

Classification of images with K-NN

**2020**

## Code :

## Code in JavaScript for ImageJ

Initially we worked on the core algorithm of the k-NN. The main steps are the below:

1. **Datasets preparation**
   1. We need two sets of data, one for the training and one for the testing, both being associated with their targets, or Labels for each row
2. **Training, or “fit”**
   1. For such algorithm, it’s more about loading the reference Data rather than “training”
3. **Prediction**
   1. At this stage, the testing data are used to calculate the Euclidean Distance with the training Data, row by row, keeping the “k” nearest neighbors
   2. The result is returned as an array of prediction for each row (one row corresponding to one symbol)
4. **Accuracy**
   1. The accuracy of the prediction is calculated by comparing the prediction of the testing Data with their known labels. Dividing the count of right answer by the count of rows gives the percentage of accuracy

As our goal was to run many times the script, using different parameters and changing the Data, we have added a couple of improvements that make the usage more friendly:

* Selection of the source image with a dialogue box
* Selection of the CSV file containing the Labels
* Choice of the ImageJ process to perform
* Noise addition
* …

Also adding user interactions avoids many direct changes into the code for each run, for instance:

* Selection of the features, or columns to keep in the Dataset for the k-NN
* Value for “k”
* …

Since the full code is written in JavaScript, most of the functions relating to ImageJ actions are using the ImageJ API “IJ”, and the code exploits as well the DataFrame extension installed as plugin for ImageJ.

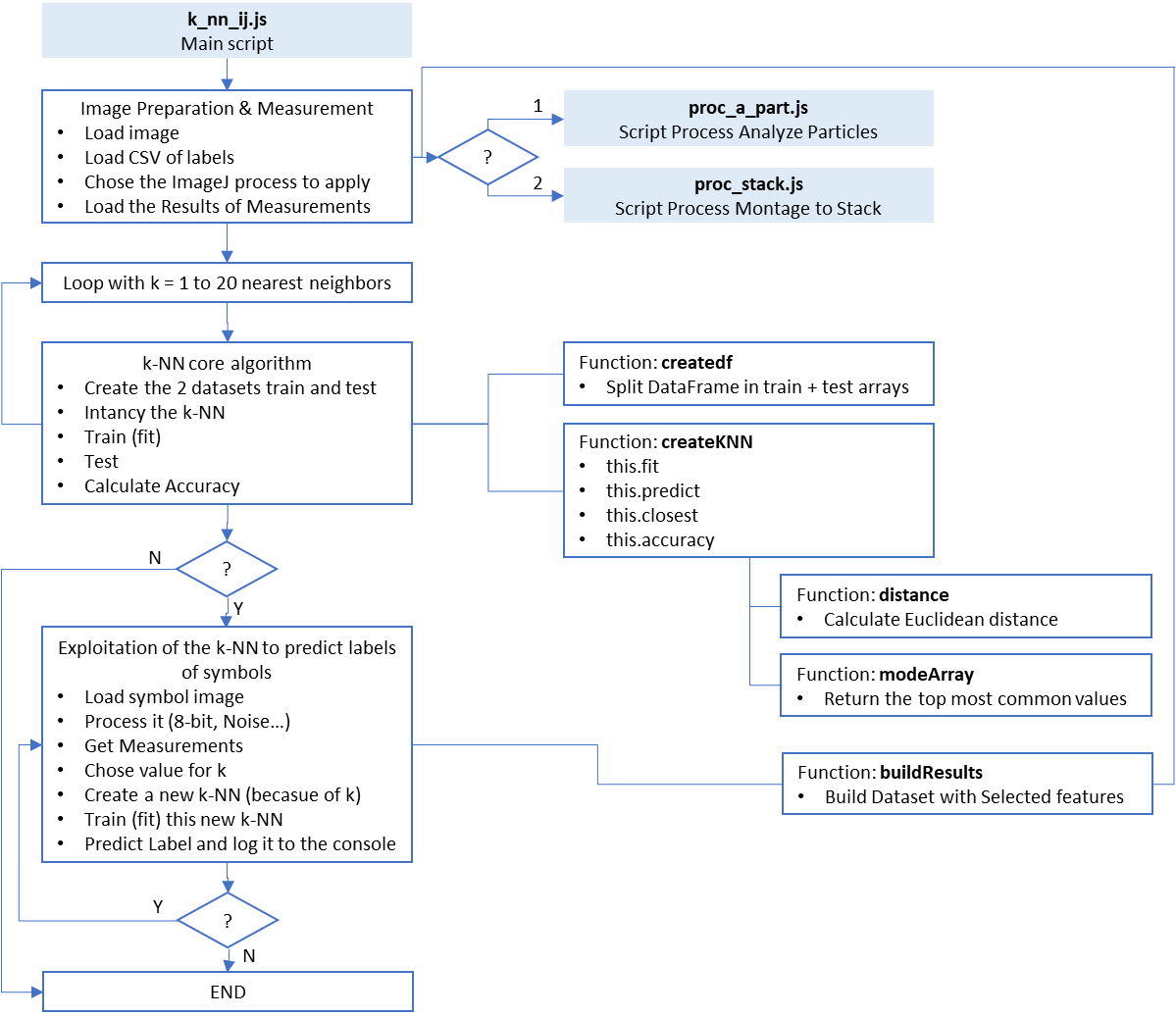
Finally, for convenience, we decided to keep outside the main .js script the two scripts used to prepare the context for the k-NN. We use them to process images outside the k-NN, for testing or troubleshooting purpose for instance.

* The script called proc\_a\_part.js process the Shapes image with Analyze Particles
* The script called proc\_stack.js process the image Circles with Montage to Stack

To produce one single script, they can be easily integrated in the core script – in the if statement line 284.

### Overview of Code Structure

Here is an overview of the structure of the final code:



### k\_nn\_ij.js - Main Script

/\*

k-NN algorithm for ImageJ

This program can use either analyze particule or montage to stack process

main steps are:

  -preparation of the Data with ImageJ

  -training (fit)

  -testing

  -accuracy measurement

  -exploitation (using stack images to predict)

requirements:

  -ImageJ v1.52s, Java SDK 9.0.4

external js scripts:

  -proc\_stack.js, proc\_a\_part.js

Alexandre Cornier - Jan. 2020

\*/

// Import helper functions (used for instance for DataFrame() and console.log())

const IJ\_PLUGINS = IJ.getDir('plugins');

load(`${IJ\_PLUGINS}/javascript/nashorn\_polyfill.js`);

load(`${IJ\_PLUGINS}/javascript/tml.js`);

/\*---------------------------------------------------------------------------

Set Global const and var

----------------------------------------------------------------------------\*/

// Set the 4 DataFrames needed for Training & Test with Data & Labels

const df\_train\_data = [];

const df\_train\_label = [];

const df\_test\_data = [];

const df\_test\_label = [];

// Set the subdirectory hosting proc\_a\_part.js & proc\_stack.js plugins

const pluginsHome = "/0\_Home/";

/\*---------------------------------------------------------------------------

Functions

----------------------------------------------------------------------------\*/

/\*

Create the 4 Dataframes needed from the loaded dataset

Argument:

- dataset: array of arays containing all the Data inclusing Labels

Return nothing but fillin the 4 df\_\* const with the appropriate Data

----------------------------------------------------------------------------\*/

function createdf(label, dataset) {

  // Check lenght of both arrays for equality

  if (label.length != dataset.length) {

    console.log("ERROR: both tables for labels and data don't have the same size: label=" + label.length + ", data=" + dataset.length);

  }

  // Use first half of the data for training

  // Math.trunc is not supported by ImageJ so Math.floor is used instead

  let isplit = Math.floor(dataset.length / 2);

  for (let i = 0; i <= isplit; i++) {

    df\_train\_data.push([dataset[i]]);

    df\_train\_label.push(label[i]);

  }

  isplit = isplit + 1;

  // Use second half of the data for testing

  for (i = isplit; i < dataset.length; i++) {

    df\_test\_data.push([dataset[i]]);

    df\_test\_label.push(label[i]);

  }

}

/\*

Get Euclidean distance between two points

Arguments:

- first point: coordintates as array

- Second point: coordinates as array

Return distance or -1 if the two points don't have the same dimension

----------------------------------------------------------------------------\*/

function distance(a, b) {

    let total = 0, diff = 0, fDist = 0;

    if (a.length != b.length) {

      console.log("ERROR: Distance - both points don't have the same dimension")

      console.log(a);

      console.log(b);

      return -1;

    }

    for (let i = 0; i < a.length; i++) {

        if (a[i] > b[i]) {

            diff = a[i] - b[i];

          } else {

            diff = b[i] - a[i];

          }

        total += diff \* diff;

    }

    fDist = Math.sqrt(total);

    return fDist;

}

/\*

Get the most common value found in an array

Argument:

- arrLabel: array of values

Return one single value that is the most common label

----------------------------------------------------------------------------\*/

function modeArray(arrLabel) {

  let most\_freq = 1;

  let icount = 0;

  let label = arrLabel[0];

  for (let i = 0; i< arrLabel.length; i++) {

    for (let j = i; j< arrLabel.length; j++) {

      if (arrLabel[i] == arrLabel[j]) {

        icount++;

      }

      if (most\_freq < icount) {

        most\_freq = icount;

        label = arrLabel[i];

      }

    }

    m=0;

  }

  return label;

}

/\*

Build the table of results with the selected features from a DataFrame

Argument:

- df: DataFram from ImageJ Results

- ft\_str: CSV String containing selected features separated by comma

Return an array with Measurements in Row and features in Columns

----------------------------------------------------------------------------\*/

function buildResults(df, ft\_str) {

  let features = ft\_str.split(",");

  // Create the new Dataset with selected columns (features)

  let data\_arr = [];

  if (features.length < 2) {

      // Case of one single feature, no need to combine columns

      let df\_single = new DataFrame();

      df\_single = df.select(features[0]);

      data\_arr = df\_single.array();

  } else {

      // Case of multiple features

      let df2 = [];

      let df2\_arr = [];

      // Get the columns corresponding to features selected

      for (let f = 0; f < features.length; f++) {

          df2[f] = df.select(features[f]);

          df2\_arr[f] = df2[f].array();

      }

      // Create new rows and push them in the final table

      for (let i = 0; i < df2\_arr[0].length; i++) {

          let arr\_temp = [];

          for (let j = 0; j < features.length; j++) {

              arr\_temp.push(df2\_arr[j][i]);

          }

          data\_arr.push(arr\_temp);

      }

  }

  return data\_arr;

}

/\*

createKNN - main "Class" containing the fit and predict functions

Argument:

- k: number of closest neighbors

----------------------------------------------------------------------------\*/

function createKNN(k) {

  this.k = k; // number of neighbors to consider

  this.prediction = []; // array of labels predicted using the test Data

  /\*

  initialize DataFrame for Training

  Arguments:

  - df: DataFrame containing the Data

  - labels: array of numbers containing the labels of the observations in df

  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \*/

  this.fit = function(df, labels) {

    this.x\_train = df;

    this.y\_train = labels;

  }

  /\*

  predict Labels

  Argument:

  - df: DataFrame of unlabeled test Data

  Return an array of predicted Labels

  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \*/

  this.predict = function(df) {

    for(let i = 0; i < df.length; i++) {

      let k\_labels = this.closest(df[i]);

      let label = modeArray(k\_labels);

      this.prediction.push(label);

    }

  }

  /\*

  Get closest neighbors

  Arguments:

  - row: data to evaluate

  Return an array of k labels from closest neighbors

  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \*/

  this.closest = function(row) {

    let dist = 0;

    let best\_dist = [], best\_index = [], k\_labels = [];

    // set arrays of resulting distances & indexes

    for(let n = 0; n < this.k; n++) {

      best\_dist.push(distance(row, this.x\_train[0]));

      best\_index.push(0);

      k\_labels.push(0);

    }

    // compute distance with each row in training Dataframe

    for(let i = 0; i < this.x\_train.length; i++) {

      dist =  distance(row, this.x\_train[i]);

      // record and sort the distance - if shorter - in array of results

      for(n = 0; n < this.k; n++) {

        if (dist < best\_dist[n]) {

          best\_dist[n] = dist; // new best distance as result

          best\_index[n] = i; // corresponding index into the training Dataframe

          break;

        }

      }

    }

    //get array of labels for closest neighbors

    for(let n = 0; n < this.k; n++) {

      k\_labels[n] = this.y\_train[best\_index[n]];

    }

    return k\_labels;

  }

  /\*

  Get accuracy of prediction

  Arguments:

  - test\_labels: array of known Labels for the test DataFrame

  Return percentage of accuracy in a range[0..1]

  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \*/

  this.accuracy = function(test\_labels) {

    let label\_match = 0, acc\_percentage = 0;

    if (test\_labels.length != this.prediction.length) {

      console.log("ERROR: Prediction and Test tables don't have the same length.")

      acc\_percentage = -1;

    } else {

      for (let i = 0; i < test\_labels.length; i++) {

        if (test\_labels[i] == this.prediction[i]) {

          label\_match++;

        }

      }

      acc\_percentage = label\_match / test\_labels.length;

    }

    return acc\_percentage;

  }

}

/\*---------------------------------------------------------------------------

Core algorithm - ImageJ - Image preparation & Measurement

----------------------------------------------------------------------------\*/

// Get the image fullpath from a dialog box

let path = IJ.getFilePath("Select the image to open");

// Open image

let image = IJ.openImage(path);

image.show("Image"); // display it

// Get the CSV file fullpath containing labels from a dialog box

path = IJ.getFilePath("Select the CSV file containing the labels for this image");

// Open file

let CSV\_labels = ResultsTable.open(path);

CSV\_labels.show("Results");

// Load DataFrame from the Result Window

let table = ResultsTable.getResultsTable();

let df\_label = new DataFrame();

df\_label.fromIJ(table);

// Close the Result Window used to load the CSV Labels

IJ.selectWindow("Results");

IJ.run("Close");

// Create an array containing the Labels and load the Data from the DataFrame

let label\_arr = df\_label.column('Label').array();

// Get the process to apply, to choose between image to stack and analyze particles

let iprocess = IJ.getNumber("ImageJ process to apply to the image: 1 for analyze particles, 2 for montage to stack, ", 1);

// Get the plugin directory

let plugin\_path = IJ.getDirectory("plugins");

let plugin = "";

// Run the selected process script

if (iprocess == 1) {

  // Case of Shapes measured using Analyze Particles

  plugin = plugin\_path + pluginsHome + "proc\_a\_part.js"

  IJ.runMacroFile(plugin);

} else if (iprocess == 2) {

  // Case of Circles measured using Montage to Stack

  plugin = plugin\_path + pluginsHome + "proc\_stack.js"

  IJ.runMacroFile(plugin);

}

// Load DataFrame from the Result Window after Measurement

let table\_results = ResultsTable.getResultsTable();

let df\_results = new DataFrame();

df\_results.fromIJ(table\_results);

// Get the features to use

let q\_str = df\_results.headings;

let features\_str = IJ.getString("Select the features to use for the k-NN", q\_str);

// Build the target Dataset containing only selected features

let results\_arr = buildResults(df\_results, features\_str);

/\*---------------------------------------------------------------------------

Core algorithm of k-NN using values [1..20] for k

----------------------------------------------------------------------------\*/

// Create the 4 DataFrames for train & Test with Data & Labels in distincts arrays

createdf(label\_arr, results\_arr);

let faccuracy = 0;

for (let k = 1; k < 21; k++) {

  // Create the k-NN object

  let classifier = new createKNN(k);

  // Load the Data & Labels for the training into the k-NN object

  classifier.fit(df\_train\_data, df\_train\_label);

  // Predict Labels using test Data (without providing Labels)

  classifier.predict(df\_test\_data);

  //Measure accuracy by comparing test & predicted labels

  faccuracy = classifier.accuracy(df\_test\_label);

  console.log(faccuracy);

}

/\*---------------------------------------------------------------------------

Exploitation of the k-NN to predict labels of symbols

----------------------------------------------------------------------------\*/

// Get answer for predicting an image or not

let bpredict = IJ.getString("Do you want to predict the label for an image? (Y or N)", "Y");

if (bpredict == "Y") {

  // Close the Stack window

  IJ.selectWindow("Stack");

  IJ.run("Close");

  do {

    // Close the Result window

    IJ.selectWindow("Results");

    IJ.run("Close");

    // Get the image fullpath from a dialog box

    let path\_img = IJ.getFilePath("Select the image to open for prediction");

    // Open image

    let img\_test = IJ.openImage(path\_img);

    img\_test.show("Img\_test");

    // Instance current image

    let imp = IJ.getImage();

    // Convert to Gray 8-bit

    IJ.run(imp, "8-bit", "" );

    // Get noise level to apply

    let fnoise = IJ.getNumber("Standard deviation level of Noise to apply, 0 for no Noise", 0);

    // Add noise if the deviation entered is > 0

    if (fnoise > 0) {

      // Get the number of cycles to apply Noise

      let fnb = IJ.getNumber("Number of cycles to apply Noise", 1);

      // Divide by 2 the pixels intensity

      IJ.run(imp, "Divide...", "value=2");

      // Cycle noise

      let std\_noise = "standard=" + fnoise;

      for (let i = 0; i < fnb; i++) {

        IJ.run(imp, "Add Specified Noise...", std\_noise);

      }

    }

    //Set Measurements

    IJ.run(imp, "Set Measurements...", "area mean min centroid shape redirect=None decimal=3");

    // Get results

    IJ.run(imp, "Measure Stack...", "");

    // Load DataFrame from the Result Window after Measurement

    table\_results = ResultsTable.getResultsTable();

    df\_results = new DataFrame();

    df\_results.fromIJ(table\_results);

    // Build the Dataset containing only selected features

    let results\_arr = [];

    results\_arr.push([buildResults(df\_results, features\_str)]);

    // Create an array containing the Data and load from the DataFrame

    //let results\_arr = [];

    //results\_arr.push([df\_data.array(0)]);

    // Get the number of nearst neighbors for the k-NN

    let k = IJ.getNumber("How many neighbors for the k-NN?", 1);

    // Create the k-NN object

    let classifier = new createKNN(k);

    // Load the Data & Labels for the training into the k-NN object

    classifier.fit(df\_train\_data, df\_train\_label);

    // Predict Labels using test Data (without providing Labels)

    classifier.predict(results\_arr);

    console.log("Symbol " + results\_arr + " is of type: " + classifier.prediction);

    // Get answer for predicting another image or not

    bpredict = IJ.getString("Do you want to predict the label for another image? (Y or N)", "Y");

    // Close the previous image

    imp.close();

  } while (bpredict == "Y");

}

//-------------------------------- END --------------------------------------

### proc\_a\_part.js – script to process the Shapes image with Analyze Particles

/\*

Process the current image in Imagej with Analyze Particles

This script is to be used as a plugin in ImageJ

Can be use alone or called by the main k\_NN\_js script

Process:

- convert to 8 bit

- invert

- resize to 1000x700

- set threshold to Otsu dark

- set measurements

- analyze particles to get results

Alexandre Cornier - Jan. 2020

\*/

// Instance current image

let imp = IJ.getImage();

// Convert to Gray 8-bit

IJ.run(imp, "8-bit", "" );

// Invert

IJ.run(imp, "Invert", "" );

// Increase canvas size

IJ.run(imp, "Canvas Size...", "width=1000 height=700 position=Center zero");

// Set Threshold

IJ.setAutoThreshold(imp, "Otsu dark");

// Set Measurements

IJ.run(imp, "Set Measurements...", "area centroid shape redirect=None decimal=3");

// Analyze particles

IJ.run(imp, "Analyze Particles...", "  show=Outlines display");

### proc\_stack.js – script to process the Shapes image with Analyze Particles

/\*

Process the current image in Imagej with Montage to Stack

This script is to be used as a plugin in ImageJ

Can be use alone or called by the main k\_NN\_js script

Process:

- convert to 8 bit

- ask for a Noise Standard Deviation, if entered

  - divide by 2 the pixels intensity

  - ask for a number of cycle to apply the std deviation

  - apply n times the std deviation

- set measurements

- run the montage to stack

- measure stack to get results

Alexandre Cornier - Jan. 2020

\*/

// Instance current image

let imp = IJ.getImage();

// Convert to Gray 8-bit

IJ.run(imp, "8-bit", "" );

// Get noise level to apply

let fnoise = IJ.getNumber("Standard deviation level of Noise to apply, 0 for no Noise", 0);

// Add noise if the deviation entered is > 0

if (fnoise > 0) {

  // Get the number of cycles to apply Noise

  let fnb = IJ.getNumber("Number of cycles to apply Noise", 1);

  // Divide by 2 the pixels intensity

  IJ.run(imp, "Divide...", "value=2");

  // Cycle noise

  let std\_noise = "standard=" + fnoise;

  for (let i = 0; i < fnb; i++) {

    IJ.run(imp, "Add Specified Noise...", std\_noise);

  }

}

// Set Measurements

IJ.run(imp, "Set Measurements...", "area mean min centroid shape redirect=None decimal=3");

// Create stack from image

IJ.run(imp, "Montage to Stack...", "columns=32 rows=32 border=0");

// Instance the stack

let stack = IJ.getImage();

// Get results

IJ.run(stack, "Measure Stack...", "");