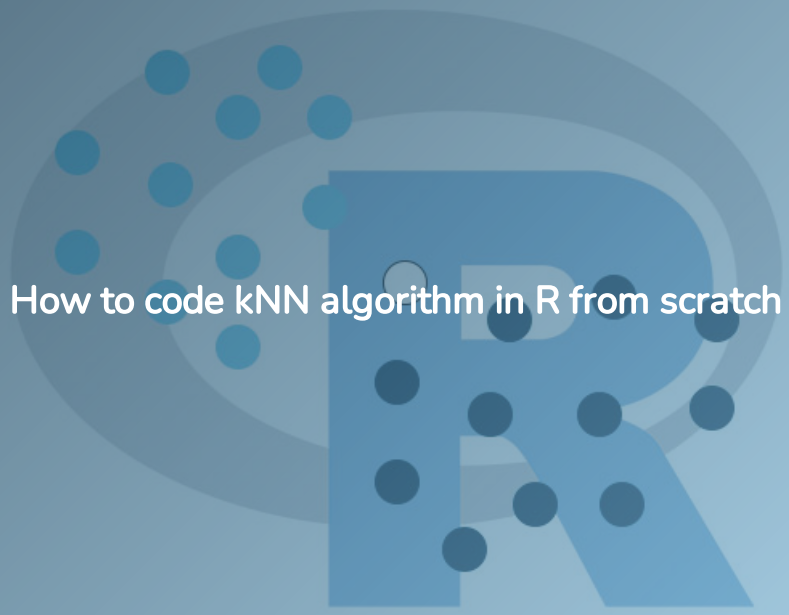


How to code kNN algorithm in R from scratch

The kNN algorithm is one of the most known algorithms in the world of machine learning, widely used, among other things, in the imputation of missing values. Today we are going to code a kNN algorithm from scratch in R so that you understand perfectly how it works in detail and how you should use it. Go for it!

Introduction to k Nearest Neighbors (kNN) in R

The idea behind the kNN algorithm is very simple: I save the training data table and when new data arrives, I find the *k* closest neighbors (observations), and I make the prediction based on the observations that are close to the new one. After all, you would expect that observations close to the new observation will have similar target values.

As you can see, here we see a big difference with respect to most supervised algorithms and that is that **kNN is a non-parametric algorithm**. In other words, while for most algorithms you need to find some parameters (such as beta in linear and logistic regression or W in neural nets), in the case of kNN you don't have to find any value. Thus, kNN algorithm is not trained.

Moreover, the key to the kNN algorithm that we code program in R is based on three key aspects that we must know:

1. Know the different distance measures that exist, how they work and when to use each of the measures.
2. Understand how to choose the number of k neighbors to observe.
3. Know how the kNN algorithm makes predictions.

That being said, lets learn how to code kNN algorithm from scratch in R!

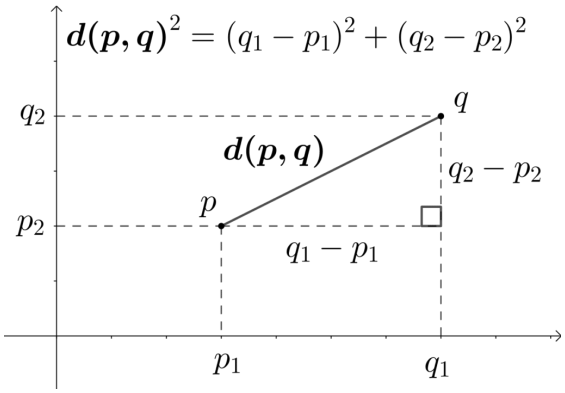
Distance measurements that the kNN algorithm can use

Within the kNN algorithm, the most used distance measures are: Euclidean distance, Minkowski distance, Manhattan distance, Cosine distance and Jaccard distance. You can use other distances, but these are the most common ones.

Euclidean distance

The Euclidean distance is something that we have already seen in this blog when programming the K-means algorithm both in R and in Python. The Euclidean distance is based on the Pythagoras theorem, according to which, the hypotenuse squared is equal to the sum of the sides squared.

This formula will work regardless of the number of variables there are and can be used to find the distance in a straight line between two points. In the following image, we can see how the distance between points *p* and *q* would be calculated.



So, since the Euclidean distance is one of the possible distance measures that the kNN algorithm can use, let's code the Euclidean distance in R:

```
euclidean_distance = function(a, b){
  #  We check that they have the same number of observation
  if(length(a) == length(b)){
    sqrt(sum((a-b)^2))
  } else{
    stop('Vectors must be of the same length')
  }
}

euclidean_distance(1:10, 11:20)
```

```
[1] 31.62278
```

Manhattan Distance

The Manhattan distance does not measure the distance measures the direct distance. Instead, it considers the distance to be the sum of the sides (considering the Pythagoras theorem). This might be a bit difficult to understand, I think it is easier to see how it works with an image:

As you can see, in this case, we are trying to measure the distance between the points (x1, y1) and (x2, y2). The blue line represents the Euclidean distance, while the red line represents the Manhattan distance, which goes block by block, following a more "natural" path.

The calculation of this distance is the result of making the sum of the sides, which can be done with the following formula:

\[Mdis = |x_2 – x_1| + |y_2 – y_1| \]

So, we are going to code the Manhattan distance from 0 into R so that we can use it in our kNN algorithm:

```
manhattan_distance = function(a, b){
  # We check that they have the same number of observation
  if(length(a) == length(b)){
    sum(abs(a-b))
  } else{
    stop('Vectors must be of the same length')
  }
}

manhattan_distance(1:10, 11:20)
```

```
[1] 100
```

Cosine similarity

As we saw in the post on how to code a recommendation system in R from scratch, the cosine similarity measures the angle between two vectors, so that we know whether or not they point in the same direction. The cosine similarity formula is calculated with the following formula:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

However, in our case, we do not want to measure the similarity, but rather the distance. The cosine similarity will be 1 when the angle between two vectors is 0, that is, they point in the same direction. In this case, the distance should be 0. Therefore, to obtain the distance based on the similarity of the cosine, we simply have to subtract the similarity from 1.

So, we can program the distance based on cosine similarity in R from 0:

```
cos_similarity = function(a,b){
  if(length(a) == length(b)){
    num = sum(a *b, na.rm = T)
    den = sqrt(sum(a^2, na.rm = T)) * sqrt(sum(b^2, na.rm = T))
    result = num/den

    return(1-result)
  } else{
    stop('Vectors must be of the same length')
  }
}

cos_similarity(1:10, 11:20)
```

```
[1] 0.0440877
```

Jaccard Coefficient

Jaccard's coefficient measures the degree of similarity between two vectors, giving a value of 1 when all values are equal and 0 when values are different.

The Jaccard coefficient can be calculated as follows:

\[J(A, B) = \frac{|A∩B|}{|A∪B|} = \frac{|A∩B|}{|A|+|B|-|A∩B|} \]

So, we are going to program the Jaccard coefficient in R to be able to use it in our kNN algorithm programmed from 0:

```
jaccard = function(a, b){
  if(length(a) == length(b)){
    intersection = length(intersect(a,b))
    union = length(a) +  length(b) - intersection
    return(intersection/union)
  } else{
```

```
      stop('Vectors must be of the same length')
    }
  }

jaccard(1:10, 11:20)
```

```
[1] 0
```

Minkowski Distance

The Minkowski distance is a type of distance that generalizes the Euclidean and Manhattan distances. Basically, the Minkowski distance is a distance that requires a p parameter, when $p = 2$ we get the Euclidean distance, and if $p = 1$ we get the Manhattan distance.

However, although the most typical values of the Minkowski distance are usually 1 and 2, it can also take other values.

The Minkowski distance formula is as follows:

$$ Minkowski = \Bigg(\sum^d_{l=1}|x_{il}-x_{jl}|^{1/p}\Bigg)^p $$

So we can program the Minkowski distance from 0 into R:

```
minkowski_distance = function(a,b,p){
  if(p<=0){
   stop('p must be higher than 0')
  }

  if(length(a)== length(b)){
    sum(abs(a-b)^p)^(1/p)
  }else{
     stop('Vectors must be of the same length')

  }
}

minkowski_distance(1:10, 11:20, 1)
```

```
[1] 100
```

```
minkowski_distance(1:10, 11:20, 2)
```

```
[1] 31.62278
```

As we can see, the results for $p = 1$ we get the Manhattan distance and with $p = 2$ they we get the Euclidean distance.

When to use each type of distance

With this you already know the main distance measurements that are usually used in the kNN algorithm. However, when should we use each of them?

Like everything in the data world, **the distance metric we use will depend on the type of data we have, the dimensions we have and the business objective**.

For example, **if we want to find the closest route that a taxi must take** or the distances on a chessboard, it seems clear that **our own data leads us to use the Manhattan function** since it is the only distance that makes sense.

Likewise, **when we have a high level of dimensionality,** that is, when there are many variables, according to this paper , the **Manhattan distance** works better than the Euclidean distance. And, with high dimensionality, everything is far from everything, so another option is usually to look at the direction of the vectors, that is, use the **cosine distance**.

On the other hand, using the **Jaccard distance or the cosine distance will depend on the duplication of the data**. If data duplication does not matter, then the Jaccard distance will be used, otherwise, the cosine distance will be used. **These distances are typically typical for data involving words (NLP) and recommender systems**.

Now that we know the different distance measures and when to use each of them, we are going to use these distances to find the k closest neighbors and, from there, see how the prediction is made.

Find the k nearest neighbors

We are going to code a function that, given a measure of distance and a series of observations, returns the *k* neighbors of an observation that we pass to it.

This is very interesting since it will allow us to use the kNN function simply to find the closest neighbors, without having to make any predictions. This is something other machine learning libraries (like scikit learn in Python) allow to do and is very useful for building recommendation systems.

In any case, the function to find the nearest neighbors will work as follows:

- Calculate the distance of the observation with respect to all the observations.
- Filter and return the k observations with the smallest distance.

Therefore, we are going to code this very important part of our kNN algorithm in R. To make it easily fit with the rest of the code, instead of returning a dataset, we will return the indices of the observations.

```
nearest_neighbors = function(x,obs, k, FUN, p = NULL){

  # Check the number of observations is the same
  if(ncol(x) != ncol(obs)){
    stop('Data must have the same number of variables')
  }

  # Calculate distance, considering p for Minkowski
  if(is.null(p)){
    dist = apply(x,1, FUN,obs)
  }else{
    dist = apply(x,1, FUN,obs,p)
  }

  # Find closest neighbours
  distances = sort(dist)[1:k]
  neighbor_ind = which(dist %in% sort(dist)[1:k])

  if(length(neighbor_ind)!= k){
    warning(
      paste('Several variables with equal distance. Used k:',length(neighbor_ind)
    )
  }

  ret = list(neighbor_ind, distances)
  return(ret)
}
```

Now, we can check that our `nearest_neighbors` function works. To do this, we are going to use the iris dataset, from which we will use the last observation with which to find 3 closest neighbors:

```
x = iris[1:(nrow(iris)-1),]
obs = iris[nrow(iris),]

ind = nearest_neighbors(x[,1:4], obs[,1:4],4, euclidean_distance)[[1]]
as.matrix(x[ind,1:4])
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width
102          5.8         2.7          5.1         1.9
128          6.1         3.0          4.9         1.8
139          6.0         3.0          4.8         1.8
143          5.8         2.7          5.1         1.9
```

As we can see, the 3 neighbors are quite similar. In fact, if we compare it with the observation used, we see that they are very similar:

```
obs[,1:4]
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width
150          5.9           3          5.1         1.8
```

As we can see, we already know how to find the *k* nearest neighbors. However, how does the algorithm make the prediction? Let's see it!

Code the prediction of the kNN algorithm

The kNN algorithm is used for classification and regression problems. Obviously, how the prediction is done will depend on what kind of problem it is.

Prediction of the kNN algorithm in classification problems

In the case of classification problems, the kNN algorithm is based on finding the mode of the variable, as if it were a voting system. Following our case, if most of our neighbors are Iris Setosa, the algorithm's prediction will be Iris Setosa.

However, this has a problem, and that is, what happens if there are two (or more) classes with the same number of votes? In that case, there would be no predominant value, so this method would not work.

In those cases, the algorithm increases the *k* by 1, that is, it adds a new neighbor that will (probably) tiebreak. If it doesn't, we would continue increasing k by 1, until we get a tiebreak.

Considering this, let's create the prediction function for the case of categorical data:

```
knn_prediction = function(x,y){

  groups = table(x[,y])
  pred = groups[groups == max(groups)]
  return(pred)

}

knn_prediction(x[ind,], 'Species')
```

```
virginica
         4
```

Thus, we see that the kNN algorithm predict that our plant is a Virginica. We check to see if the prediction is right or not:

```
obs[,'Species']
```

```
[1] virginica
Levels: setosa versicolor virginica
```

Indeed, we see that the algorithm has made the prediction correct. Anyway, we still haven't fixed the problem that there are two classes with the same number of neighbors. We will fix this later when mounting the algorithm.

Having seen the prediction in the case of classification, we can see how the kNN algorithm makes the prediction in the case of a regression:

Prediction of the kNN algorithm in regression problems

In the case of the kNN algorithm for regressions, we can choose two approaches:

1. Make the prediction based on the mean, as is done in the case of decision trees, as we saw in this post .
2. Make the prediction based on a weighted average that has Take into account the distance of the rest of the observations with respect to *target* , in such a way that those observations that are closer have more weight than those that are further away.

There are different ways to calculate the weighted average (link), although the most common is to weight based on the inverse of distances.

So, let's modify the prediction function above so that:

1. Take into account the type of variable on which the prediction is made.
2. In case of a regression , accept that the prediction is made by both a simple mean and a weighted mean.

```
knn_prediction = function(x,y, weights = NULL){

  x = as.matrix(x)

  if(is.factor(x[,y]) | is.character(x[,y])){
    groups = table(x[,y])
    pred = names(groups[groups == max(groups)])
  }

  if(is.numeric(x[,y])){

    # Calculate weighted prediction
    if(!is.null(weights)){
      w = 1/weights/ sum(weights)
      pred = weighted.mean(x[,y], w)

    # Calculate standard prediction
    }else{
      pred = mean(x[,y])
    }

  }

  # If no pred, then class is not correct
  if(try(class(x[,y])) == 'try-error'){
    stop('Y should be factor or numeric.')
  }

  return(pred)

}
```

Now, let's try to make the prediction. To do this, we are going to use the `gapminder` dataset, with which we will try to predict the life expectancy of a country for 2007, using its population and its GDP per Capita:

```
cat('Prediction:', pred,'\n',    'Valor real: ', obs$lifeExp,    sep ='')
```

```
Prediction:54.5806
Valor real: 43.487
```

As we can see, we have obtained a prediction that is quite close to what we could have expected. We check the same, but in the case that we had done a weighted kNN:

```
pred = knn_prediction(x[neighbors[[1]],],'lifeExp',weights = neighbors[[2]])

cat('Prediction:', pred,'\n',
    'Valor real: ', obs$lifeExp,
    sep ='')
```

```
Prediction:51.67521
Valor real: 43.487
```

As we can see, the prediction has improved a bit. This is because the closest observations are the ones that have had the most weight and, from what it seems, are the ones that best predict (which makes sense and, precisely for that reason, the kNN algorithm that we just programmed in R).

With this, we would already know how the kNN algorithm makes the prediction. Now, we are going to put everything together so that we finish programming our algorithm and that using kNN is simply calling a function. Let's get to it!

Finishing coding the kNN algorithm from 0 in R

In order to finish programming our kNN algorithm in R, we must take into account how this algorithm is used. In general, the prediction is not usually made on one observation, but on several at the same time. Therefore, we will have to allow our algorithm to receive several observations on which to predict and to return a vector of predictions.

With what we have done so far I am very simple, since we will simply have to iterate over the input data to and make an *append < / em> from the results.*

Also, we are going to program another question that we have previously left in the pipeline, and that is when the prediction is made in a classification problem, there is no single class with more votes than the others, but there is a tie between two or more classes.

As we have the algorithm programmed, solving that is very simple if we apply recursion. So when you get a prediction you just have to check the number of response classes. If it is higher than one, for that same observation you call the algorithm again, but with a higher k.

Let's see how to do it:

```
knn = function(x_fit, x_pred, y, k,
               func = euclidean_distance,weighted_pred = F, p = NULL){

  # Inicilizamos las predicciones
  predictions = c()

  y_ind = which(colnames(x_pred) == y)

  # Para cada observaciones, obtenemos la prediccion
  for(i in 1:nrow(x_pred)){

    neighbors = nearest_neighbors(x_fit[,-y_ind],
                                  x_pred[i,-y_ind],k,FUN = func)

    if(weighted_pred){
      pred = knn_prediction(x_fit[neighbors[[1]], ],y, neighbors[[2]])
    } else{
      pred = knn_prediction(x_fit[neighbors[[1]], ],y)
    }

    # If more than 1 predictions, make prediction with 1 more k
    if(length(pred)>1){
      pred = knn(x_fit, x_pred[i,],y, k = k+1,
                 func = func, weighted_pred = weighted_pred, p == p)
    }

    predictions[i] = pred

  }
  return(predictions)

}
```

As you can see, setting up the function has been very simple. Let's see how it works. In order to measure the performance in the classification, I will use the iris data, but in this case, sampled.

```
set.seed(1234)

n_fit = 20
train_ind = sample(1:nrow(iris),n_fit)

x_fit = iris[-train_ind,]
x_pred = iris[train_ind,]

predictions = knn(x_fit, x_pred, 'Species', k = 5)
predictions
```

```
[1] 'setosa'     'versicolor' 'virginica'  'virginica'  'virginica'  'virginica'
 [9] 'versicolor' 'virginica'  'versicolor' 'versicolor' 'versicolor' 'virginica'
[17] 'versicolor' 'setosa'     'virginica'  'setosa'
```

Now, we can compare the predictions with the real data, to see how the algorithm has behaved. Being a classification problem, we will make a confusion matrix, for which we will use the confusionMatrix function of `caret`.

```
library(caret)

predictions = factor(predictions, levels = levels(x_pred$Species))

confusionMatrix(as.factor(predictions), x_pred$Species)
```

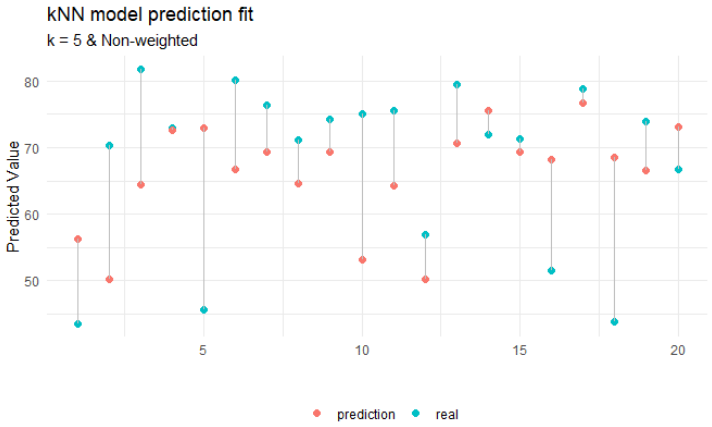
```
Confusion Matrix and Statistics

            Reference
Prediction   setosa versicolor virginica
  setosa          4          0         0
  versicolor      0          6         0
  virginica       0          0        10

Overall Statistics

               Accuracy : 1
                 95% CI : (0.8316, 1)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 9.537e-07

                  Kappa : 1

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                    1.0               1.0              1.0
Specificity                    1.0               1.0              1.0
Pos Pred Value                 1.0               1.0              1.0
Neg Pred Value                 1.0               1.0              1.0
Prevalence                     0.2               0.3              0.5
Detection Rate                 0.2               0.3              0.5
Detection Prevalence           0.2               0.3              0.5
Balanced Accuracy              1.0               1.0              1.0
```

As we can see, our algorithm is 100% correct, although it is true that it is a simple case that is particularly good at to this algorithm, but hey, at least it helps us to check that the algorithm works.

In addition, we can also check how our algorithm works when solving prediction problems. To do this, we will use the `gapminder` dataset that we have used previously. To evaluate it, we will use both RMSE and a visualization of the error using `ggplot`.

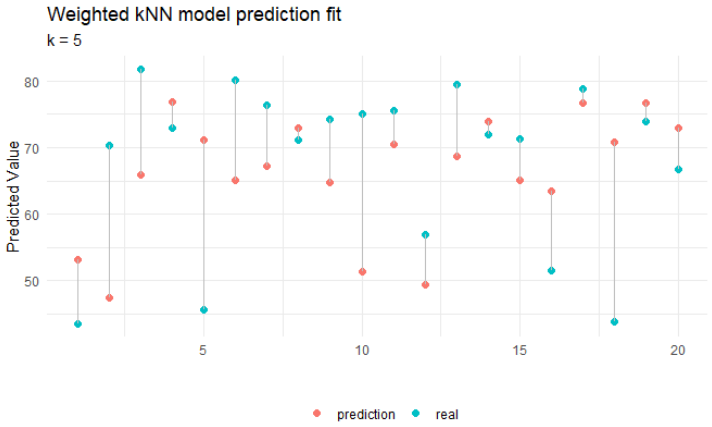
```
library(ggplot2)
library(dplyr)
library(tidyr)

set.seed(12345)

n_fit = 20
train_ind = sample(1:nrow(gapminder_2007),n_fit)

x_fit = gapminder_2007[-train_ind, 3:6]
x_pred = gapminder_2007[train_ind, 3:6]

predictions = knn(x_fit, x_pred, 'lifeExp', k = 5)

results = data.frame(real = x_pred$lifeExp,
                     prediction = predictions)
```

```
results %>%
  mutate(id = row_number()) %>%
  pivot_longer(cols = -c('id'), names_to = 'variable', values_to = 'valor') %>%
  ggplot(aes(id, valor, col = variable)) +
  geom_point(size = 2) +  geom_line(aes(group = id),color='grey') +
  theme_minimal() + theme(legend.position = 'bottom') +
  labs(x = '', y = 'Predicted Value', col = '',
       title = 'kNN model prediction fit', subtitle = 'k = 5 & Non-weighted')
```



As we can see, the prediction seems to be quite correct in some cases, although very remote in others. Let's now see how the weighted kNN behaves, so that we can see if there is any significant variation:

```
predictions_w = knn(x_fit, x_pred, 'lifeExp', k = 5, weighted_pred = T)

results_w = data.frame(real = x_pred$lifeExp,
                       prediction = predictions_w)
results_w %>%
  mutate(id = row_number()) %>%
  pivot_longer(cols = -c('id'),
               names_to = 'variable',
               values_to = 'valor') %>%
  ggplot(aes(id, valor, col = variable)) +
  geom_point(size = 2) +  geom_line(aes(group = id),color='grey') +
  theme_minimal() + theme(legend.position = 'bottom') +
  labs(x = '', y = 'Predicted Value', col = '',
       title = 'Weighted kNN model prediction fit', subtitle = 'k = 5')
```



As we can see, the predictions seem somewhat minor. In any case, in this case we can use the RMSE to compare both models:

```
rmse = function(y_pred, y_real){
  sqrt(mean((y_pred-y_real)^2))
}
```

```
rmse_model1 = rmse(results$prediction, results$real)
rmse_model2 = rmse(results_w$prediction, results_w$real)

cat(
  'RMSE kNN: ', rmse_model1,'\n',
  'RMSE weighted kNN: ', rmse_model2, sep=''
)
```

```
RMSE kNN: 13.47053
RMSE weighted kNN: 13.54268
```

As we can see, in this case the weighted kNN has had a little bit than the normal kNN. But, could it be because we are not applying the algorithm well? And, so far we have seen how it is programmed from 0 in R. And yes, we already know how it works. But .. how do you choose the number of k, for example? In which cases should we use the kNN algorithm? Let's see it!

How and when to use the kNN algorithm

Choice of the number of neighbors to check

One of the most important questions about the kNN algorithm is how many neighbors should we check. As a general rule, there are two main problems when choosing k:

- If the k is too low, the algorithm will use too few neighbors, which will tend to *overfitting*.
- If the k is too high, the prediction will increasingly tend to resemble the mean, so you will have an *underfitting*.

As a general rule, the *k* is usually chosen using the following formula:

$$ k = \sqrt{n} $$

Out of curiosity, we are going to check what the RMSE of our model would be by adjusting the *k* in this way in the case of regression:

```
k = round(sqrt(nrow(x_fit)))

# Make predictions
predictions = knn(x_fit, x_pred, 'lifeExp', k )
predictions_w = knn(x_fit, x_pred, 'lifeExp', k , weighted_pred = T)

# Get RMSE for both cases
rmse_model1 = rmse(predictions, x_pred$lifeExp)
rmse_model2 = rmse(predictions_w, x_pred$lifeExp)


cat(
  '-- k correctly selected --','\n',
  'selected value of k: ', k,'\n',
  'RMSE kNN: ', rmse_model1,'\n',
  'RMSE weighted kNN: ', rmse_model2, sep=''
)
```

```
-- k correctly selected --
selected value of k: 11
RMSE kNN: 12.15191
RMSE weighted kNN: 12.69471
```

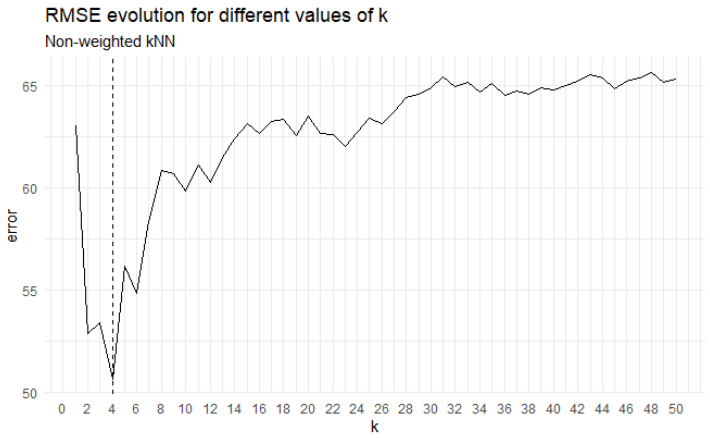
As we can see, choosing *k* correctly, our predictions have improved significantly by reducing RMSE in 1 point.

Although this rule is quite simple and fast, another way to choose the number of ks is by iterating the model with different ks. In this way, we can ensure that the k we choose is the one that maximizes the results (at least for that dataset).

Let's see how it's done:

```
n_iterations =  50

errors = c()
```

```
for(i in 1:n_iterations){
  prediction = knn(x_fit, x_pred, 'lifeExp', k = i)
  errors[i] = prediction
  if(i%%10==0){print(i)}
}

error = data.frame(k = c(1:n_iterations), error = errors)

error %>%
  ggplot(aes(k,error)) + geom_line() +
  geom_vline(xintercept = 4, linetype = 'dashed') +
  theme_minimal() +
  scale_x_continuous(breaks = seq(0,n_iterations,2)) +
  labs(title = 'RMSE evolution for different values of k',
       subtitle = 'Non-weighted kNN'
       )
```



As we can see, with k = 4 we get the least amount of RMSE. Before that, the prediction is suffering from *overfitting* and with k> 4, we predict worse and worse until k= 8 when the model stops generalizing and starts to suffer from *underfitting*.

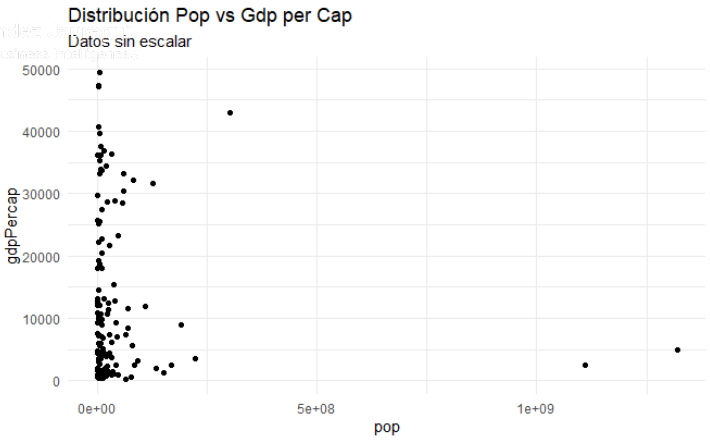
However, the downside of obtaining the number of *k* in this way is that it is computationally very expensive, which makes lose one of the kNN algorithm values.

Normalization of the data

As we saw with the k-Means algorithm, for the kNN algorithm to work properly we must normalize the data on which we are going to apply it.

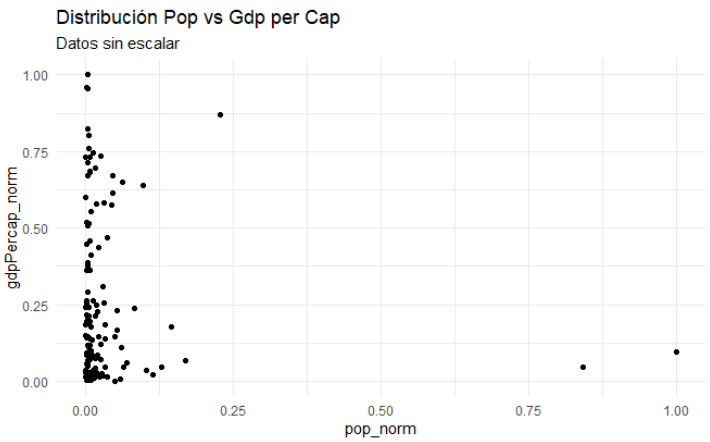
The reason is that if we have two variables with very different scales (see population and GDP per Capita), the variable with higher values will be the one that determines the distance. This means that, in practice, instead of the algorithm taking both variables into account, it is focusing only on one variable to make the prediction.

```
gapminder_2007 %>%
  ggplot(aes(pop, gdpPercap)) + geom_point() +
  theme_minimal() +
  labs(title = 'Distribución Pop vs Gdp per Cap',
       subtitle = 'Datos sin escalar')
```



If we scale the data, we will see how finding the closest neighbors will be easier:

```
normalization = function(x){
  (x - min(x))/(max(x) - min(x))
}

gapminder_2007$pop_norm = normalization(gapminder_2007$pop)
gapminder_2007$gdpPercap_norm = normalization(gapminder_2007$gdpPercap)

gapminder_2007 %>%
  ggplot(aes(pop_norm, gdpPercap_norm)) + geom_point() +
  theme_minimal() +
  labs(title = 'Distribución Pop vs Gdp per Cap',
       subtitle = 'Datos sin escalar')
```



The data seems to have not changed (due to the outliers), but if we check the RMSE we will see how it has been significantly reduced:

```
x_fit = gapminder_2007[-train_ind, c(4,7,8)]
x_pred = gapminder_2007[train_ind, c(4,7,8)]

prediction = knn(x_fit, x_pred, 'lifeExp', k = 11)

cat(
  'RMSE kNN not normalized data: ', rmse_model1,'\n',
  'RMSE kNN normalized data: ', rmse(prediction, x_pred$lifeExp), sep=''
)
```

```
RMSE kNN not normalized data: 12.15191
RMSE kNN normalized data: 7.510772
```

As we can see, the prediction has improved a lot and this is simply due to the fact of having normalized the data.

Use categorical variables in kNN

If you have noticed, in the examples used so far I have only used numeric variables. And, in order to use categorical variables in the kNN algorithm that we have just programmed in R, there are two options:

- Convert categories into numeric variables, applying techniques such as *one-hot encoding* (typical of neural networks).
- Use a distance measure that does allow you to work with categorical variables. And it is that, while some distance measures (such as the Euclidean distance or the Manhattan distance) only admit numerical variables, other distance measures from Hamming or the distance from Gower .

Conclusion

Now that we know everything there is to know about the kNN algorithm: we have programmed it from 0 in R, the different distance measurements it can use and when to use each of them, how to use it correctly, how to choose the k and we know how to make it use categorical variables.

I hope this post has been useful to you. If you would like to continue learning about different algorithms in R, I would recommend you subscribe to the newsletter to be notified each time I upload a new post. See you next time!