First Steps With Julia

- Kaggle

Description

This competition is designed to help you get started with Julia. If you are looking for a good programming language for data science, or if you are already accustomed to one language, we encourage you to also try Julia. Julia is a relatively new language for technical computing that attempts to combine the strengths of other popular programming languages.

Here we introduce two tutorials to highlight some of Julia's features. The first is focused on the basics of the language. In the second, a complete implementation of the K Nearest Neighbor algorithm is presented, highlighting features such as parallelization and speed.

Both tutorials show that it is easy to write code in Julia, due to its intuitive syntax and design. The tutorials also describe some basics of image processing and some concepts of machine learning such as cross validation. After reviewing them, we hope you will be motivated to write your own machine learning algorithms in Julia.

This tutorial focuses on the task of identifying characters from Google Street View images. It differs from traditional character recognition because the data set contains different character fonts and the background is not the same for all images.

Acknowledgements

The data was taken from the Chars74K dataset, which consists of images of characters selected from Google Street View images. We ask that you cite the following reference in any publication resulting from your work:

T. E. de Campos, B. R. Babu and M. Varma, Character recognition in natural images, *Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP)*, Lisbon, Portugal, February 2009.

This tutorial was developed by Luis Tandalla during his summer 2014 internship at Kaggle.

Evaluation

Your model should identify the character in each image in the test set. The possible characters are 'A-Z', 'a-z', and '0-9'.

The predictions will be evaluated using Classification Accuracy.

$$\text{Accuracy} = \frac{\sum_{i=1}^{N} \text{true}_i = \text{prediction}_i}{N}$$

Submission File

For every image in the dataset, submission files should contain two columns: ImageId and Class (character predicted) .

The file should contain a header and have the following format:

```
ImageId,Class
6284,A
6285,b
6286,0
...
```

Julia Tutorial

This tutorial introduces the Julia language for data science tasks. It teaches some basics of image processing and uses a popular machine learning algorithm to identify the character from pictures.

IJulia or Julia Studio may help you to write and run Julia code. You may also use Forio if you want to run Julia code without installing it on your computer.

Image Loading

First, we need to install and load the required packages.

```
Pkg.add("Images")
Pkg.add("DataFrames")
using Images
using DataFrames
```

All the images should be read and stored into a matrix of real numbers. For simplicity, we use the trainResized and testResized data files, as all the images have the same size.

imread() allows us to read the image. float32sc() changes the image into real values. The result could be a single matrix if the image is black/white, or a triple array that contains three matrices representing each color (Red, Green, Blue).

It is easier to work with images with the same representation, so we convert all of the color images to grayscale by averaging the values across the three color matrices.

```
img = imread(nameFile)
temp = float32sc(img)
if ndims(temp) == 3
 temp = mean(temp.data, 1)
end
```

The result is a single matrix per image. Changing each image matrix into a vector allows us to save all results in a single matrix that contains the data for all images.

```
x[i, :] = reshape(temp, 1, imageSize)
```

The result is a data matrix where each row is an image instance and each column is the value for a specific pixel in the image. Each column is also interpreted as a feature.

These steps can be combined into a single function allowing us to easily repeat the process for other images. Note that we can access the string representation of a variable with syntax "$(var)" or "$var".

```
#typeData could be either "train" or "test.
#labelsInfo should contain the IDs of each image to be read
#The images in the trainResized and testResized data files
#are 20x20 pixels, so imageSize is set to 400.
#path should be set to the location of the data files.

function read_data(typeData, labelsInfo, imageSize, path)
 #Intialize x matrix
 x = zeros(size(labelsInfo, 1), imageSize)

 for (index, idImage) in enumerate(labelsInfo["ID"])
  #Read image file
  nameFile = "$(path)/$(typeData)Resized/$(idImage).Bmp"
  img = imread(nameFile)

  #Convert img to float values
  temp = float32sc(img)

  #Convert color images to gray images
  #by taking the average of the color scales.
  if ndims(temp) == 3
   temp = mean(temp.data, 1)
  end
```

```
  #Transform image matrix to a vector and store
  #it in data matrix
  x[index, :] = reshape(temp, 1, imageSize)
 end
 return x
end
```

Training and test matrices can now be loaded using function read_data(). Information about the labels can be read using the readtable() function:

```
imageSize = 400 # 20 x 20 pixel

#Set location of data files, folders
path = ...

#Read information about training data , IDs.
labelsInfoTrain = readtable("$(path)/trainLabels.csv")

#Read training matrix
xTrain = read_data("train", labelsInfoTrain, imageSize, path)

#Read information about test data ( IDs ).
labelsInfoTest = readtable("$(path)/sampleSubmission.csv")

#Read test matrix
xTest = read_data("test", labelsInfoTest, imageSize, path)
```

The labels are characters, but the algorithms recognize numbers, so we will map each character to an integer. The data is loaded into a string type by default. We take the first element of the string (the actual character) and convert it to an integer number.

```
#Get only first character of string (convert from string to character).
#Apply the function to each element of the column "Class"
yTrain = map(x -> x[1], labelsInfoTrain["Class"])

#Convert from character to integer
yTrain = int(yTrain)
```

Training model

Since we now have both the images data and labels represented vectors of real numbers, we are ready to apply a machine learning algorithm. The algorithm should learn the patterns in the images that identify the character in the label.

Here we will use the Julia version of the popular Random Forest algorithm. This algorithm can usually achieve high performance without the need of tuning many parameters (more information about the algorithm can be found at Random Forest). The model requires that we set three parameters: the **number of features** to choose at each split, the **number of trees**, and the **ratio of subsampling**. The number of features to try at each split is usually chosen to be

$$\sqrt{\text{number of features}}$$

which in this case would

$$\sqrt{400} = 20$$

The numbers of trees is chosen arbitrarily. Larger is better, but it takes more time to train. The ratio of subsampling is usually chosen to be 1.0. However, you may change any of these numbers and chose the ones that produce the highest performance.

Let's now train the model:

```
Pkg.add("DecisionTree")
using DecisionTree

#Train random forest with
#20 for number of features chosen at each random split,
#50 for number of trees,
#and 1.0 for ratio of subsampling.
model = build_forest(yTrain, xTrain, 20, 50, 1.0)
```

With the model trained, we use it to identify the characters in the test data:

```
#Get predictions for test data
predTest = apply_forest(model, xTest)
```

The result will be an array of integers, so we need to convert them back to characters. Afterwards, we save and write the results into a file:

```
#Get predictions for test data
predTest = apply_forest(model, xTest)

#Convert integer predictions to character
labelsInfoTest["Class"] = char(predTest)

#Save predictions
writetable("$(path)/juliaSubmission.csv", labelsInfoTest, separator=',', header=true)
```

Finally, the submission file has been written, which you can upload to get a score. Let's see some of the predictions!

The following picture corresponds to image 6284. According to our prediction file juliaSubmission.csv, the prediction for this image is an 'H', which in fact it is.

The following picture corresponds to image 6310. Our prediction for this image is an 'E', but the character is in fact a 'B'. This image of a B does have strong visual similarity with an E, so it is harder for the algorithm to recognize it correctly.

In most Kaggle competitions, you are only allowed a fixed number of submissions per day, and ideally the test set should not be observed while building the model. **n-fold cross validation** is used to test the performance of a model without using the test data. You can use it to measure the performance of several models and upload only the ones with the highest performance. The random forest implementation in Julia Forest already includes a function for n-fold cross validation. We can run it using 4 folds:

```
accuracy = nfoldCV_forest(yTrain, xTrain, 20, 50, 4, 1.0);
println ("4 fold accuracy: $(mean(accuracy))")
```

The result should be similar to the one obtained by submitting to the leaderboard.

You may want to re-run this code, trying different values for the different parameters and choosing the best combination. You may also try different machine learning algorithms that are already written in Julia, or write one yourself. We'll do just that in the next tutorial.

Knn Tutorial

Implement and customize a machine learning algorithm

In this tutorial, we use Julia to implement the K-Nearest Neighbor (k-NN) algorithm with Leave-One-Out-Fold Cross Validation (LOOF-CV). k-NN and LOOF-CV are available in many standard machine learning libraries, but they do not take the advantage of the fact that LOOF-CV is particularly fast with k-NN. Here we customize the implementation to use this property and efficiently tune the parameter k, the number of neighbors. For cases where a custom implementation is necessary, Julia is an attractive language choice because prototypes can be easily written and the language is fast without needing external code. This tutorial also introduces parallelization in Julia, which allows us to speed up the program. Parallelization is not unique to Julia, but Julia is designed such that it is remarkably easy implement.

A brief background of some concepts in machine learning is given. You may safely skip any section if you already know the material.

Measuring the performance of a model

One of the most important steps when building a model is to measure its performance. One popular method is k-fold cross validation, in which the training data is split in k partitions or folds. The model is tested on each of the k folds after being trained on the remaining (k-1) folds. The cv-error is the average of the errors in the k folds. This method gives a good approximation of the error, but it increases run time by a factor of k (since we need to train the model k times). It also ignores a portion of the data at each training, so the error may overfit a particular split of the data.

Leave-One-Out-Fold-Cross-Validation (LOOF-CV) is similar to k-fold CV, but k is set to be equal to the number of training points. The model is tested on each individual data point after being trained with the remaining points in the data. Because LOOF-CV uses all but one of the data points for training, it is not biased to any particular point or split, and could arguably give a better estimation for the performance of the model. It is sometimes intractable to perform LOOF-CV, but it can be done efficiently with the K-Nearest Neighbors algorithm.

Brief Description of KNN

In the case of 1 Nearest Neighbor, the label of a test point is set to be the label of the nearest training point. In the case of k-NN, the label of a test point is set to be the most common label of the k nearest training points.

Preparing data

This tutorial assumes that you have loaded the data using the code described in the previous tutorial, so you have the XTrain, XTest, and yTrain matrices in your workspace. As a reminder, the X* matrices contain the data with each row being a data point, which in this case is an image.

We start by transposing the XTrain and XTest matrices:

```
xTrain = xTrain'
xTest = xTest'
```

The transposed matrices now have the columns representing data points (images) and the rows representing features. An iteration from one image to the next is an iteration from column to column, which is desirable since iteration over columns is faster in Julia than iteration over rows (performance tips). In the previous tutorial, the random forest function expected the other representation.

Implementing k-NN with LOOF-CV

To find the nearest point to a given point, we need the distance between the given point and each of the training points. First, we must define a distance function that measures the similarity between two data points, in this case two images. There are several options for distance functions. You may even design one yourself.

Vectorized operations and for loops for computing distance

A common choice for similarity measures is Euclidean distance:

$$\text{distance} = \sum_{i=1}^{N} (a_i - b_i)^2 = (a-b)\cdot(a-b)$$

Vectorized operations are common in many languages, so let's first write the vectorized form of euclidean distance. This is simply the dot product between the difference of the two vectors:

```
function euclidean_distance(a, b)
 return dot(a-b, a-b)
end
```

That was simple! But what if the vectorized form is harder to write than the for-loop equivalent? What if you only know how to write for loops and not vectorized operations? In general, loops are much slower in languages such as Python, R, and Matlab. Thanks to the typing and clever compiler decisions that happen behind the scenes in Julia, the opposite is often true: *for loops can be faster than vectorized operations!*

Additionally, why create an entire vector if only one number, such as its sum, is needed? In the example above, the new vector (a - b) is created, but is later discarded since only one number is returned by the function.

Let's rewrite Euclidean distance using a for loop:

$$\text{distance} = \sum_{i=1}^{N} (a_i - b_i)^2$$

```
function euclidean_distance(a, b)
 distance = 0.0
 for index in 1:size(a, 1)
  distance += (a[index]-b[index]) * (a[index]-b[index])
 end
 return distance
end
```

The function above takes extra lines, but no knowledge of vectorized operations is needed. It uses the definition of Euclidean distance and does not incur additional memory costs (e.g., an intermediary vector is not created and later discarded). We will use this version of Euclidean distance for the remainder of the tutorial.

Continuation of k-NN

Next we define a function that finds the k nearest neighbors of a data point:

```
#This function finds the k nearest neighbors of the i^th data point.
function get_k_nearest_neighbors(x, i, k)
```

```
    nRows, nCols = size(x)

    #Let's initialize a vector image_i. We do this so that
    #the image i^th is accessed only once from the main X matrix.
    #The program saves time because no repeated work is done.
    #Also, creating an empty vector and filling it with each
    #element at a time is faster than copying the entire vector at once.
    #Creating empty array (vector) of nRows elements of type Float32(decimal)
    imageI = Array(Float32, nRows)

    for index in 1:nRows
     imageI[index] = x[index, i]
    end

    #For the same previous reasons, we initialize an empty vector
    #that will contain the j^th data point
    imageJ = Array(Float32, nRows)

    #Let's also initialize an empty vector that will contain the distances
    #between the i^th data point and each data point in the X matrix.
    distances = Array(Float32, nCols)

    for j in 1:nCols
     #The next for loop fills the vector image_j with the j^th data point
     #from the main matrix. Copying element one by one is faster
     #than copying the entire vector at once.
     for index in 1:nRows
      imageJ[index] = x[index, j]
     end
     #Let's calculate the distance and save the result
     distances[j] = euclidean_distance(imageI, imageJ)
    end

    #The following line gives the indices sorted by distances.
    sortedNeighbors = sortperm(distances)

    #Let's select the k nearest neighbors. We start with the
    #second closest. See explanation below.
    kNearestNeighbors = sortedNeighbors[2:k+1]
    return kNearestNeighbors
   end
```

Since the code calculates the distance between the i^th data point and all the points in the training data, the closest point to the i^th point is itself with a distance of zero. Hence, we exclude it and select the next k points. At this stage, we apply LOOF-CV by excluding the test point and only that point.

Now that we have found the k nearest neighbors, let's assign a label to our test point. The most popular label in the selected neighbors is chosen for the test point.

```
#This function assigns a label to the i^th point according to
#the labels of the k nearest neighbors. The training
#data is stored in the X matrix, and its labels are stored in y.

function assign_label(x, y, k, i)
 kNearestNeighbors = get_k_nearest_neighbors(x, i, k)

 #let's make a dictionary to save the counts of
 #the labels
 # Dict{}() is also right .
 # Int,Int indicates the dictionary to expect integer values
 counts = Dict{Int, Int}()

 #The next two variables keep track of the
 #label with the highest count.
```

```
 highestCount = 0
 mostPopularLabel = 0

 #Iterating over the labels of the k nearest neighbors
 for n in kNearestNeighbors
  labelOfN = y[n]
  #Adding the current label to our dictionary
  #if it's not already there
  if !haskey(counts, labelOfN)
   counts[labelOfN] = 0
  end
  #Add one to the count
  counts[labelOfN] += 1

  if counts[labelOfN] > highestCount
   highestCount = counts[labelOfN]
   mostPopularLabel = labelOfN
  end
 end
 return mostPopularLabel
end
```

The next step is to apply the previous function for each point in the training data:

```
#In this example, we use a value of 1 for k.
k=1
yPredictions = [assign_label(xTrain, yTrain, k, i) for i in 1:size(xTrain, 2)]
```

Finally, we can measure the accuracy of the model by comparing our predictions with the true labels.

```
#The . makes an element-wise comparison
loofCvAccuracy = mean(yPredictions .== yTrain)
println("The LOOF-CV accuracy of 1NN is $(loofCvAccuracy)")

#This is also an example of a vectorized
#operation that Julia is also capable of doing.
```

Parallelization in Julia

Let's speed up the program with parallelization. Instead of calculating the label for each point at a time, we can easily calculate them in parallel. To use parallelization, we need to run this line at the beginning of our code:

```
addprocs(2)
#This line adds 2 parallel processes to the program
#increasing the speed by a factor of approximately 2.
#You can choose a different number if you have more
#available cores in your machine.
```

We also need to add @everywhere before each of our functions. This makes the functions available to each of our processes.

```
@everywhere function euclidean_distance(a, b)
 distance = 0.0
 for index in 1:size(a, 1)
  distance += (a[index]-b[index]) * (a[index]-b[index])
 end
 return distance
end

@everywhere function get_k_nearest_neighbors(x, i, k)
 nRows, nCols = size(x)
 imageI = Array(Float32, nRows)
 for index in 1:nRows
  imageI[index] = x[index, i]
```

```
 end
 imageJ = Array(Float32, nRows)
 distances = Array(Float32, nCols)
 for j in 1:nCols
  for index in 1:nRows
   imageJ[index] = x[index, j]
  end
  distances[j] = euclidean_distance(imageI, imageJ)
 end
 sortedNeighbors = sortperm(distances)
 kNearestNeighbors = sortedNeighbors[2:k+1]
 return kNearestNeighbors
end

@everywhere function assign_label(x, y, k, i)
 kNearestNeighbors = get_k_nearest_neighbors(x, i, k)
 counts = Dict{Int, Int}()
 highestCount = 0
 mostPopularLabel = 0
 for n in kNearestNeighbors
  labelOfN = y[n]
  if !haskey(counts, labelOfN)
   counts[labelOfN] = 0
  end
  counts[labelOfN] += 1
  if counts[labelOfN] > highestCount
   highestCount = counts[labelOfN]
   mostPopularLabel = labelOfN
  end
 end
 return mostPopularLabel
end
```

The @ keyword in Julia indicates a macro. Macros are analogous to functions with respect to expressions. Functions take values as arguments and perform some operations with those values. Macros take expressions and perform some operations with those expressions. In the previous example, the macro @everywhere make its arguments, the definition of the functions, available to each process.

Julia is ready to run code in parallel after the previous steps. We only need to add @parallel (vcat) to make a parallel for-loop:

```
#This line runs the for loop in parallel
#and saves the results in yPredictions.
k = 1
yPredictions = @parallel (vcat) for i in 1:size(xTrain, 2)
 assign_label(xTrain, yTrain, k, i)
end

#Next line added for comparison
yPredictions = [ assign_label(xTrain, yTrain, 1, i) for i in size(xTrain, 2)]
```

(vcat) combines our results in a vector. We can also use (+) to add up all the values and calculate the accuracy directly.

```
k = 1
sumValues = @parallel (+) for i in 1:size(xTrain, 2)
 assign_label(xTrain, yTrain, k, i) == yTrain[i, 1]
end
loofCvAccuracy = sumValues / size(xTrain, 2)

#Next lines added for comparison
yPredictions = [ assign_label(xTrain, yTrain, k, i) for i in size(xTrain, 2)]
loofCvAccuracy = mean(yPredictions .== yTrain)
```

That wasn't complicated! Julia runs in parallel by adding only 3 steps:

1. addprocs() before running the code
2. @everywhere before each function
3. @parallel before each for loop

In other languages, the programmer needs to make sure that the data is available to all the parallel workers. In our code, we didn't need to worry about that. In Julia, each parallel process is able to access the data by default.

Tuning the value of k

The calculation of CV performance of a model is mostly used to compare it with other models. You may use the leaderboard score, but you may be deceived and overfit to the leaderboard. This is why we use cross validation. Even if you do not compare your model with others, most ML algorithms have hyper-parameters that need to be tuned for best performance. In the case of k-NN, we need to tune k. The CV performance helps us to find the optimal values for those parameters. Standard libraries usually do CV for each value independently. With k-NN we can calculate the LOOF-CV performance for different values of k at once and avoid re-training the model. We now customize our implementation to take advantage of this feature.

The k nearest neighbors are already located in the function *assign_label* that returns the label based on the k neighbors. We only need to modify it so that it returns the labels based on 1, 2, 3, ..., k-1, k neighbors. The new function returns a vector of k labels. When iterating over the k neighbors, the current best label is stored into the vector of labels. Let's look at the code:

```
#Similar to function assign_label.
#Only changes are commented
@everywhere function assign_label_each_k(x, y, maxK, i)
 kNearestNeighbors = get_k_nearest_neighbors(x, i, maxK)

 #The next array will keep the labels for each value of k
 labelsK = zeros(Int, 1, maxK)

 counts = Dict{Int, Int}()
 highestCount = 0
 mostPopularLabel = 0

 #We need to keep track of the current value of k
 for (k, n) in enumerate(kNearestNeighbors)
  labelOfN = y[n]
  if !haskey(counts, labelOfN)
   counts[labelOfN] = 0
  end
  counts[labelOfN] += 1
  if counts[labelOfN] > highestCount
   highestCount = counts[labelOfN]
   mostPopularLabel = labelOfN
  end
  #Save current most popular label
  labelsK[k] = mostPopularLabel
 end
 #Return vector of labels for each k
 return labelsK
end
```

We then apply our new function to each data point in parallel:

```
maxK = 20 #Any value can be chosen
yPredictionsK = @parallel (vcat) for i in 1:size(xTrain, 2)
```

```
 assign_label_each_k(xTrain, yTrain, maxK, i)
end
```

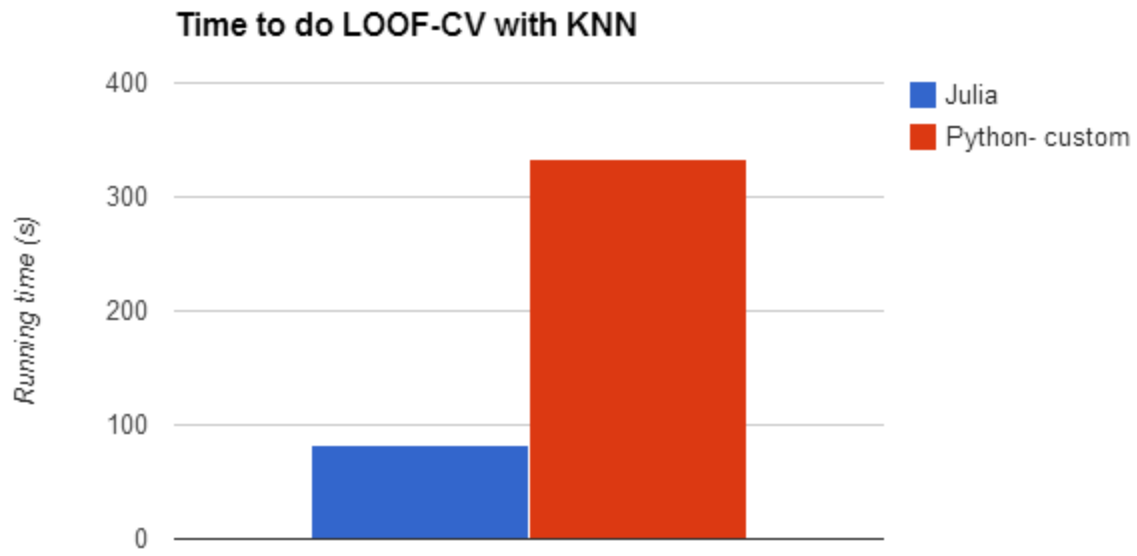
The result is a matrix with the predictions using 1NN in the first column, using 2NN in the second column, and so on. Now, let's calculate the accuracy for each column:

```
for k in 1:maxK
 accuracyK = mean(yTrain .== yPredictionsK[:, k])
 println("The LOOF-CV accuracy of $(k)-NN is $(accuracyK)")
end
```

The accuracies for each value of k are printed. In this case, k=3 produces the highest accuracy. We therefore choose that number as our final value for k-NN.

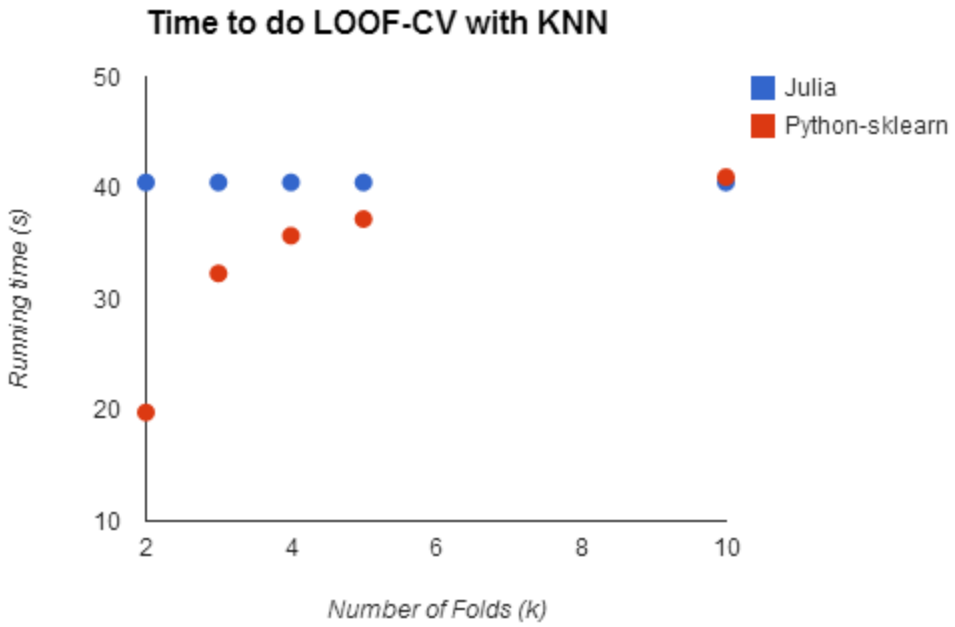
Some comparisons

The Julia code on this page is available for download from the Data page. There are also two Python scripts for comparison. One version uses a personalized version of k-NN similar to the Julia code, and the other version uses the k-NN implementation from scikit-learn. Looking at each source code, both are relatively easy to write, but let's investigate the run time.

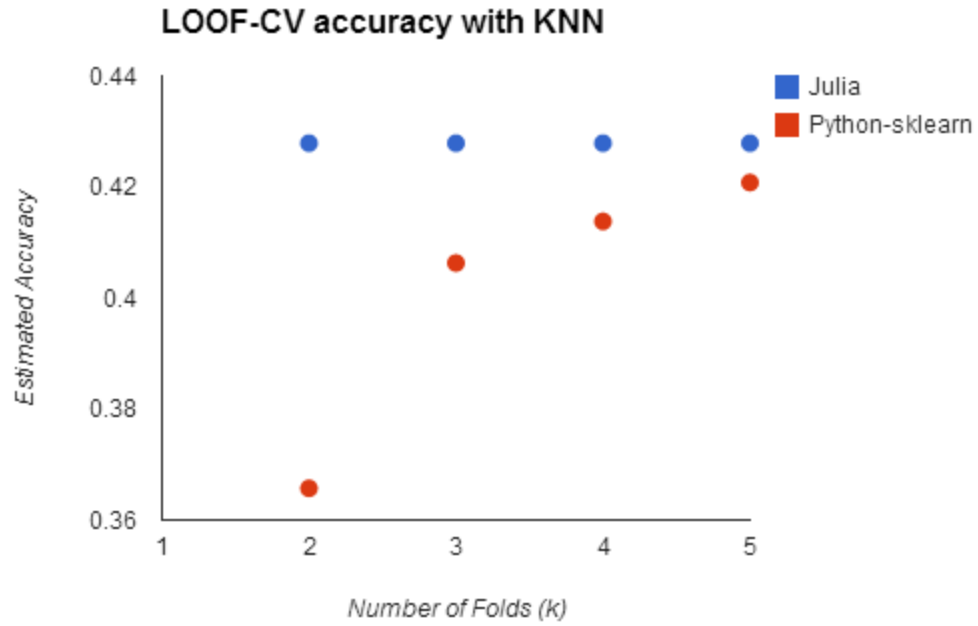


Our custom implementation of LOOF-CV in Julia takes 83 seconds to run, while a similar implementation written in Python takes 340 seconds. In this case, Julia runs 4 times faster.

Let's compare the Julia implementation with the Python implementation that uses scikit-learn. We use KFold-CV in the Python code because there is not a direct implementation of LOOF-CV with KNN in sklearn.

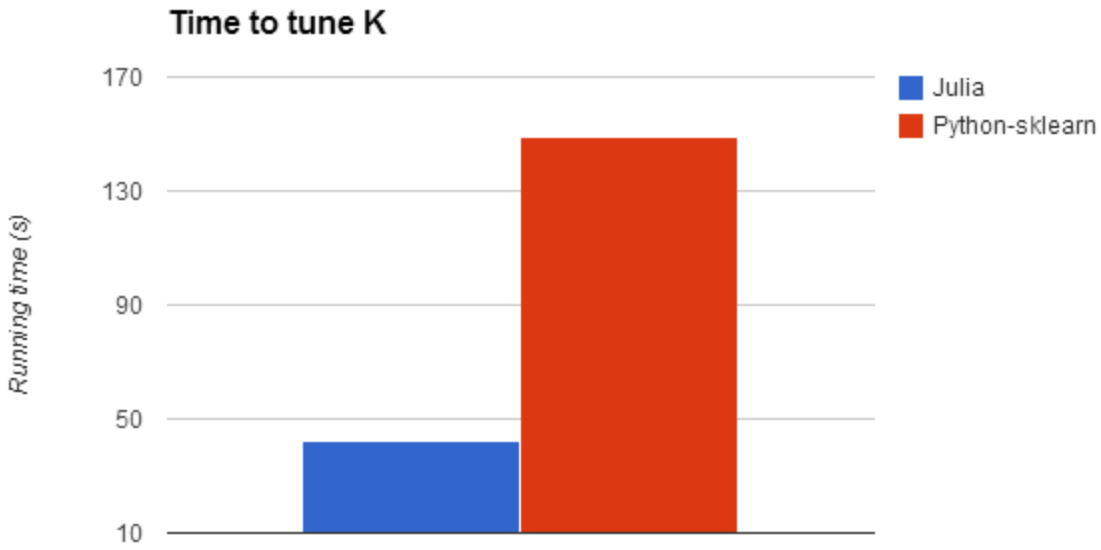


In this case, the Python code using sklearn runs faster than the Julia code running in parallel. This may be expected because sklearn is a highly optimized library with advanced algorithms written in low-level languages, such as C. 2-fold CV with Python is the fastest, but let's also look at the values of the accuracy.



The first plot shows that 2-fold CV with Python is faster than LOOF-CV with Julia, but the second plot shows that 2-fold CV underestimates the accuracy of the KNN model. 5-fold accuracy is similar to LOOF-CV accuracy, so 5 may be a good value for k. With a value of 5, the Python-scikit code is only slightly faster than the Julia code.

Now, let's compare our implementation that tunes the value of the number of neighbors. We use LOOF-CV with Julia and 5-fold CV with python-sklearn. The Julia code tries values from 1 to 20 for k. The Python code tries only values from 1 to 5 for k.



Our custom Julia code took 42 seconds to calculate all the LOOF-CV accuracies for each value of k from 1 to 20. It takes almost the same time as running k-NN for one single value of k. On the other hand, the Python code took 149 seconds to calculate the LOOF-CV accuracies for each value of k from 1 to 5.

Even though the library used by Python is highly optimized, our Julia code is much faster for tuning k because we added the special optimization in the implementation. As you can see from this tutorial, it is easy to write a custom and fast implementation of a machine learning model in Julia.

Final k-NN

In the previous code, we implement LOOF-CV with k-NN. The actual code that calculates the predictions for the test set is written below:

```
@everywhere function get_k_nearest_neighbors(xTrain, imageI, k)
 nRows, nCols = size(xTrain)
 imageJ = Array(Float32, nRows)
 distances = Array(Float32, nCols)
 for j in 1:nCols
  for index in 1:nRows
   imageJ[index] = xTrain[index, j]
  end
  distances[j] = euclidean_distance(imageI, imageJ)
 end
 sortedNeighbors = sortperm(distances)
 kNearestNeighbors = sortedNeighbors[1:k]
 return kNearestNeighbors
end

@everywhere function assign_label(xTrain, yTrain, k, imageI)
 kNearestNeighbors = get_k_nearest_neighbors(xTrain, imageI, k)
 counts = Dict{Int, Int}()
 highestCount = 0
 mostPopularLabel = 0
 for n in kNearestNeighbors
  labelOfN = yTrain[n]
  if !haskey(counts, labelOfN)
   counts[labelOfN] = 0
  end
  counts[labelOfN] += 1 #add one to the count
  if counts[labelOfN] > highestCount
   highestCount = counts[labelOfN]
   mostPopularLabel = labelOfN
```

```
    end
   end
   return mostPopularLabel
  end

  k = 3 # The CV accuracy shows this value to be the best.
  yPredictions = @parallel (vcat) for i in 1:size(xTest, 2)
   nRows = size(xTrain, 1)
   imageI = Array(Float32, nRows)
   for index in 1:nRows
    imageI[index] = xTest[index, i]
   end
   assign_label(xTrain, yTrain, k, imageI)
  end

  #Convert integer predictions to character
  labelsInfoTest["Class"] = char(yPredictions)

  #Save predictions
  writetable("$(path)/juliaKNNSubmission.csv", labelsInfoTest, separator=',', header=true)
```

Citation

joycenv, Will Cukierski. (2014). First Steps With Julia. Kaggle. https://kaggle.com/competitions/street-view-getting-started-with-julia