KNN

Albert Mata

25/12/2018

Contents

Lazy Learning - Classification Using Nearest Neighbors

Things that are alike are likely to have properties that are alike. Nearest neighbor classifiers classify unlabeled examples by assigning them the class of similar labeled examples. They are well-suited for classification tasks, where relationships among the features and the target classes are numerous, complicated, or extremely difficult to understand, yet the items of similar class type tend to be fairly homogeneous. However, if the data is noisy and thus no clear distinction exists among the groups, the nearest neighbor algorithms may struggle to identify the class boundaries.

Classification algorithms based on the nearest neighbor methods are considered **lazy learning** algorithms because, technically speaking, no abstraction occurs. The abstraction and generalization processes are skipped altogether. A lazy learner is not really learning anything. The training phase doesn't train anything and so is really quick. The downside is that the process of making predictions tends to be relatively slow in comparison to training. Due to the heavy reliance on the training instances rather than an abstracted model, lazy learning is also known as **instance-based learning** or **rote learning**.

The method is said to be in a class of **non-parametric** learning methods, as no parameters are learned about the data. Without generating theories about the underlying data, non-parametric methods limit our ability to understand how the classifier is using the data. On the other hand, this allows the learner to find natural patterns rather than trying to fit the data into a preconceived and potentially biased functional form.

The k-NN algorithm

The letter k is a variable term implying that any number of nearest neighbors could be used. After choosing k, the algorithm requires a training dataset made up of examples that have been classified into several categories and labeled by a nominal variable. Then, for each unlabeled record in the test dataset, k-NN identifies k records in the training data that are the "nearest" in similarity. The unlabeled test instance is assigned the class of the majority of the k nearest neighbors.

Strengths:

- Simple and effective.
- Makes no assumptions about the underlying data distribution.
- Fast training phase.

Weaknesses:

- Does not produce a model, limiting the ability to understand how the features are related to the class.
- Requires selection of an appropriate k.

- Slow classification phase.
- Nominal features and missing data require additional processing.

Locating an element's nearest neighbors requires a distance function, or a formula that measures the similarity between two instances. There are many different ways to calculate this distance, but the k-NN algorithm often uses **Euclidean distance**, which is the distance one would measure if it were possible to use a ruler to connect two points by the shortest direct route.

$$dist(p, q) = \sqrt{(p_1 - q_1)^2 + ... + (p_n - q_n)^2}$$

En el fons ve a ser aplicar el Teorema de Pitàgores.

Choosing an appropriate *k*

Once we have calculated all distances between the element and nearest neighbors, we need to **choose an appropriate *k***. With $k = 1$ (1-NN classification) the element is assigned the class of its single nearest neighbor. With $k = 3$, a vote among the three nearest neighbors is performed.

The decision of how many neighbors to use for k-NN determines how well the model will generalize to future data. The balance between overfitting and underfitting the training data is a problem known as **bias-variance trade-off**. Choosing a large *k* reduces the impact or variance caused by noisy data, but can bias the learner so that it runs the risk of ignoring small, but important patterns: with *k* equal to the number of observations, the model would always predict the majority class, regardless of the nearest neighbors. On the opposite extreme, using a single nearest neighbor allows the noisy data or outliers to excessively influence the classification of examples.

In practice, choosing *k* depends on the difficulty of the concept to be learned, and the number of records in the training data. There are different approaches:

1. Begin with *k* equal to the square root of the number of training examples.
2. Test several *k* values on a variety of test datasets and choose the one that delivers the best classification performance.
3. Choose a larger *k* and apply a weighted voting process in which the vote of the closer neighbors is considered more authoritative than the vote of the far away neighbors.

But, unless the data is very noisy, a large training dataset can make the choice of *k* less important, as even subtle concepts will have a sufficiently large pool of examples to vote as nearest neighbors.

Preparing data for use with k-NN

If certain features have a much larger range of values than the others, the distance measurements would be strongly dominated by the features with larger ranges. In consequence, prior transformation to a standard range is mandatory.

The traditional method of rescaling features for k-NN is **min-max normalization** (all features falling in a 0 to 1 range):

$$X_{new} = \frac{X - min(X)}{max(X) - min(X)}$$

Another common transformation is called **z-score standardization** (how many standard deviations each feature falls above or below the mean value):

$$X_{new} = \frac{X - \mu}{\sigma} = \frac{X - Mean(X)}{StdDev(X)}$$

Future examples may have different ranges and normal distributions. So, we can either use min-max normalization with expected minimum and maximum values rather than observed ones or we can use z-score standardization assuming normal distributions won't be so different.

For nominal data, we need to convert them into a numeric format. A typical solution utilizes **dummy coding** (0 for one category and 1 for the other): `gender` becomes `male` with values 1 or 0; `temperature` (hot, medium or cold) becomes `hot` and `medium` with values 1 or 0 (both set to 0 for cold). For ordinal nominal data, it's also possible to number the categories and apply normalization, but this should be done only when the steps between groups are equal.

Example - diagnosing breast cancer with the k-NN algorithm

Step 1 - collecting data

We'll take the data from Breast Cancer Wisconsin (Diagnostic) (Dheeru and Karra Taniskidou 2017).

The breast cancer data includes 569 examples of cancer biopsies, each with 32 features:

- An identification number.
- The cancer diagnosis ("M" for malignant, "B" for benign).
- 30 numeric-valued laboratory measurements with mean, standard error and worst (largest) value for 10 different characteristics of the digitized cell nuclei:
  - Radius
  - Texture
  - Perimeter
  - Area
  - Smoothness
  - Compactness
  - Concavity
  - Concave points
  - Symmetry

– Fractal dimension

Step 2 - exploring and preparing the data

```
wbcd <- read.csv("wisc_bc_data.csv", stringsAsFactors = FALSE)
str(wbcd, vec.len = 3, list.len = 6)
```

```
## 'data.frame':    569 obs. of  32 variables:
##  $ id               : int  87139402 8910251 905520 868871 9012568 906539 925291 87880 ...
##  $ diagnosis        : chr  "B" "B" "B" ...
##  $ radius_mean      : num  12.3 10.6 11 11.3 ...
##  $ texture_mean     : num  12.4 18.9 16.8 13.4 ...
##  $ perimeter_mean   : num  78.8 69.3 70.9 73 ...
##  $ area_mean        : num  464 346 373 385 ...
##   [list output truncated]
```

The first variable is an integer variable named `id`. As this is simply a unique identifier (ID) for each patient in the data, it does not provide useful information, and we will need to exclude it from the model:

```
wbcd <- wbcd[-1]
```

Next variable, `diagnosis`, is of particular interest as it is the outcome we hope to predict. Many R machine learning classifiers require that the target feature is coded as a factor:

```
wbcd$diagnosis <- factor(wbcd$diagnosis,
                         levels = c("B", "M"),
                         labels = c("Benign", "Malignant"))
round(prop.table(table(wbcd$diagnosis)) * 100, digits = 1)
```

```
##
##    Benign Malignant
##      62.7      37.3
```

Now we need to normalize all numeric data:

```
normalize <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}

wbcd_n <- as.data.frame(lapply(wbcd[2:31], normalize))
summary(wbcd_n$area_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.1174  0.1729  0.2169  0.2711  1.0000
```

Finally, we need to create training and test datasets. So we divide our data into two portions: a training dataset that will be used to build the k-NN model (first 469 records) and a test dataset that will be used to estimate the predictive accuracy of the model (remaining 100 records):

```
# These include no diagnosis value.
wbcd_train <- wbcd_n[1:469, ]
wbcd_test  <- wbcd_n[470:569, ]

# These are the diagnosis values.
wbcd_train_labels <- wbcd[1:469, 1]
wbcd_test_labels  <- wbcd[470:569, 1]
```

Els registres d'aquest dataset estaven intencionadament desordenats. Si no hagués estat així i seguissin algun ordre específic, els subsets que acabem de fer no serien vàlids.

Step 3 - training a model on the data

As we said at the beginning of this chapter, the training phase for the k-NN algorithm actually involves no model building; the process of training a lazy learner like k-NN simply involves storing the input data in a structured format.

Training and classification using the `knn()` function from `class` package is performed in a single function call, which just requires the data structures we already have and a value for k. As our training data includes 469 instances, we might try $k = 21$, an odd number roughly equal to the square root of 469:

```
library(class)
wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                      cl = wbcd_train_labels, k = 21)
```

Step 4 - evaluating model performance

The next step of the process is to evaluate how well the predicted classes in the `wbcd_ test_pred` vector match up with the known values in the `wbcd_test_labels` vector. We do that using the `CrossTable()` function in the `gmodels` package:

```
library(gmodels)
CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
           prop.chisq = FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                  | wbcd_test_pred
## wbcd_test_labels |    Benign | Malignant | Row Total |
## -----------------|-----------|-----------|-----------|
##           Benign |        61 |         0 |        61 |
##                  |     1.000 |     0.000 |     0.610 |
##                  |     0.968 |     0.000 |           |
##                  |     0.610 |     0.000 |           |
## -----------------|-----------|-----------|-----------|
##        Malignant |         2 |        37 |        39 |
##                  |     0.051 |     0.949 |     0.390 |
##                  |     0.032 |     1.000 |           |
##                  |     0.020 |     0.370 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        63 |        37 |       100 |
##                  |     0.630 |     0.370 |           |
## -----------------|-----------|-----------|-----------|
##
##
```

- Top-left cell → true negative results (100%).
- Bottom-right cell → true positive results (94.9%).
- Top-right cell → false positive results (0%).
- Bottom-left cell → false negative results (5.1%).

Errors in bottom-left are the most dangerous, as they might lead a patient to believe that she is cancer-free while the disease may continue to spread.

Step 5 - improving model performance

A total of 2 out of 100 (2%) of masses were incorrectly classified by the k-NN approach. We can try to improve this and reduce the number of values that have been incorrectly classified, particularly because the errors were dangerous false negatives. First, we will employ an alternative method for rescaling our numeric features. Second, we will try several different values for *k*.

First, we're going to try **z-score standardization** using R's built-in `scale()` function, which, by default, rescales values using the z-score standardization:

```
wbcd_z <- as.data.frame(scale(wbcd[-1]))
summary(wbcd_z$area_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.4532 -0.6666 -0.2949  0.0000  0.3632  5.2459
```

The mean of a z-score standardized variable should always be zero, and the range should be fairly compact. A z-score greater than 3 or less than -3 indicates an extremely rare value. So the transformation seems to have worked. Let's go with the rest of the steps now:

```
wbcd_train <- wbcd_z[1:469, ]
wbcd_test <- wbcd_z[470:569, ]

wbcd_train_labels <- wbcd[1:469, 1]
wbcd_test_labels <- wbcd[470:569, 1]

wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                      cl = wbcd_train_labels, k = 21)

CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
           prop.chisq = FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                  | wbcd_test_pred
## wbcd_test_labels |    Benign | Malignant | Row Total |
## -----------------|-----------|-----------|-----------|
##           Benign |        61 |         0 |        61 |
##                  |     1.000 |     0.000 |     0.610 |
##                  |     0.924 |     0.000 |           |
##                  |     0.610 |     0.000 |           |
## -----------------|-----------|-----------|-----------|
##        Malignant |         5 |        34 |        39 |
##                  |     0.128 |     0.872 |     0.390 |
##                  |     0.076 |     1.000 |           |
##                  |     0.050 |     0.340 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        66 |        34 |       100 |
##                  |     0.660 |     0.340 |           |
## -----------------|-----------|-----------|-----------|
##
##
```

Unfortunately, the results of our new transformation in the table above show a slight decline in accuracy, with more dangerous false negatives.

So, instead of going with z-score standardization, we can try instead different values for *k*. If we do it, we'll see that although the classifier is never perfect, the 1-NN approach is able to avoid some of the false negatives at the expense of adding false positives. But it is important to keep in mind that it would be unwise to tailor our approach too closely to our test data. After all, a different set of 100 patient records is likely to be somewhat different from those used to measure our performance.

A Quick Introduction to K-Nearest Neighbors Algorithm (Bronhstein 2017)

KNN is **non-parametric**. Saying a technique is non-parametric means that it does not make any assumptions on the underlying data distribution. In other words, the model structure is determined from the data. If you think about it, it's pretty useful, because in the "real world", most of the data does not obey the typical theoretical assumptions made (as in linear regression models, for example). Therefore, KNN could and probably should be one of the first choices for a classification study when there is little or no prior knowledge about the distribution data.

KNN is also a **lazy algorithm** (as opposed to an eager algorithm). This means that it does not use the training data points to do any generalization. In other words, there is no explicit training phase or it is very minimal. This also means that the training phase is pretty fast. Lack of generalization means that KNN keeps all the training data. To be more exact, all (or most) the training data is needed during the testing phase.

KNN can be used for classification: the output is a class membership (predicts a class, a discrete value). An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. But it can also be used for regression: the output is the value for the object (predicts continuous values). This value is the average (or median) of the values of its k nearest neighbors.

The algorithm can be summarized as:

- A positive integer k is specified, along with a new sample.
- We select the k entries in our database which are closest to the new sample.
- We find the most common classification of these entries.
- This is the classification we give to the new sample.

A few other features of KNN:

- KNN stores the entire training dataset which it uses as its representation.
- KNN does not learn any model.
- KNN makes predictions just-in-time by calculating the similarity between an input sample and each training instance.

References

Bronhstein, Adi. 2017. “A Quick Introduction to K-Nearest Neighbors Algorithm.” https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighborsalgorithm-62214cea29c7.

Dheeru, Dua, and Efi Karra Taniskidou. 2017. “UCI Machine Learning Repository.” University of California, Irvine, School of Information; Computer Sciences. http://archive.ics.uci.edu/ml.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.