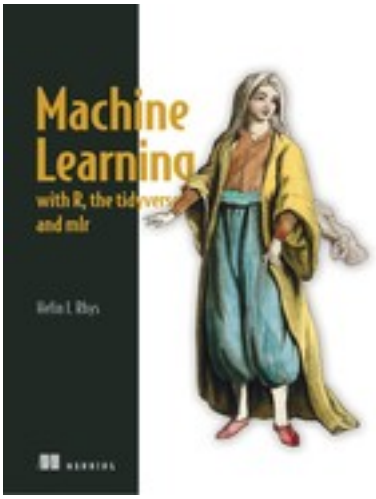
kNN Example (mlr)

This notebook shows how to uild a kNN classifier that we can use to predict diabetes status from measurements of future patients. It is based on chapter 3 of *Machine Learning with R, the tidyverse, and mlr* by Hefin Rhys (2020).



Building a machine learning model has three main stages:

- Define the task. The task consists of the data and what we want to do with it. In this case, the data is contained in a tibble: `diabetesTib`, and we want to classify the data with the class variable as the target variable.
- Define the learner. The learner is simply the name of the algorithm we plan to use, along with any additional arguments the algorithm accepts. We're going to be using the k-Nearest Neighbour (kNN) learning algorithm.
- Train the model. This stage is what it sounds like: you pass the task to the learner, and the learner generates a model that you can use to make future predictions.

First off we'll load in the diabetes data from the `mclust` package and create a tibble to store the data, we'll also draw some summary graphs to understand the data. Remember we're going to find a set of feature variables that we can use to train the model so that we can predict a target variable. Specifically for this dataset we're looking at insulin, glucose and sspg variables as predictors of the class of diabetes a given patient may have. It could equally be survival chances for passengers on the Titanic, varieties of iris, etc.

```
library(mlr)
```

```
## Loading required package: ParamHelpers
```

```
## 'mlr' is in maintenance mode since July 2019. Future development
## efforts will go into its successor 'mlr3' (<https://mlr3.mlr-org.com>).
```

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------------
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
```

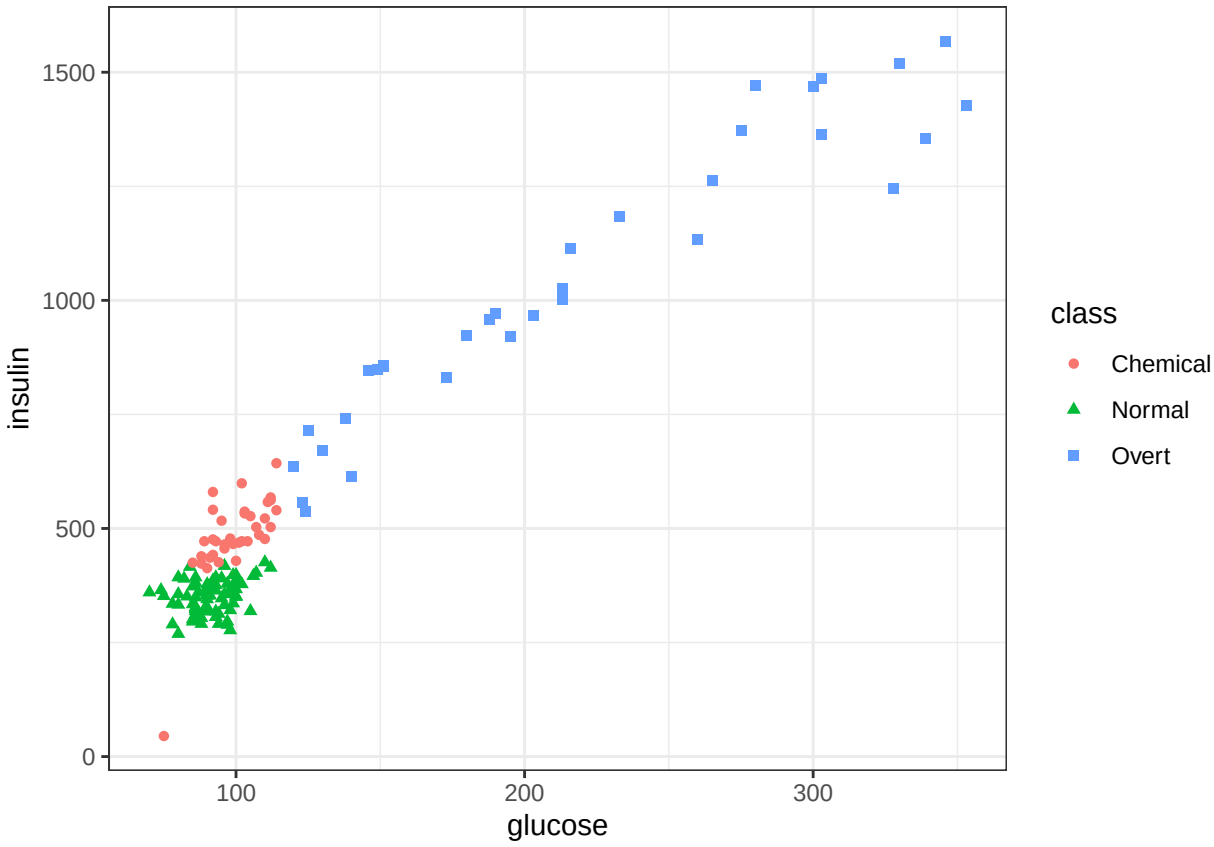
```
## v readr   1.3.1     v forcats 0.5.0
## -- Conflicts ---------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
data(diabetes, package = "mclust")
diabetesTib <- as_tibble(diabetes)
```

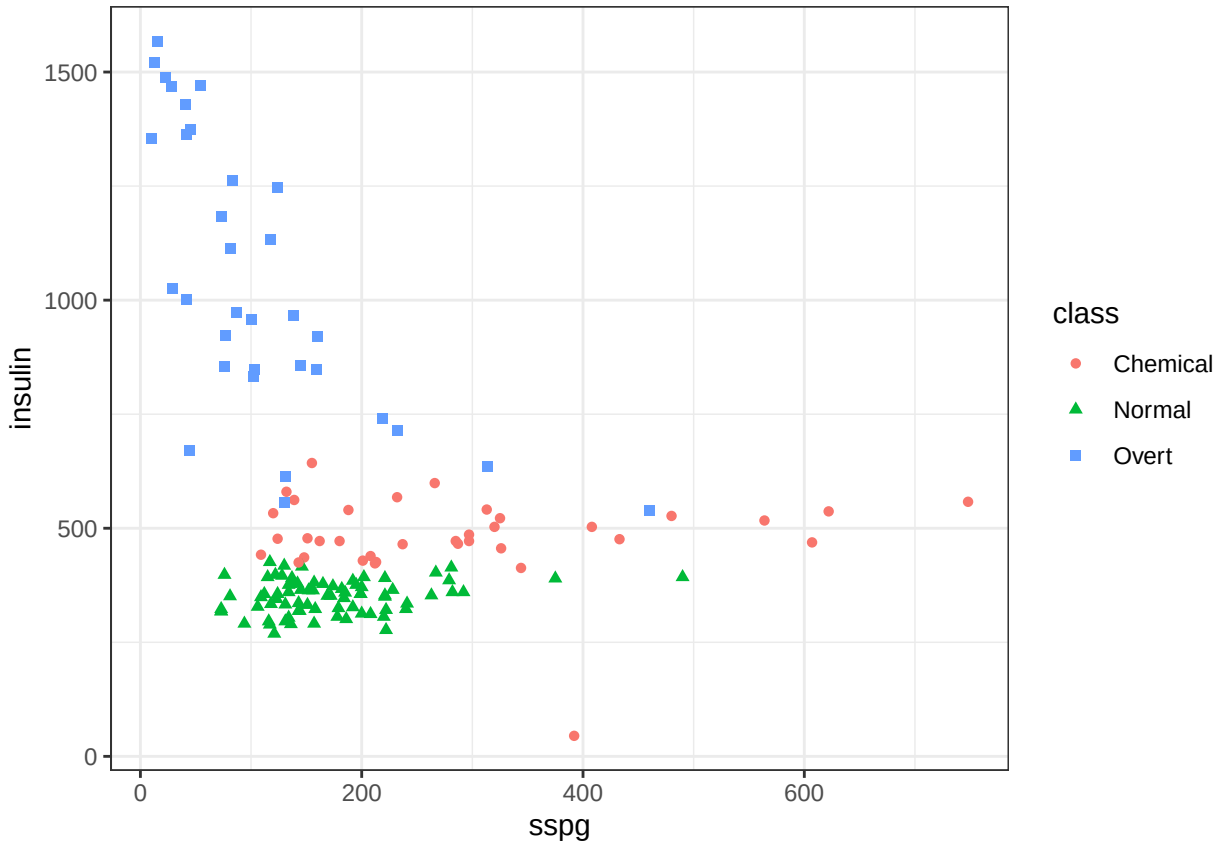
```
summary(diabetesTib)
```

```
##       class       glucose       insulin            sspg
##  Chemical:36   Min.   : 70   Min.   :  45.0   Min.   : 10.0
##  Normal  :76   1st Qu.: 90   1st Qu.: 352.0   1st Qu.:118.0
##  Overt   :33   Median : 97   Median : 403.0   Median :156.0
##                Mean   :122   Mean   : 540.8   Mean   :186.1
##                3rd Qu.:112   3rd Qu.: 558.0   3rd Qu.:221.0
##                Max.   :353   Max.   :1568.0   Max.   :748.0
```

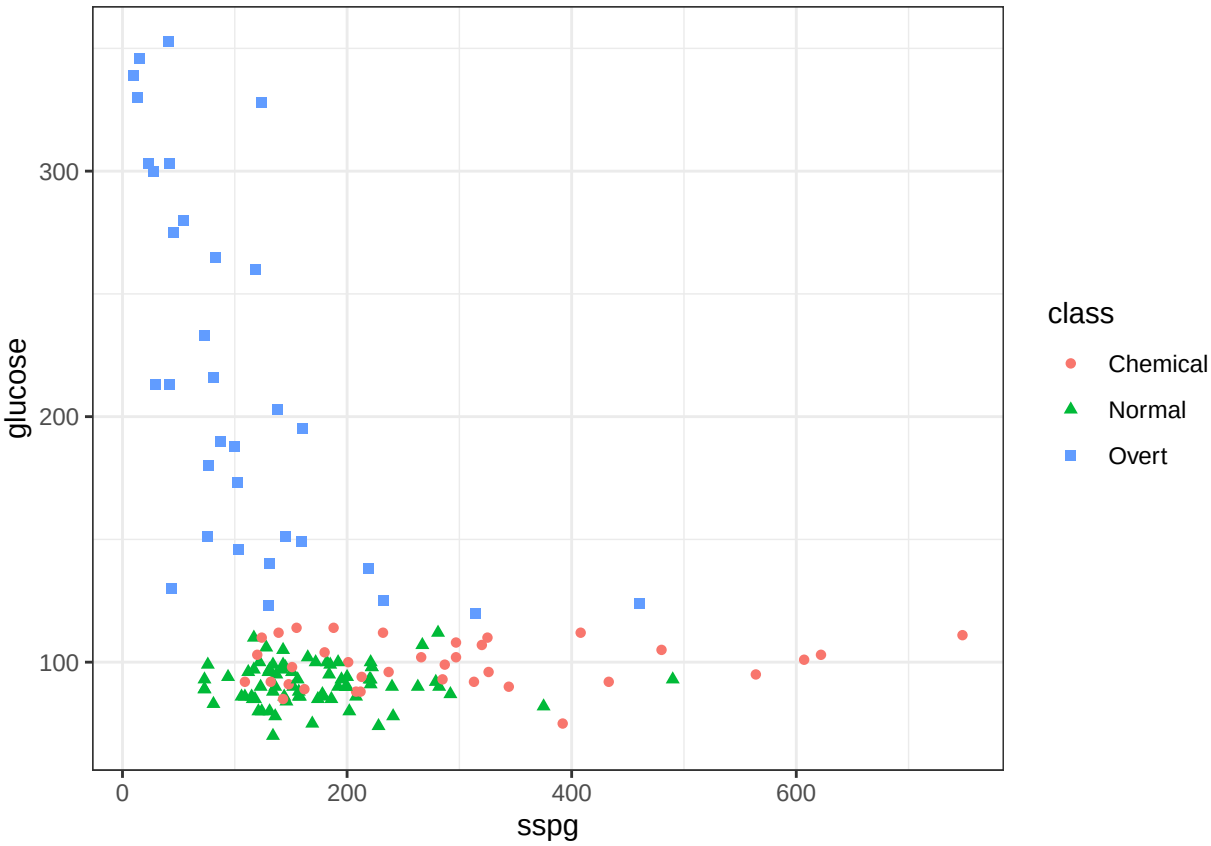
```
ggplot(diabetesTib,
       aes(glucose, insulin, shape = class, color = class)) +
  geom_point() +
  theme_bw()
```



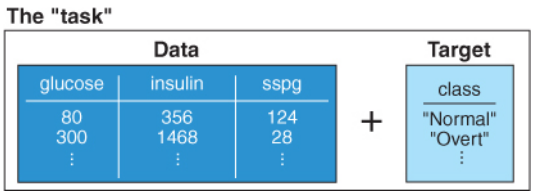

```
ggplot(diabetesTib,
       aes(sspg, insulin, shape = class, color = class)) +
  geom_point() +
  theme_bw()
```




```
ggplot(diabetesTib,
       aes(sspg, glucose, shape = class, color = class)) +
  geom_point() +
  theme_bw()
```



Now we can create the classification task for this. A task is an mlr contsruct that contains the data with the predictor variables, and the target variable we want to predict. In this case the predictor variables are the insulin, glucose and sspg variables, and we want to target the class of diabetes (Normal, Chemical or Overt).



Create a new task with the `makeClassifTask` function:

```
diabetesTask <- makeClassifTask(data = diabetesTib,
                                target = "class")
```

```
## Warning in makeTask(type = type, data = data, weights = weights, blocking =
## blocking, : Provided data is not a pure data.frame but from class tbl_df, hence
## it will be converted.
```

```
diabetesTask
```

```
## Supervised task: diabetesTib
## Type: classif
## Target: class
## Observations: 145
## Features:
##    numerics     factors     ordered functionals
##           3           0           0           0
## Missings: FALSE
```

```
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 3
## Chemical   Normal    Overt
##       36       76       33
## Positive class: NA
```

We're going to train a K-nearest neighbour classifier, so we setup a learner to do this, setting the value of `k` (which is termed a 'hyperparameter') to 2:

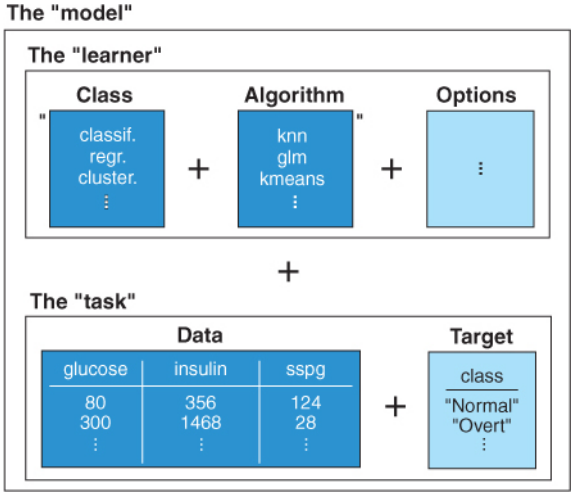
```
knnLearner <- makeLearner("classif.knn", par.vals = list("k" = 2))
knnModel <- train(knnLearner, diabetesTask)
```

Training a learner means fitting a model to a given data set. Subsequently, we want to predict the label for new observations. As you can see from the above, we've trained our model using kNN:

```
knnModel
```

```
## Model for learner.id=classif.knn; learner.class=classif.knn
## Trained on: task.id = diabetesTib; obs = 145; features = 3
## Hyperparameters: k=2
```

Think of the model as the combination of the task and learner:



Subsequently, we want to predict the label for new observations, and evaluate the performance of the model.

```
knnPred <- predict(knnModel, newdata = diabetesTib)
```

```
## Warning in predict.WrappedModel(knnModel, newdata = diabetesTib): Provided data
## for prediction is not a pure data.frame but from class tbl_df, hence it will be
## converted.
```

```
performance(knnPred, measures = list(mmce, acc))
```

```
##       mmce        acc
## 0.04827586 0.95172414
```

Cross-Validation

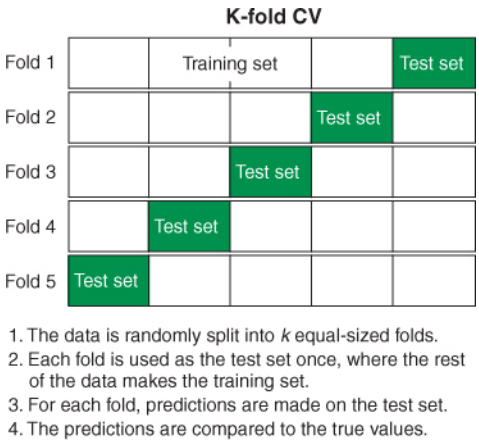
In order to understand how good our model is, whether it is underfitting or overfitting the data, we need to perform some cross-validation. We'll use k-fold cross-validation here, setting the number of folds to 10.

Cross-validation is a way of evaluate the performance of your model on data it hasn't seen yet by repeatedly sifting the data that is used for the training and test sets.

```
print(kFoldCV$aggr)
```

```
## mmce.test.mean  acc.test.mean
##       0.108626       0.891374
```

The following diagram shows how the folding works:



Finally, lets look at the confusion matrix for one of the resamplings, and see how this compares with the one we generated previously:

```
calculateConfusionMatrix(kFoldCV$pred, relative = TRUE)
```

```
## Relative confusion matrix (normalized by row/column):
##           predicted
## true       Chemical  Normal    Overt     -err.-
##   Chemical 0.79/0.78 0.10/0.05 0.11/0.12 0.21
##   Normal   0.04/0.08 0.96/0.95 0.00/0.00 0.04
##   Overt    0.16/0.15 0.00/0.00 0.84/0.88 0.16
##   -err.-        0.22      0.05      0.12 0.11
##
##
## Absolute confusion matrix:
##           predicted
## true       Chemical Normal Overt -err.-
##   Chemical     1425    180   195    375
##   Normal        144   3656     0    144
##   Overt         269      0  1381    269
##   -err.-        413    180   195    788
```

Hyperparameter Tuning

With the kNN learner, there is a single hyperparameter that we have to decide on: k. But what value of k is best to use? This is where hyperparameter tuning comes in. We can define a range of values for k which can then be run as a parameter sweep.

```
knnParamSpace <- makeParamSet(makeDiscreteParam("k", values = 1:12))
```

The `makeParamSet()` function defines the hyperparameter space we'll be using. In this example the space just contains a single hyperparameter: k. Next we setup a grid search method to run the parameter sweep. This simply tries every single value in the parameter space when looking for the best-performing value.

```
gridSearch <- makeTuneControlGrid()
```

The principle with running the parameter sweep is that for every value in the parameter space (integers 1 to 12), we perform repeated k-fold CV. For each value of k, we take the average performance measure across all those iterations and compare it with the average performance measures for all the other values of k we tried. In this way we find the 'best' k. We call the `tuneParams()` function to perform the tuning. We'll reuse the k-fold resampling we used previously. This will take some time as it repeatedly iterates through the 'grid' of k-values from 1-12, running k-fold resamplings for each.

```
tunedK <- tuneParams("classif.knn", task = diabetesTask,
                     resampling = kFold,
                     par.set = knnParamSpace, control = gridSearch)
```

```
## [Tune] Started tuning learner classif.knn for parameter set:
```

```
##       Type len Def                     Constr Req Tunable Trafo
## k discrete   -   - 1,2,3,4,5,6,7,8,9,10,11,12   -    TRUE     -
```

```
## With control class: TuneControlGrid
```

```
## Imputation value: 1
```

```
## [Tune-x] 1: k=1
```

```
## [Tune-y] 1: mmce.test.mean=0.1057987; time: 0.0 min
```

```
## [Tune-x] 2: k=2
```

```
## [Tune-y] 2: mmce.test.mean=0.1044617; time: 0.0 min
```

```
## [Tune-x] 3: k=3
```

```
## [Tune-y] 3: mmce.test.mean=0.0847070; time: 0.0 min
```

```
## [Tune-x] 4: k=4
```

```
## [Tune-y] 4: mmce.test.mean=0.0854098; time: 0.0 min
```

```
## [Tune-x] 5: k=5
```

```
## [Tune-y] 5: mmce.test.mean=0.0811646; time: 0.0 min
```

```
## [Tune-x] 6: k=6
```

```
## [Tune-y] 6: mmce.test.mean=0.0798220; time: 0.0 min
```

```
## [Tune-x] 7: k=7
```

```
## [Tune-y] 7: mmce.test.mean=0.0740517; time: 0.0 min
```

```
## [Tune-x] 8: k=8
```

```
## [Tune-y] 8: mmce.test.mean=0.0849370; time: 0.0 min
```

```
## [Tune-x] 9: k=9
```

```
## [Tune-y] 9: mmce.test.mean=0.0861311; time: 0.0 min
```

```
## [Tune-x] 10: k=10
```

```
## [Tune-y] 10: mmce.test.mean=0.0869291; time: 0.0 min
```

```
## [Tune-x] 11: k=11
```

```
## [Tune-y] 11: mmce.test.mean=0.0837417; time: 0.0 min
```

```
## [Tune-x] 12: k=12
```

```
## [Tune-y] 12: mmce.test.mean=0.0841683; time: 0.0 min
```

```
## [Tune] Result: k=7 : mmce.test.mean=0.0740517
```

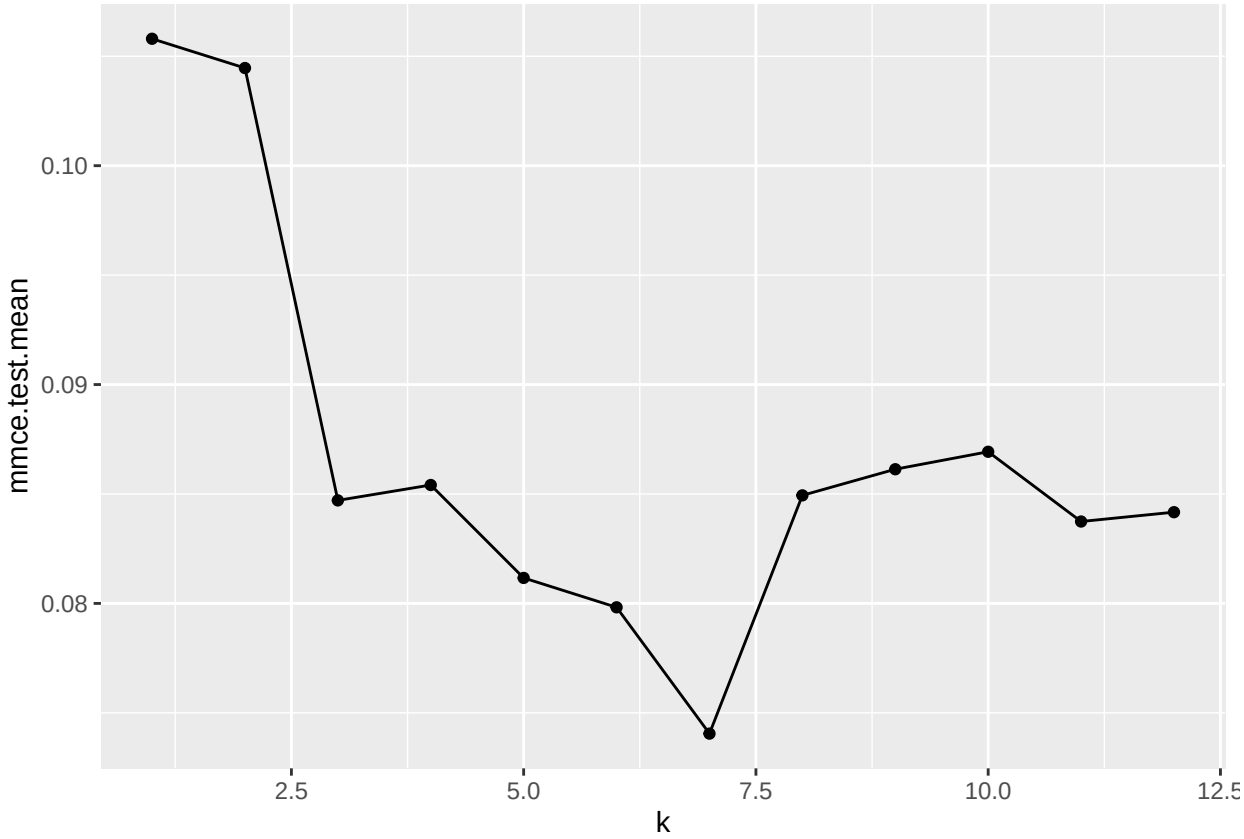
We can see that the best value is k=7 (along with its average MMCE value of 0.076):

```
tunedK
```

```
## Tune result:
## Op. pars: k=7
## mmce.test.mean=0.0740517
```

We can nicely visualise the tuning process like this:

```
knnTuningData <- generateHyperParsEffectData(tunedK)
plotHyperParsEffect(knnTuningData,
                    x = "k", y = "mmce.test.mean",
                    plot.type = "line")
```



Now we are ready to train our final model, using a k of 7 (rather than the $k = 2$ we chose originally):

```
tunedKnn <- setHyperPars(makeLearner("classif.knn"), par.vals = tunedK$x)
tunedKnnModel <- train(tunedKnn, diabetesTask)
```