Advanced Topics in Statistics
K-nearest neighbour classification in R and Python

K-Nearest Neighbour

The k-Nearest Neighbour algorithm is one of the simplest model available and typically used as a baseline to benchmark other ML algorithms. The rationale behind kNN is simple; the class label for a particular test point is the majority vote of the surrounding training data:

1. Compute the distance between test point and every training data point.
2. Find the k training points closest to the test point.
3. Assign test point the majority vote of their class label.

We use the k-nearest neighbour algorithm to classify the observations of the iris dataset. The dataset contains measurements from individuals of three different subspecies of the iris flower. The classification aims to identify the correct subspecies based on the available measurements.

R:

First load the data and the necessary packages.

Make sure you familiarise yourself with the data. (i.e. make sure you know which variable is the label we are trying to predict, and which variables are the ones we are using as predictors).

```
library(caret) # function for creating train/test datasets, and confusion matrices
library(class) # this has the knn function
library(ggplot2)
data(iris)
```

The next step is to split the dataset into a training and test dataset, so later we can test the performance of the built model.

We could build our own function for this, but the `caret` package has a `createDataPartition` function, which can do the splitting for us.

The `createDataPartition` function takes the vector of outcomes (this can be the vector of subspecies in our iris dataset) and the percentage of the data that should go to training (here we use 70%, that is `p=0.7`) as input, and produces a vector of indices as an output. Note that by default, the output is given as a list, thus we also set `list=F` to ensure that the output is in vector format.

Once we have the indices of the training dataset (which essentially tells us the row numbers corresponding to the training dataset), we can split the `iris` dataset up into training and test datasets. We will have a separate object for the predictors (these can be found in the first 4 columns of the iris data), and the target label (which is in the 5th column of our dataset).

Note that we also set the seed to ensure reproducibility.

```
# Split test/train
set.seed(12345) # for reproducibility
ii <- createDataPartition(iris[, 5], p=.7, list=F) ## returns indices for train data

# split the data using the indices returned by the createDataPartition function
xTrain <- iris[ii, 1:4] # predictors for training
yTrain <- iris[ii, 5]   # class label for training
xTest <- iris[-ii, 1:4] # predictors for testing
yTest <- iris[-ii, 5]   # class label for testing

# check the dimensions (is the split is indeed 70-30%?)
dim(xTrain)
dim(xTest)
```

Finally, we fit our first kNN model using $k = 3$ (the chosen `k` is quite arbitrary here, the sole purpose is to show how the fitting procedure works).

We use the `knn` function of the `class` package to fit the model.

We have to specify the matrix of training predictors (`train`), the matrix of test predictors (`test`), the train set labels (`cl`), and the number of neighbours considered (`k`).

The output of the `knn` function is a vector of predicted labels for the test dataset.

To check the accuracy of the model we produce the confusion matrix of the model fit using the `confusionMatrix` function of the `caret` package. This needs the set of predicted labels and the set of true test labels as input.

Note that the `confusionMatrix` function needs the input to be in factor format. Whenever numbers are used as labels, we first have to convert these into factors.

```
# fit the model
fit = knn(train=xTrain,test=xTest,cl=yTrain,k=3)
# fit = knn(xTrain,xTest,yTrain,k=3)

# produce the confusion matrix
confusionMatrix(fit,yTest)

# when numbers are used as class labels we might need
# confusionMatrix(as.factor(fit),as.factor(yTest))
```

This gives an accuracy of 93.33%. Since only three labels are mispredicted, we can see that the model performs really well.

However this is not always the case. The model performance can often be improved by optimising the number of neighbours we use in the fitting procedure. Here `k=3` was arbitrary. Can we find a `k` that gives a model with higher accuracy? To answer this question we will use K-fold cross validation.

K-fold cross validation

Machine learning algorithms can deal with nonlinearities and complex interactions amongst variables because the models are flexible enough to fit the data (as opposed to rigid linear regression models, for example). However, this flexibility needs to be constrained to avoid fitting to noise (overfitting). This is achieved by tuning the model's hyperparameters.

Hyperparameters are parameters that are not directly learnt by the machine learning algorithm, but affect its structure. For example, consider a simple polynomial regression model:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_p x^p.$$

The β's are the model parameters that are inferred/learnt from the data. The degree of the polynomial p, however, is a hyperparameter that dictates the complexity of the model. Hyperparameters are tuned by cross-validation to strike a balance between underfitting and overfitting, known as the bias-variance trade-off.

In the k-nearest neighbour classification algorithm, `k` is a hyperparameter.

K Fold Cross Validation is a technique in which the data set is divided into K Folds or K partitions. The Machine Learning model is trained on $K-1$ folds and tested on the Kth fold i.e.we will have $K-1$ folds for training data and 1 for testing the ML model.

The K Fold Cross Validation estimates the average validation error that we can expect on a new unseen test data. More precisely, the performance is measured on the part left out in the training process. The average prediction error is computed from the K runs and the hyperparameters that minimise this error are used to build the final model.

Some of the more popular predictive performance measures are

- Regression: root mean squared error (RMSE), R-squared
- Classification: area uder the receiver operating characteristic (ROC) curve, confusion matrix

Note that to make cross-validation insensitive to a single random partitioning of the data, **repeated cross-validation** is typically performed, where cross-validation is repeated on several random splits of the data.

Repeated cross-validation is what we will use to tune the hyperparameter of the kNN classification..

We build this model on the previously created training dataset. The optimal k for the k-nearest neighbour algorithm is found using repeated cross validation.

When we want to combine the fitting procedure with the tuning of the hyperparameter, it's not the `knn` function that we use anymore. Instead we use the `train` function of the `caret` package, which will do the fitting and tuning simultaneously.

The `train` function sets up a grid of tuning parameters (`tuneGrid`), these are essentially the `k` values we want to try. It fits the model for each tuning parameter, and calculates a resampling based performance measure. The resampling method we want to use is given by the `trControl` option of the `train` function. Here we will use 10-fold cross validation (`number=5`) repeated 5 times (`repeats=10`).

Note that a normal 10-fold cross validation (without repeats) would have `trainControl(method="cv", number=10)` as `trControl` parameter.

```
# Set training options
# Repeat 5-fold cross-validation, ten times
opts <- trainControl(method='repeatedcv', number=10, repeats=5)

# Find optimal k (model)
mdl <- train(x=xTrain, y=yTrain,                # training data
             method='knn',                      # machine learning model
             trControl=opts,                    # training options
             tuneGrid=data.frame(k=seq(2, 15))) # range of k's to try

print(mdl) # print the outcome
```

This identifies the k that achieves the best accuracy, `k=9` in this case.

Since we wanted to see how the cross validation improves the model fit, we only used the previously created training data to fit the model with cross validation. Thus we test this model on the test dataset.

We could now use the `knn` function to fit a model with `k=9` (the optimised parameter), and see its performance of test set. An easier approach however is to use the `predict` function, which takes the previous fitted model (`mdl`) as an input an applies that to the dataset specified by the `newdata` argument. The output is a vector of predicted labels for this new dataset.

Then we use the `confusionMatrix` function to compare the predicted labels with the true labels, just as before.

```
# Test model on testing data
yTestPred <- predict(mdl, newdata=xTest)
confusionMatrix(yTestPred, yTest) # predicted/true
```

Now only one observation of the test dataset is mispredicted (as opposed to 3, when we just picked an arbitrary `k` value without any tuning).

Of course here the model performance was already very good. In most classification problems, tuning the parameter can result in a much larger improvement.

Same in Python...

Python:

First we import the iris dataset and the necessary packages. Python also has a built in function for splitting the dataset, `train_test_split`.

We again use a 70/30 split to get the training and test datasets.

We also set the seed for reproducibility.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import RepeatedKFold, GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn import datasets
iris = datasets.load_iris()

# Split test/train while setting the seed
xTrain, xTest, yTrain, yTest = train_test_split(iris.data, iris.target,
                                                train_size=0.7,random_state=123)
# check training and test data dimensions
print(xTrain.shape)
print(xTest.shape)
```

Next we fit the model with our arbitrary `k=3` neighbour number.

```
# create an instance of the KNN classifier
knn = KNeighborsClassifier(n_neighbors=3)

# fit the model
knn.fit(xTrain, yTrain)
```

```
# predict test dataset
yTestPred = knn.predict(xTest)

# print associated confusion matrix
print(confusion_matrix(yTest, yTestPred))
```

We have 2 mispredicted test observations. Can we do better?

In order to tune the parameter we again use a 10 fold cross validation repeated 5 times is used to find the best value of k (out of $2, 3, \ldots, 15$). The type of cross validation is specified separately, then passed onto the `GridSearchCV` function, which looks at all the possible values of k in the specified range, and using cross validation finds the predictive performance associated to each k. The k with the highest accuracy can be extracted by the command `mdl.best_estimator_`.

```
# Repeat 10-fold cross-validation, five times, again we set the seed
cv = RepeatedKFold(n_splits=10, n_repeats=5, random_state=1040)

# create an instance of GridSearchCV
mdl = GridSearchCV(estimator=KNeighborsClassifier(),
                   param_grid={'n_neighbors': range(2, 15)}, cv=cv) # Set search grid for k

# Find optimal k (model)
# Fit the previously defined GridSearchCV to our taining dataset.
mdl.fit(X=xTrain, y=yTrain)

print(mdl.best_estimator_)
```

The cross-validation chooses `k=11` as the optimal hyperparameter.

Finally we use the trained model on the test dataset to evaluate its performance.

```
yTestPred = mdl.predict(xTest) # evaluate performance on test data
print(confusion_matrix(yTest, yTestPred)) # true/predicted
```

Now only one of the test observations is mispredicted.