the function *ContrastPictures* below:

```
# This function receives a picture under its matrix form and returns a 2-dimensions matrix of the same height / length with nu
mbers corresponding to the contrast between the pixel and its four neighbour pixels.

ContrastPicture <- function(matrixPic) {

  contrastFlower <- matrix(0, nrow = dim(matrixPic)[1], ncol = dim(matrixPic)[2])

# First and last lines and columns are excluded from the computation as they have only 2 or 3 neighbours.

  for(i in (2:(nrow(matrixPic)-1))) {

    for(j in (2:(ncol(matrixPic)-1))) {

# Compute the distance between a point and its adjacent ones to estimate the local contrast.

      contrastFlower[i, j] <- ((matrixPic[i, j, 1] - matrixPic[i+1, j, 1])^2 + (matrixPic[i, j, 2] - matrixPic[i+1, j, 2])^2 +
(matrixPic[i, j, 3] - matrixPic[i+1, j, 3])^2)^(1/2) +
                              ((matrixPic[i, j, 1] - matrixPic[i-1, j, 1])^2 + (matrixPic[i, j, 2] - matrixPic[i-1, j, 2])^2 +
(matrixPic[i, j, 3] - matrixPic[i-1, j, 3])^2)^(1/2) +

                              ((matrixPic[i, j, 1] - matrixPic[i, j+1, 1])^2 + (matrixPic[i, j, 2] - matrixPic[i, j+1, 2])^2 +
(matrixPic[i, j, 3] - matrixPic[i, j+1, 3])^2)^(1/2) +
                              ((matrixPic[i, j, 1] - matrixPic[i, j-1, 1])^2 + (matrixPic[i, j, 2] - matrixPic[i, j-1, 2])^2 +
(matrixPic[i, j, 3] - matrixPic[i, j-1, 3])^2)^(1/2)

    }
  }

  contrastFlower
}
```

The function takes a picture under its matrix form and returns a 2-dimensions matrix with the same numbers of rows and colums, each value corresponds to the local contrast. The value is in the range [0 - $4\sqrt{3}$], 0 when a pixel is surrounded by pixels of the same color (no contrast), $4\sqrt{3}$ (around 6.92) when a white pixel is surrounded by black pixels or the other way round (highest possible contrast). By curiosity, the resulting matrix can be converted to a picture after normalization (to avoid values >1):

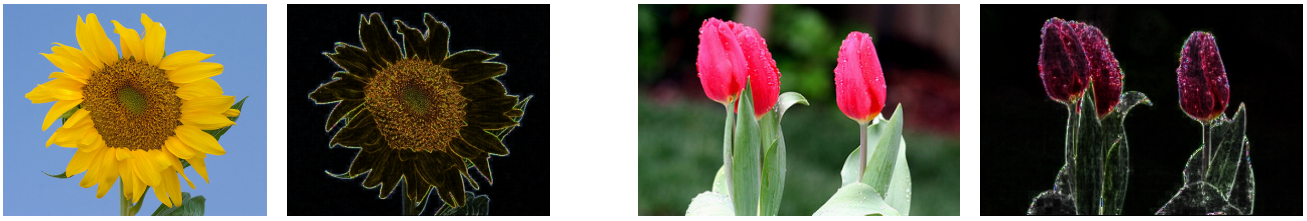


Example of sunflower (left) and tulips (right) before and after contrast processing

The whitest parts correspond to the zones of highest contrast: the petals contours and the seeds. We can go further by multiplying this contrast matrix to the original picture matrix. The function *HighlightHighContrastZones* will do the job:

```
# This function receive a matrix form and returns the picture multiplicated by its contrast matrix and normalized.

HighlightHighContrastZones <- function(matrixPic) {

highestContrastFlower <- array(0, dim = dim(matrixPic))

# Get the contrast matrix.

contrastPic <- ContrastPicture(matrixPic)

# Normalization.

highestContrastFlower <- highestContrastFlower / max(contrastPic)

# Multiplicate the contrast matrix by the original picture.

highestContrastFlower[,,1] <- matrixPic[,,1] * contrastPic
highestContrastFlower[,,2] <- matrixPic[,,2] * contrastPic
highestContrastFlower[,,3] <- matrixPic[,,3] * contrastPic


highestContrastFlower

}
```

We can apply *HighlightHighContrastZones* to the same examples as before:



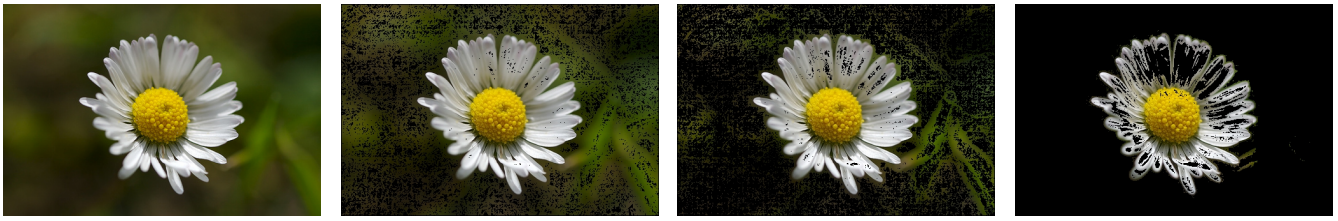
Same examples as earlier after the multiplication of the contrast matrix with the original matrix

This method highlights the flower contours, but generates a picture with artificial colors that should not be used for training as it would lead to train a model from non-genuine colors. The contrast picture can however be used as a basis to filter from the original picture zones with the lowest contrast.

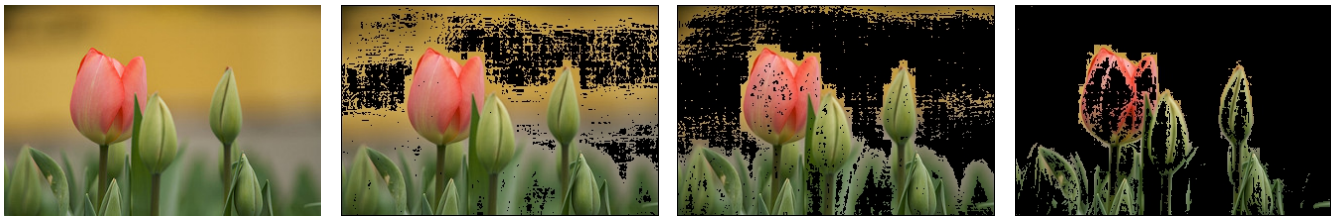
The function *FilterHighestContrastZones* will do the job: It receives a picture (under its matrix form) and a *contrastQuantile* (decimal between 0 and 1 correspondng to a quantile), and returns the original picture where the pixels being under the specified contrast quantile points are transformed in black. For example, with a specified ratio 0.4, all pixels having a contrast value part of the lowest 40% contrast values of the full pictures will be set to black.

```
# This function receives a matrix form and a quantile (decimal number) and filters from the picture points being in the lowest
contrast quantile.

FilterHighestContrastZones <- function(matrixPic, contrastQuantile) {

  filteredHighContrastFlower <- array(0, dim = dim(matrixPic))

  contrastScalePic <- ContrastPicture(matrixPic)

  contrastTreshold <- quantile(contrastScalePic, prob = contrastQuantile)

  contrastMonochrome <- ifelse(contrastScalePic > contrastTreshold, 1, 0)

  filteredHighContrastFlower[,,1] <- matrixPic[,,1] * contrastMonochrome
  filteredHighContrastFlower[,,2] <- matrixPic[,,2] * contrastMonochrome
  filteredHighContrastFlower[,,3] <- matrixPic[,,3] * contrastMonochrome

  filteredHighContrastFlower

}
```

Let’s try the tool on some examples with different contrast quantiles filter:



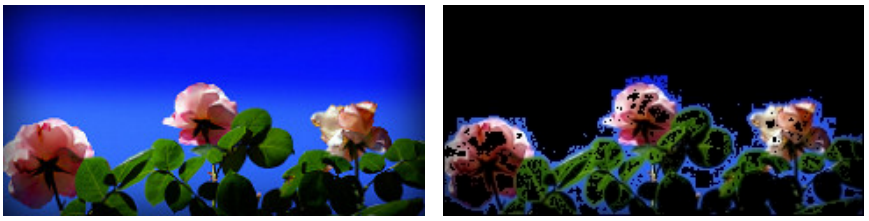
Example of contrast filter applied to a daisy picture, from left to right: original, with quantile 0.2, with quantile 0.5, with quantile 0.8



Example of blue filter applied to a tulip picture, from left to right: original, with quantile 0.2, with quantile 0.5, with quantile 0.8

We can apply this method on the pictures of pink roses on the sky background we used earlier to see the impact on the predictors:

The original rose picture (left), transformed in the processed picture with a contrast quantile of 0.7 (right):



The color range is now the following:

```
##   flowerType col1 col10 col100 col101 col11 col110 col111 col112 col12 col2
## 1       rose   10    17     10      1    10      8     16      5    10    1
##   col211 col212 col221 col222
## 1      5      1      1      5
```

is now divided in:

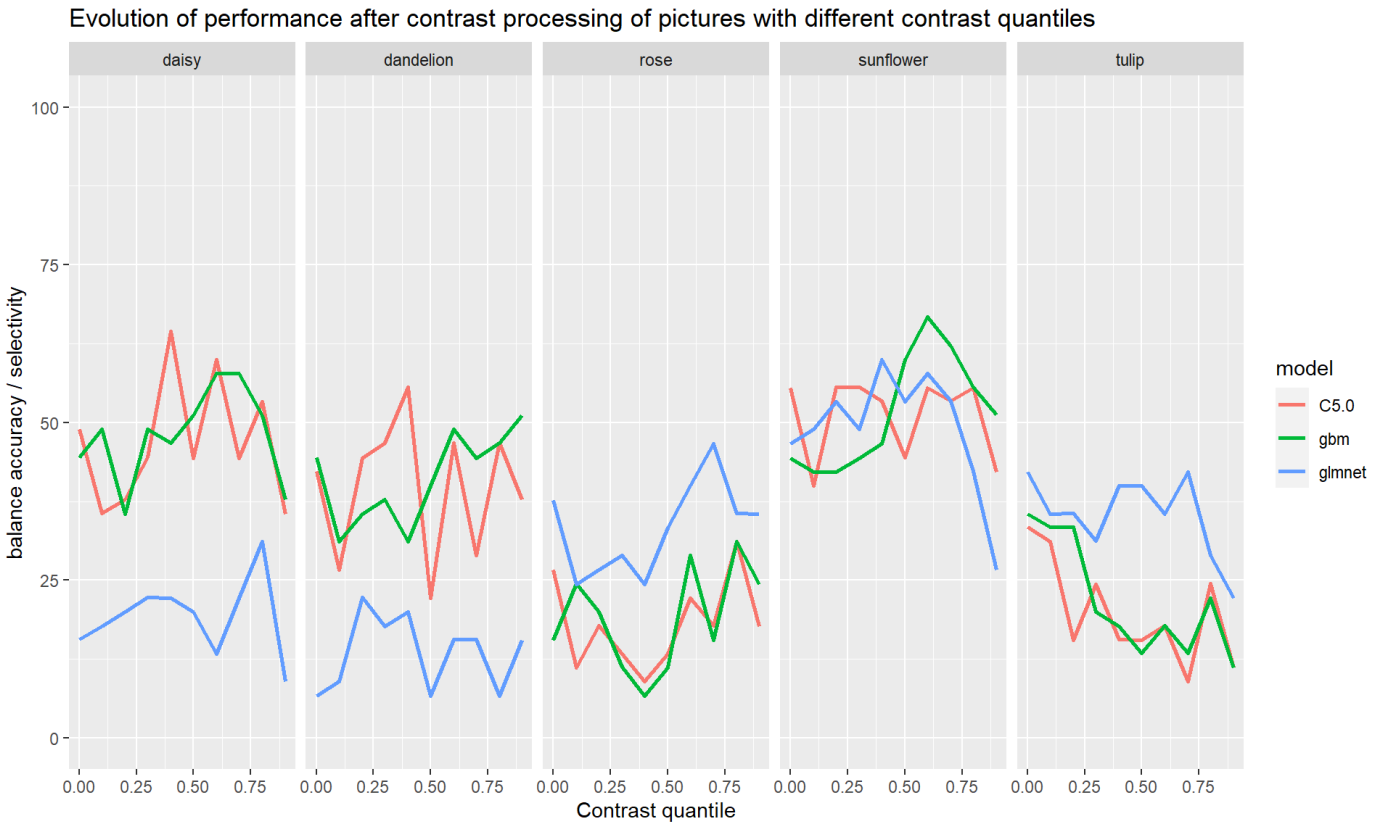
- 17% of dark green (col10)
- 16% of grey (col111)
- 10% of dark green-blue (col11)
- 10% of dark red (col100)
- 10% of light blue (col12)
- 10% of dark blue (col1)
- 8% of dark yellow (col110)
- 5% of light purple (col112)
- 5% of white (col222)
- 5% of light red (col211)
- 1% of full blue (col2)
- 1% of light pink (col212)
- 1% of light yellow (col221)

Even though the number of colored pixels has been reduced, we now have a much higher ratio of pixels corresponding to the flower colors. Earlier the blue was the dominent color, now it is the green (leaves) and color flowers (white, dark red, grey, etc.) reach altogether around 50%.

The filter is a very powerful tool to remove soft backgrounds but also tend to crop central parts of the petals, so to prevent a too strong filtering, the contrast quantile should be carefully chosen. To pick the best value, we will try to apply on the training and test sets the filter with values from 0 to 1 and define experimentally the optimal contrast quantile for the different models we kept earlier:

```
# Let's keep the results of the previous training for models and nr of color levels we kept. The original pictures correspond
 to a processus with a contrast quantile = 0.

compareTrainModels <- compareTrainModels %>%
                      filter(model %in% trainModels & nrLevels == optimalNrColors) %>%
                      mutate(contrastQuantile = 0) %>%
                      select(- nrLevels)

# Now use the Temp pictures as source for the training/ prediction.

trainSet <- trainSet %>% mutate(source = temp)
testSet <- testSet %>% mutate(source = temp)


# The processing below apply the contrast filter to the original picture and save the resulting pictures in the Temp folder.

for(contrastQuantile in seq(0.1, 0.9, 0.1)) {

# Processing the training set.

for(n in 1:nrow(trainSet)) {

  matrixPic <- readJPEG(trainSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, contrastQuantile)

  writeJPEG(matrixPic, target = trainSet$temp[n], quality = 1.0, bg = "white")

}


# Processing the test set.

for(n in 1:nrow(testSet)) {

  matrixPic <- readJPEG(testSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, contrastQuantile)

  writeJPEG(matrixPic, target = testSet$temp[n], quality = 1.0, bg = "white")

}


  filteredValuesTraining <- GetThePredictors(trainSet, optimalNrColors, TRUE)

  filteredValuesTest <- GetThePredictors(testSet, optimalNrColors, FALSE)


  for (m in 1:length(trainModels)) {

  set.seed(1, sample.kind="Rounding")

  trainingResults <- train(flowerType ~ ., filteredValuesTraining, method = trainModels[m])

  prediction <- predict(trainingResults, filteredValuesTest)

  compareTrainModels <- rbind(compareTrainModels, data.frame(model = trainModels[m], name = testSet$source, flowerType = testS
et$flowerType, prediction, contrastQuantile))

  }

}



rm(trainingResults, prediction, finalPredictorsTest, filteredValuesTest, filteredValuesTraining, matrixPic, contrastQuantile,
 exampleOfContrastQuantile, examples, m, n)
```

Now let's summarize the results and see them graphically:



We can notice on the graph high-level tendencies, however at first sight this one looks a bit choatique and random (especially for model C5.0), but there are several reasons for that:

- The test set is made of only 45 pictures of each flower type. So each single variation lead to more than 2 pts of variation (100/45);
- Here we don't measure the accuracy but the balance accuracy / selectivity, and the balance is not between 0 and 100, but between -400 and 100 (-400 when a flower is always wrong but when other flowers are identiied as this one), so finally the fluctuation we have here is only on a small part of the full range;
- By design, the balance is much more volatile than the accuracy as each flower type balance depends of the four other flower types results. If for example from a contrast quantile to the next only one more of the 45 flowers get wrongly identified and one of each other type of flower get identified as this one (which is a small overall variation), it leads to a global variation of balance of more than 11 points ((1+4)*100/45).

The optimal value of the contrast quantile is computed as the value giving the highest average balance for all flowers, in this case it is **0.6**. Implementation of the contrast filter with this optimal value increases particularly performance on daisies and dandelions, however is not so good for tulips:

```
##              step  model daisy dandelion rose sunflower tulip
## 1 Contrast Filter   C5.0  60.0      46.7 22.2      55.5  17.8
## 2 Contrast Filter    gbm  57.8      48.9 28.9      66.7  17.8
## 3 Contrast Filter glmnet  13.3      15.6 40.0      57.8  35.5
## 4    Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 5    Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 6    Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

The contrast filter increases slightly the overall accuracy

```
##              step model overallAccuracy
## 1 Contrast Filter   gbm           72.00
## 2    Raw Pictures  C5.0           70.68
```

Pictures are reworked with this optimal contrast quantile and the resulting are stored in folder *ProcessedPictures* to use already processed at the next step:

```
# Processed pictures are to be stored in ProcessedPictures at this stage.

for(n in 1:nrow(trainSet)) {

  matrixPic <- readJPEG(trainSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, optimalContrastQuantile)

  writeJPEG(matrixPic, target = trainSet$processed[n], quality = 1.0, bg = "white")

  }


# Processing the test set.

for(n in 1:nrow(testSet)) {

  matrixPic <- readJPEG(testSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, optimalContrastQuantile)

  writeJPEG(matrixPic, target = testSet$processed[n], quality = 1.0, bg = "white")

  }

# Now the processed pictures become the source, and target is set back to the temp folder.

rm(matrixPic, n, finalResults)
```

Another challenge we face is the background contrast. On many pictures, the background is also a source of contrast that remains even after this first processing. Considering that most of the pictures have as background either grass or sky, and considering that none of the flower are either green or blue and that all flowers have a green stem and green leaves, green and blue don't appear to be an element of distinction between the different flowers types. So a quick way to get rid of the background noise is to filter shades of green and blue from the processed pictures. This should however be performed with caution to not discard a too wide range of colors and crop the flower too much.

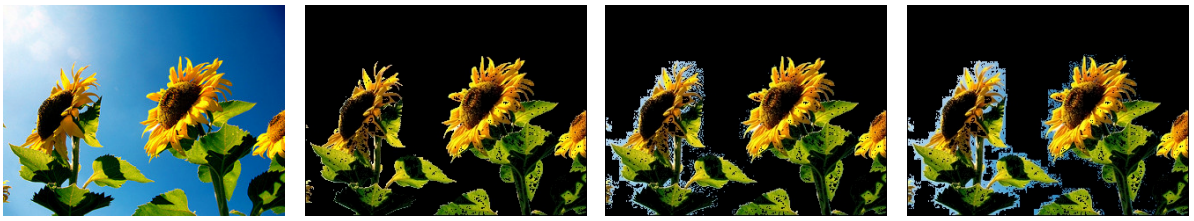
The following *FilterColor* function should do the job. It takes as input a picture, a color, and a coefficient filter. This coefficient corresponds to the level of dominance of the green / blue on the other colors from which all shades of blue / green will be discarded. For example if the function is called for blue and coef 1.1, all pixels of the picture where the blue layer (third slice of the matrix) is 1.1 times higher than the red and the green layers will be transformed to black. In this example, a pixel with values [100, 100, 110] will be set to black, where a pixel with value [100, 100, 109] will be kept unchainged.

```
# This function receives a picture under it matrix form and discard the green or blue pixels with dominance higher than a cert
ain treshold specified as input.

FilterColor <- function(matrixPic, color, coef) {

  boolPic <- matrix(0, nrow = dim(matrixPic)[1], ncol = dim(matrixPic)[2])
  filteredSample <- array(0, dim = dim(matrixPic))

  # Filter pixels where blue is dominant

  if(color == "green") { boolPic <- ifelse(matrixPic[,,2] > coef * matrixPic[,,1] & matrixPic[,,2] > coef * matrixPic[,,3], 0,
1) }
  if(color == "blue")  { boolPic <- ifelse(matrixPic[,,3] > coef * matrixPic[,,1] & matrixPic[,,3] > coef * matrixPic[,,2], 0,
1) }


  filteredSample[,,1] <- matrixPic[,,1] * boolPic
  filteredSample[,,2] <- matrixPic[,,2] * boolPic
  filteredSample[,,3] <- matrixPic[,,3] * boolPic

  filteredSample

}
```

The function will have to be called twice, once for green, once for blue, and the optimal value of the green and blue coefficients will have to be determined experimentally.

Let's apply the function on a few examples (after application of the contrast filter):

Example of green filter applied to a dandelion picture, from left to right: original, with coef 1, with coef 1.25, with coef 1.5



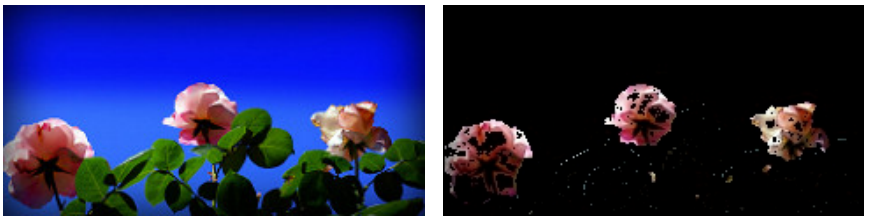
Example of blue filter applied to a daisy picture, from left to right: original, with coef 1, with coef 1.25, with coef 1.5

Those filters are to be applied after the contrast filter of the previous step with the optimal contrast quantile.

We can once again apply the two filters on the pictures of pink roses on the sky background we used earlier to see the impact on the predictors:

```
# Example of predictors computation for a random rose picture.

exampleList <- data.frame(source = paste(getwd(), "/ExamplePictures/18464075576_4e496e7d42_n_p.jpg", sep = ""), flowerType =
"rose", stringsAsFactors = FALSE)

nrLevels <- 2
greenF <- 1.05
blueF <- 1.05

matrixPic <- readJPEG(exampleList$source)

matrixPic <- FilterColor(matrixPic, "green", greenF)

matrixPic <- FilterColor(matrixPic, "blue", blueF)

exampleList$source <- paste(getwd(), "/ExamplePictures/18464075576_4e496e7d42_n_q.jpg", sep = "")

writeJPEG(matrixPic, target = exampleList$source, quality = 1.0, bg = "white")

examplePred <- GetThePredictors(exampleList, nrLevels, TRUE)
```

The original rose picture (left), transformed in the processed picture with the contrast filter, a green and blue filters with coefficient 1.05 (right):



The color range is now the following:

```
##   flowerType col100 col101 col110 col111 col211 col212 col221 col222
## 1       rose     30      2      6     29     15      3      2     13
```

is now divided in:

- 30% of dark red (col100)
- 29% of grey (col111)
- 15% of light red (col211)
- 13% of white (col222)
- 6% of dark yellow (col110)
- 3% of light pink (col212)
- 2% of light yellow (col221)
- 2% of dark purple (col101)

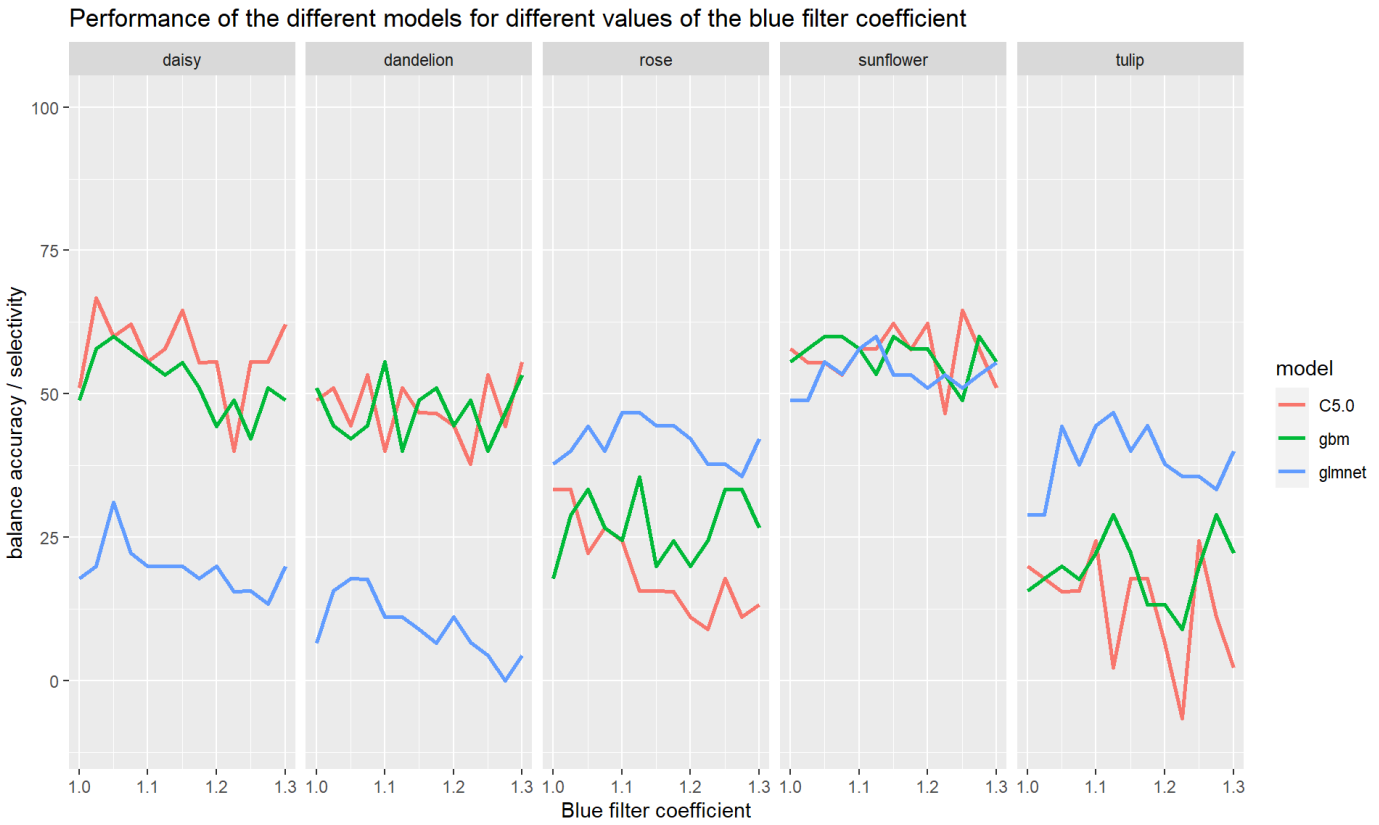
We now have predictors exclusively dedicated to the relevant flower colors, we hence expect a better color recognition.

The value of the two coefficients (green filter and blue filter) will have to be chosen to get the best compromise in order to remove superfluous parts but not crunch the flower colors too much.

The following loop will train the algorithm from pictures processed with the green filter at different coefficients and store the predictions of each step in the data frame *compareTrainModels*:

```
compareTrainModels <- data.frame(model = character(), flowerType = character(), name = character(), prediction = character(),
greenCoef = numeric())

# Processing pictures coming form the contrast filter with the blue filters with different coefficients.

for(blueCoef in seq(1, 1.3, 0.025)) {

# Processing the training set.

for(n in 1:nrow(trainSet)) {

  matrixPic <- readJPEG(trainSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "blue", blueCoef)

  writeJPEG(matrixPic, target = trainSet$temp[n], quality = 1.0, bg = "white")

}


# Processing the test set.

for(n in 1:nrow(testSet)) {

  matrixPic <- readJPEG(testSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "blue", blueCoef)

  writeJPEG(matrixPic, target = testSet$temp[n], quality = 1.0, bg = "white")

}



  filteredValuesTraining <- GetThePredictors(trainSet, optimalNrColors, TRUE)

  filteredValuesTest <- GetThePredictors(testSet, optimalNrColors, FALSE)



# Train and predict using the different models selected before.

for (m in 1:length(trainModels)) {

set.seed(1, sample.kind="Rounding")

trainingResults <- train(flowerType ~ ., filteredValuesTraining, method = trainModels[m])

prediction <- predict(trainingResults, filteredValuesTest)

compareTrainModels <- rbind(compareTrainModels, data.frame(model = trainModels[m], name = testSet$source, flowerType = testSet
$flowerType, prediction, blueCoef))

}

}

rm(prediction, trainingResults, filteredValuesTest, filteredValuesTraining, matrixPic, blueCoef, m, n)
```

Now we summarise the results to see the performance evolution accross the blue filter coefficient and show it graphically:



We can compute the optimal blue filter coefficient:

```
# Keep the blue filter coefficient giving the best results.

optimalBlueFilterCoef <- finalResults %>%
                         group_by(blueCoef,model) %>%
                         summarize(avgBalance = mean(balance)) %>%
                         arrange(desc(avgBalance)) %>%
                         ungroup() %>%
                         top_n(1) %>%
                         select(blueCoef) %>%
                         arrange(blueCoef) %>%     # In case of equality, keep the lowest.
                         top_n(1) %>%
                         pull(blueCoef)
```

The optimal value of the blue filter coefficient is computed as the value giving the highest average balance for all flowers for the best model, in this case **1.025**.

```
##              step  model daisy dandelion rose sunflower tulip
## 1     Blue Filter   C5.0  66.7      51.1 33.3      55.5  17.8
## 2     Blue Filter    gbm  57.8      44.5 28.8      57.8  17.8
## 3     Blue Filter glmnet  20.0      15.6 40.0      48.9  28.9
## 4 Contrast Filter   C5.0  60.0      46.7 22.2      55.5  17.8
## 5 Contrast Filter    gbm  57.8      48.9 28.9      66.7  17.8
## 6 Contrast Filter glmnet  13.3      15.6 40.0      57.8  35.5
## 7    Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 8    Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 9    Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

The performance increase of the algorithm on pictures processed with the blue filter appear to be very moderate and a bit diappointing. It increases the daisies and rose for C5.0, but also harm performance of the glmnet model. One of the hypothesis we can make about this low enhancement is that the amount of pictures with a sky background is equaly dispatched among the different types of flowers, hence not being a relevant color and then from the beginning not a relevant predictor.

The overall performance finally becomes:

```
##              step model overallAccuracy
## 1     Blue Filter  C5.0           72.44
## 2 Contrast Filter   gbm           72.00
## 3    Raw Pictures  C5.0           70.68
```

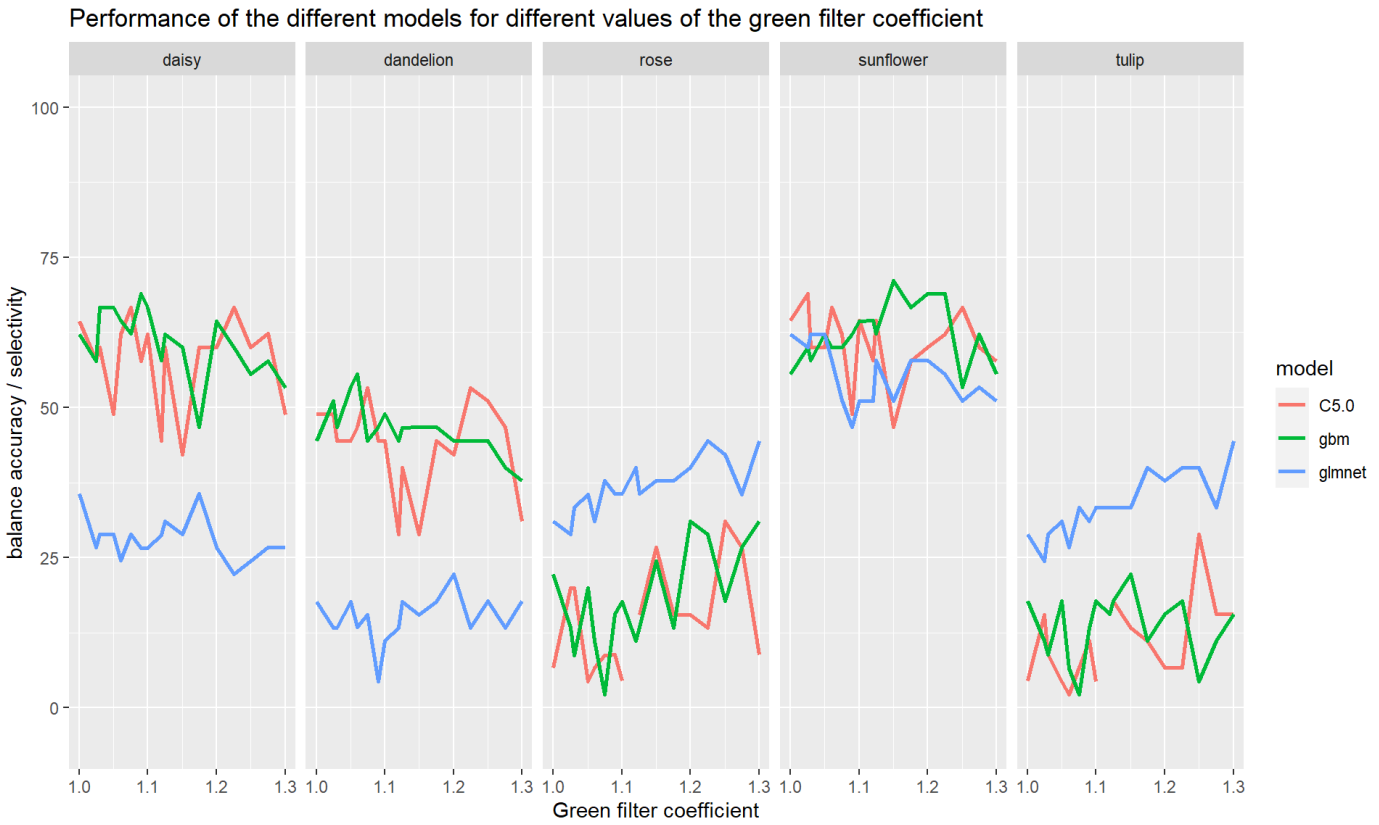
Now we generate pictures reworked with this optimal blue filter in folder ProcessedPictures for future usage:

```
# Processing the processed pictures and replacing them.

for(n in 1:nrow(trainSet)) {

  matrixPic <- readJPEG(trainSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "blue", optimalBlueFilterCoef)

  writeJPEG(matrixPic, target = trainSet$processed[n], quality = 1.0, bg = "white")

}


# Processing the test set.

for(n in 1:nrow(testSet)) {

  matrixPic <- readJPEG(testSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "blue", optimalBlueFilterCoef)

  writeJPEG(matrixPic, target = testSet$processed[n], quality = 1.0, bg = "white")

}

rm(matrixPic, finalResults, n)
```

The same method is applied to determine the best green filter coefficient:

```
compareTrainModels <- data.frame(model = character(), flowerType = character(), name = character(), prediction = character(), greenCoef = numeric())

# Process the pictures with different levels of green filter coef, store it in the temp folder.

for(greenCoef in seq(1., 1.3, 0.025)) {

# Processing the training set.

for(n in 1:nrow(trainSet)) {

  matrixPic <- readJPEG(trainSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "green", greenCoef)

  writeJPEG(matrixPic, target = trainSet$temp[n], quality = 1.0, bg = "white")

}


# Processing the test set.

for(n in 1:nrow(testSet)) {

  matrixPic <- readJPEG(testSet$processed[n])

  matrixPic <- FilterColor(matrixPic, "green", greenCoef)

  writeJPEG(matrixPic, target = testSet$temp[n], quality = 1.0, bg = "white")

}



  filteredValuesTraining <- GetThePredictors(trainSet, optimalNrColors, TRUE)

  filteredValuesTest <- GetThePredictors(testSet, optimalNrColors, FALSE)



for (m in 1:length(trainModels)) {

set.seed(1, sample.kind="Rounding")

trainingResults <- train(flowerType ~ ., filteredValuesTraining, method = trainModels[m])

prediction <- predict(trainingResults, filteredValuesTest)

compareTrainModels <- rbind(compareTrainModels, data.frame(model = trainModels[m], name = testSet$source, flowerType = testSet$flowerType, prediction, greenCoef))

}

}

rm(prediction, trainingResults, finalPredictorsTest, filteredValuesTest, filteredValuesTraining, matrixPic, greenCoef, m, n)
```

Results are summarized graphically in the graph below:



Once again here it is not easy to get a high-level tendency. Lines appear to be a bit flat, except maybe the glmnet curve for roses and tulips that seems to get an optimal for higher values.

```
# Keep the green filter coefficient giving the best average value for the different flowers.

optimalGreenFilterCoef <- finalResults %>%
                          group_by(greenCoef, model) %>%
                          summarize(avgBalance = mean(balance)) %>%
                          arrange(desc(avgBalance)) %>%
                          ungroup() %>%
                          top_n(1) %>%
                          select(greenCoef) %>%
                          arrange(greenCoef) %>%     # In case of equality, keep the lowest.
                          top_n(1) %>%
                          pull(greenCoef)
```

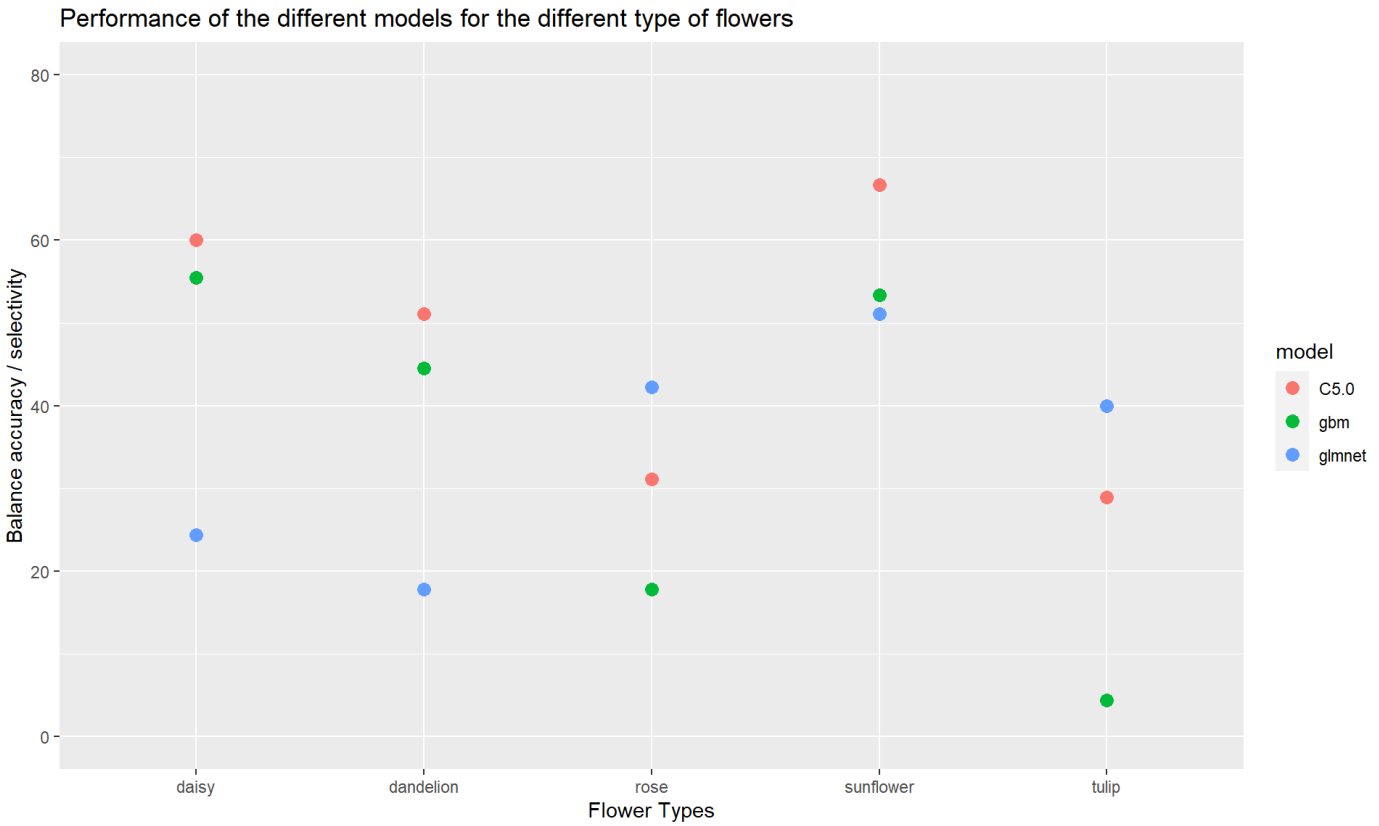
The optimal value of the green filter coefficient is computed as the value giving the highest average balance for all flowers and the best model, in this case **1.25**. Let's seen now how the implementation of the green filter makes the results evolve:

```
##              step  model daisy dandelion rose sunflower tulip
## 1     Green Filter   C5.0  60.0      51.1 31.1      66.7  28.9
## 2     Green Filter    gbm  55.5      44.5 17.8      53.4   4.4
## 3     Green Filter glmnet  24.4      17.8 42.2      51.1  40.0
## 4      Blue Filter   C5.0  66.7      51.1 33.3      55.5  17.8
## 5      Blue Filter    gbm  57.8      44.5 28.8      57.8  17.8
## 6      Blue Filter glmnet  20.0      15.6 40.0      48.9  28.9
## 7  Contrast Filter   C5.0  60.0      46.7 22.2      55.5  17.8
## 8  Contrast Filter    gbm  57.8      48.9 28.9      66.7  17.8
## 9  Contrast Filter glmnet  13.3      15.6 40.0      57.8  35.5
## 10    Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 11    Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 12    Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

Balance for sunflowers and tulips is increased, balance for daisies is a bit reduced. C5.0 remains the best model, geenral accuracy is a bit better, however overall improvement remains moderate:

```
##             step model overallAccuracy
## 1    Green Filter  C5.0          73.78
## 2     Blue Filter  C5.0          72.44
## 3 Contrast Filter   gbm          72.00
## 4    Raw Pictures  C5.0          70.68
```

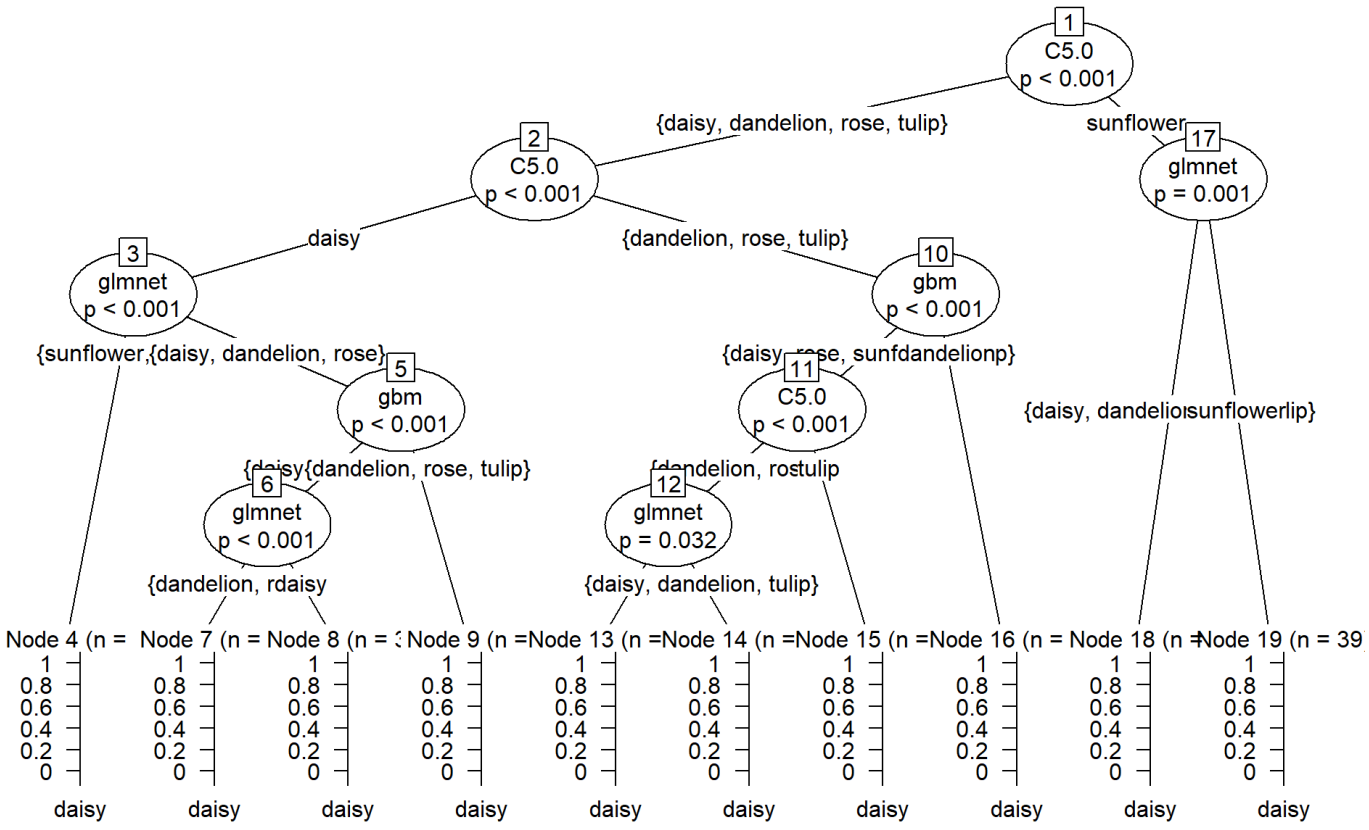
When taking the best values we got so far (green filter), we notice that still some models work better for different kind of flowers:



From the beginning we have noticed that gbm and C5.0 work better for daisies, dandelions and sunflowers, and that glmnet works better for roses and tulips.

To take the most benefit of this, we will add as a final stage a decision tree that will ponderate the results given by the different models. We will try to build two different trees from available packages: *rpart* and *ctree*. We can reuse directly the results coming from the previous steps by filtering the results using the optimal values of coefficients we found before.

Use of *ctree* lead to the following arborescence:



This model considers first results of C5.0 before switching in a lower node to the glmnet results. It is a bit surprising, as I would have expected a split the other way round, consider if the glmnet gives a rose or a tulip, and if not check result of C5.0 and gbm.

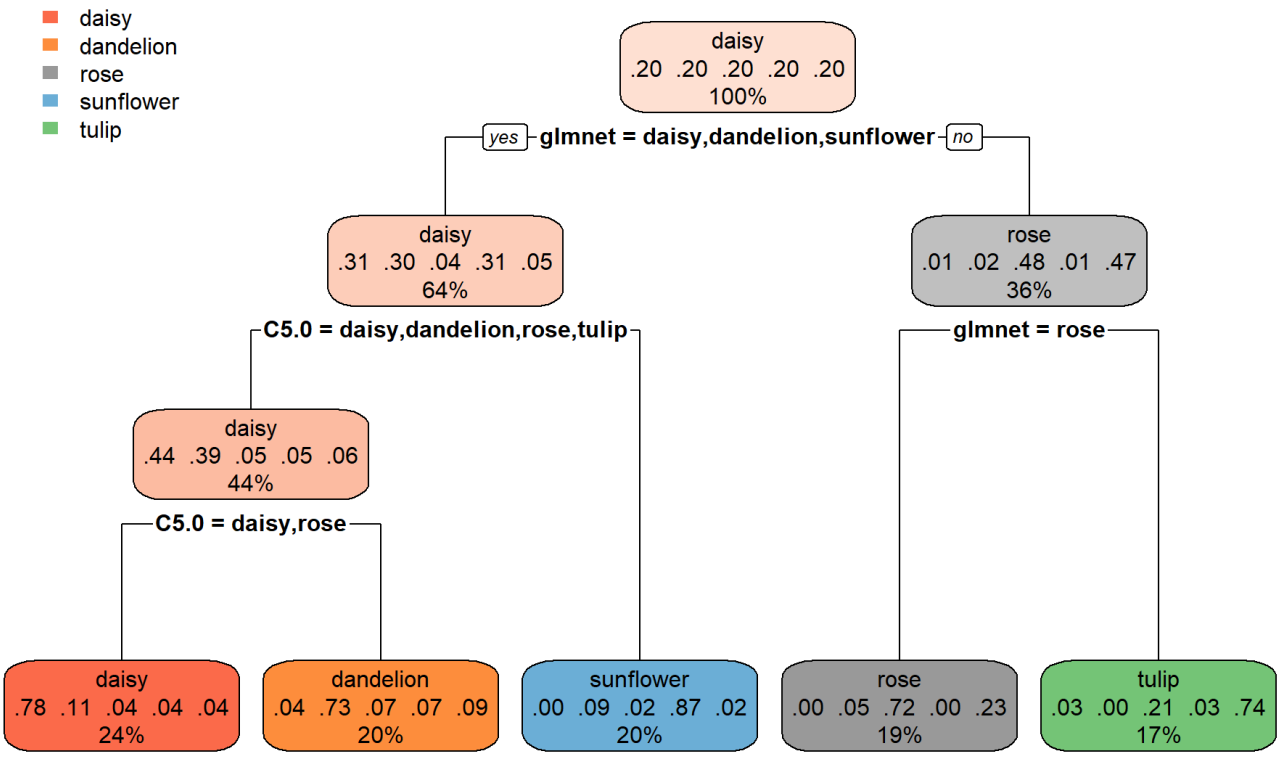
Finally, predictions made with the ctree lead to the following results:

```
##              step  model daisy dandelion rose sunflower tulip
## 1   Decision Tree  cTree  60.0      53.3 37.7      66.7  20.0
## 2    Green Filter   C5.0  60.0      51.1 31.1      66.7  28.9
## 3    Green Filter    gbm  55.5      44.5 17.8      53.4   4.4
## 4    Green Filter glmnet  24.4      17.8 42.2      51.1  40.0
## 5     Blue Filter   C5.0  66.7      51.1 33.3      55.5  17.8
## 6     Blue Filter    gbm  57.8      44.5 28.8      57.8  17.8
## 7     Blue Filter glmnet  20.0      15.6 40.0      48.9  28.9
## 8  Contrast Filter   C5.0  60.0      46.7 22.2      55.5  17.8
## 9  Contrast Filter    gbm  57.8      48.9 28.9      66.7  17.8
## 10 Contrast Filter glmnet  13.3      15.6 40.0      57.8  35.5
## 11    Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 12    Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 13    Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

And it doesn't lead to any improvment at all, overall accuracy remains stable:

```
##              step model overallAccuracy
## 1   Decision Tree cTree           73.78
## 2    Green Filter  C5.0           73.78
## 3     Blue Filter  C5.0           72.44
## 4 Contrast Filter   gbm           72.00
## 5    Raw Pictures  C5.0           70.68
```

Now we can try the same using the *rpart* package:



Approach if this tree seems to be a bit more rationnal, consider glmnet for roses and tulips, C5.0 otherwise. We end up with the following results:

```
##              step  model daisy dandelion rose sunflower tulip
## 1   Decision Tree  rpart  66.6      46.6 42.2      73.4  40.0
## 2   Decision Tree  cTree  60.0      53.3 37.7      66.7  20.0
## 3    Green Filter   C5.0  60.0      51.1 31.1      66.7  28.9
## 4    Green Filter    gbm  55.5      44.5 17.8      53.4   4.4
## 5    Green Filter glmnet  24.4      17.8 42.2      51.1  40.0
## 6     Blue Filter   C5.0  66.7      51.1 33.3      55.5  17.8
## 7     Blue Filter    gbm  57.8      44.5 28.8      57.8  17.8
## 8     Blue Filter glmnet  20.0      15.6 40.0      48.9  28.9
## 9  Contrast Filter   C5.0  60.0      46.7 22.2      55.5  17.8
## 10 Contrast Filter    gbm  57.8      48.9 28.9      66.7  17.8
## 11 Contrast Filter glmnet  13.3      15.6 40.0      57.8  35.5
## 12    Raw Pictures   C5.0  48.9      42.3 26.7      55.5  33.4
## 13    Raw Pictures    gbm  44.5      44.5 15.5      44.4  35.5
## 14    Raw Pictures glmnet  15.6       6.7 37.7      46.6  42.2
```

This time the overall accuracy is increased:

```
##              step model overallAccuracy
## 1   Decision Tree rpart           76.88
## 2   Decision Tree cTree           73.78
## 3    Green Filter  C5.0           73.78
## 4     Blue Filter  C5.0           72.44
## 5 Contrast Filter   gbm           72.00
## 6    Raw Pictures  C5.0           70.68
```

This is the best overall accuracy we have so far, we will keep those to assess the performance with the verification set.

Part 5: Performance on the validation set

We now have determined experimentally the different parameters of our model, which models to keep, we can measure the performance of our algorithm on the validation set. We group the training and the test sets used so far into a single set that will become the training set, process the pictures, train the model and see the accuracy of predictions:

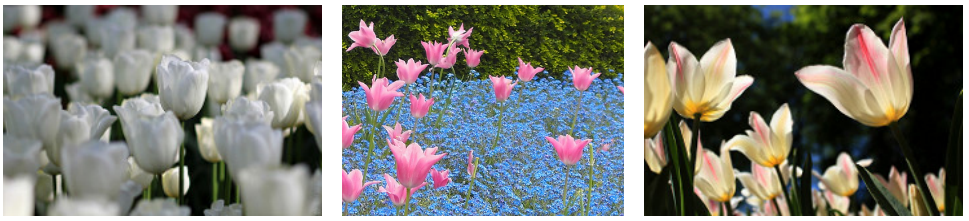
```
# training and test data sets are grouped in a single one that will become the final training set. Pictures are processed with the different steps using the optimal parameters.

compareTrainModels <- data.frame(model = character(), name = character(), flowerType = character(), prediction = character())

uniqueTrainingSet <- rbind(trainSet, testSet)


# Processing of training pictures with optimal coefficients found earlier.

for(n in 1:nrow(uniqueTrainingSet)) {

  matrixPic <- readJPEG(uniqueTrainingSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, optimalContrastQuantile)

  matrixPic <- FilterColor(matrixPic, "blue", optimalBlueFilterCoef)

  matrixPic <- FilterColor(matrixPic, "green", optimalGreenFilterCoef)

  writeJPEG(matrixPic, target = uniqueTrainingSet$processed[n], quality = 1.0, bg = "white")

}


# Processing the validation set pictures.

for(n in 1:nrow(validationSet)) {

  matrixPic <- readJPEG(validationSet$original[n])

  matrixPic <- FilterHighestContrastZones(matrixPic, optimalContrastQuantile)

  matrixPic <- FilterColor(matrixPic, "blue", optimalBlueFilterCoef)

  matrixPic <- FilterColor(matrixPic, "green", optimalGreenFilterCoef)

  writeJPEG(matrixPic, target = validationSet$processed[n], quality = 1.0, bg = "white")

}

# Now the processed pictures should be used

uniqueTrainingSet <- uniqueTrainingSet %>%
                     mutate(source = str_replace(source, "Original", "Processed"))

validationSet <- validationSet %>%
                   mutate(source = str_replace(source, "Original", "Processed"))



filteredValuesTraining <- GetThePredictors(uniqueTrainingSet, optimalNrColors, TRUE)

filteredValuesValidation <- GetThePredictors(validationSet, optimalNrColors, FALSE)


for (m in 1:length(trainModels)) {

  set.seed(1, sample.kind="Rounding")

  trainingResults <- train(flowerType ~ ., filteredValuesTraining, method = trainModels[m])

  prediction <- predict(trainingResults, filteredValuesValidation)

  compareTrainModels <- rbind(compareTrainModels, data.frame(model = trainModels[m], name = validationSet$source, flowerType = validationSet$flowerType, prediction))

}

# Spreading the results to make them ready for the tree.

spreadValidation <- spread(compareTrainModels, model, prediction)

spreadValidationLight <- spreadValidation %>% select(-name, -flowerType)
```

```
# Use the best tree selected earlier, if cTree, the prediction is direct, if rpart, result has to be converted.

if(cTreeAccuracy > rpartAccuracy)           # Select the tree giving the best results.
{treePrediction <- predict(bestTree, spreadValidationLight) } else
{treePrediction <- predict(bestTree, spreadValidationLight)
 treePrediction <- colnames(treePrediction)[max.col(treePrediction)]} #if the best is rpart, a line need to be added to transf
orm the prediction from a probability to a factor.


validationResults <- data.frame(name = spreadValidation$name, flowerType = spreadValidation$flowerType, prediction = treePredi
ction, stringsAsFactors = FALSE)

validationFinalResults <- validationResults %>%
                          mutate(correct = flowerType == prediction) %>%
                          group_by(flowerType) %>%
                          summarise(accuracy = mean(correct))


validationResultsOverview <- spread(validationFinalResults, flowerType, accuracy)

rm(trainSet, matrixPic, testSet, n, prediction, trainingResults, compareTrainModels)
```

The application on the validation set gives an accuracy of **0.62**. Result apperas to be a bit disappointing, likely suffered of overtraining. The detail of accuracies for each type of flower is displayed below (for final results we can work with accuracies, no longer the balance accuarcy / selectivity):

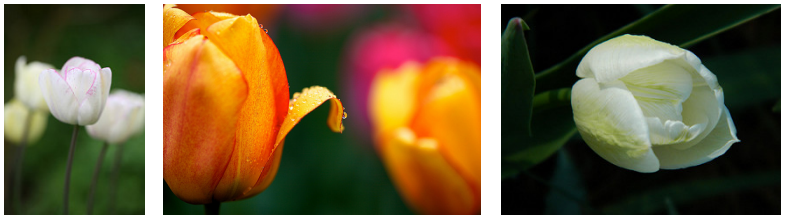
```
##   daisy dandelion rose sunflower tulip
## 1  0.76      0.58 0.36       0.9  0.52
```

We can have a look to some examples of pictures wrongly identified as daisies:



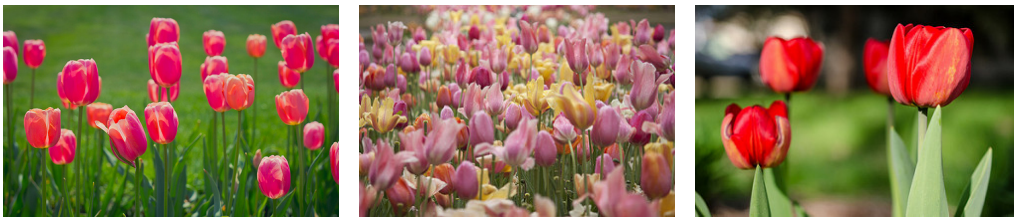
Example of pictures wrongly identified as daisies

Example of pictures wrongly identified as dandelions:



Example of pictures wrongly identified as dandelions

Example of pictures wrongly identified as roses:



Example of pictures wrongly identified as roses

Example of pictures wrongly identified as sunflowers:

Example of pictures wrongly identified as sunflowers

Example of pictures wrongly identified as tulips:

Example of pictures wrongly identified as tulips

Part 6: Conclusion and perspectives

We report here an example of development of a machine-learning algorithm finally able to distinguish five different kinds of flowers only based on their color. We have slowly enhanced the accuracy step by step and finally end up with an somewhat acceptable accuracy, still far from perfect for sure. I guess a larger data set would help to reduce the overtraining. Eventually it should also be possible to run the algorithm multiple times with different training / test sets with a variation of the seed used to generate the training and test sets. The color only may not be enough to identify flowers, elements of shape should also be considered.

We could also imagine other possibilities to enhance this algorithm:

- The probably most obvious enhancement would be to finetune the different models used for the training. Here we make use the *caret* package with its default parameters but it would likely be useful to finetune the parameters available (number of trees, of iterations, etc.) for each of the several models used;
- A major enhancement would consist in training the algorithm to ditinguish the core of the flower from the petals. Many cases of wrong identification come from the fact that some flowers share the same colors (daisies core and sunflowers petals are both yellow for example). The use of the function *ContrastPicture* designed in first part of chapter 4 of this report could allow the system to make the distinction of the core (small zone of high contrast for daisies, dandelions and sunflowers) from the petals (large zones of low contrast but encircled by a thin zone of high contrast). Also following the distinction of petals, their length and width could be used for identification as the shape strongly differ from a flower to another. First attempts on this topic show very interesting results;
- We could imagine to use the existing function *FilterColor* to focus on green zones and extract information from the leaves and the stems (size, colors, shapes) useful for identification;
- At some point, doing the distinction between the dandelion seedhead and flower and split this single flower type in two may help to increase the accuracy for dandelions. This requires a tedious classification from the original data set but seems to be an interesting and easy way to make improvements;
- One of the hard points remains identification of tulips, mostly because their variety of colors. An interesting fact about tulips is that they are very common in parterres, where individual flower can easily be spotted using the function *FilterHighestContrastZones* with a high-level of contrast filter. The resulting processed picture has specific patterns of high-contrast zones that can be a useful information in the identification of tulips;

Following enhancements listed before, the same algorithm could be tried on a wider range of flowers.

Thank you for you review!