**The IRIS Data Set**

Lets plot the data before rushing into classification so that we can have a deeper understanding of the problem at hand. R has a beautiful visualization tool called ggplot2 that we will use to create 2 quick scatter plots of **sepal width vs sepal length** and **petal width vs petal length**.

*# ============================== R code ==============================*

*# loading packages*

library(ggplot2)

library(magrittr)

*# sepal width vs. sepal length*

iris **%>%**

ggplot(aes(x**=**Sepal.Length, y**=**Sepal.Width, color**=**Species)) **+**

geom\_point()

*# petal width vs. petal length*

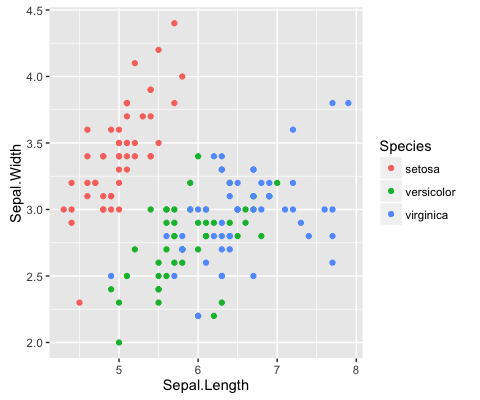
iris **%>%**

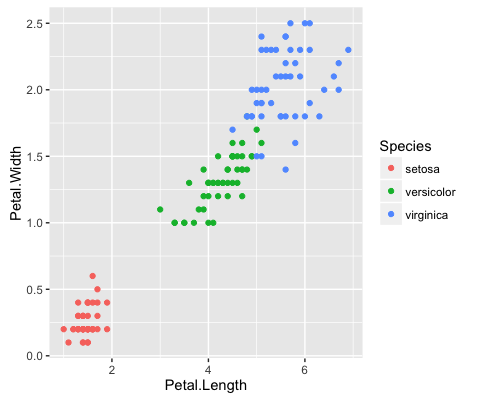
ggplot(aes(x**=**Petal.Length, y**=**Petal.Width, color**=**Species)) **+**

geom\_point()

*# =====================================================================*

Note that we’ve accessed the iris dataframe which comes preloaded in R by default.





A quick study of the above graphs reveals some strong classification criterion. We observe that setosas have small petals, versicolor have medium sized petals and virginica have the largest petals. Furthermore, setosas seem to have shorter and wider sepals than the other two classes. Pretty interesting right? Without even using an algorithm, we’ve managed to intuitively construct a classifier that can perform pretty well on the dataset.

Let’s not remain on this high-level overview of the data! R gives you the opportunity to go more in-depth with the summary() function. This will give you the minimum value, first quantile, median, mean, third quantile and maximum value of the data set Iris for numeric data types. For the class variable, the count of factors will be returned:

# Summary overview of `iris`

summary(iris)

# Refined summary overview

summary(iris[c("Petal.Width", "Sepal.Width")])

Many of the algorithms used in machine learning are not incorporated by default into R. You will most probably need to download the packages that you want to use when you want to get started with machine learning.

**The KNN algorithm**

This tutorial works with the **package class**:

library(class)

**Normalization**

As a part of your data preparation, you might need to normalize your data so that its consistent. For this introductory tutorial, just remember that normalization makes it easier for the KNN algorithm to learn. So when do you need to normalize your dataset?

**In short: when you suspect that the data is not consistent.**

You can easily see this when you go through the results of the summary() function. Look at the minimum and maximum values of all the (numerical) attributes. If you see that one attribute has a wide range of values, you will need to normalize your dataset, because this means that the distance will be dominated by this feature.

For example, if your dataset has just two attributes, X and Y, and X has values that range from 1 to 1000, while Y has values that only go from 1 to 100, then Y’s influence on the distance function will usually be overpowered by X’s influence.

When you normalize, you actually adjust the range of all features, so that distances between variables with larger ranges will not be over-emphasised.

Tip: go back to the result of summary(iris) and try to figure out if normalization is necessary.

The Iris data set doesn’t need to be normalized: the Sepal.Length attribute has values that go from 4.3 to 7.9 and Sepal.Width contains values from 2 to 4.4, while Petal.Length’s values range from 1 to 6.9 and Petal.Width goes from 0.1 to 2.5.

All values of all attributes are contained within the range of 0.1 and 7.9, which you can consider acceptable.

Nevertheless, it’s still a good idea to study normalization and its effect, especially if you’re new to machine learning. You can perform feature normalization, for example, by first making your own **normalize()** function.

You can then use this argument in another command, where you put the results of the normalization in a data frame through as.data.frame() after the function lapply() returns a list of the same length as the data set that you give in. Each element of that list is the result of the application of the normalize argument to the data set that served as input:

# Build your own `normalize()` function

normalize <- function(x) {

num <- x - min(x)

denom <- max(x) - min(x)

return (num/denom)

}

# Normalize the `iris` data

iris\_norm <- as.data.frame(lapply(iris[1:4], normalize))

# Summarize `iris\_norm`

summary(iris\_norm)

**Training And Test Sets**

In order to assess your model’s performance later, you will need to divide the data set into two parts: a training set and a test set.

The first is used to train the system, while the second is used to evaluate the learned or trained system. In practice, the division of your data set into a test and a training sets is disjoint: the most common splitting choice is to take 2/3 of your original data set as the training set, while the 1/3 that remains will compose the test set.

One last look on the data set teaches you that if you performed the division of both sets on the data set as is, you would get a training class with all species of “Setosa” and “Versicolor”, but none of “Virginica”. The model would therefore classify all unknown instances as either “Setosa” or “Versicolor”, as it would not be aware of the presence of a third species of flowers in the data.In short, you would get incorrect predictions for the test set.

You thus need to make sure that all three classes of species are present in the training model. What’s more, the amount of instances of all three species needs to be more or less equal so that you do not favour one or the other class in your predictions. To make your training and test sets, you first set a seed. This is a number of R’s random number generator. The major advantage of setting a seed is that you can get the same sequence of random numbers whenever you supply the same seed in the random number generator.

**set.seed(1234)**

Then, you want to make sure that your Iris data set is shuffled and that you have an equal amount of each species in your training and test sets.

You use the sample() function to take a sample with a size that is set as the number of rows of the Iris data set, or 150. You sample with replacement: you choose from a vector of 2 elements and assign either 1 or 2 to the 150 rows of the Iris data set. The assignment of the elements is subject to probability weights of 0.67 and 0.33.

**ind <- sample(2, nrow(iris), replace=TRUE, prob=c(0.67, 0.33))**

Remember that you want your training set to be 2/3 of your original data set: that is why you assign “1” with a probability of 0.67 and the “2”s with a probability of 0.33 to the 150 sample rows.You can then use the sample that is stored in the variable ind to define your training and test sets:

# Compose training set

iris.training <- iris[ind==1, 1:4]

# Inspect training set

head(iris.training)

# Compose test set

iris.test <- iris[ind==2, 1:4]

# Inspect test set

head(iris.test)

Note that, in addition to the 2/3 and 1/3 proportions specified above, you don’t take into account all attributes to form the training and test sets. Specifically, you only take Sepal.Length, Sepal.Width, Petal.Length and Petal.Width. This is because you actually want to predict the fifth attribute, Species: it is your target variable. However, you do want to include it into the KNN algorithm, otherwise there will never be any prediction for it.

You therefore need to store the class labels in factor vectors and divide them over the training and test sets:

# Compose `iris` training labels

iris.trainLabels <- iris[ind==1,5]

# Inspect result

print(iris.trainLabels)

# Compose `iris` test labels

iris.testLabels <- iris[ind==2, 5]

# Inspect result

print(iris.testLabels)

**Building Your KNN Classifier**

After all these preparation steps, you have made sure that all your known (training) data is stored. No actual model or learning was performed up until this moment. Now, you want to find the k nearest neighbors of your training set.

An easy way to do these two steps is by using the knn() function, which uses the Euclidian distance measure in order to find the k-nearest neighbours to your new, unknown instance. Here, the k parameter is one that you set yourself.

As mentioned before, new instances are classified by looking at the majority vote or weighted vote. In case of classification, the data point with the highest score wins the battle and the unknown instance receives the label of that winning data point. If there is an equal amount of winners, the classification happens randomly.

Note: the k parameter is often an odd number to avoid ties in the voting scores.

To build your classifier, you need to take the knn() function and simply add some arguments to it, just like in this example**:**

# Build the model

iris\_pred <- knn(train = iris.training, test = iris.test, cl = iris.trainLabels, k=3)

# Inspect `iris\_pred`

iris\_pred

You store into iris\_pred the knn() function that takes as arguments the training set, the test set, the train labels and the amount of neighbours you want to find with this algorithm. The result of this function is a factor vector with the predicted classes for each row of the test data.Note that you don’t want to insert the test labels: these will be used to see if your model is good at predicting the actual classes of your instances!

**KNN-Evaluation**

An essential next step in machine learning is the evaluation of your model’s performance. In other words, you want to analyze the degree of correctness of the model’s predictions.For a more abstract view, you can just compare the results of iris\_pred to the test labels that you had defined earlier:

# Put `iris.testLabels` in a data frame

irisTestLabels <- data.frame(iris.testLabels)

# Merge `iris\_pred` and `iris.testLabels`

merge <- data.frame(iris\_pred, iris.testLabels)

# Specify column names for `merge`

names(merge) <- c("Predicted Species", "Observed Species")

# Inspect `merge`

merge

**Machine Learning-KNN in R with caret Package**

In the previous section, you have gotten started with supervised learning in R via the KNN algorithm. Machine learning in R can get really complex, as there are various algorithms with various syntax, different parameters, etc.

That’s where the caret package can come in handy: it’s short for “Classification and Regression Training” and offers everything you need to know to solve supervised machine learning problems: it provides a uniform interface to a ton of machine learning algorithms. If you’re a bit familiar with **Python** machine learning, you might see similarities with **scikit-learn.**

You already know what’s next! Let’s split up the data in a training and test set. In this case, though, you handle things a little bit differently: you split up the data based on the labels that you find in iris$Species. Also, the ratio is in this case set at 75-25 for the training and test sets.

Create index to split based on labels

index <- createDataPartition(iris$Species, p=0.75, list=FALSE)

# Subset training set with index

iris.training <- iris[index,]

# Subset test set with index

iris.test <- iris[-index,]

Caret is an extremely large project that includes a lot of algorithms. If you’re in doubt on what algorithms are included in the project, you can get a list of all of them. Pull up the list by running names(getModelInfo()), just like the code chunk below demonstrates. Next, pick an algorithm and train a model with the train() function:

# Overview of algorithms supported by caret

names(getModelInfo())

# Train a model

model\_knn <- train(iris.training[, 1:4], iris.training[, 5], method='knn')

***Note that making other models is extremely simple when you have gotten this far; You just have to change the method argument.***

Now that you have trained your model, it’s time to predict the labels of the test set that you have just made and evaluate how the model has done on your data:

# Predict the labels of the test set

predictions<-predict.train(object=model\_knn,iris.test[,1:4], type="raw")

# Evaluate the predictions

table(predictions)

# Confusion matrix

confusionMatrix(predictions,iris.test[,5])

Additionally, you can try to do the same test as before, to examine the effect of preprocessing, such as **scaling and centering**, on your model

# Train the model with preprocessing

model\_knn <- train(iris.training[, 1:4], iris.training[, 5], method='knn', preProcess=c("center", "scale"))

# Predict values

predictions<-predict.train(object=model\_knn,iris.test[,1:4], type="raw")

# Confusion matrix

confusionMatrix(predictions,iris.test[,5])