Lecture 5

Lazy Learning – Classification Using Nearest Neighbors

Classification

Classification is the problem of identifying to which set of classes a new observation (or instance) belongs on the basis of a training data set containing observations (or instances) whose class membership is known.

In general, the goal of classification is to learn a mapping from inputs x (data instances) to outputs y (classes), where $y \in \{C_1, \ldots, C_N\}$, with N being the number of classes. If N= 2, this is called **binary classification**; if N > 2, this is called **multiclass classification**. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), the classification is called **multi-label classification**. When we use the term "classification", we will mean multiclass classification with a single output (single class), unless we state otherwise.

One way to formalize classification is as function learning. The labeling function is y = f(x), it is unknown and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions. Let's denote the estimate function that we learn by f*. In general, f* might not be the same as f. For example, we can predict f(x) for some x as y* = f*(x). (We use the "*" symbol to denote an estimate.)

Classification *(cont.)*

In other words, we are trying to learn a function f*, which is an approximation of the "real" function f. Our main goal is to make predictions on novel inputs, meaning ones that we have not seen before (this is called generalization) since predicting the response on the training set is easy (we can just lookup the answer). Of course, we want our predications to be accurate, i.e., we want the function f* to be as close as possible to the "real" function f.

Instance-based learning is one of the simplest and oldest classification algorithms. In instance-based learning, there is a set of labeled data (i.e., a set of data instances with known classes). The goal of the algorithm is to assign a class (or to label) to a new data instance using the information from the labeled data set.

The **Nearest-Neighbor** algorithm employs instance-based learning and it works as follows**.** When given a new data item, the algorithm finds a data instance in its training set which is closest (as similar as possible) to the new data instance and uses the class (the label) of the closest data instance as a label for the new data instance.

In other words, if x_0 is the new data item and x_n is the closest data item in the training set, then $f^*(x_0) = f(x_n)$. Note that we have $f(x_n)$, not $f^*(x_n)$, because we know $f(x_n)$ for sure.

k-Nearest Neighbors (kNN) algorithm is an improvement of the Nearest-Neighbor algorithm, in which k (k is a preset number) closest data instances to the new data instance are found, and the class of the new data instance is determined as a function of the classes of the k closest data instances. For example, the function could be the mean, the mode, an weighted mean, etc.

Classification *(cont.)*

Instance-based learning is also called **lazy learning** because it does not do much work at learning time. At learning time, instance-based learning merely stores the training data and does not do anything else. The upside is that the training time is zero. The downside is that the whole work is postponed until the moment when a new data instance must be labeled. In other words, instance-based learning waits for a new data instance to arrive and forms a generalization just for the purpose of classifying the new data instance. In contrast, **eager learning** does not wait for a new data instance and generalizes ahead of time using the whole training data. For this reason, eager learning algorithms do the actual classification faster. The next lecture will be about the Naïve Bayes algorithm, which is an eager classifier.

Eager algorithms must generalize all data using global approximations. In contrast, lazy methods work with local data and use local approximations. As a result, lazy methods could be useful for learning complex functions.

Instance-based learning is a **non-parametric** learning method because no parameters are learned (the model does not have any parameters besides k).

Advantages and disadvantages of kNN

Advantages:

- Zero learning time
- Can learn complex functions (class mappings). That is, kNN works well when the target classes are difficult to define, but you can easily recognize them.
- Simple
- Makes no assumption about underlying data distribution
- The training data is not lost. It is always available.

Disadvantages:

- Slow classification phase. The speed of classification can grow with the size of the training data.
- Requires storage to keep all the training data (problem for large training set)
- Does not produce a model to explain how features are related to classes
- Requires selection of appropriate k
- Can be misguided by irrelevant features because similarity measures (or closeness measures) take into account all features including irrelevant ones. In other words, the nearest neighbor might not always be the most relevant.

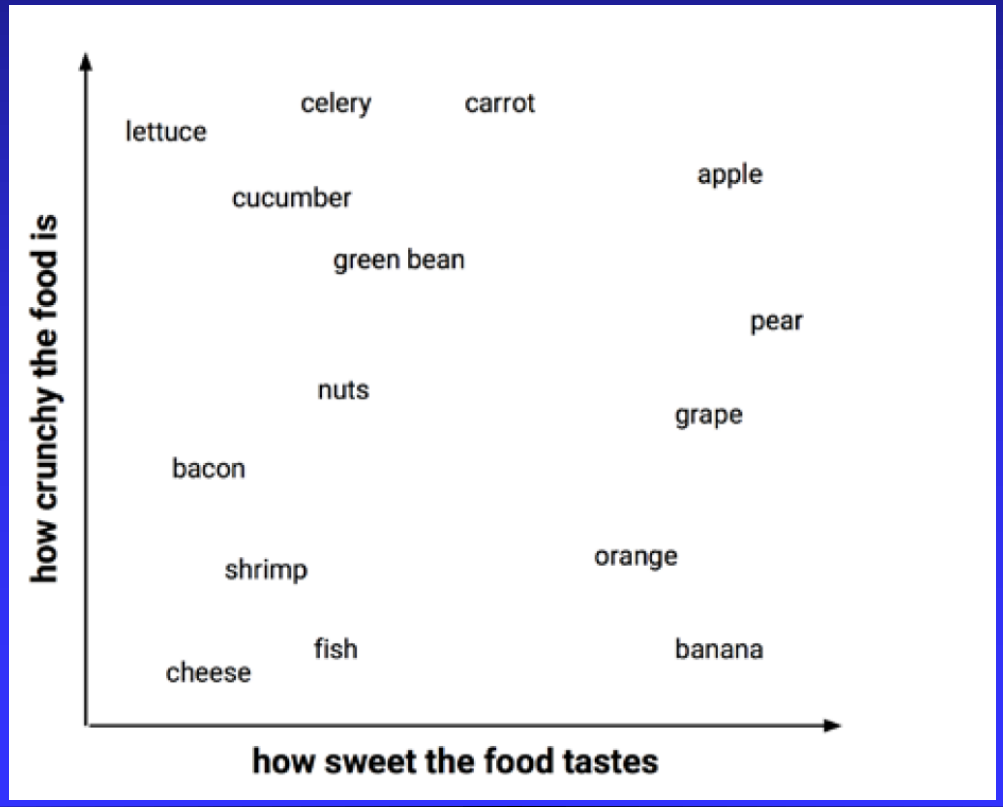
Example: food classification

Suppose we want to classify food items into three classes: fruits, vegetables and proteins based on two features: sweetness and crunchiness. Each feature is measured on a scale from 1 to 10. The training set consists of several labeled food instances:

Ingredient	Sweetness	Crunchiness	Food type
apple	10	9	fruit
bacon	1	4	protein
banana	10	1	fruit
carrot	7	10	vegetable
celery	3	10	vegetable
cheese	1	1	protein

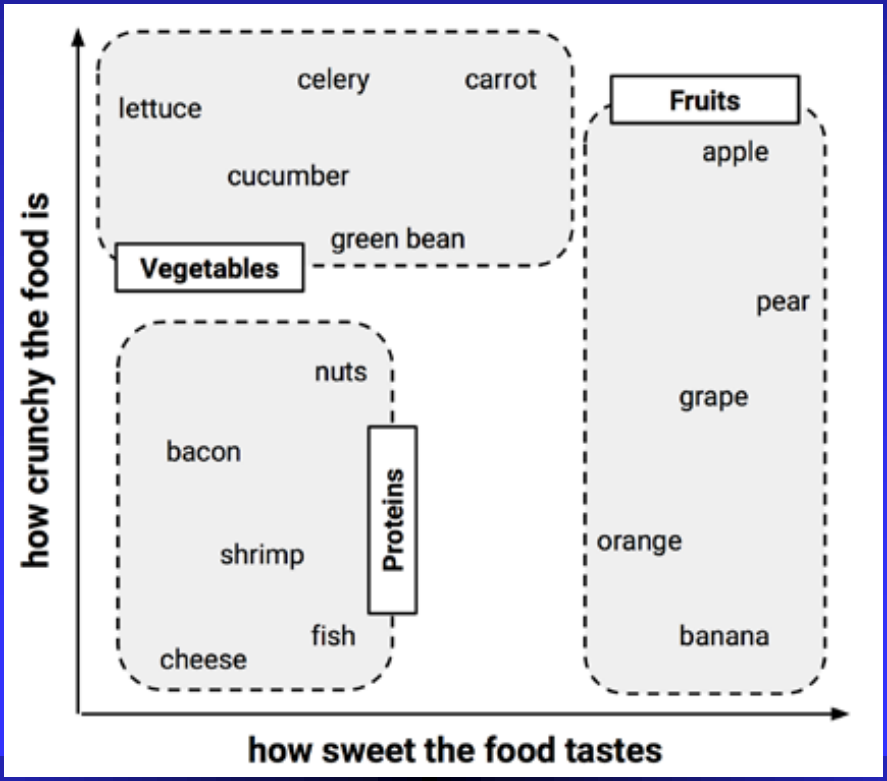
Example: food classification *(cont.)*

Since we have only two features, we can plot the two-dimensional data on a scatter plot, with the *x* dimension indicating the ingredient's sweetness and the *y* dimension, the crunchiness.



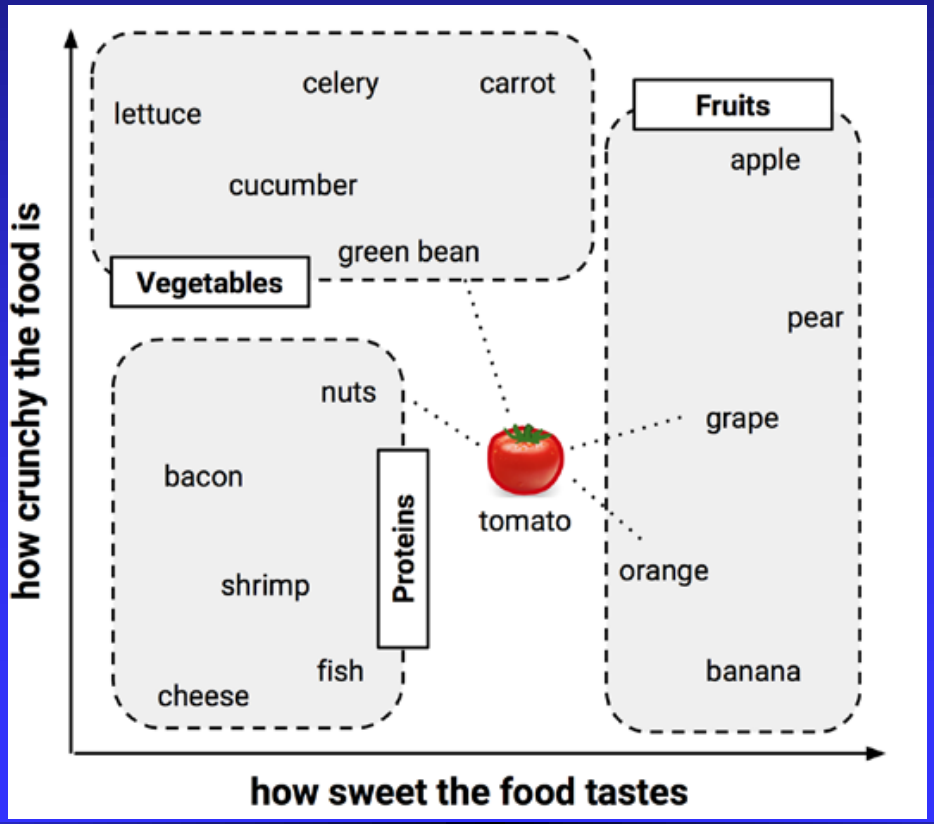
Example: food classification *(cont.)*

The scatterplot reveals a noticeable pattern: similar foods have similar feature characteristics, i.e., vegetables tend to be crunchy but not sweet, fruits tend to be sweet and either crunchy or not crunchy, while proteins tend to be neither crunchy nor sweet. On the scatterplot, the food classes present themselves as clusters:



Example: food classification *(cont.)*

Suppose that the new food item that must be labeled is tomato. We can use the nearest neighbor approach to determine which class is a better fit. To do this, we need first to find the k nearest neighbors to the tomato instance:



Distance measures

How to find the k nearest neighbors? The fact that each example can be represented by a point in an *n*-dimensional space makes it possible to calculate the distance between any pair of examples. The closer to each other the examples are in the instance space, the greater their mutual similarity. This is how the *nearest-neighbor classifier* got its name: the training example with the smallest distance from **x** in the instance space is, geometrically speaking, **x**'s nearest neighbor.

The success of the kNN algorithm depends on the choice of the distance/similarity function, i.e., on how well the distance function reflects the class differences.

In mathematics, a distance function, d, defined on a set of objects A is any function that takes real values, i.e., d : AxA → **R** , and that satisfies the following three conditions:

- $d(x,y) \geq 0$, and $d(x,y) = 0$ if and only if $x = y$. In other words, the distance is always positive and it is zero if its is measured from a point to itself.

- $d(x,y) = d(y,x)$. In other words, the distance is symmetric. The distance between *x* and *y* is the same as the distance between y and x.

- $d(x,z) \leq d(x,y) + d(y,z)$. This is the so called triangle inequality: the distance between two points is the shortest path between them, i.e., going through a third point only increases the length of the path between the two points.

Examples of numeric distance functions

For any two n-dimensional numeric vectors (x_1, x_2, ...,x_n) and (y_1, y_2, ...,y_n), we have the following distance functions:

1-norm distance $= \sum_{i=1}^{n} |x_i - y_i|$ also called Manhattan distance

2-norm distance $= \left(\sum_{i=1}^{n} |x_i - y_i|^2 \right)^{1/2}$ also called Euclidean distance

p-norm distance $= \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$

infinity norm distance $= \max \left(|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_n - y_n| \right).$

The Manhattan distance is sometimes called taxicab distance because it is the distance a car would drive in a city laid out in square blocks, i.e., the car will drive along one dimension, then it will drive along the other dimension and so on.

Example of symbolic distance functions

The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. For example, the Hamming distance between “ABC” and “AAD” is 2 (the strings differ in two positions; and the Hamming distance between <red, blue, green> and <blue, blue, blue> is 2.

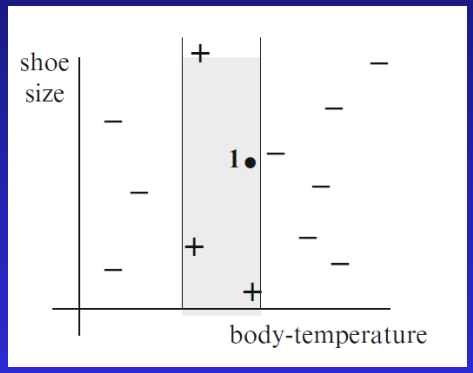
You can also use weighted Hamming distance in which different positions carry different weights. For example, if the first position is twice as important as the remaining positions, the distance between <bmw, 523> and <bmw,323> is 1 (the weight of the second position is 1, and the weight of the first position is 2), whereas the distance between <bmw,523> and <Volvo, s60> is 2+1 = 3.

Problems with distance measures: mixing numeric and categorical data

Mixing continuous and discrete attributes can be risky. Suppose our examples are described by three attributes: size, price, and season. Of these, the first two are continuous, and the last is categorical. Suppose that the data instance is **x** = (1, 100, summer) and **y** = (3, 150, winter). Suppose further that winter is modeled as 1 and summer is modeled as 0. A thoughtless application of the Euclidian distance, for example, can result in a situation where the difference between two prices (e.g., price1= 100 and price2 =150, which means that the difference between prices is 50) can totally dominate the difference between two seasons (which could not exceed 1). This observation is closely related to the problem of *scaling,* which we will discuss later.

Problems with distance measures: irrelevant attributes

Suppose we have two numeric attributes: body-temperature (the horizontal axis) and shoe-size (the vertical axis). The black dot stands for the object that the *k*-NN classifier is expected to label as healthy (pos) or sick (neg).



You can see that all positive examples find themselves in the shaded area delimited by two critical points along the “horizontal” attribute: temperatures exceeding the maximum indicate fever, and those below the minimum, hypothermia. As for the “vertical” attribute, we see that the positive and negative examples alike are distributed along the entire range, show-size being unable to contribute anything to a person’s health. The object we want to classify is in the highlighted region, and common sense requires that it should be labeled as positive—despite the fact that its nearest neighbor happens to be negative.

Problems with distance measures: scaling

Suppose we want to evaluate the similarity of two examples, **x** = (*t*, 0.2, 254) and **y** = (*f* , 0.1, 194), described by three attributes, of which the first is Boolean, the second is continuous with values from interval [0,1], and the third is continuous with values from interval [1, 1000]. The Euclidian distance between the two examples is:

$$\sqrt{(1-0)^2 + (0.2-0.1)^2 + (254-194)^2}$$

The expression shows that the third attribute completely dominates, reducing the other two to virtual insignificance. No matter how we modify their values within their ranges, the distance, *d*(**x**; **y**), will hardly change. Therefore, the scales of the attribute values can radically affect the *k*-NN classifier's behavior.

One way out of this problem is to *normalize* the attributes, i.e., to rescale them in a way that makes all values fall into the same interval, [0,1]. The simplest normalization method is perhaps *min-max normalization*. For any given attribute, it find its maximum (*MAX*) and minimum (*MIN*), and then replaces each value, *x*, of this attribute with:

$$x = \frac{x - MIN}{MAX - MIN}$$

Example: food classification *(cont.)*

Let's go back to our example and label tomato. We need to measure the distance between tomato and the other food instances:

Ingredient	Sweetness	Crunchiness	Food type	Distance to the tomato
grape	8	5	fruit	*sqrt((6 - 8)^2 + (4 - 5)^2) = 2.2*
green bean	3	7	vegetable	*sqrt((6 - 3)^2 + (4 - 7)^2) = 4.2*
nuts	3	6	protein	*sqrt((6 - 3)^2 + (4 - 6)^2) = 3.6*
orange	7	3	fruit	*sqrt((6 - 7)^2 + (4 - 3)^2) = 1.4*

If we use 1-NN classification (i.e., k=1 and you are using the nearest neighbor), then the predicted food type for tomato is fruit because tomato's nearest neighbor is orange (with distance 1.4) and the class of orange is fruit.

If we use 3-NN classification (i.e., k =3) with a simple majority vote among the nearest neighbors, then the predicted class for tomato is again fruit because the closest 3 neighbors are orange (class=fruit), grape (distance = 2.2, class = fruit), and nuts (distance = 3.6, class = proteins) and the majority class among the three neighbors is fruit.

Variance bias tradeoff

There are two sources of the classifier's errors: variance and bias. Because different samples of training data can be used to create a classifier, the sample that is used as a training set to create a particular classifier almost never captures all aspects of the underlying classes. This is partly due to the way the training set has been created. In some applications, the training set has been created at random. In other cases, it consists of examples available at the given moment. In addition, the training set could have been created by an expert who is inevitably subjective. The problem is that different training sets will produce different classifiers that will differ in their predictions. This is what we mean by **variance** and by saying that variance in the training data is an important source of error.

A classifier is **biased** if the average predictions of different classifiers (based on different training sets) systematically miss the target. For example, an algorithm with high bias can miss important relations between features. Often, the bias stems from erroneous assumptions in the learning algorithm.

Overfitting vs underfiting

A learning algorithm might **overfit** data if is it is very accurate on training data but less accurate in predicting new data. In other words, overfittig produces a model that works too well on training data but fails to generalize to unseen data. Overfitting is often a result of complex learning models or of learning models that adjust to nosy or random irrelevant features of the training data. Overfitting can make the model sensitive to small fluctuations in the training set and, therefore, increase the variance error.

A learning algorithm is said to **underfit** training data if it is less accurate in fitting training data. For example, the algorithm might fail to capture important data regularities. Underfitting often leads to less variance and higher bias. In the extreme case, a classifier might not fit any data at all and might output the same constant prediction for every new data instance. As a result, the classifier has zero variance and high bias.

Variance bias tradeoff *(cont.)*

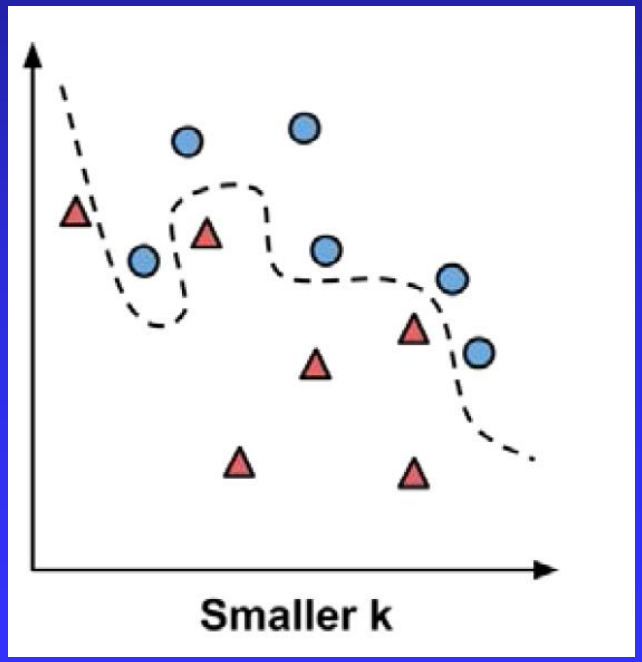
The bias–variance tradeoff is a central problem in machine learning. In general, we want to create a model that performs very well on training data and generalizes well to unseen data. Unfortunately, it is typically impossible to reduce both errors. That is, reducing bias comes at the cost of increasing variance and vice versa.

Models that work very well on their training sets are at risk of overfitting noisy or unrepresentative training data and tend to have high variance. In contrast, algorithms that do not perform very well on their training sets don't tend to *overfit*, but may underfit their training data, thereby creating high bias.

In other words, one of the problems is to find the right balance between bias and variance.

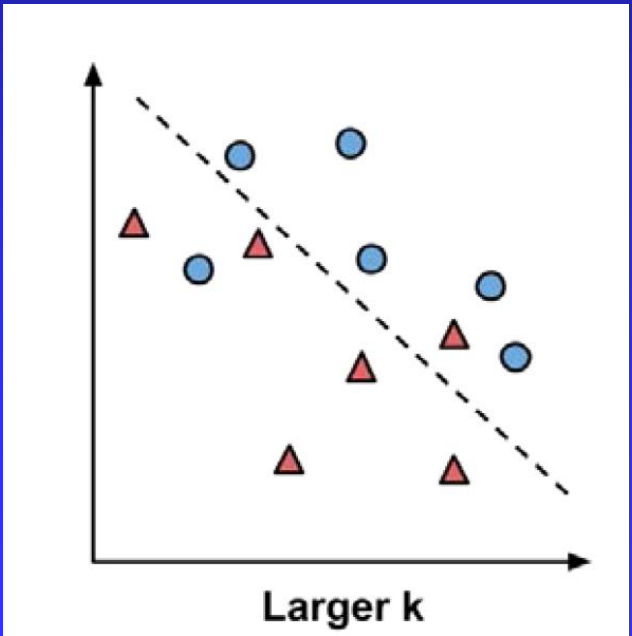
How to choose k?

The value of k in the kNN algorithm can have a large effect on the behavior of the algorithm. When k=1, the resulting prediction surface is very wiggly. As a result, the method might not work well on predicting future data.



How to choose k? *(cont.)*

As k increases, the prediction surface becomes smoother until, in the limit of k=N, we end up with predicting the majority label of the whole data set.



kNN: Variance bias tradeoff

Wen k is large, the algorithm loses its precision and attention to detail because some important regularities can get neglected. In other words, large k tend to produce more bias and less variability. In the extreme case when k=N, the algorithm would always predict the majority class, regardless of the nearest neighbors. Although the variability is zero (the prediction is always the same), the result is biased: the algorithm prediction tends to differ from the correct one.

On the other hand, when k is small, the algorithm might overfit data, allowing unimportant features to influence the classification of examples. In other words, the algorithm tens to produce more variability and less bias.

Avoiding overfitting in kNN

A training set includes two types of data instances: instances that are relevant for the classification and irrelevant instances also called noise. The more noisy the training set is, the more difficult the prediction is because more noise must be ignored. The problem is how to find noisy instances. One heuristic often used is the following: remove a data instance if all its neighbors are of another class.

A learning algorithm that can reduce the chance of fitting noise is called **robust**.

Another technique used to adjust k in order to avoid overfitting is cross-validation. In general, in cross-validation, we use one sample to train a classifier and another sample to test its performance. Two commonly used cross-validation methods are 2-fold cross-validation and k-fold cross validation.

In 2-fold cross validation, the initial data set is randomly divided into two sets of equal size A and B. The classifier is then trained on A and tested on B, followed by training on B and testing on A.

In k-fold cross-validation, the initial data set is partitioned into k equally-sized subsets. Each subset is used as a test set for a classifier trained on the remaining k -1 subsets. The empirical accuracy is given by the average of the accuracies of these k classifiers.

Preparing data for use with k-NN

In order to use distance measures in k-NN features must be transformed to a standard range.

One method for rescaling features for k-NN is **min-max normalization.**

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

All normalized features fall in the range between 0 and 1, and each feature shows how far (from 0 percent to 100 percent) the original value fell along the range between the original minimum and maximum.

Another way to rescale features is by using **z-score standardization:**

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

The resulting value is **the z-score,** which tells how many standard deviations the feature falls above or below its mean value.

Preparing data for use with k-NN *(cont.)*

If the futures are nominal, they can be converted into numeric format using dummy coding, in which binary flags are used to indicate that a feature belongs to a specific category. For example, if the nominal variable *gender* has two levels, *male* and *female*, it can be coded with one flag, say *male* that indicates membership to the male class. That is *male=1* if the data instance is male and *male=0* if the data instance is not *male* (therefore it is *female*).

As another example, consider the case where temperature variable has three levels, *hot*, *medium* and *cold*. The variable can be coded with two flags. The first flag indicates if the data instance is *hot* and the second flag indicates if the data instance is *medium*. For instance, a data instance with hot temperature must be encoded as *hot=1* and *medium=0*. A data instance with cold temperature must have *hot=0* and *medium=0*.

Ordinal features could be coded with ordinal numbers. For example, if the levels of sweetness are coded on a scale from 1 to 10, one must ensure that the steps between different values represent equal intensity of sweetness, i.e., the difference between sweetness levels 7 and 6 is the same as the difference between sweetness levels 2 and 1.

Diagnosing breast cancer with k-NN: Collecting data

We will apply the k-NN algorithm to measurements of biopsied cells from women with abnormal breast masses in order to automatically detect cancer.

The textbook uses the Wisconsin Breast Cancer Diagnostic dataset from the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml. The data includes the measurements from digitized images of fine-needle aspirate of breast mass.

You can download the data set from Blackboard. Save the file wisc_bc_data.csv in your working directory.

The breast cancer data includes 569 examples of cancer biopsies, each with 32 features. One feature is an identification number, another is the cancer diagnosis, and there are 30 numeric-valued laboratory measurements. The diagnosis is coded as "M" to indicate malignant or "B" to indicate benign.

The other 30 numeric measurements comprise the mean, standard error, and worst (that is, largest) value for 10 different characteristics of the digitized cell nuclei. These include: Radius, Texture, Perimeter, Area, Smoothness, Compactness, Concavity, Concave points, Symmetry, and Fractal dimension.

Diagnosing breast cancer with k-NN: exploring and preparing the data

First, we will import the data into the wbcd data frame:

```
>  wbcd <- read.csv("wisc_bc_data.csv", stringsAsFactors =
+ FALSE)
```

The str(wbcd) command gives us a brief look at the data:

```
> str(wbcd)
'data.frame':   569 obs. of  32 variables:
 $ id                : int  87139402 8910251 905520 868871 9012568
906539 925291 87880 862989 89827 ...
 $ diagnosis         : chr  "B" "B" "B" "B" ...
 $ radius_mean       : num  12.3 10.6 11 11.3 15.2 ...
 $ texture_mean      : num  12.4 18.9 16.8 13.4 13.2 ...
 $ perimeter_mean    : num  78.8 69.3 70.9 73 97.7 ...
```

.....

The first variable is the patient ID. Its numeric value is useless for the purposes of k-NN, i.e., we cannot use the patient ID as an attribute to the distance function. We will make a copy of the data frame which does not include the patient ID:

```
> wbcd <- wbcd[-1]    # since the first column has a minus sign
                      # it is excluded from the copy
```

Diagnosing breast cancer with k-NN: exploring and preparing the data *(cont.)*

The variable diagnosis indicates whether the patient has a benign or malignant mass. The next command labels the two levels of the diagnosis variable, “B” and “M”, as “Benign” and “Malignant”:

```
> wbcd$diagnosis<- factor(wbcd$diagnosis, levels = c("B", "M"),
+ labels = c("Benign", "Malignant"))
```

We can see how many records are benign and how many are malignant:

```
 > table(wbcd$diagnosis)
    Benign Malignant
       357       212
```

The same result can be shown in terms of rounded proportions:

```
> round(prop.table(table(wbcd$diagnosis)) * 100, digits = 1)
   Benign Malignant
     62.7      37.3
```

Diagnosing breast cancer with k-NN: normalizing numeric data

The remaining 30 features are all numeric and each one consists of three different measurements (the mean, the standard error and the largest value) of ten characteristics. We need to determine if the numeric features are appropriate for distance measuring in k-NN and if they need to be rescaled. The *str()* function shows that many features are measured on different scales. For example, area_mean is measured in terms of hundreds, whereas smoothness_mean is less than 1. This observation is also confirmed by the *summary()* function:

```
> summary(wbcd[c("radius_mean", "area_mean",
"smoothness_mean")])
  radius_mean       area_mean      smoothness_mean
 Min.   : 6.981   Min.   : 143.5   Min.   :0.05263
 1st Qu.:11.700   1st Qu.: 420.3   1st Qu.:0.08637
 Median :13.370   Median : 551.1   Median :0.09587
 Mean   :14.127   Mean   : 654.9   Mean   :0.09636
 3rd Qu.:15.780   3rd Qu.: 782.7   3rd Qu.:0.10530
 Max.   :28.110   Max.   :2501.0   Max.   :0.16340
```

Diagnosing breast cancer with k-NN: normalizing numeric data *(cont.)*

We will normalize features using min-max normalization. First, we define *normalize()* function, which takes a vector x of numeric values, and for each value in x, subtracts the minimum value in x and divides by the range of values in x.

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

Instead of normalizing each of the 30 numeric variables one by one, we will use the *lapply()* function to do the normalization at once. The function *lapply(x, fun)* returns a list of the same length as x, each element of which is the result of applying *fun* to the corresponding element of x.

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

Here, we have applied the normalize function to each feature and have converted the list returned by *lapply()* to a data frame, using the *as.data.frame()* function. The result was stored in *wbcd_n*. We skipped the first column because it contains the labels.

Diagnosing breast cancer with k-NN: creating training and test datasets

The next step is to split the data set into two sets, a training set and a test set. The test set will be used as a validation set to evaluate the performance of the algorithm. We will use the first 469 records for the training dataset and the remaining 100 for the test set. We can assume that the records in the original data set have been ordered randomly, and our choice of consecutive records in the training and the test sets are not biased. In general, both the training and the test set must be selected randomly. If the training and the test set are not random, it might happen that we train the model on one type of data and test it on another type, leading to discrepancy in results. The following two commands store the training and the test set into variables wbcd_train and wbcd_test, respectively.

```
> wbcd_train <- wbcd_n[1:469, ]    # selects all columns of the
                                   # first 469 rows
> wbcd_test <- wbcd_n[470:569, ] #selects all columns of the
                                  # last 100 rows
```

Before we can run the k-NN algorithm we must store the labels, i.e., the values of the diagnosis variables for each patient record:

```
> wbcd_train_labels <- wbcd[1:469, 1] # selects the first
                                      # column from rows 1:468
> wbcd_test_labels <- wbcd[470:569, 1] #selects the first
                                      # column from rows 470:569
```

Diagnosing breast cancer with k-NN: running the algorithm

One implementation of kNN is the *knn()* function from the package *class*. The function uses the majority vote of the k nearest neighbors to decide the label of a new data instance, with ties broken at random. The package *class* usually comes preinstalled. Before we can run the *knn()* function, we need to load the package:

```
> library(class)
```

The general format of the knn() function is as follows:

knn(train, test, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE)

Where

train – matrix/vector or data frame of training set cases.

test – matrix/vector or data frame of test set cases.

cl - factor of true classifications of training set

k – the number of neighbors considered.

The attributes l, prob and use.all are optional. l is the minimum vote for definite decision, in other words, less than k-l dissenting votes are allowed. If prob is true, the proportion of the votes for the winning class are returned as attribute prob. use.all controls handling of ties. If true, all distances equal to the k-th largest are included. Otherwise, a random selection of distances equal to the k-th is chosen.

The *knn()* function returns a factor of classifications of the test set.

Diagnosing breast cancer with k-NN: running the algorithm *(cont.)*

Usually, one runs the *knn()* function with the following typical attributes:

knn(train, test, cl, k)

We run the knn() function, storing the vector of classified labels into wbcd_test_pred:

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

The `wbcd_test_pred` is the prediction vector that holds the predicted values. What remains to be done is to evaluate the algorithm performance by comparing the prediction vector wbcd_test_pred with the test vector wbcd_test. Since both vectors are nominal (with values of Benign or Malignant), we can create a crosstab of predicted and true labels using the CrossTable function from the gmodels package.

```
> library(gmodels)  # loads the gmodels package
> CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
+ prop.chisq=FALSE)
```

The crosstab is shown on the next slide.

Diagnosing breast cancer with k-NN: evaluating model performance

```
   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 |           |
-----------------|-----------|-----------|-----------|
```

Diagnosing breast cancer with k-NN: evaluating model performance *(cont.)*

We can see a mismatch of 2 on the intersection of malignant row and Benign column. In general, the following errors are possible:

- False positive – a benign record is classified as malignant
- False negative – a malignant record is classified as benign.

In this particular scenario, false negatives are more dangerous than false positives because ill people will not get diagnosed correctly on time, allowing the cancer cells to spread.

The algorithm made two false negative errors (the entry on the intersection of malignant row and benign column) and zero false positive errors (the entry on benign row and malignant column) . In general the error rate is 2%: 98% of the examples were correctly classified.

Diagnosing breast cancer with k-NN: improving model performance *(cont.)*

One way to improve the k-NN algorithm performance is by adjusting k. Unfortunately, in our case, trying different k values with the same training and validation sets does not have significant impact on the algorithm performance. The worst case is k=1. The following diagram shows the number of false positives and false negatives for different values of k.

k value	False negatives	False positives	Error total
1	1	3	4
5	2	0	2
10	4	0	4
15	3	0	3
25	3	0	3
30	4	0	4
40	4	0	4
60	4	0	4

Diagnosing breast cancer with k-NN: evaluating model performance *(cont.)*

Another technique that can be used to improve model performance is z-score normalization. As we know, the z-score measures how far (in terms of standard deviations) is each value is from the mean. Using z-score instead of min-max normalization allows the outliers to be weighted more heavily in the distance calculation since most values are usually within ±3 standard deviations from the mean. There is a built-in *scale()* function, which rescales values using the z-score standardization. The function can be applied directly to a data frame and there is no need to use the lapply() function:

```
> wbcd_z <- as.data.frame(scale(wbcd[-1]))
```

We have excluded the diagnosis value (column 1) from normalization since it is nominal. The result of the normalization is stored into wbcd_z.

We use the same records as training and validation sets:

```
> wbcd_train <- wbcd_z[1:469, ]    # training set is extracted
> wbcd_test <- wbcd_z[470:569, ]     # validation set is
                                     # extracted
> wbcd_train_labels <- wbcd[1:469, 1]      # training labels
> wbcd_test_labels <- wbcd[470:569, 1]     # test labels
```

We get 5 false negatives and z-score normalization does not improve performance in this particular case.

Diagnosing breast cancer with k-NN: evaluating model performance *(cont.)*

```
> wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
+ cl = wbcd_train_labels, k = 21)       # runs the k-NN algorithm
> CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
+ prop.chisq = FALSE)                   # builds a crosstab
```

wbcd_test_labels	wbcd_test_pred: Benign	wbcd_test_pred: 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	

Diagnosing breast cancer with k-NN: evaluating model performance *(cont.)*

After the z-score normalization, we can try running the knn function with different values of k:

k value	False negatives	False positives	Error total
1	3	2	5
5	2	1	3
10	3	1	4
15	3	0	3
25	5	0	5
30	5	0	5
40	4	0	4
60	5	0	5