Artists Report

vvorre

5/11/2020

Contents

1 Overview and Dataset

1.1 Introduction

Image classification is one of the most widely used tasks in image processing. The objective is to label/classify a new image using an algorithm that learns the features of labels/classes from given input images. Some examples are hand-written letter recognition in postal carriers, facial recognition in law enforcement, and object detection in image search engines. Machine learning is a commonly used approach to solve image classification problems. Neural networks, specifically convolution neural networks (CNNs) have increased the accuracy of classifying images with advances in computing power and deeper network architectures.

Artist identification is a difficult problem in image classification. Some of challenges that make deciphering an artist from a painting includes: changes in a particular artist's style over time, similar styles among artists who have influenced each other, and that diversity of artist pieces that may include both black an white sketches as well as color paintings.

In this report, we use machine learning techniques to guess the artist from a given painting. We use the paintings of 10 popular artists scraped from WikiArt to train various models and report their validation accuracies. We observe that using a 18 layer CNN Resnet18 with tranfer learning achieves a 84.8% validation accuracy. Next, we extract 512 features from the images using the CNN and train them using other machine learning models. We achieve a validation accuracy of 83.4%, 85.4%, 86.2%, 83.6%, 83.4% for Naive Bayes, Logistic Regression, Support Vector Machines, Random forest, and XGBoost, respectively. The ensemble of all the methods results in an accuracy of 86%. Finally, we apply the best model, the ensemble, on the test set and observe an accuracy of 83.6%. The report demonstrates that combinations of various machine learning models could lead to better models. Similar approaches could be used with larger sets of artists as well as image classification problems where the feature vectors have large variations.

1.2 Goal of the project

The dataset consists of 4000 paintings from 10 popular artists where each artist has 400 paintings. We divide the dataset into three parts: train (300), validation (50) and test (50), in a ratio of 6:1:1. Each painting is labelled with the artist name. The train dataset is used to build various machine learning models whereas the validation dataset is used to compare the performance of the models. We use accuracy, the percentage of correct identification of the artist given a painting, as a metric to benchmark the model. The objective is to build a model that maximizes the accuracy of the validation dataset.

1.3 Scraping data from WikiArt

We use the code given in Appendix 5.1 to scrape paintings from 10 popular artists in the last 30 days who have at least 600 paintings.

1.4 Preprocessing and Data exploration

We look at the data frame consisting of information about the downloaded data

```
# Read the CSV file containing information about artists and the images
art_info_df <- read.csv2('data/artists.csv',sep=',',stringsAsFactors = FALSE)

# Glimpse at the data
glimpse(art_info_df)
```

```
Observations: 10
Variables: 4
$ id                <chr> "pablo-picasso", "claude-monet", "salvador-d...
$ name              <chr> "Pablo Picasso", "Claude Monet", "Salvador D...
$ numberofpaintings <int> 1160, 1369, 1162, 1931, 708, 1008, 1404, 759...
$ url               <chr> "https://www.wikiart.org/en/pablo-picasso", ...
```

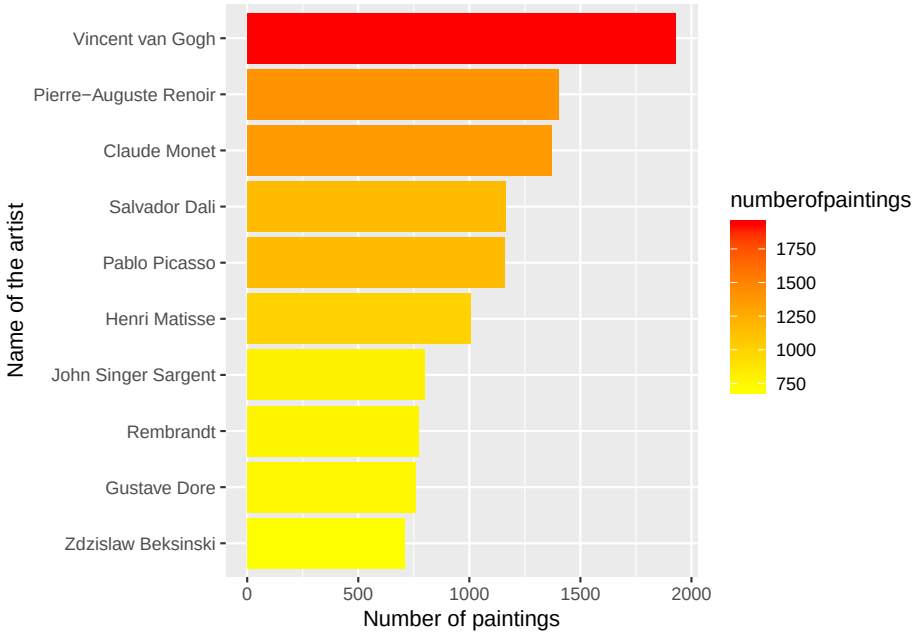

Figure 1: Distribution of Paintings per artist

The art_info_df data frame consists of 4 variables:

id - Unique ID given to an artist

name - Name of the artist

numberofpaintings - number of paintings downloaded for each artist

url - url of the artist in WikiArt

The artists in the set are:

```
[1] "Pablo Picasso"         "Claude Monet"
[3] "Salvador Dali"         "Vincent van Gogh"
[5] "Zdzislaw Beksinski"    "Henri Matisse"
[7] "Pierre-Auguste Renoir" "Gustave Dore"
[9] "John Singer Sargent"   "Rembrandt"
```

We look at the distribution of paintings per artist in Figure 1 and read some samples from Picasso in Figure 2.

We observe that there is an uneven distribution of paintings per artist, which would lead to bias in models with a larger number of paintings. There exists many methods to solve this problem such as downsampling, data augmentation or bootstrapping. Here, we choose downsampling as it introduces less bias, although we might loose some information from the removal of paintings in the dataset. We randomly choose 400 samples from each artist to form a new dataset.

Another issue we observe is that the images are of varying sizes and have to be resized to work with machine learning algorithms. Resizing all the images to the same dimension would change the aspect ratio. We assume that the style of the artist lies in the colors used, the geometries of the shapes and the brush strokes. To avoid changing the aspect ratio, we crop a square box from the center of the images of dimensions equal to the smallest dimension of the image. For example a crop of 168 x 168 would be applied to a 168 x 200 image from its center. Next, we resize all the images to 224 x 224 and preserve the aspect ratio. The dimension of 224 x 224 is chosen as the CNN Resnet18 used to extract features takes this dimension of input images as

Figure 2: Some sample images from Picasso

default. The above method also results in information loss from a painting. Other methods such as random repeated crop could be applied, but may introduce some bias for larger paintings.

Next, we look at a few paintings after we processed the paintings in Figures 3 and 4.

The processed data has a total of 3000, 500 and 500 images of dimensions 224 x 224 in the train, validation, and test directories. Each of these directories have folders corresponding to 10 artists where the images are stored. The directory structure for the images is given below

```
                levelName
1  data
2   °--procdata
3       ¦--train
4       ¦   °--artist
5       ¦       °--images
6       ¦--validation
7       ¦   °--artist
8       ¦       °--images
9       °--test
10          °--artist
11              °--images
```

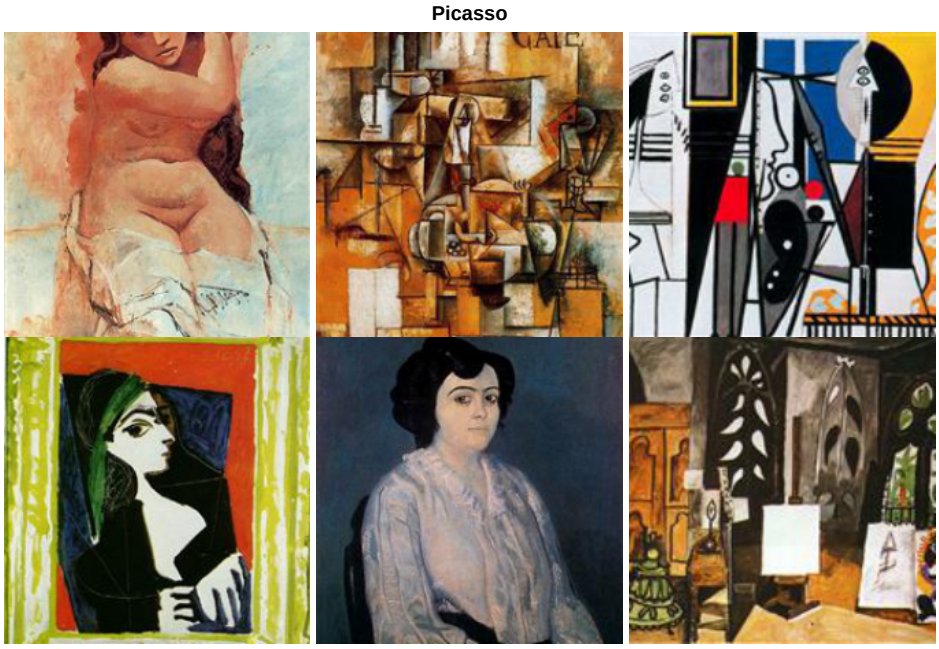

Figure 3: Some processed images from Picasso

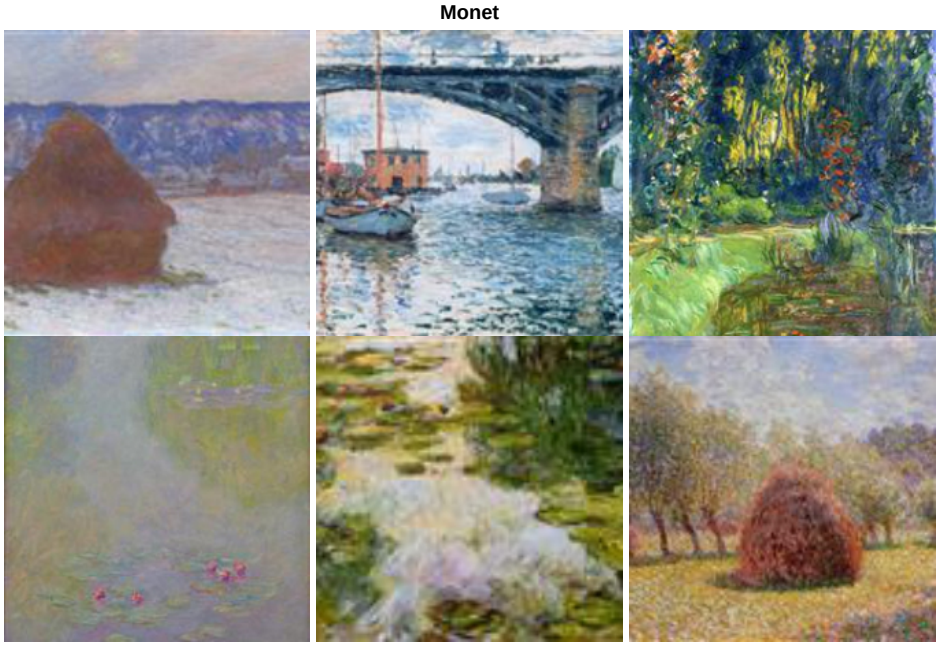

Figure 4: Some processed images from Monet

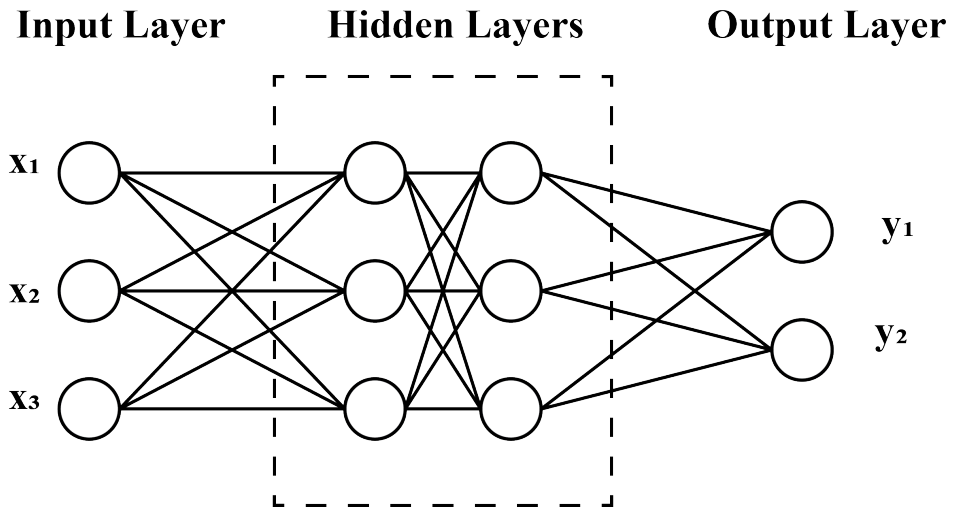

Figure 5: Neural network

2 Methods and Analysis

2.1 Convolution neural networks (CNNs)

Neural networks are a popular machine learning technique inspired by the transmission of neurons in the human brain. A layer in a neural network consists of nodes similar to neurons. The nodes between the layers are connected through weights and the node outputs are computed through a non-linear function of the inputs. A typical neural network consists of an input layer, output layer and a few hidden layers as shown in Figure 5. The nodes in the input layer corresponds to input samples whereas the output layer corresponds the number of classes to be classified. The neural network learns the weights of the model as more training data is provided. The predictions are made based on the maximum probability of the output class given an input dataset. Neural networks perform well where complex non-linear relations exists in data since each layer adds complexity.

Convolution neural networks (CNNs) are a special class of neural networks which are designed to extract features from an image format. In contrast to standard neural networks, CNNs can easily extract spatial features in the images. Convolution networks require little preprocessing as the features of the images are extracted on the fly compared to other image classification algorithms which rely on designing custom filters.

The CNNs extract the features in the image through convolution of kernels on the images. The convolution of kernels is described in Figure 6. The kernels in this example are of size 2x2 where as the input image is of size 4x4x3 consisting of red, blue and green channels. The kernel slides over each component (R,G,B) of the input image performing an inner product to produce a 3x3 feature vector. By combining four different kernels, one could transform from 4x4x3 input layer to 3x3x4 feature layer. Essentially, we have extracted important features using convolution such as edges or colors in the image.

A typical CNN consists of convolution layers, pooling layers, and fully connected layers as shown in Figure 7. The convolution layers extracts the localized information whereas pooling layers reduce the feature space and also prevent over-fitting. The fully connected layers connects the features extracted to a neural network where the output layer contains the probability score of each label in the classification problem. Using this architecture, CNNs can reduce the feature space effectively for images and produce good predictions for classification problems.

2.2 Resnet18

Deeper CNNs have an underlying problem that increasing layers makes them harder to train due to a vanishing gradient. The back-propagation algorithm used to calculate the weights in the model, can reduce

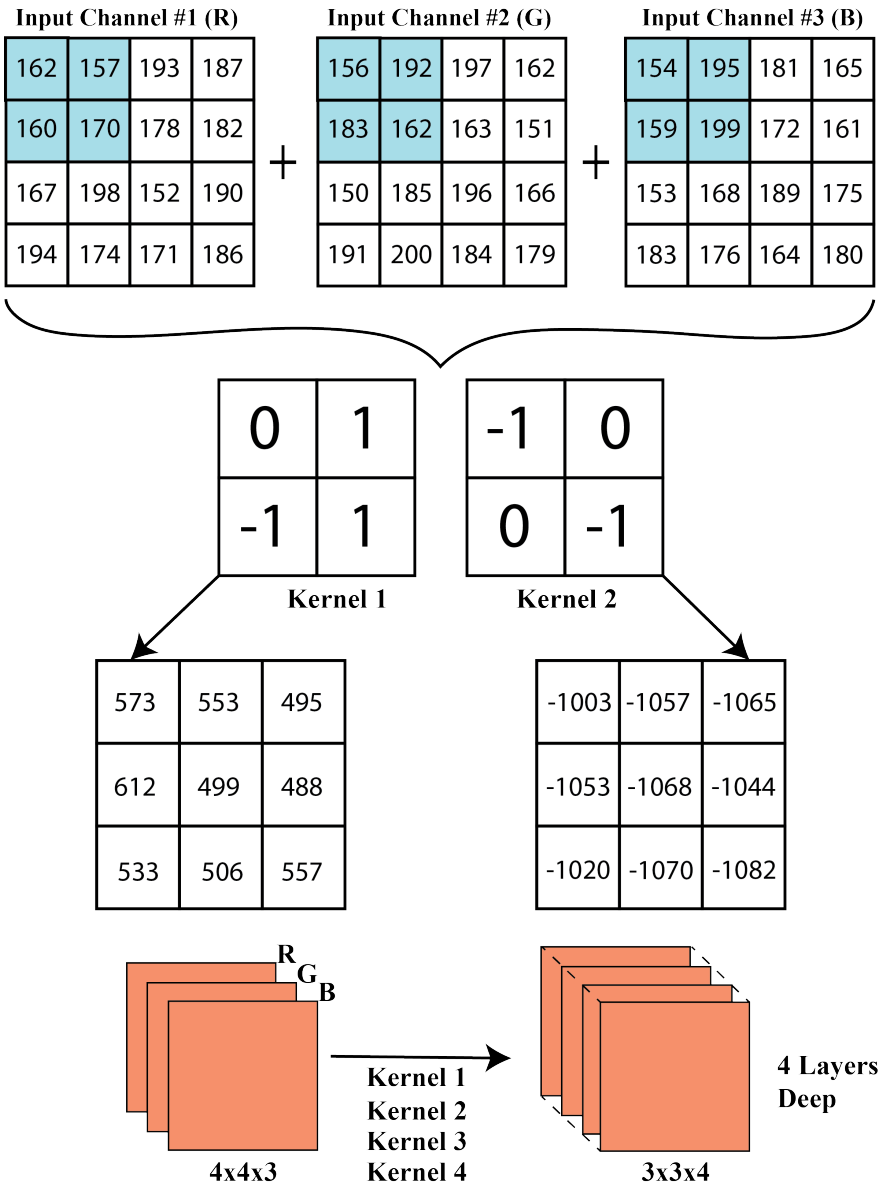

Figure 6: Convolution of kernel in a CNN.

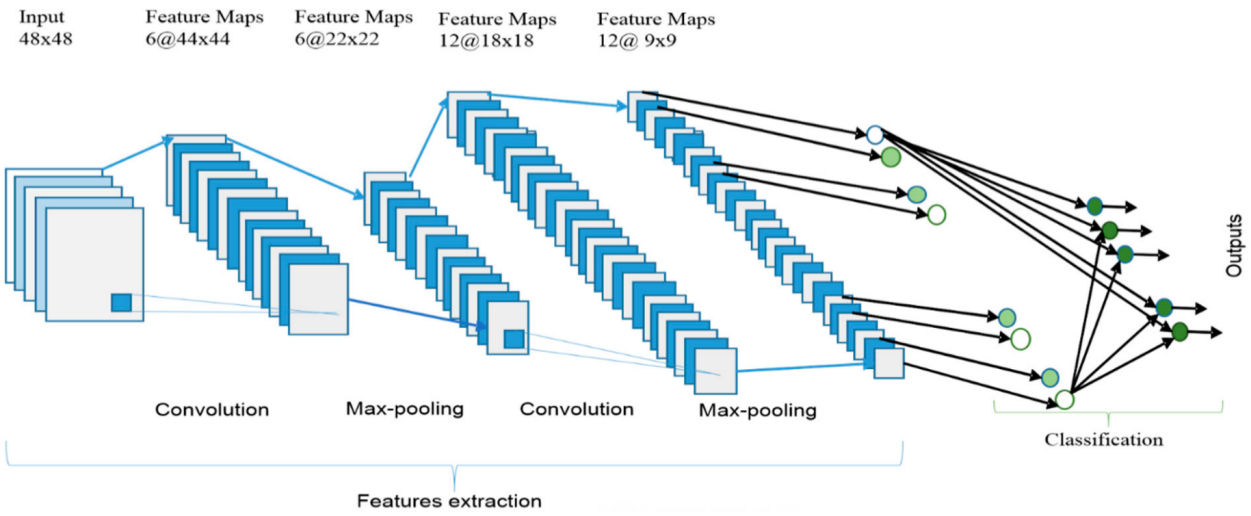

Figure 7: Convolution Neural network. Source: https://doi.org/10.3390/electronics8030292

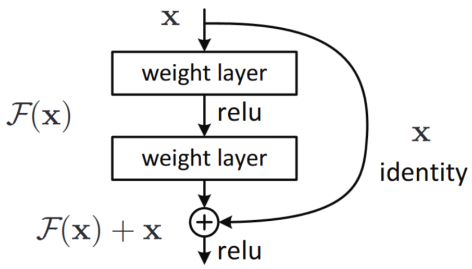

Figure 8: Building block of Resnet architecture Source: https://arxiv.org/abs/1512.03385

the gradients close to zero as the layers increase and saturates the model. The Resnet architecture was proposed to solve the vanishing gradient by skipping one or more layers. The basic building block of the Resnet architecture is given in Figure 8 where gradients can flow through identity connections. This allows the model to perform better for deeper networks.

Resnet18 is an 18 layer CNN with Resnet architecture which has about 11.5 million parameters. We use Resnet18 architecture in our model due to its smaller size and faster processing. The networks takes an input size of 224x224x3 and produces an 7x7x512 feature layer before the fully connected layer.

2.3 Transfer learning using Resnet18

Transfer learning is a method of using previously learned knowledge from a problem to solve a new problem. The original Resnet18 was created to solve a popular image classification problem called ImageNet challenge. ImageNet consisted of thousands of images in the training set whose objective was to classify objects belonging to 1000 categories. The pre-trained weights of ImageNet is optimized to extract most of the relevant features in any given image. This implies that pre-trained weights of ImageNet would also extract most of the relevant features in our artist classification problem. The idea of transfer learning is to use the pre-trained weights from the ImageNet as a starting point to solve our classification algorithm. The implementation for the algorithm is given below.

1. Start with the base model of Resnet18 whose weights are pre-trained ImageNet values. The base model excludes the top layer and produces a feature vector of 7x7x512
2. On top of the base model, add a 2D global averaging pooling layer to flatten it to a 512 node layer. Next, add a 10 node output layer with a softmax (probability mapping) activation corresponding to the 10 artists.
3. Freeze the weights of the base model and train the model to find the weights of the last layer of the network.
4. Unfreeze the weights of the base model and fine tune all the layers at a reduced learning rate.

We add a couple of preprocessing steps such as featurewise centering and featurewise standard normalization on all the images. We also use data augmentation on the training samples to reduce the overfitting and increase the sample size. Specifically, we use a horizontal flip and random zoom between 80% and 120% on the training set while training the model. The code implementation of transfer learning is given in Appendix 5.2.

2.4 Feature extraction

The features of the images are extracted after the global averaging 2D layer in the CNN. This produces a 512 feature vector for each image. Using the feature extraction we build training (train_x), validation (valid_x) and test (test_x) set of dimension 3000 x 512 , 500 x 512 and 500 x 512 respectively. We also create the corresponding labels in train_y, valid_y and test_y. Code to implement feature extraction in given in 5.3.

```
dim(train_x)
```

```
[1] 3000  512
```

```
dim(valid_x)
```

```
[1] 500 512
```

```
dim(test_x)
```

```
[1] 500 512
```

```
dim(train_y)
```

```
[1] 3000    1
```

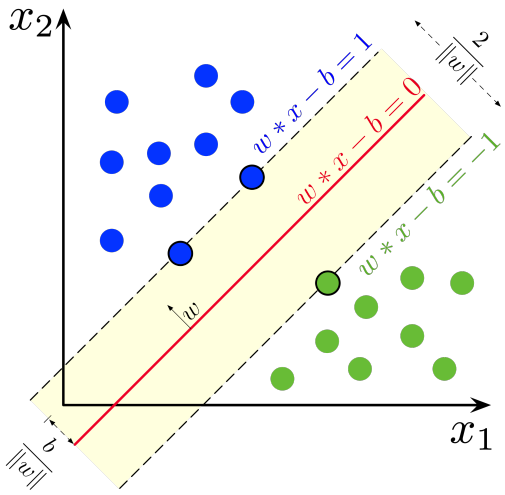

Figure 9: Linear SVM classifier. Source: Wikipedia

```
dim(valid_y)
```

```
[1] 500   1
```

```
dim(test_y)
```

```
[1] 500   1
```

We use the extracted features to train other machine learning models.

2.5 Naive Bayes

Naive Bayes is an approach which assumes that all the features in the data are independent of each other and calculates the probabilities. The model predicts using the following equation.

$$\hat{y} = argmax_k \left(p(C_k) \prod_n p(x_i|C_k) \right)$$

where C_k, x_i are the labels and features respectively. The approximation assumes $p(C_k|x_1, x_2, .., x_n) = p(C_k) \prod_n p(x_i|C_k)$ which is only true when the features are independent. We use the naive bayes packages to implement this model.

2.6 Logistic regression

Logistic regression is machine learning technique that assumes that there is a linear relationship between the features and the log-odds of an event. The model for a binary outcome is given by $log(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ...$ where, p is the probability that the event occurs. It can be extended to multinomial classification where the result is arrived at using multiple independent binary logistic regressions. We use the glmnet package to implement the multinomial logistic regression.

2.7 Support vector machines

Support vector machines (SVM) use hyperplanes to maximally separate classes. For example, the hyperplane separating the two classes (green and blue) is given by red line as shown in Figure 9. The SVM algorithm finds the hyperplane that maximizes the margin between the two classes shown in dotted lines. SVM uses a kernel trick to classify non-linear boundaries. The feature space is mapped to a higher dimension where a linear hyperplane could be used to separate the classes. The higher dimensional feature space

is then mapped back to the the original feature space. One popular kernel is a radial/Gaussian function $K(x_i, x_j) = exp(-\gamma||x_i - x_j||^2)$ for $\gamma > 0$. We use the kernlab package to implement SVM with a radial kernel.

2.8 Random forest

Random forest is an ensemble of decision trees where different models are fit to bootstrapped resamples of data. Another variation from decision trees is that the algorithm samples the number of variable to be included at each split of tree. The algorithm is given below.

1. Bootstrap (with replacement) a subsample from number of input samples.
2. At each split of the tree, sample n features without replacement from the features.
3. Build the decision tree T_m.
4. Repeat 1 to 3 for M trees.

The final prediction is given by $\hat{y} = (1/M)\sum_m \hat{T}_m$. We use the random forest package in our implementation.

2.9 Extreme gradient boosting

Boosting is another ensemble model where a series of decision tree models are fit to data. However, it differs from random forest in such that the successive models are built to minimize the error of previous models. The algorithm works as follows.

1. Choose M, the total number of models to be fit and T_1,T_2,...T_M to be the fitted models. Initialize the weights of samples, $w_i = 1/N$ where i = 1,2..N and N is the number of samples.

2. Train a model T_m using weights w_i that minimizes the weighted error e_m. e_m is the sum of the weights for misclassified observations for model m.
3. Update the weights w_i such that misclassified weights have larger values.
4. Go to next model in the iteration.

The final prediction is given by $\hat{y} = \sum_m \alpha_m \hat{T}_m$ where $\alpha_m = \frac{log(1-e_m)}{e_m}$

A commonly used boosting method is stochastic gradient boosting and is implemented in XGBoost package, which we use in our analysis.

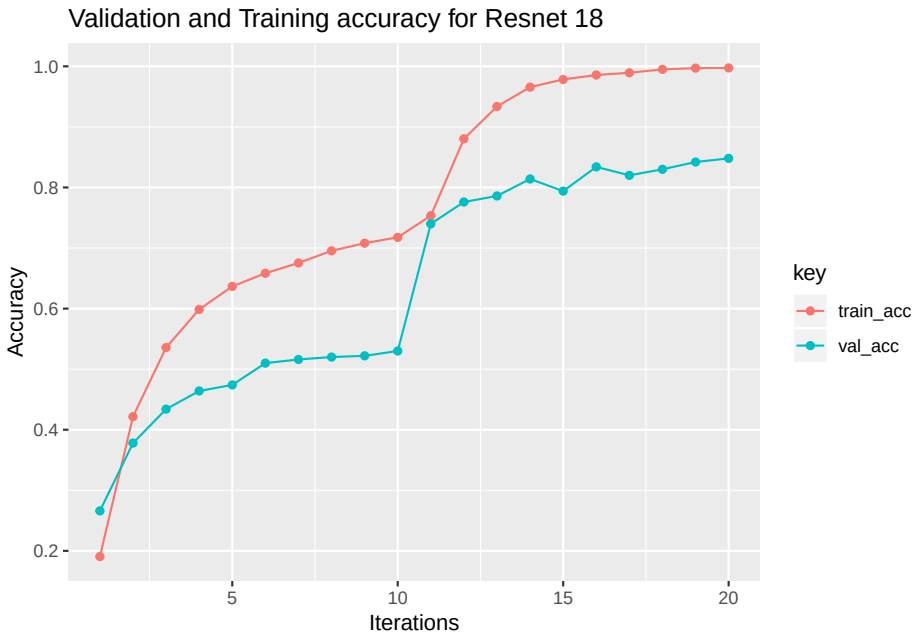

Figure 10: Accuracy vs iterations in Resnet18

3 Results

We present the results for each model and its corresponding validation accuracy. For training the models, we use the method of 10-fold cross-validation repeated 3 times. We optimize the parameters for each model and calculate the validation accuracy for the best model. The 512 extracted features from the images are used to train all the models.

3.1 Accuracy of Validation dataset

3.1.1 Model 1: Resnet18 using transfer learning

As described in the methods section, we initially train the CNN by the freezing the base layer until 10 iterations. We use an adam optimizer with learning rate = 1e-3, $\beta_1 = 0.9$, $\beta_2 = 0.999$ to train the model. After 10 iterations, we unfreeze all the layers and fine tune the model at a low learning rate of 1e-4. The plot of training and validation accuracy and losses are given in Figure 10.

The Resnet18 CNN trained using transfer learning gives a validation accuracy of 84.8%.

3.1.2 Model 2: Naive Bayes

In model 2, we use Naive Bayes to train the network using the naivebayes package.

We obtain a validation accuracy of 83.4%.

3.1.3 Model 3: Logistic Regression

In model 3, we use Logistic Regression to train the network using the glmnet package. The *alpha* corresponds to the elastic-net mixing parameter and when *alpha* = 1, the regression is lasso i.e. the regularization term is on the squares of the features. The regularization parameter *lambda* is tuned from 5e-5 to 5e-3 to find the optimal model.

We find the best *lambda* to be 5e-4 and obtain the corresponding validation accuracy to be 85.4%.

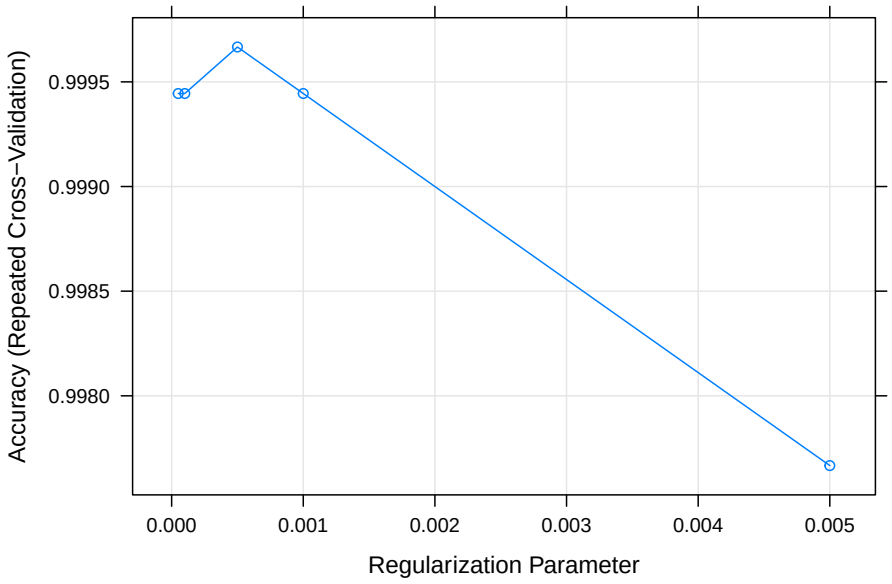

Figure 11: Accuracy vs lambda in logistic regression

3.1.4 Model 4: Support Vector Machines

In model 4, we use support vector machines with Gaussian kernel to train the network using the kernlab package. We use the inbuilt parameter selection to tune over 10 automatically selected values. The selection fixes *sigma* (the inverse of standard deviation of the Gaussian kernel) and tunes regularization parameter *C*.

We find the best *C* and *sigma* to be 2 and 0.001039198 respectively. The validation accuracy is measured to be 86.2%.

3.1.5 Model 5: Random forests

In model 5, we use random forests to train the network using the randomForest package. We tune the *mtry* parameter which is the number of variables randomly sampled at each split from 11 to 35. We also fix the *ntree* parameter to 1000, which are number of trees to grow.

We find the best *mtry* to be 11 and obtain the corresponding validation accuracy to be 83.6%.

3.1.6 Model 6: Extreme gradient boosting

In model 6, we use Extreme gradient boosting (XGBoost) to train the network using the xgboost package. The xgboost have more parameters and we list them here:

eta - Step size shrinkage

gamma - regularization parameter

colsample_bytree - fraction of columns to be sampled for every tree constructed

nrounds - number of iterations

subsample - Subsampling of training samples at each iteration

max_depth - maximum depth of a tree

min_child_weight - minimum sum of instance weights

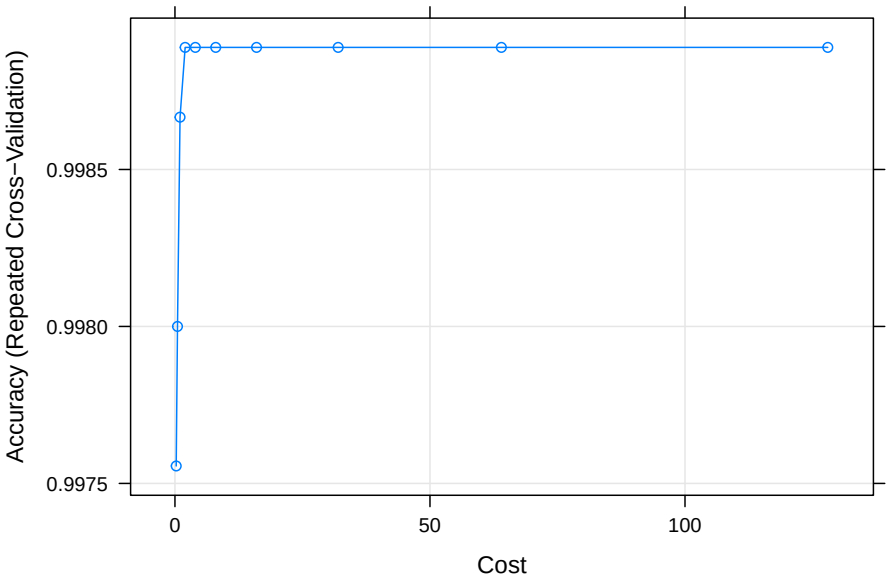

Figure 12: Accuracy vs C in support vector machine

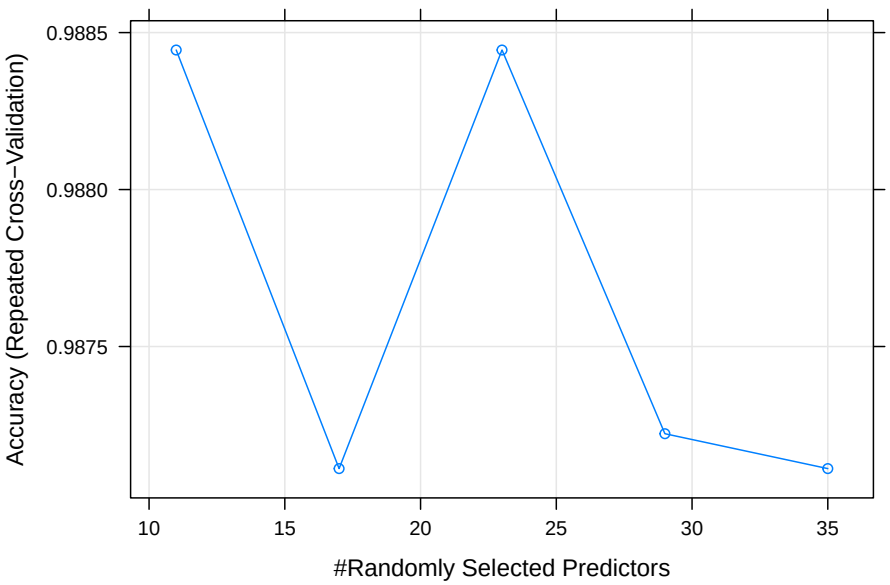

Figure 13: Accuracy vs mtry in random forest

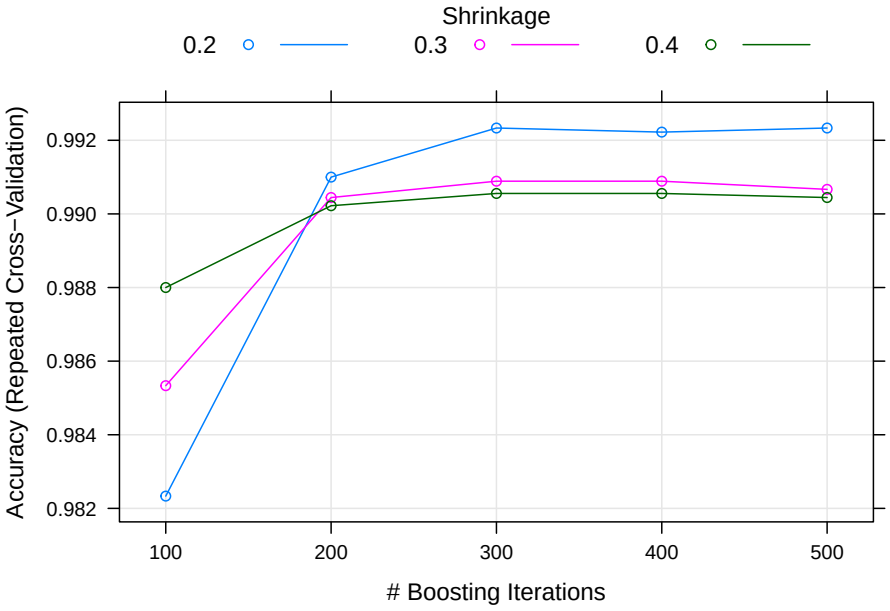

Figure 14: Accuracy vs eta vs gamma in Xgboost

We tune the *eta* between .2, .3, .4 and *nrounds* to vary from 100 to 500 in our model. The parameters *gamma*, *colsample_bytree*, *max_depth*, *min_child_weight* and *subsample* are fixed to be 0, .8, 1, 1, and .8 respectively.

We find the best *eta* and *nrounds* to be .2 and 300 respectively. The corresponding validation accuracy is measured to be 83.4%.

3.1.7 Model 7: Ensemble

Next, we use all the above methods to create an ensemble model. For a given row in the data, the ensemble predicts the class that occurs the most in the models. In case of a tie, we choose the first class in the list. The validation accuracy of the ensemble is 86%. The ensemble produces a more resilient model and improves the limitations in individual models.

We report the validation accuracy of all the models below in a table.

method	accuracy
CNN Resnet 18	0.848
Naive Bayes	0.834
Logistic regression	0.854
SVM with Radial	0.862
Random Forest	0.836
Extreme gradient boosting	0.834
Ensemble	0.860

3.2 Accuracy of Test dataset

We use the ensemble model on the test set and obtain an accuracy of 83.6%.

method	accuracy
Ensemble Test Accuracy	0.836

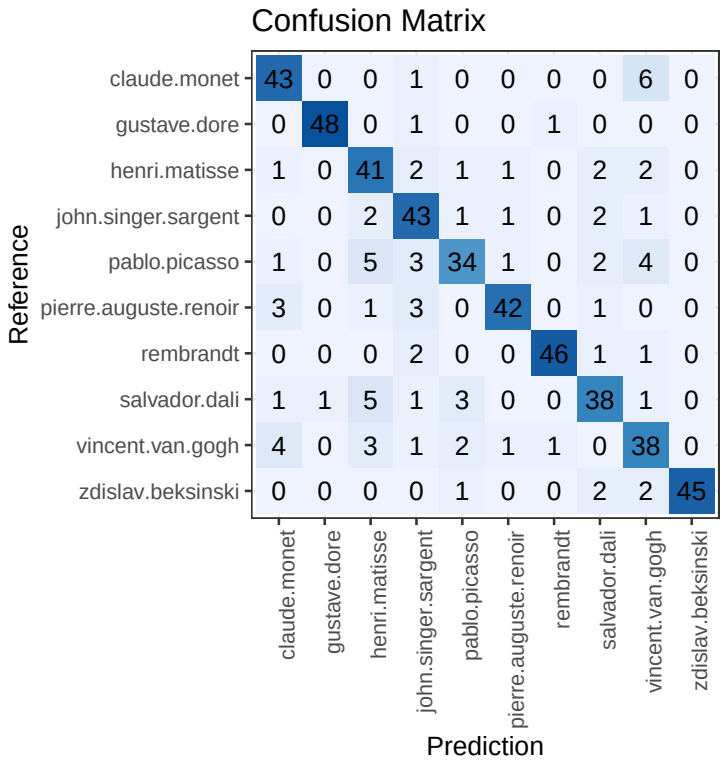

Figure 15: Confusion matrix of test set

3.3 Confusion Matrix and Discussion

We plot the confusion matrix on the test dataset to find the incorrect classifications of individual artists.

Next, we look at some misclassification of images in the test set.

We observe that the model performed poorly on the artists who have multiple styles. For example, Picasso's work is classified into multiple periods where the artist changes the style. This leads to lower number of training samples per style and hence model could not train well. Another set of misclassification results are due to influences of styles between artists. Van gogh and Monet were both impressionists and used landscapes and similar colors in their paintings. Many of these errors could be improved by using higher resolution paintings and larger training samples.

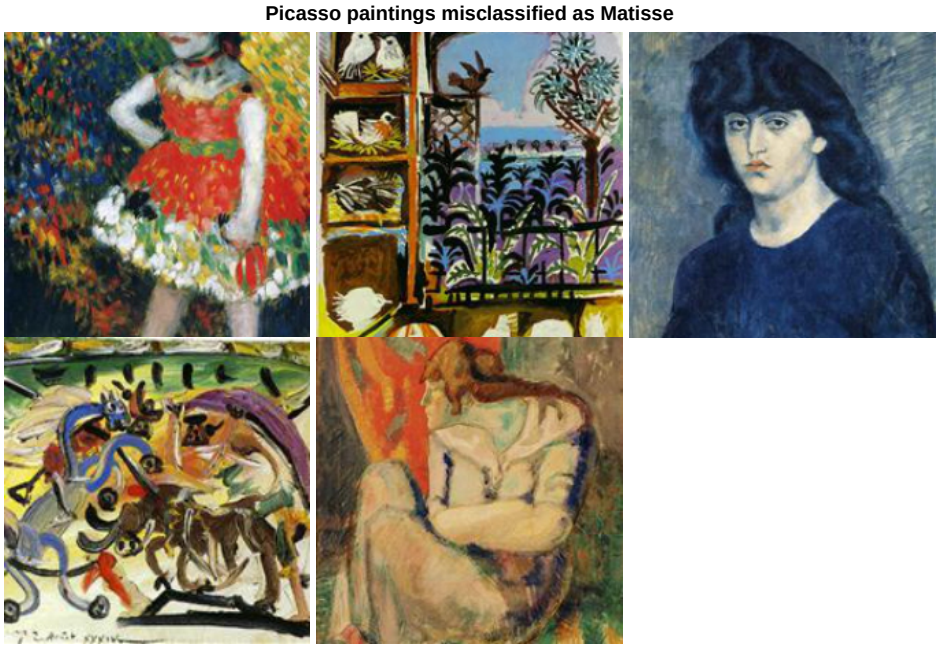

Figure 16: Picasso paintings misclassified as Matisse

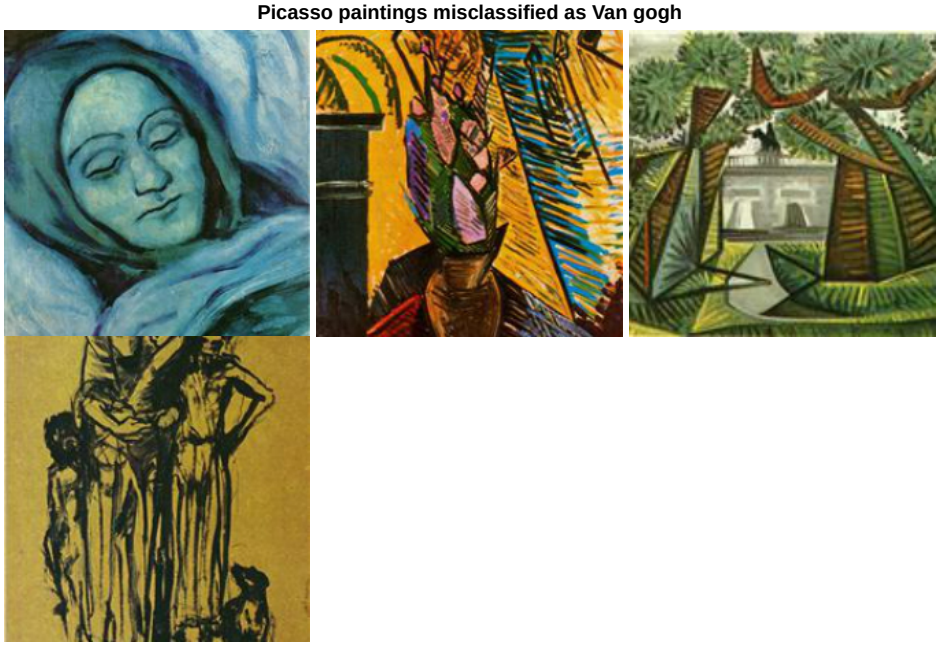

Figure 17: Picasso paintings misclassified as Van gogh

Figure 18: Van gogh paintings misclassified as Monet

4 Conclusion

In this report, we have trained several models to predict the artist from a given painting. We observed that some models built on the extracted features from the CNN Resnet18 performed superior to just using the neural network. Using an ensemble approach produces a better model and resulted in an accuracy of 83.6% on the test set. We also observed that some incorrect classifications are due to the style similarity between artists. Here, we only used one CNN Resnet18 due to its small size and fast operations. However, there are other advanced CNNs such as Inception, Alex, VGGNet as discussed in this article. Fine tuning these advanced architectures or building a custom CNN could also improve the accuracy. We could also use the data augmentation and boot strapping techniques to include more data and make use of the imbalanced classes. Lastly, higher resolutions of paintings could be used in cropped format to get the finer features which might also improve the model. We conclude the report on the note that combining several machine learning approaches can result in a better model.

5 Appendix

5.1 A: Code to scrape the paintings

We use the following code to scrap paintings from Wikiart from 10 popular artists in the last 30 days who have at least 600 paintings.

```
################################
# Scrap artists
################################

# JSON file for popular artists in the last 30 days
json_file <- "https://www.wikiart.org/en/App/Search/popular-artists?json=3&amp;layout=new"

json_data <- fromJSON(file=json_file)

# Read the html page of the 60 popular artists
h <- read_html(json_data$ArtistsHtml)

# Number of artists to scrape
Nartists <- 10

# Contains the number of paintings and the name of the artist
temp_list <- h %>% html_nodes('a') %>% html_attr('title')

# Get the urls of the artists
urls  <- h %>% html_nodes('a') %>% html_attr('href') %>% .[seq(1,length(temp_list),3)]

# Make name-id to be the unique identifier
id <- substr(urls,5,nchar(urls))

# Add the prefix to complete the url
urls <- paste0('https://www.wikiart.org',urls)

# Get the artist name
name = temp_list[seq(1,length(temp_list),3)]

# Get the number of paintings
numberofpaintings <- temp_list[seq(2,length(temp_list),3)] %>%
  gsub("(\\d*).*","\\1",.) %>% as.numeric()

# Create a data frame to write the information collected in a csv file
csv_df <- data.frame(id = id, name = name,
                     numberofpaintings = numberofpaintings,
                     url = urls,stringsAsFactors = FALSE)

# Select the first 10 artists with more than 600 paintings
csv_df  <- csv_df %>% filter(numberofpaintings > 600) %>% .[1:Nartists,]

# Clean up
rm(temp_list, numberofpaintings, name, urls,id)

# Create a directory data to store the csv file and the images
if(!dir.exists('data')) dir.create('data')
```

```r
# Create a sub-directory data to store images
if(!dir.exists('data/rawdata')) dir.create('data/rawdata')

# Write to a csv file
write.csv(csv_df,file = paste0('data/artists','.csv'),row.names = FALSE)

# Get images for each artist
for (i in 1:nrow(csv_df)){

  # url list for paintings
  url <- paste0(csv_df$url[i],'/all-works/text-list')
  # read the html
  h <- read_html(url)

  # Node containing url for the paintings
  url_node_data <- h %>% html_nodes('a') %>% html_attr('href')

  # Create a directory with artist name
  if(!dir.exists(paste0('data/rawdata/',csv_df$id[i])))
    dir.create(paste0('data/rawdata/',csv_df$id[i]))

  # Loop through all the painting pages
  for (j in 1:csv_df$numberofpaintings[i]){

    # Url for each painting
    url1 <-  paste0('https://www.wikiart.org',url_node_data[41+j])
    # Read the html page
    h1 <- read_html(url1)

    # Image Url
    image_link <- h1 %>% html_nodes('img.ms-zoom-cursor') %>% html_attr('src')

    # Get the smaller version of the URL
    temp_link <- unlist(str_split(image_link,"!", n= 2))[1]

    # Links have three formats, jpg, jpeg and .png, Store depending on the format
    if(str_detect(temp_link,regex('.Jpg', ignore_case = T))){
      image_link <- paste0(temp_link,'!PinterestSmall.jpg')
    } else if(str_detect(temp_link,regex('.Jpeg', ignore_case = T))){
      image_link <- paste0(temp_link,'!PinterestSmall.jpeg')
    } else(str_detect(temp_link,regex('.png', ignore_case = T))){
      image_link <- paste0(temp_link,'!PinterestSmall.png')
    }
    # Download the url
    if(str_detect(temp_link,regex('.Jpg', ignore_case = T)) |
       str_detect(temp_link,regex('.Jpeg', ignore_case = T))){
      download.file(image_link,paste0('data/rawdata/',csv_df$id[i],'/',
                                      csv_df$id[i],'_',j,'.jpg'), mode = 'wb',
                    method = 'curl')
    } else{
      download.file(image_link,paste0('data/rawdata/',csv_df$id[i],'/',
                                      csv_df$id[i],'_',j,'.png'), mode = 'wb',
                    method = 'curl')
```

```
    }
    # Wait for .25 sec between loops to reduce error between consecutive requests
    Sys.sleep(.25)
  }
}
```

5.2 B: Code to train the CNN

```
# Image dimensions and batchsize to be used for processing cnn
img_width <- 224
img_height <- 224
batchsize <- 100

# Paths to directory
train_dir <- "./data/procdata/train/"
validation_dir <- "./data/procdata/validation/"
test_dir <- "./data/procdata/test/"

# Generators for train, test and validation datasets with feature normalization
# Data augmentation using horizontal flip and zoom for training set
train_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE,
  zoom_range = 0.2, # zoom by +_ .2
  horizontal_flip = TRUE, # horizontal flip
  fill_mode = "nearest"
)
validation_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE
)
test_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE
)
# Controls to read images from the train, test and validation directories
train_generator <- flow_images_from_directory(
  train_dir,                  # Target directory
  train_datagen,              # Data generator
  target_size = c(img_width, img_height),  # Size is 224 x 224
  batch_size = batchsize,
  class_mode = "categorical", # categorical labels
  shuffle = TRUE ,           # Shuffle the classes
  seed = 1
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(img_width, img_height),
  batch_size = batchsize,
  shuffle = FALSE,
  class_mode = "categorical"
)
```

```
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(img_width, img_height),
  batch_size = batchsize,
  shuffle = FALSE,
  class_mode = "categorical"
)

# Number of samples in each set
train_samples <- train_generator$n
validation_samples <- validation_generator$n
test_samples <- test_generator$n

## Load Resnet 18- downloaded from https://github.com/qubvel/classification_models
base_model <- load_model_hdf5(filepath = './data/resnet18_224.h5')
# model on top of the base model
model <- keras_model_sequential() %>%
  base_model %>%
  layer_global_average_pooling_2d() %>% # global averaging
  layer_dense(units = 10, activation = "softmax") # 10 layer node

# Freeze weights for all the weights of the model and the global averaging layer
freeze_weights(base_model)
for (layer in base_model$layers)
  layer$trainable <- FALSE
for (layer in model$layers[2])
  layer$trainable <- FALSE

# Compile the model with the adam optimizer
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(lr = 1e-3, beta_1 = 0.9, beta_2 = 0.999),
  metrics = c("accuracy")
)

# Callbacks to the model to store and earstop the model, if val accuraccy doesn't improve
callbacks <- list(
  callback_early_stopping(monitor = "val_acc", min_delta = .005,
                          # Early stopping if the validation accuracy does not improve
                          patience = 3, restore_best_weights = TRUE),
  callback_model_checkpoint(filepath = paste0("./data/","resnet18_coarsetune",
                                              "weights.{epoch:02d}-{val_acc:.3f}.hd5"),
                            # Save weights of every run
                            monitor = "val_acc"))
# Store the history
history_coarse <- model %>% fit_generator(
  train_generator,
  steps_per_epoch = round(train_samples/batchsize),
  epochs = 10,
  validation_data = validation_generator,
  validation_steps = round(validation_samples/batchsize),
  callbacks = callbacks
```

```
)
# Save the history and best model
save(history_coarse,file = './data/resnet18_coarse_hist.Rda')
save_model_hdf5(model,'./data/resnet18_coarse_model.hd5', overwrite = TRUE)

# Fine tuning the model
# Unfreeze all the layers
unfreeze_weights(base_model)
for (layer in base_model$layers)
  layer$trainable <- TRUE
for (layer in model$layers[2])
  layer$trainable <- TRUE

# Callbacks
callbacks <- list(
  callback_early_stopping(monitor = "val_acc", min_delta = .005,
                          # Early stopping if the validation accuracy does not improve
                          patience = 3, restore_best_weights = TRUE),
  callback_model_checkpoint(filepath = paste0("./data/","resnet18_finetune",
                                              "weights.{epoch:02d}-{val_acc:.3f}.hd5"),
                            # Save weights of every run
                            monitor = "val_acc"))
# Compile the model with reduced learning rate
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(lr = 1e-4, beta_1 = 0.9, beta_2 = 0.999),
  metrics = c("accuracy")
)
# Train the model and store the history
history_fine <- model %>% fit_generator(
  train_generator,
  steps_per_epoch = round(train_samples/batchsize),
  epochs = 20,
  validation_data = validation_generator,
  validation_steps = round(validation_samples/batchsize),
  initial_epoch = 10, callbacks = callbacks
)

save(history_fine,file = './data/resnet18_fine_hist.Rda')
save_model_hdf5(model,'./data/resnet18_fine_model.hd5', overwrite = TRUE)
```

5.3 C: Code to implement feature extraction

```
# Extract feature vectors
# Generators for train without any augmentation
train_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE
)
validation_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE
)
```

```
test_datagen <- image_data_generator(
  featurewise_center = TRUE,
  featurewise_std_normalization = TRUE
)
# Controls to read images from the train, test and validation directories
train_generator <- flow_images_from_directory(
  train_dir, # Target directory
  train_datagen, # Data generator with preprocessing
  target_size = c(img_width, img_height), # input image size
  batch_size = batchsize, # batch size
  shuffle = FALSE, # if the data needs to be shuffled
  class_mode = "categorical"
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(img_width, img_height),
  batch_size = batchsize,
  shuffle = FALSE,
  class_mode = "categorical"
)
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(img_width, img_height),
  batch_size = batchsize,
  shuffle = FALSE,
  class_mode = "categorical"
)
# Extract the model with the output layer as the global averaging 512 elements
model_extract <- keras_model(inputs = model$input,
                             outputs = get_layer(model,
                            'global_average_pooling2d_1')$output)

# Extract the features by using the model
test_x <- as.data.frame(predict_generator(model_extract,
                                          test_generator,
                                          steps = round(test_samples/batchsize)))

valid_x <- as.data.frame(predict_generator(model_extract,
                                           validation_generator,
                                           steps = round(validation_samples/batchsize)))

train_x <- as.data.frame(predict_generator(model_extract,
                                           train_generator,
                                           steps = round(train_samples/batchsize)))
# Extract the corresponding labels
test_y <- data.frame(label = factor(test_generator$classes,
                  labels = colnames(data.frame((test_generator$class_indices)))))

valid_y <- data.frame(label = factor(validation_generator$classes,
                  labels = colnames(data.frame((validation_generator$class_indices)))))
```

```
train_y <- data.frame(label = factor(train_generator$classes,
                  labels = colnames(data.frame((train_generator$class_indices)))))

# Save the data to a file
save(train_x,train_y,valid_x,valid_y,test_x,test_y,
     file="./data/preprocesseddata_resnet18.Rda")
```

References

- Irizzary,R. (2020),*Introduction to Data Science*, book, https://rafalab.github.io/dsbook/
- He, K., Zhang, X., Ren, S., & Sun, J. (2016), *Deep residual learning for image recognition*, In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778), https://arxiv.org/abs/1512.03385
- Anwar, A. (2019),*Difference between AlexNet, VGGNet, ResNet and Inception*, blog, https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96
- Boehmke, B., Greenwell, B. (2020), *Hands-On Machine Learning with R* book, https://bradleyboehmke.github.io/HOML/svm.html
- Viswanathan, N. (2017), *Artist identification with convolutional neural networks*, http://cs231n.stanford.edu/reports/2017/pdfs/406.pdf
- Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., ... & Asari, V. K. (2019), *A state-of-the-art survey on deep learning theory and architectures.*, Electronics, 8(3), 292
- WikiArt, webpage, https://www.wikiart.org
- R interface to Keras, webpage, https://keras.rstudio.com
- Kuhn, M. (2019), *The caret Package*, webpage, https://topepo.github.io/caret/index.html
- Hastie, T., Qian, J. (2014), *Glmnet Vignette*, webpage, https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html
- Extreme Gradient Boosting (2020), manual, https://cran.r-project.org/web/packages/xgboost/xgboost.pdf