Stat 432 Final Project

CAPTCHA Images classification

Team: STAT Never Giveup

Member: Liding Li, Shuxun Zhou, Patrick Bai

Date: 2018/12/15

Introduction

The CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) dataset was built in 1997 as a way for user to identify and block computers. The dataset contains 1030 PNG files with a scale of 200x50. Each image is five letter words combined with numbers and alphabets. It also contains noises such as blur and a line bypassing all 5 characters.

The object of the project is to use two separate methods to correctly identify each captcha in the image. Both methods involves segmenting the captcha to 5 characters first. The first method focuses on using K-nearest neighbour (KNN) on each character while the second method uses Convolutional Neural Network (CNN) to identify each captcha.

Literature Review

1. OpenCV Word Segmenting on CAPTCHA Images
   reference: https://www.kaggle.com/fournierp/opencv-word-segmenting-on-captcha-images
   Mathematical Morphology in Any Number of Dimensions
   reference: https://www.r-project.org/nosvn/pandoc/mmand.html

   We used above sources as our methods to remove noise in each captcha image. Mathematical Morphology is an image processing technique used to remove certain features in binary or grayscale image. In our case, we tried to remove horizontal line across the whole image, so using dilate() function in the package "mmand" would be a suitable approach. This function expands some areas into neighbourhood pixels. The segmentation method mentioned in the first source, however, does not seem to perform well on overall images. Therefore, we develop our own way, constrained k-mean, to segment image.

2. CNN impelemtation in R

reference:https://www.researchgate.net/post/How_to_implement_Convolutional_Neural_Network_in_R

reference: https://www.r-bloggers.com/image-recognition-tutorial-in-r-using-deep-convolutional-neural-networks-mxnet-package/

The above two sources provide R tutorials of performing CNN with package MXNet. We used the tutorials as ways to perform CNN on our dataset.

Methodology and Analysis

Image loading

```
library(colorspace)
library(png)
library(mmand)

# the list of file names in zip-folder.
list_ = list.files(path ="C:/Users/Shuxun Zhou/Desktop/captcha-version-2-images/432data")
list_ = list_[-898]
# a list storing pixel of each observation
datalist = list()

# storing pixel of observation into datalist # showing the five observations.
for (i in 1:length(list_)){
  a = readPNG(paste("C:/Users/Shuxun Zhou/Desktop/captcha-version-2-images/432data/",list_[i], sep = ""))
  datalist[[i]] = a

}

# let's take a look at our first sample before modifying it.
a = datalist[[3]]
plot(c(0, 200), c(0, 50), xaxt = 'n', yaxt = 'n', bty = 'n', pch = '', ylab = '', xlab = '')
rasterImage(a, 0, 0, 200, 50)
```

1. Noise Removal

First, our team tried to remove the noise of the captcha images. As shown in the data visualization, all the captchas have a line passing through all the characters as noise. The background color of the images is grey. Some boundaries of the characters are blurred. All these are considered to be the noises. To remove noises, we first transformed the image to black and white so that that background color and blurring of boundaries will not distract the machine from learning the true value (for KNN only, we transformed back to grayscale in CNN). Specifically, we defined a threshold of 0.5 on the gray scale image. When the grey scale is less than 0.5, it is considered to be a black pixel. Otherwise, the pixel is considered to be white. Then, we performed erosion on the image in order to remove the line across the characters in a captcha. Erosion is one of the morphological methods that often used in image processing to remove noises. This method is specifically stated in our "literature review section". This method applies a "filter" which scans through the image and whiten the pixels that do not fit the "filter". This method successfully removed a substantial amount of the noises. Although a small portion of characters are also removed and cause information loss, the loss is rather insignificant.

Binary conversion

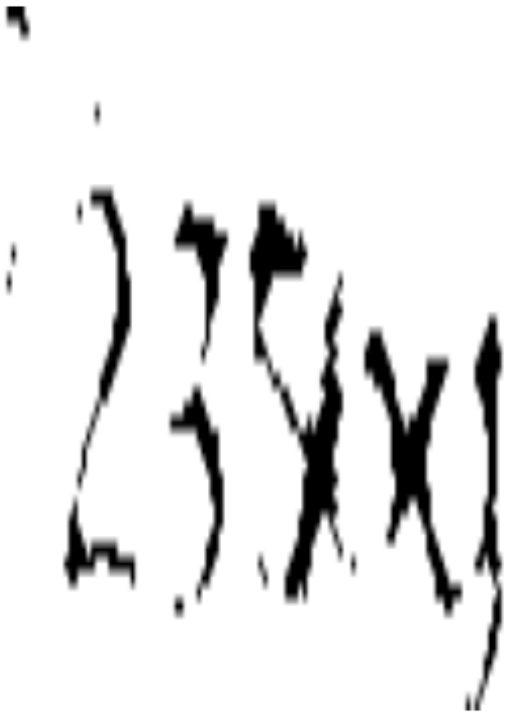
```
binary_conversion = function(img){
  img_expand = apply(img, 3, c)
  new_img = img[,,1]
  for (i in 1:200) {
    for (j in 1:50) {
      if (img[j,i,1] > 0.5) { new_img[j,i] = 1}
      else if (img[j,i,1] < 0.5) { new_img[j,i] = 0 }
    }
  }

  return(new_img)
}
```

erosion

```
erosion = function(img){
  k <- shapeKernel(c(4,4), type="disc")
  new_image = dilate(img,k)
  return(new_image)
}

#output sample
image = binary_conversion(datalist[[3]])
print(list_[3])
```

```
## [1] "2356g.png"
```

```
image = erosion(image)

par(mar=rep(0.2, 4))
plot(c(0, 200), c(0, 50), xaxt = 'n', yaxt = 'n', bty = 'n', pch = '',
ylab = '', xlab = '')
rasterImage(image, 0, 0, 200, 50)
```



2. Image segmentation with a constrained K-mean

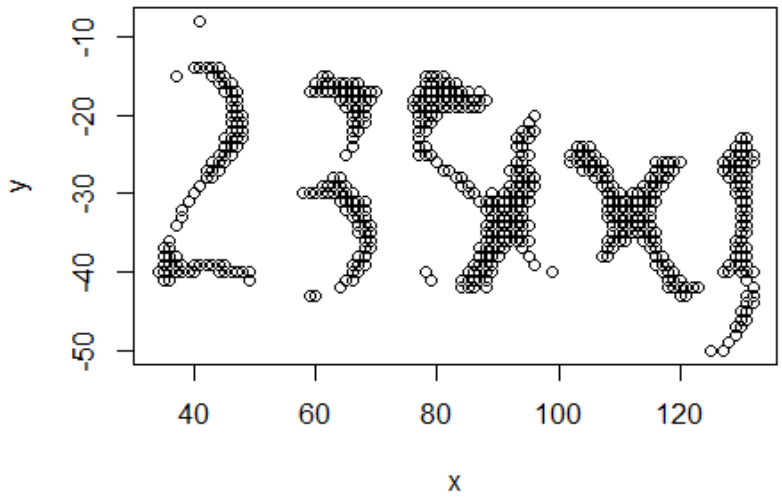
After we removed all the noises, we transformed all the pixels of 1 and 0 to their respective coordinate. If the pixel was black, we recorded the coordinate of that pixel. Then we projected all the x-y coordinate on the x-axis and performed K-mean on the projected points. The y-coordinates were not used in performing k-

mean because the characters are not tilted. We made the important assumption that the centers of the 5 characters are uniformly distributed in a certain region. As a result, we are able to set the initial condition of the k-mean algorithm in a rational manner. We chose k-mean instead of segmenting the image strictly because the characters varies in length and sizes. For example, in most captcha, 'w' and 'm' are wider than all other characters. Performing K-mean provides flexibility in adjusting the size of individual characters. It was also better than performing column sums of points in a pixel because some characters were sticked together. Lastly, we also constrained the movement of the centers because most characters were not symmetric and some of them sticked together. For example, when 'd' and 'b' sticks together, the 2 centers would be close to the middle since there are more points in the strokes of the 2 characters.

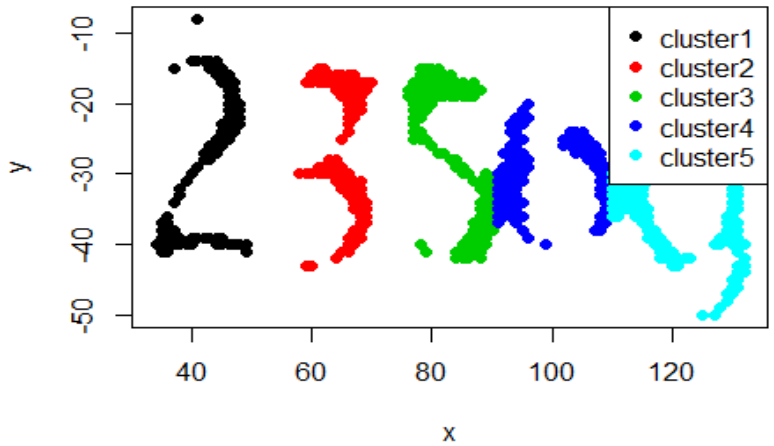
constrained k-mean

```
con_k_mean = function(X){
  #after doing erosion, we make assumptions that the center are uniform
ly distributed and thus calculate the expected value of 5 centers
  len_ = max(X[,1])-min(X[,1])
  len_ = len_/5

  ini_mean_x = min(X[,1]) + c(len_/2, len_/2*3, len_/2*5, len_/2*7, len
_/2*9)
  ini_mean_y = rep(-25,5)

  mean_x = rep(0,5)
  mean_y = rep(0,5)

  #perform k-mean only on x asis with constraint that the x-coordinate
of center should not deviate then expected center by len_/4
  cluster = rep(0,length(X[,1]))
  #first random shuffle our X
  for(i in 1:length(X[,1])){
    cluster[i] = i%%5 + 1
  }

  mean_x = ini_mean_x
  #update distance matrix to the mean
  distance = matrix(0,nrow = length(cluster), ncol = 5)
  for(i in 1:length(cluster)){
    for(j in 1:5){
      distance[i,j] = abs(X[i,1]-mean_x[j])
    }
    cluster[i] = which.min(distance[i,])
  }

  #update new_mean
  mean_x_ori = mean_x
  for(i in 1:5){
```

```
    a = sum(X[which(X[,1]==i),1])/sum(cluster==i)
    if((ini_mean_x[i] - a) > len_/4){
      a = ini_mean_x[i]-len_/4
    }else if((a - ini_mean_x[i]) > len_/4){
      a = ini_mean_x[i]+len_/4
    }
    mean_x[i] = a
  }

  while(abs(sum(mean_x)-sum(mean_x_ori))>0.001){
    for(i in 1:length(cluster)){
      for(j in 1:5){
        distance[i,j] = abs(X[i,1]-mean_x[j])
      }
    cluster[i] = which.min(distance[i,])
    }
    mean_x_ori = mean_x
    for(i in 1:5){
      a = sum(X[which(X[,1]==i),1])/sum(cluster==i)
      if(ini_mean_x[i] - a >len_/8){
        a = ini_mean_x[i]-len_/8
      }else if(a - ini_mean_x[i] > len_/6){
        a = ini_mean_x[i]+len_/8
    }
    mean_x[i] = a
    }
  }

  return (cbind(X,cluster))
}

#output sample,after  using constrained k-mean
x= integer(0)
y = integer(0)

for(i in 1:200){
  for(j in 1:50){
    if(image[j,i] == 0 && i >= 30){
      x = c(x,i)
      y = c(y,-j)
    }
  }
}

plot(x,y)
```




```
X = cbind(x,y)
result = as.data.frame(table(x))

a = con_k_mean(X)

km_fit_ = con_k_mean(X)
color = as.numeric(km_fit_[,3])
plot(X, col = color, pch = 19)
legend("topright", c("cluster1", "cluster2", "cluster3", "cluster4", "c
luster5"), col = 1:5, pch = 19)
```



3. Centering the new graph

After we divided the graph into 5 clusters, we can move the points of every cluster to an image of 50*40. When performing KNN, we have to ensure that the number of predictors are the same, so we transformed the coordinates back to the binary representations of 1 and 0. Also, we have calculated the center of every cluster so we are able to center every cluster to the middle of the new image. This is an essential step because KNN is very sensitive to the position of black pixels.

```
#@parame km_fit, char->actual result
#return a design matrix with actual result attached to the front
merge_to_design = function(kmfit, char){

  Xmatrix = matrix(1, ncol=2000, nrow = 5)
  for(i in 1:5){
    x_cor = kmfit[which(kmfit[,3]==i),1]
    y_cor = abs(kmfit[which(kmfit[,3]==i),2])
    x_mid = as.integer((max(x_cor)+min(x_cor))/2)
    y_mid = as.integer((max(y_cor)+min(y_cor))/2)

    x_movement = x_mid - 20
    y_movement = y_mid - 25

    for(j in 1:length(x_cor)){
      x_final = x_cor[j] - x_movement
      y_final = y_cor[j] - y_movement

      Xmatrix[i,(y_final-1)*40+x_final] = 0
    }

  }

  return(cbind(char,Xmatrix))
}

#output sample
chara = as.vector(utf8ToInt(list_[3]))[1:5]
m = merge_to_design(km_fit_, chara)
plot(c(0, 40), c(0, 50), xaxt = 'n', yaxt = 'n', bty = 'n', pch = '', y
lab = '', xlab = '')
rasterImage(matrix(m[5,-1], nrow = 40, ncol = 50), 0, 0, 40, 50)
```

4. Performing simple KNN

We first divide the dataset into training and testing set. Each set consists of half of the whole dataset. For each image of captcha, we performed the operations above and divided every captcha into 5 separate observations. However, we did not want to perform KNN on such a large dataset because KNN involves sorting and makes the algorithm inefficient. Furthermore, since we are using 1-NN, overfitting is very likely to occur because there are 2000 predictors.

```
##break half to training and half to testing

training = matrix(0, nrow = 0, ncol = 2001)
testing = matrix(0, nrow = 0, ncol = 2001)

for(k in 1:length(list_)){
  image = binary_conversion(datalist[[k]])
  image = erosion(image)

  #convert image the black and white image to x-y coordinate
  x= integer(0)
  y = integer(0)

  for(i in 1:200){
    for(j in 1:50){
      if(image[j,i] == 0 && i >= 30){
        x = c(x,i)
        y = c(y,-j)
      }
    }
  }

  X = cbind(x,y)
  km_fit_ = con_k_mean(X)
  chara = as.vector(utf8ToInt(list_[k]))[1:5]
```

```
  m = merge_to_design(km_fit_, chara) #a 5 by n matrix with first colum
n as the result
  if(k%%2 == 0){
    training = rbind(training,m)
  }else{
    testing = rbind(testing,m)
  }
}
```

Therefore, we averaged all the pixels of a given character. For example, we averaged all the pixels of the letter 'g'. The grey scale picture is a visualization of such technique. Darkness represents the probability of occurrence of the pixel. Darker regions represent that the pixel occurs more frequently and lighter regions represent the pixel occurs less frequently. As a result, we are able to condense the 5000 training characters to 36 (a-z, 0-9). When we perform KNN to predict the testing set, the prediction accuracy for individual character is xxx while the prediction accuracy for individual captcha is xxx. On average, the model performed fairly well.

```
#It's very inefficient to do KNN given a dataset with large predictors.
 Therefore let's do a simple average then compute the KNN to see the re
sult
pred = as.vector(utf8ToInt("abcdefghijklmnopqrstuvwxyz0123456789"))
statistics = matrix(0,ncol = 2001, nrow = 0)
for(i in 1:36){
  index = which(training[,1] == pred[i])
  subset_ = training[index,]
  subset_ = subset_[,-1]
  subset_mean = colMeans(subset_)
  subset_mean = c(pred[i],subset_mean)
  statistics = rbind(statistics,subset_mean)

}

#perform KNN and see the result
#Let's first take a look at the avg picture
i = 16
intToUtf8(statistics[i,1])

## [1] "p"

plot(c(0, 40), c(0, 50), xaxt = 'n', yaxt = 'n', bty = 'n', pch = '', y
lab = '', xlab = '')
rasterImage(matrix(statistics[i,-1], nrow = 40, ncol = 50), 0, 0, 40, 5
0)
```

```
prediction = vector(length = 0)
#Perform KNN
for(i in 1:nrow(testing)){
  obs = testing[i,-1]
  result = vector(length = 0)
  for(j in 1:36){
    result = c(result,sum((obs-statistics[j,-1])^2))
  }
  prediction = c(prediction,pred[which.min(result)])
}

#calculate the prediction accuracy
mean(prediction==testing[,1])

## [1] 0.6838462

accuracy = integer(0)
for(i in 1:(length(prediction)/5)){
  for(j in 1:5){
    idx = (i-1)*5 + j

    if(prediction[idx] != testing[idx,1]){
      accuracy = c(accuracy,0)
      break;
    }
    accuracy = c(accuracy,1)
  }
}
length(accuracy)

## [1] 1545

mean(accuracy)

## [1] 0.7430421
```

5. Adding credibility to our model

Since performing simple average is like using a kernel then perform KNN, can we loose our assumptions about it? We proposed the weight "credibility score" as weight. Instead of doing a simple average, our team change it to the weighted average and the credibility score is the weight. The intuition is that the segmentation of characters are often imperfect and this causes problem in KNN. According to our observation, the first and last character usually have good segmentation while it sometimes messed up in the middle. Therefore, we assign highest score to the first and last character of a captcha image, and less weight to the middle characters. However, the prediction accuracy actually drops slightly when we use a weight of (1, 0.3, 0.5, 0.7, 0.8). But this model is a more generalized one because if the 5 weights are equal, it is essentially doing a simple average.

involving credability

```
credability = c(1, 0.3, 0.5, 0.7, 0.8)
credability = rep(credability, times = nrow(training)/5)
training_with_cred = cbind(credability,training)
new_training = matrix(0,nrow = 0, ncol = 2002)
for(i in 1:nrow(training_with_cred)){
  v = training_with_cred[i,-c(1,2)]*training_with_cred[i,1]
  v = c(training_with_cred[i,2],training_with_cred[i,1],v)
  new_training = rbind(new_training,v)
}

statistics = matrix(0,ncol = 2002, nrow = 0)
for(i in 1:36){
  index = which(new_training[,1] == pred[i])
  subset_ = new_training[index,]
  subset_ = subset_[,-1]
  subset_sum = colSums(subset_)
  if(subset_sum[1] == 0){
    subset_mean = subset_sum
  }else{
    subset_mean = subset_sum/subset_sum[1]
  }
  subset_mean = c(pred[i],subset_mean)
  statistics = rbind(statistics,subset_mean)

}

prediction = vector(length = 0)
#Perform KNN
for(i in 1:nrow(testing)){
  obs = testing[i,-1]
  result = vector(length = 0)
  for(j in 1:36){
    statistics[j,-c(1,2)]
    result = c(result,sum((obs-statistics[j,-c(1,2)])^2))
```

```
    result
  }
  prediction = c(prediction,pred[which.min(result)])
}

#calculate the prediction accuracy
mean(prediction == testing[,1])
```

```
## [1] 0.675
```

```
accuracy = integer(0)
for(i in 1:(length(prediction)/5)){
  for(j in 1:5){
    idx = (i-1)*5 + j

    if(prediction[idx] != testing[idx,1]){
      accuracy = c(accuracy,0)
      break;
    }
    accuracy = c(accuracy,1)
  }
}
mean(accuracy)
```

```
## [1] 0.7331169
```

6. Performing CNN

Our last model involves using CNN. We used the package "mxnet" to do the job. Our CNN model has 2 hidden layers whose nodes are fully connected. The first hidden layer has total nodes of 500 while the second layer has nodes of 200. We have also tuned the parameters manually like the learning rate (rate of change of gradient method), batch size and number of iterations. Overall, we have achieved a trainning accuracy of 0.8838 and a prediction accuracy with whole captcha of 0.8122.

CNN

```
merge_to_design_new = function(kmfit, char,img){

  Xmatrix = matrix(1, ncol=2000, nrow = 5)
  for(i in 1:5){
    x_cor = kmfit[which(kmfit[,3]==i),1]
    y_cor = abs(kmfit[which(kmfit[,3]==i),2])
    x_mid = as.integer((max(x_cor)+min(x_cor))/2)
    y_mid = as.integer((max(y_cor)+min(y_cor))/2)

    x_movement = x_mid - 20
    y_movement = y_mid - 25
```

```
    for(j in 1:length(x_cor)){
      x_final = x_cor[j] - x_movement
      y_final = y_cor[j] - y_movement

      Xmatrix[i,(y_final-1)*40+x_final] = img[y_cor[j],x_cor[j]]
    }

  }

  return(cbind(char,Xmatrix))
}

##break half to training and half to testing

training = matrix(0, nrow = 0, ncol = 2001)
testing = matrix(0, nrow = 0, ncol = 2001)

for(k in 1:length(list_)){
  image = binary_conversion(datalist[[k]])
  image = erosion(image)

  #convert image the black and white image to x-y coordinate
  x= integer(0)
  y = integer(0)

  for(i in 1:200){
    for(j in 1:50){
      if(image[j,i] <0.5 && i >= 30){
        x = c(x,i)
        y = c(y,-j)
      }
    }
  }

  X = cbind(x,y)
  km_fit_ = con_k_mean(X)
  chara = as.vector(utf8ToInt(list_[k]))[1:5]
  m = merge_to_design_new(km_fit_, chara,datalist[[k]][,,1]) #a 5 by n
matrix with first column as the result
  if(k%%2 == 0){
    training = rbind(training,m)
  }else{
    testing = rbind(testing,m)
  }
}
```

```
require(mxnet)

## Loading required package: mxnet

train.x = t(training[,-1])
train.y = training[,1]

test_x <- t(testing[, -1])
test_y <- testing[,1]
for(i in 1:length(train.y)){
  train.y[i] = which(pred == train.y[i])-1
  test_y[i] = which(pred == test_y[i])-1
}

test_array <- test_x

data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=500)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=200)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
fc3 <- mx.symbol.FullyConnected(act2, name="fc3", num_hidden=36)
softmax <- mx.symbol.SoftmaxOutput(fc3, name="sm")

devices <- mx.cpu()
mx.set.seed(100)
model <- mx.model.FeedForward.create(softmax, X=train.x, y=train.y,
                                     ctx=devices, num.round=50, array.b
atch.size=100,
                                     learning.rate = 0.02, momentum=0.7
,  eval.metric=mx.metric.accuracy,
                                     initializer=mx.init.uniform(0.05),
                                     epoch.end.callback=mx.callback.log
.train.metric(100)
                                      )

## Warning in mx.model.select.layout.train(X, y): Auto detect layout in
put matrix, use colmajor..

## Start training with 1 devices

## [1] Train-accuracy=0.0734615388254707

## [2] Train-accuracy=0.1142307693282

## [3] Train-accuracy=0.138846153393388

## [4] Train-accuracy=0.203846155021053

## [5] Train-accuracy=0.302692309308511

## [6] Train-accuracy=0.392307691849195
```

```
## [7] Train-accuracy=0.438846149123632
## [8] Train-accuracy=0.463076927340948
## [9] Train-accuracy=0.498461534197514
## [10] Train-accuracy=0.578461536994347
## [11] Train-accuracy=0.593846149169482
## [12] Train-accuracy=0.597307691207299
## [13] Train-accuracy=0.616923073163399
## [14] Train-accuracy=0.630384617126905
## [15] Train-accuracy=0.638846142933919
## [16] Train-accuracy=0.670384615659714
## [17] Train-accuracy=0.688461549006976
## [18] Train-accuracy=0.703461543871806
## [19] Train-accuracy=0.723461538553238
## [20] Train-accuracy=0.743076922801825
## [21] Train-accuracy=0.747307697167763
## [22] Train-accuracy=0.756153844870054
## [23] Train-accuracy=0.753846150178176
## [24] Train-accuracy=0.767692304574526
## [25] Train-accuracy=0.77692307646458
## [26] Train-accuracy=0.776538461446762
## [27] Train-accuracy=0.784615381405904
## [28] Train-accuracy=0.788076925736207
## [29] Train-accuracy=0.782692301731843
## [30] Train-accuracy=0.769615386541073
## [31] Train-accuracy=0.7842307640956
## [32] Train-accuracy=0.783846149077782
## [33] Train-accuracy=0.778846146968695
## [34] Train-accuracy=0.79153845172662
```

```
## [35] Train-accuracy=0.804615378379822

## [36] Train-accuracy=0.807692307692308

## [37] Train-accuracy=0.802307692857889

## [38] Train-accuracy=0.817692309617996

## [39] Train-accuracy=0.821923079398962

## [40] Train-accuracy=0.836153846520644

## [41] Train-accuracy=0.844615381497603

## [42] Train-accuracy=0.852692296871772

## [43] Train-accuracy=0.854615383423292

## [44] Train-accuracy=0.851923080591055

## [45] Train-accuracy=0.861153845603649

## [46] Train-accuracy=0.869615382873095

## [47] Train-accuracy=0.871923077564973

## [48] Train-accuracy=0.876153849638425

## [49] Train-accuracy=0.880769229852236

## [50] Train-accuracy=0.883846156872236
```

```
y = predict(model, test_x)
```

```
## Warning in mx.model.select.layout.predict(X, model): Auto detect lay
out input matrix, use colmajor..
```

```
prediction = integer(0)
for(i in 1: ncol(y)){
  column = y[,i]
  prediction = c(prediction,which.max(column)-1)
}
```

```
#prediction accuracy in terms of per character
mean(test_y == prediction)
```

```
## [1] 0.7638462
```

```
#prediction accuracy for the whole captcha
accuracy = integer(0)
for(i in 1:(length(prediction)/5)){
  for(j in 1:5){
    idx = (i-1)*5 + j
```

```
    if(prediction[idx] != test_y[idx]){
      accuracy = c(accuracy,0)
      break;
    }
    accuracy = c(accuracy,1)
  }
}

mean(accuracy)

## [1] 0.8122172

#assert y.shape == (4000, 26)
#score = mod.score(val_iter, ['acc'])
#print("Accuracy score is %f" % (score[0][1]))
```

Discussion

In terms of removing noise in the captcha image, we used morphology as our method. The method perform well for the first two characters, but because the last three characters are more compact than the other two, some important pixels are removed. For instance, number "6" in the fourth position are more like "0". In order to deal with such problem, we proposed an idea of shortest path to remove the horizontal line. In future analysis, we may consider importing or writing a graph class, and change all black pixels into graph nodes so that we can traverse the graph from left to right by using breadth first search. By using such approach, prediction accuracy may boost up for the last two to three characters in captcha images.

When we compare the simple KNN and KNN with credibility score, we expected the model with credibility score to have a higher prediction accuracy because we are trading bias with variance. Since the credibility score can be treated as tuning parameters and provides more information, it increases the flexibility of the model. The bias should be lower and variance should increase. We expected the prediction accuracy to increase but we are getting the opposite result. One likely possibility is that the credibility score is based on our subjective analysis, so we have not tuned parameters properly. One possible solution is to do a cross validation on the training set and get the best credibility score.

Comparing KNN with CNN, CNN performs better than KNN. The reason is that KNN does not extract features of characters while CNN does. KNN is basically a naive pixel to pixel comparison and subject to varies constraints. For example, when the size of the character shrinks or the character rotates, KNN is very likely to misclassify the same character. However, CNN detects features and store them as nodes. Say when CNN sees a character of 9, it might understand the graph like our human brain does, a line attached to a circle at the bottom. Therefore, CNN learns more "deeply" and hence could improve the prediction accuracy.

Conclusion

This report provided a process of segmenting captcha images consists of characters and numbers and used 2 models to read and predict the 5 characters in the captcha image. During image segmentation, the noises are first removed through image erosion and the pixels are changed to binary format. Then constrained k-mean is performed to cluster the image into 5 characters. From this process, 1 single captcha observation is divided into 5 distinct observations.

Then we tried 2 models to train and predict the captchas. The first one is KNN. We have tried both the simple KNN and KNN with credibility score. Simple KNN seems to perform better than KNN with credibility score because we did not tune our credibility score. KNN gives a prediction accuracy of xxx in reading the captcha.

Lastly, CNN is used for prediction. CNN performed sigfinicantly better than KNN with a 7.9% of increase in prediction accuracy. This is because CNN extracts features while KNN does not.