# Introduction to Machine Learning

Working Group “Computational Statistics” – Bernd Bischl et al.

# Code demo for generative classification methods

## Some Maths

When we are using a generative approach to classification, we are not modeling the conditional density $\pi_k(x) = P(y = k|x)$, i.e., the class membership probability given a certain feature vector, directly. Instead, we are modeling the "other" conditional density $f(x|y = k)$ ("probability" of a feature vector given a certain class membership, i.e. the **likelihood** of observing $x$ under the assumption that the class is $k$). Following the Bayes' rule one gets that

$$\pi_k(x) \propto \pi_k \cdot f(x|y = k).$$

The distribution defined by the parameters $\pi_k$ is called the **prior** and can be interpreted as the representation of our *a priori* knowledge about the frequencies of the target classes. In our setting we can use a straightforward approach to deriving this prior, s.t.

$$\hat{\pi}_k = \frac{n_k}{n}.$$

With prior and likelihood specified, we can use the fact that all posterior class probabilities need to sum to one to get:

$$1 = \sum_{j=1}^{g} \pi_j(x) = \sum_{j=1}^{g} \alpha\pi_j \cdot p(x|y = j) \iff \alpha = \frac{1}{\sum_{j=1}^{g} \pi_j \cdot p(x|y = j)}.$$

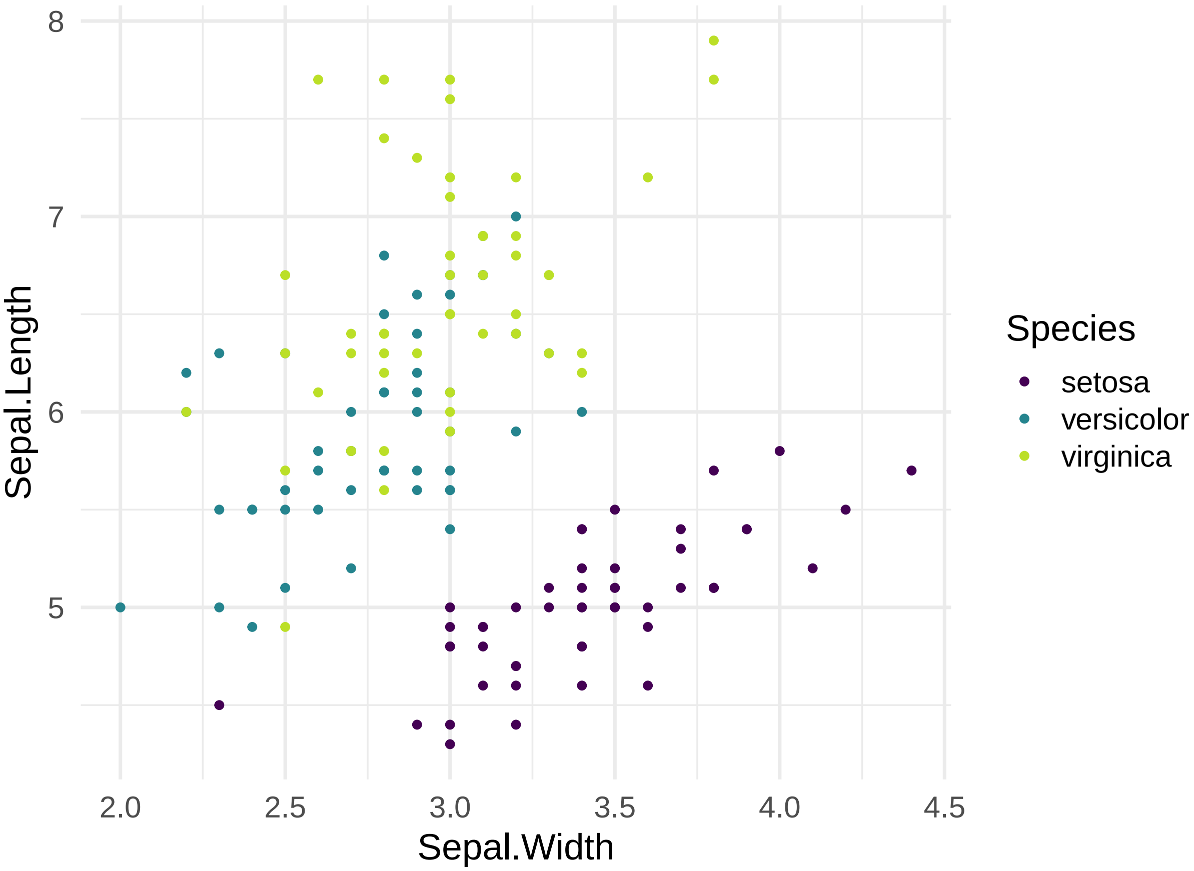
From this, we see that $\pi_k(x)$ can be expressed as

$$\pi_k(x) = \frac{\pi_k \cdot p(x|y = k)}{\sum_{j=1}^{g} \pi_j \cdot p(x|y = j)}.$$

## Some Data

In this code demo we're looking at the iris data set again:

```
library(ggplot2)

data(iris)
target <- "Species"
features <- c("Sepal.Width", "Sepal.Length")
iris_train <- iris[, c(target, features)]
target_levels <- levels(iris_train[, target])

ggplot(iris_train, aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point(aes(color = Species))
```



For the estimation of the models we will mostly use the `mlr3`-package, s.t. we firstly have to define a *task*:

```
library(mlr3)
library(mlr3learners)

iris_task <- TaskClassif$new(id = "iris_train", backend = iris_train,
                             target = target)
```

## Some Models (& More Maths & Lots of Code) (sorry…) (…not sorry.)

### Linear discriminant analysis (LDA)

In LDA, we model the likelihood as a multivariate normal distribution s.t.

$$p(x|y=k) = \frac{1}{\pi^{\frac{p}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\right).$$

With:

- $\hat{\mu}_k = \frac{1}{n_k}\sum_{i:y^{(i)}=k} x^{(i)}$,
- $\hat{\Sigma} = \frac{1}{n-g}\sum_{k=1}^{g}\sum_{i:y^{(i)}=k}(x^{(i)}-\hat{\mu}_k)(x^{(i)}-\hat{\mu}_k)^T$.

For every class, it is assumed that data is normally distributed with the same covariance matrix $\Sigma$ for all classes but different mean vectors $\mu_k$.

We train the model:

```
iris_lda_learner <- lrn("classif.lda", predict_type = "prob")
iris_lda_learner$train(task = iris_task)
```

We create a general framework for likelihoods, s.t. the we are able to visualize them:

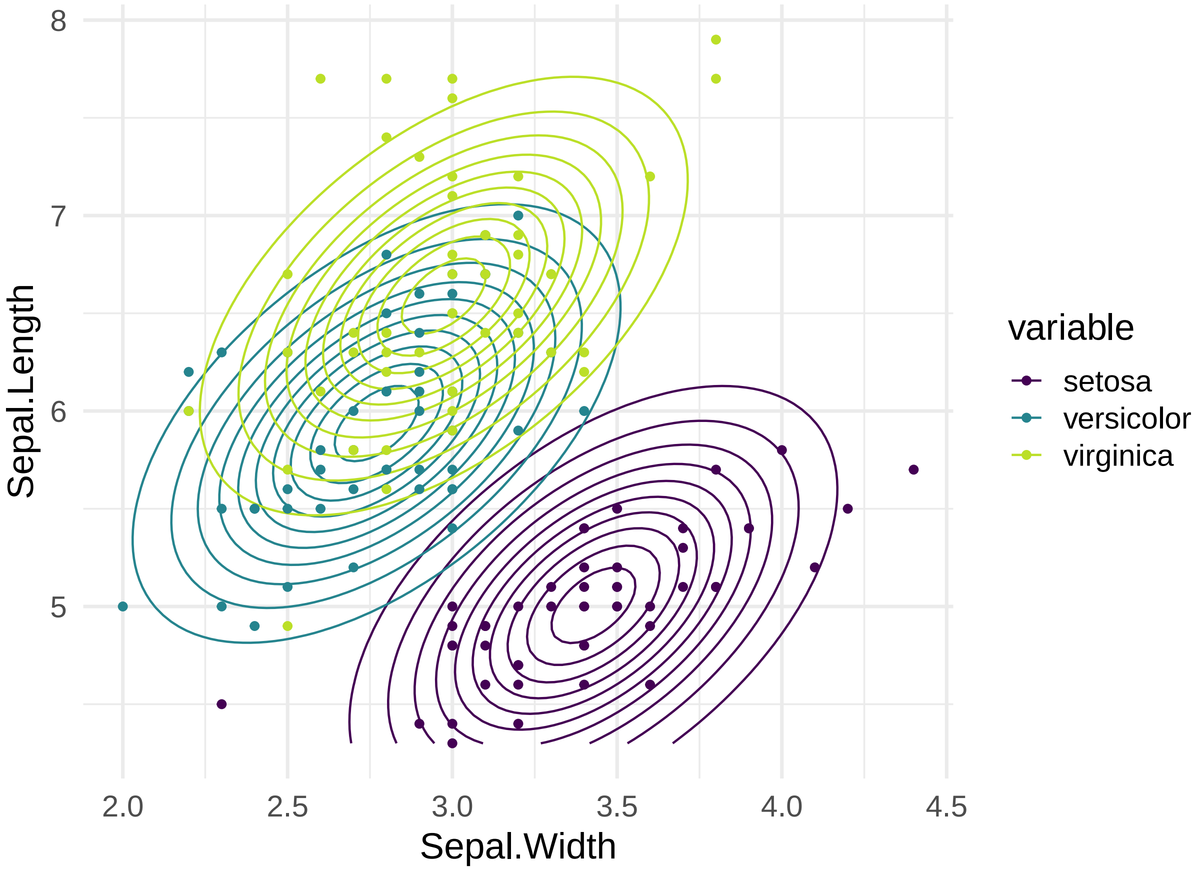
```
library(mvtnorm)

# Train lda and return means of the classes and covariance.
# This defines the likelihood completely.
get_mvgaussian_lda <- function(data, target, level, features) {
  classif_task <- TaskClassif$new(id = "mvg_task",
    backend = data[, c(features, target)],
    target = target
  )
  lda_learner <- lrn("classif.lda")
  lda_learner$train(task = classif_task)

  list(
    mean = lda_learner$model$means[level, features],
    # ?MASS::lda defines 'scaling' as
    # "a matrix which transforms observations to discriminant functions,
    # normalized so that within groups covariance matrix is spherical."
    # -->so the inverse square root of the covariance:
    sigma = solve(tcrossprod(lda_learner$model$scaling[features, ])),
    type = "mv_gaussian",
    features = features
  )
}

# creates a likelihood object by combining the data with
# the specification of the likelihood
# (this can be written way more elegantly with S3 methods)
likelihood <- function(likelihood_def, data) {
  # will be extended below when we define additional likelihood types
  switch(likelihood_def$type,
    mvgaussian_lda = get_mvgaussian_lda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    )
  )
}

# compute likelihood value at given x from a likelihood object
predict_likelihood <- function(likelihood, x) {
  # will be extended below when we define additional likelihood types
  switch(likelihood$type,
    mv_gaussian = dmvnorm(x,
      mean = likelihood$mean,
      sigma = likelihood$sigma
    )
  )
}
```

We write a plot function for multivariate likelihood functions with two features:

```
library(reshape2)

# visualize generative classification method with two features
plot_2D_likelihood <- function(likelihoods, data, X1, X2, target, lengthX1 = 100,
                               lengthX2 = 100) {
  gridX1 <- seq(
    min(data[, X1]),
    max(data[, X1]),
    length.out = lengthX1
  )
  gridX2 <- seq(
    min(data[, X2]),
    max(data[, X2]),
    length.out = lengthX2
  )

  # compute grid coordinates with cartesian product
  grid_data <- expand.grid(gridX1, gridX2)
  features <- c(X1, X2)
  target_levels <- names(likelihoods)
  names(grid_data) <- features

  # predict likelihood values for every target level on the grid
  lik <- sapply(target_levels, function(level) {
    likelihood <- likelihoods[[level]]
    predict_likelihood(likelihood, grid_data[, likelihood$features])
  })

  grid_data <- cbind(grid_data, lik)

  to_plot <- melt(grid_data, id.vars = features)

  ggplot() +
    geom_contour(
      data = to_plot,
      aes_string(x = X1, y = X2, z = "value", color = "variable")
    ) +
    geom_point(data = data, aes_string(x = X1, y = X2, color = target))
}
```

```
# compute lda likelihoods for every target level
lda_liks <- sapply(target_levels, function(level)
  likelihood(
    likelihood_def = list(
      type = "mvgaussian_lda", target = target,
      level = level, features = features
    ),
    iris_train
  ),
simplify = FALSE
)
```

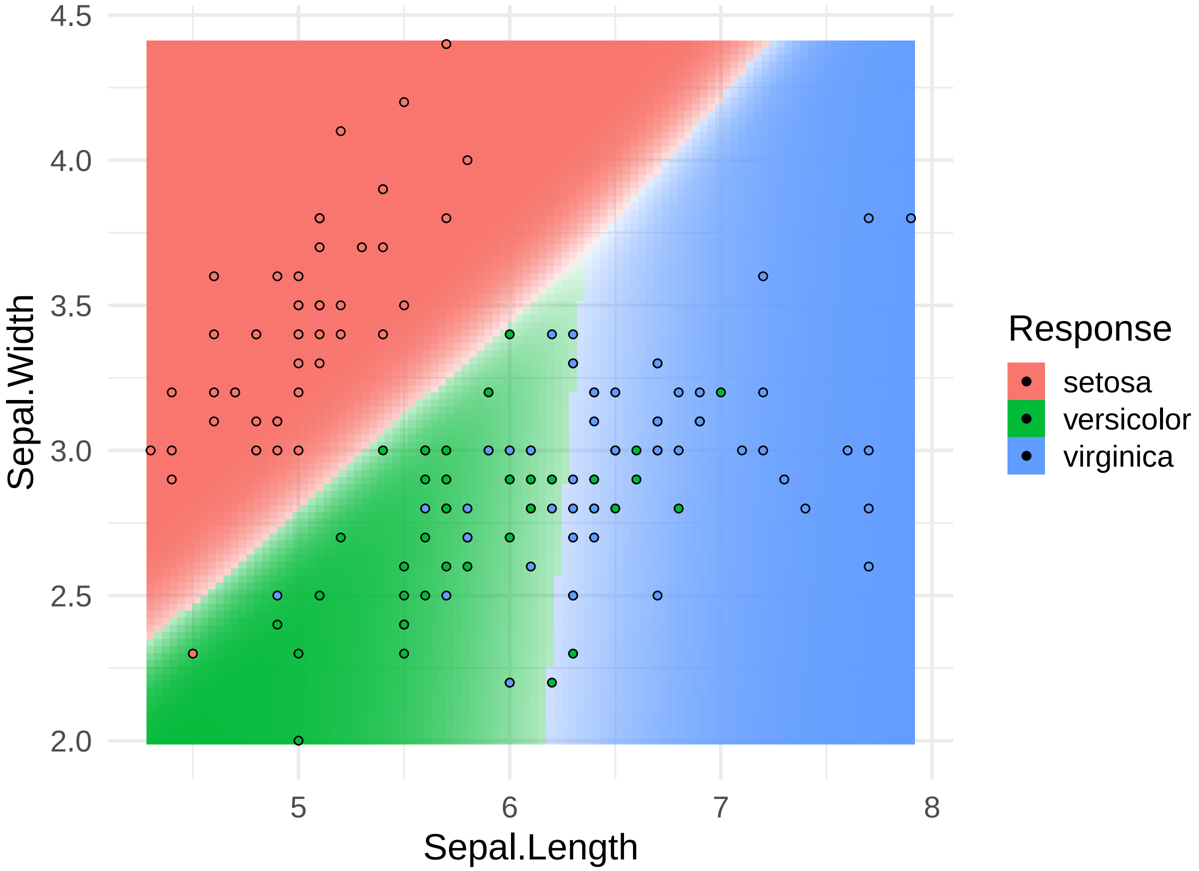
```r
plot_2D_likelihood(lda_liks, iris_train, "Sepal.Width", "Sepal.Length", target)
```



We clearly see that all class distributions are modeled with the same covariance matrix - the shape and orientation of the contour lines of all 3 class distributions is the same.

```r
library(mlr3viz)

plot_learner_prediction(iris_lda_learner, iris_task) +
  guides(alpha = "none", shape = "none")
```

```
## INFO  [10:40:09.878] Applying learner 'classif.lda' on task 'iris_train' (iter 1/1)
```



The resulting decision boundaries are linear – even though we can't really clearly see that in the contour plot above.

### Quadratic discriminant analysis (QDA)

In QDA, we model the likelihood as a multivariate normal distribution s.t.

$$p(x|y=k) = \frac{1}{\pi^{\frac{p}{2}}|\Sigma_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)$$

With:

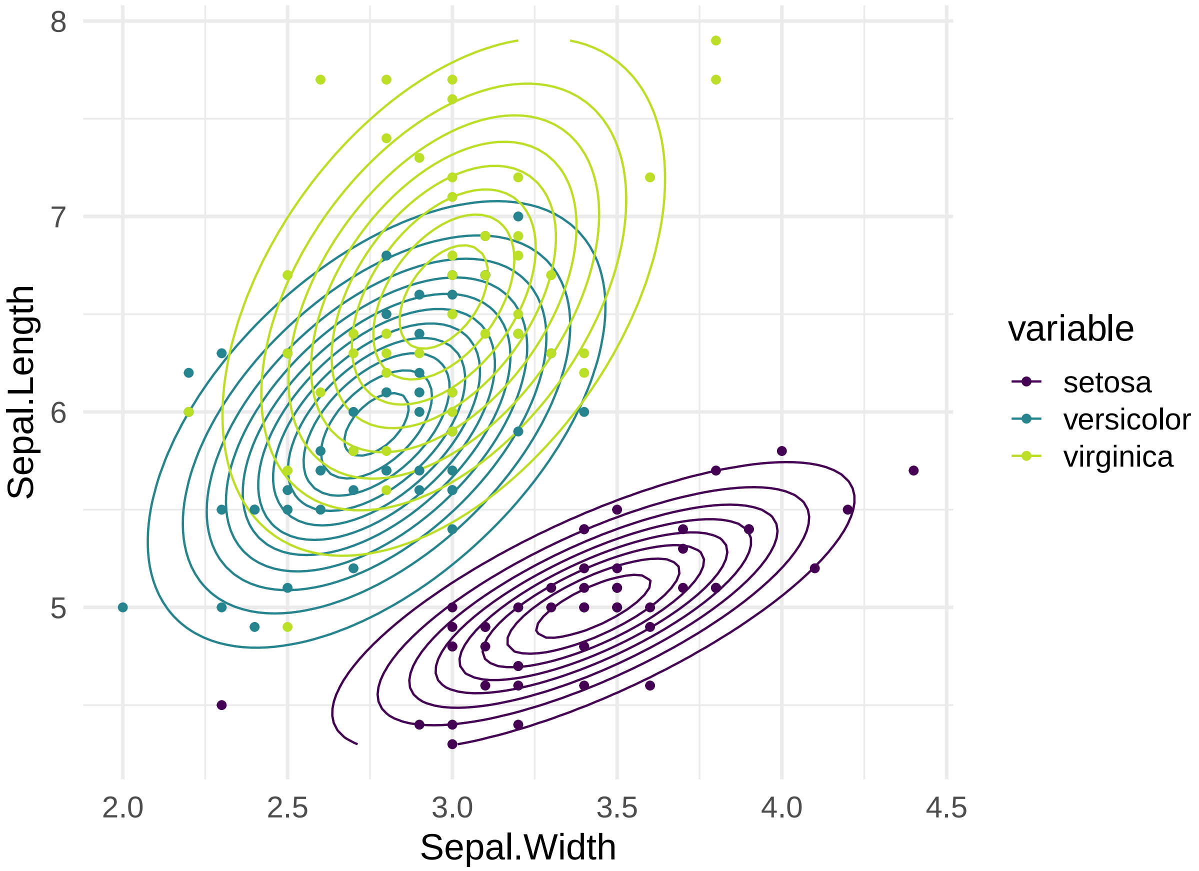
- $\hat{\mu}_k = \frac{1}{n_k}\sum_{i:y^{(i)}=k} x^{(i)}$,
- $\hat{\Sigma}_k = \frac{1}{n_k - 1}\sum_{i:y^{(i)}=k}(x^{(i)} - \hat{\mu}_k)(x^{(i)} - \hat{\mu}_k)^T$.

This means we estimate a different mean vector and covariance matrix for for every class.

```
iris_qda_learner <- lrn("classif.qda", predict_type = "prob")
iris_qda_learner$train(task = iris_task)
```

We define all we need for our likelihood framework and plot them:

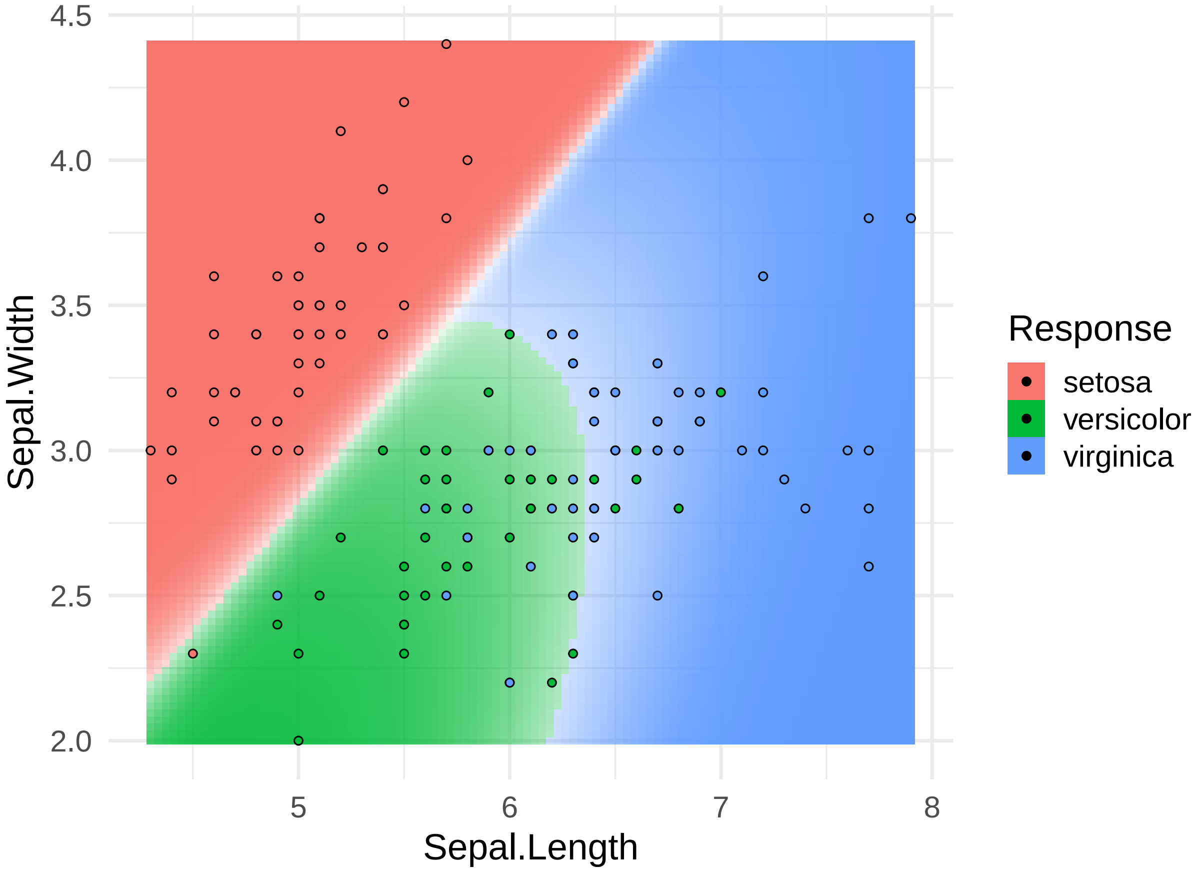
```r
# train QDA and return  class means and class covariance matrices.
# This defines the likelihood completely.
get_mvgaussian_qda <- function(data, target, level, features) {
  classif_task <- TaskClassif$new(id = "mvg_task",
    backend = data[, c(features, target)],
    target = target
  )
  qda_learner <- lrn("classif.qda")
  qda_learner$train(task = classif_task)

  list(
    mean = qda_learner$model$means[level, features],
    # ?MASS::qda "for each group i, scaling[,,i] is an array which transforms
    # observations so that within-groups covariance matrix is spherical."
    # -->so the inverse square root of the covariance
    sigma = solve(tcrossprod(qda_learner$model$scaling[features, ,
                                                        level])),
    type = "mv_gaussian",
    features = features
  )
}

# creates a likelihood object by combining the data with the specification of
# the likelihood
likelihood <- function(likelihood_def, data) {
  switch(likelihood_def$type,
    mvgaussian_lda = get_mvgaussian_lda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    ),
    mvgaussian_qda = get_mvgaussian_qda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    )
  )
}

# compute qda likelihoods for every target level
liks <- sapply(target_levels, function(level)
  likelihood(list(
    type = "mvgaussian_qda", target = target,
    level = level, features = features
  ), iris_train), simplify = FALSE)

plot_2D_likelihood(liks, iris_train, "Sepal.Width", "Sepal.Length", target)
```



As we can see, the covariance is now different in each class.

```
plot_learner_prediction(iris_qda_learner, iris_task) +
  guides(alpha = "none", shape = "none")
```

```
## INFO  [10:40:12.400] Applying learner 'classif.qda' on task 'iris_train' (iter 1/1)
```



Decision boundaries of QDA learners are quadratic (but can of course also look very linear as you can see here….).

### Naive Bayes

Here, we make a "naive" conditional independence assumption:

$$p(x|y=k) = \prod_{j=1}^{p} p(x_j|y=k)$$

Given the class of the target, the joint density/likelihood of all the features is just the product of the univariate densities of each separate feature (i.e., the features are *independent* given $y$).

Again we simply use training data class frequencies as our prior:

$$\hat{\pi}_k = \frac{n_k}{n}.$$

```r
get_priors <- function(df, target) {
  as.list(prop.table(table(df[, target])))
}
```

We now add 3 new likelihood types to our likelihood function:

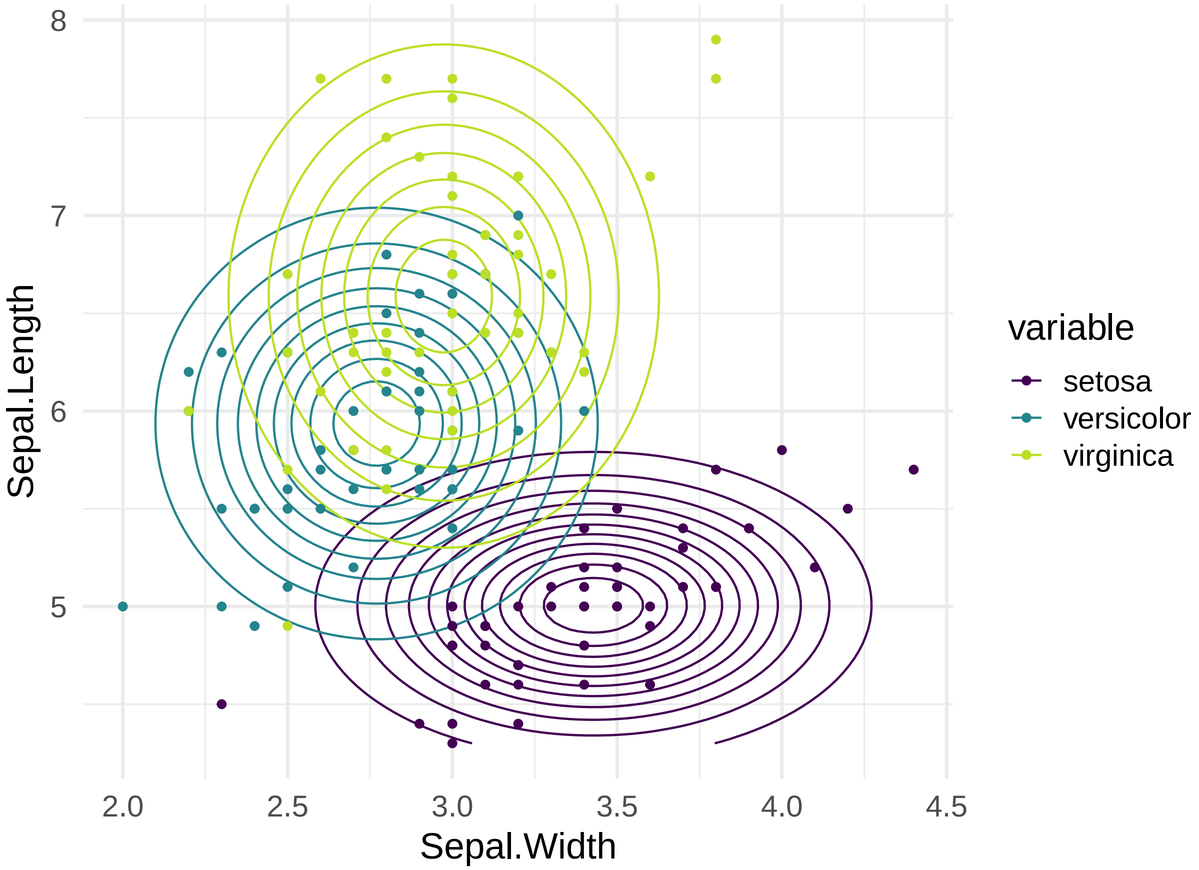
- a one-dimensional Gaussian likelihood for a numerical feature,

- a multinomial likelihood for a categorical feature,
- a "Naive Bayes" likelihood which depends on the likelihoods of the features. These could in turn be one-dimensional Gaussians or multinomial distributions or whatever.

```r
likelihood <- function(likelihood_def, data) {
  switch(likelihood_def$type,
    mvgaussian_lda = get_mvgaussian_lda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    ),
    mvgaussian_qda = get_mvgaussian_qda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    ),
    gaussian = list(
      mean = mean(data[, likelihood_def$features]),
      sd = sd(data[, likelihood_def$features]),
      features = likelihood_def$features, type = "gaussian"
    ),
    # simply computes relative frequencies in the training data:
    categorical = list(
      freq_table = prop.table(table(data[, likelihood_def$features])),
      features = likelihood_def$features, type = "categorical"
    ),
    # the Naive Bayes likelihood is defined by the univariate likelihood
    # definitions of its features. Hence we need to recursively call the
    # likelihood function for every feature.
    naivebayes = list(
      likelihoods = sapply(likelihood_def$likelihood_defs,
        function(likelihood_def)
          likelihood(likelihood_def, data),
        simplify = FALSE
      ),
      features = likelihood_def$features, type = "naivebayes"
    )
  )
}

# helper function for automatic Naive Bayes likelihood computation for every
# target level
# - default likelihood for a numerical feature: gaussian
# - default likelihood for a categorical feature: multinomial
get_naivebayes_likelihoods <- function(df, target, likelihood_defs = NULL) {
  X <- df[, !(names(df) %in% target)]
  features <- colnames(X)
  target_levels <- levels(df[, target])

  if (is.null(likelihood_defs)) {
    # determine default likelihood defintions
    likelihood_defs <- sapply(features, function(feature)
      list(
        args = list(),
        features = feature,
        type =
          ifelse(is.factor(df[, feature]),
            "categorical",
            "gaussian"
          )
      ),
    simplify = FALSE,
    USE.NAMES = TRUE
    )
  }

  # define Naive Bayes likelihood
  naivebayes_def <- list(
    type = "naivebayes",
    likelihood_defs = likelihood_defs,
    features = features
  )
  # loop over all target classes and compute conditional densities
  sapply(
    target_levels,
    function(level) {
      likelihood(naivebayes_def, X[df[[target]] == level, features])
    },
    simplify = FALSE
  )
}
```

```
# predict function extended for the new likelihood types
predict_likelihood <- function(likelihood, x) {
  if (is.data.frame(x)) x <- x[, likelihood$features]
  switch(likelihood$type,
    mv_gaussian = dmvnorm(x,
      mean = likelihood$mean,
      sigma = likelihood$sigma
    ),
    gaussian = dnorm(x, mean = likelihood$mean, sd = likelihood$sd),
    categorical = {
      which_level <- match(x, names(likelihood$freq_table))
      likelihood$freq_table[which_level]
    },
    # to compute the value of the Naive Bayes likelihood we need to predict
    # the likelihood of the every feature. Hence we need to recursively call
    # the predict function for every feature.
    # (To improve numerical stability, the product l_1*...*l_n was transformed
    # to exp(log(l_1)+...+log(l_n))
    naivebayes = exp(rowSums(log(
      sapply(likelihood$likelihoods,
             function(likelihood) predict_likelihood(likelihood, x))
    ))))
}

liks_nb <- get_naivebayes_likelihoods(iris_train, target)
```

```r
plot_2D_likelihood(liks_nb, iris_train, "Sepal.Width", "Sepal.Length", target)
```



Compare this image showing the Naive Bayes class likelihoods to the ones for LDA and QDA – what's different? Why? *Hint:* Which properties of a multivariate Gaussian distribution can you determine from the direction in which its elliptical contours are "tilted"?

```r
norm1 <- function(x) x / sum(abs(x))

# computes posterior class probabilities (if normalize == TRUE)
# or the joint density (if normalize == FALSE), which can be interpreted
# as a score function.
naivebayes <- function(X, priors, nb_likelihood, normalize = FALSE) {
  features <- colnames(X)
  target_levels <- names(priors)

  # compute likelihood for every target level
  lik <- sapply(target_levels, function(level) {
    likelihood <- nb_likelihood[[level]]
    predict_likelihood(likelihood, X[, likelihood$features])
  })

  joint_density <- unlist(priors) * lik

  if (normalize) {
    t(apply(joint_density, 1, norm1))
  } else {
    joint_density
  }
}
```

```
priors <- get_priors(iris_train, target)

X <- iris_train[, !(names(iris_train) %in% target)]
head(naivebayes(X, priors, liks_nb, normalize = TRUE))
```

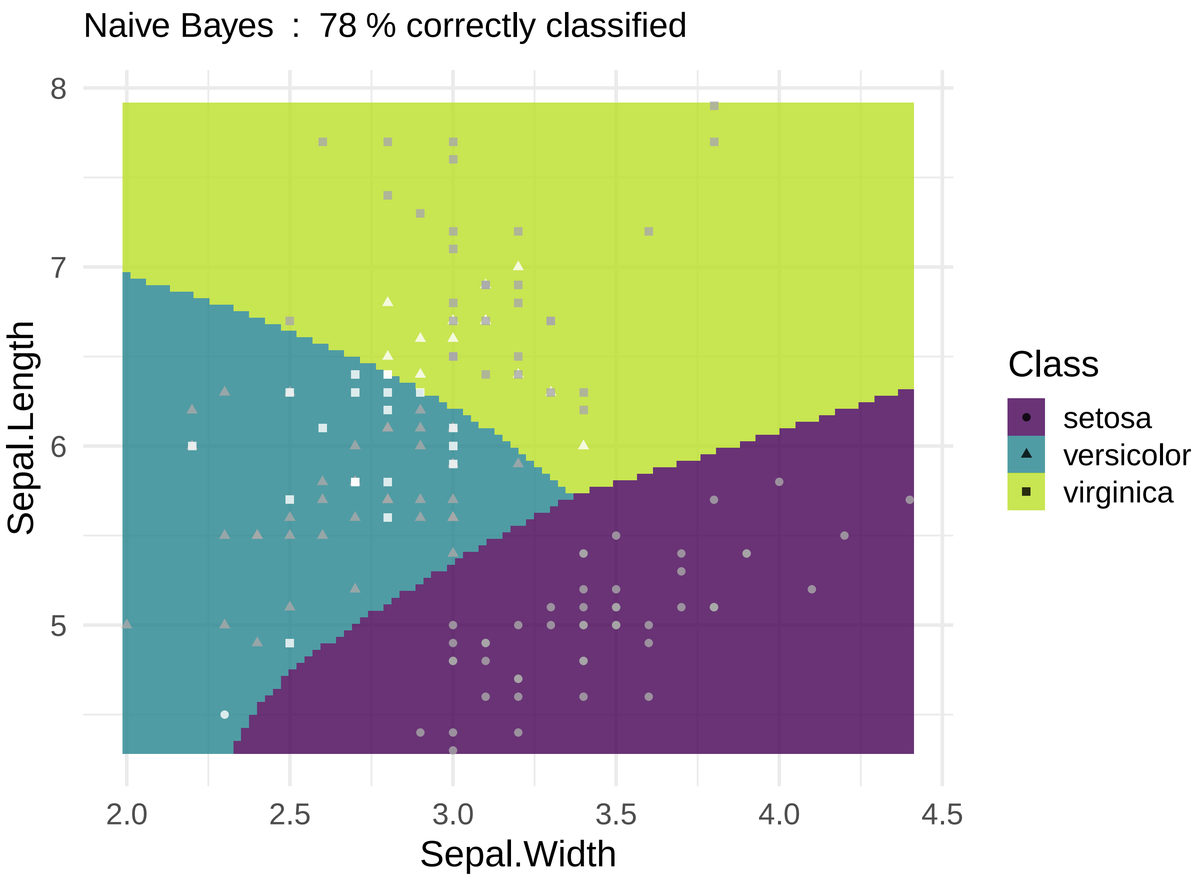
```
##      setosa versicolor virginica
## [1,]  0.973    0.01525   0.01145
## [2,]  0.830    0.13826   0.03148
## [3,]  0.959    0.03073   0.01040
## [4,]  0.944    0.04436   0.01213
## [5,]  0.990    0.00529   0.00481
## [6,]  0.990    0.00295   0.00740
```

```
# function to classify data with Naive Bayes
classify_naivebayes <- function(X, priors, likelihoods) {
  nb <- naivebayes(X, priors, likelihoods)

  list(
    prediction = names(sapply(1:nrow(nb), function(i)
      which.max(nb[i, ]))),
    levels = names(priors)
  )
}
```

```
# function for naive bayes with two features to visualize class
# boundaries X1 and X2 are the names of the two features to use.

plot_2D_naivebayes <- function(priors, likelihoods, X1 = "X1", X2 = "X2",
  # by default, we "predict" the class of the training data:
                               to_classify_labels = as.character(Y),
                               to_classify_data = data.frame(
                                 "X1" = X[, 1],
                                 "X2" = X[, 2]
                               ),
                               lengthX1 = 100, lengthX2 = 100,
                               title = '"Naive Bayes" ~',
                               plot_class_rate = TRUE) {
  plot_2D_classify(
    to_classify_labels = to_classify_labels,
    to_classify_data = to_classify_data,
    classify_method = function(to_classify_data)
      classify_naivebayes(
        X = to_classify_data,
        priors, likelihoods
      ),
    X1, X2,
    lengthX1 = lengthX1, lengthX2 = lengthX2,
    title = title,
    plot_class_rate = plot_class_rate
  )
}
```

Result of our implementation:

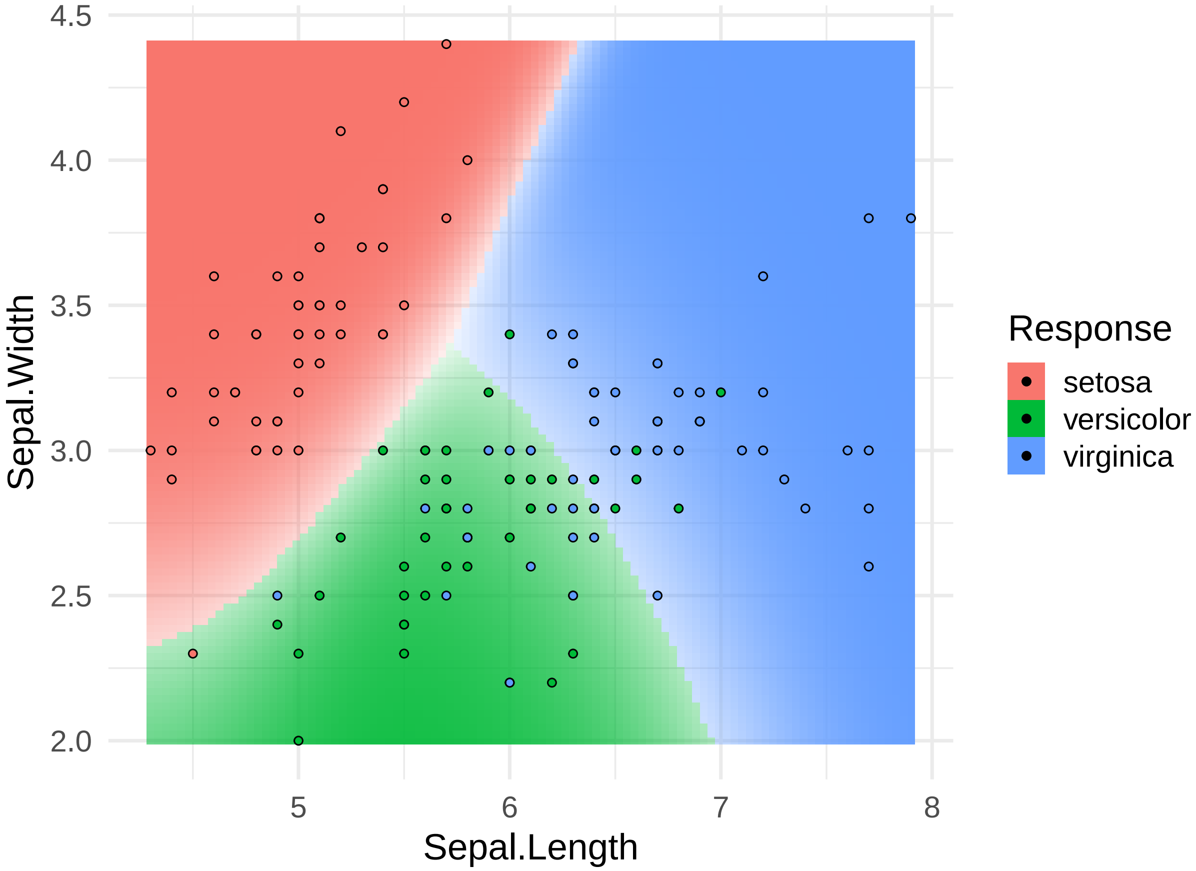
```
plot_2D_naivebayes(
  priors, liks_nb, "Sepal.Width", "Sepal.Length",
  iris_train$Species, X
)
```



Compare with `mlr` implementation:

```
# compare with mlr
iris_nb_learner <- lrn("classif.naive_bayes", predict_type = "prob")
iris_nb_learner$train(task = iris_task)

plot_learner_prediction(iris_nb_learner, iris_task) +
  guides(alpha = "none", shape = "none")
```

```
## INFO  [10:40:16.901] Applying learner 'classif.naive_bayes' on task 'iris_train' (iter 1/1)
```

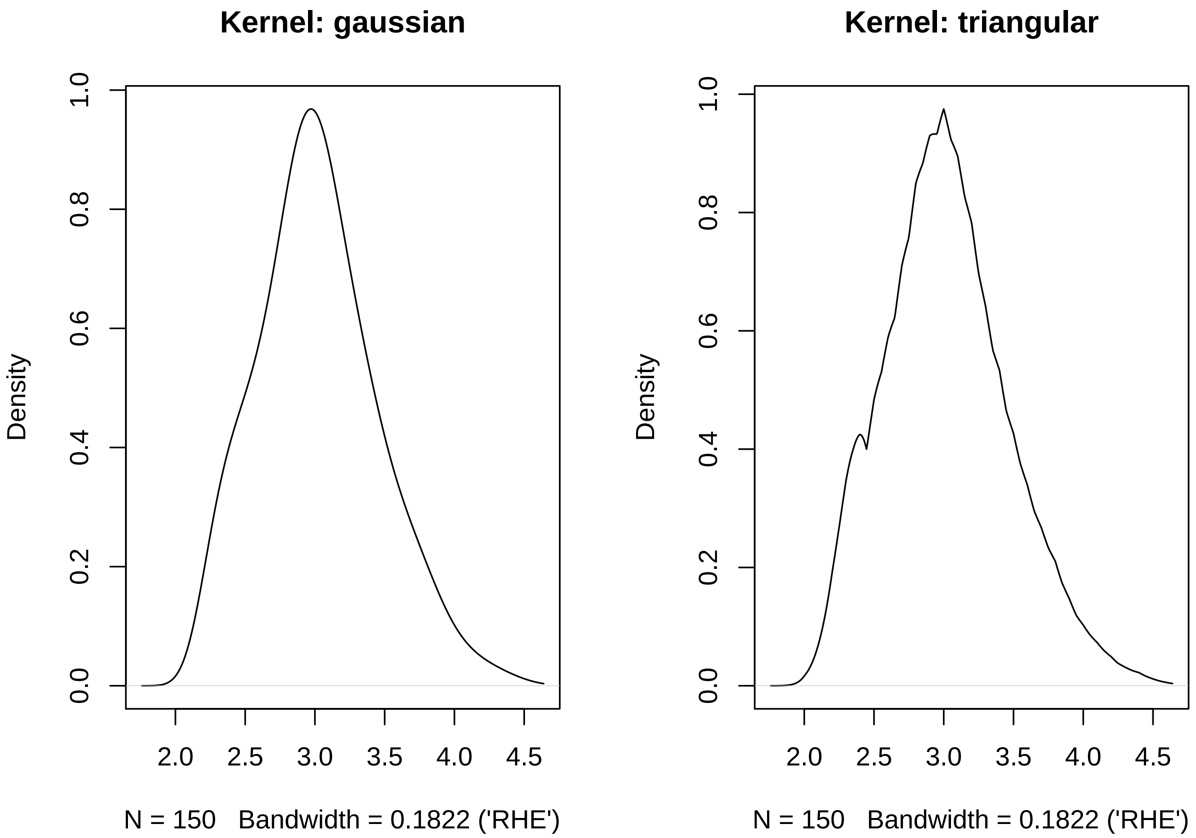


## Additional material: Naive Bayes with kernel densities

In the section above, we assumed that the class-conditional distribution of each and every numerical feature could be modeled sufficiently well with a Gaussian density. What can we do if this does not hold or we just generally want more flexibility?

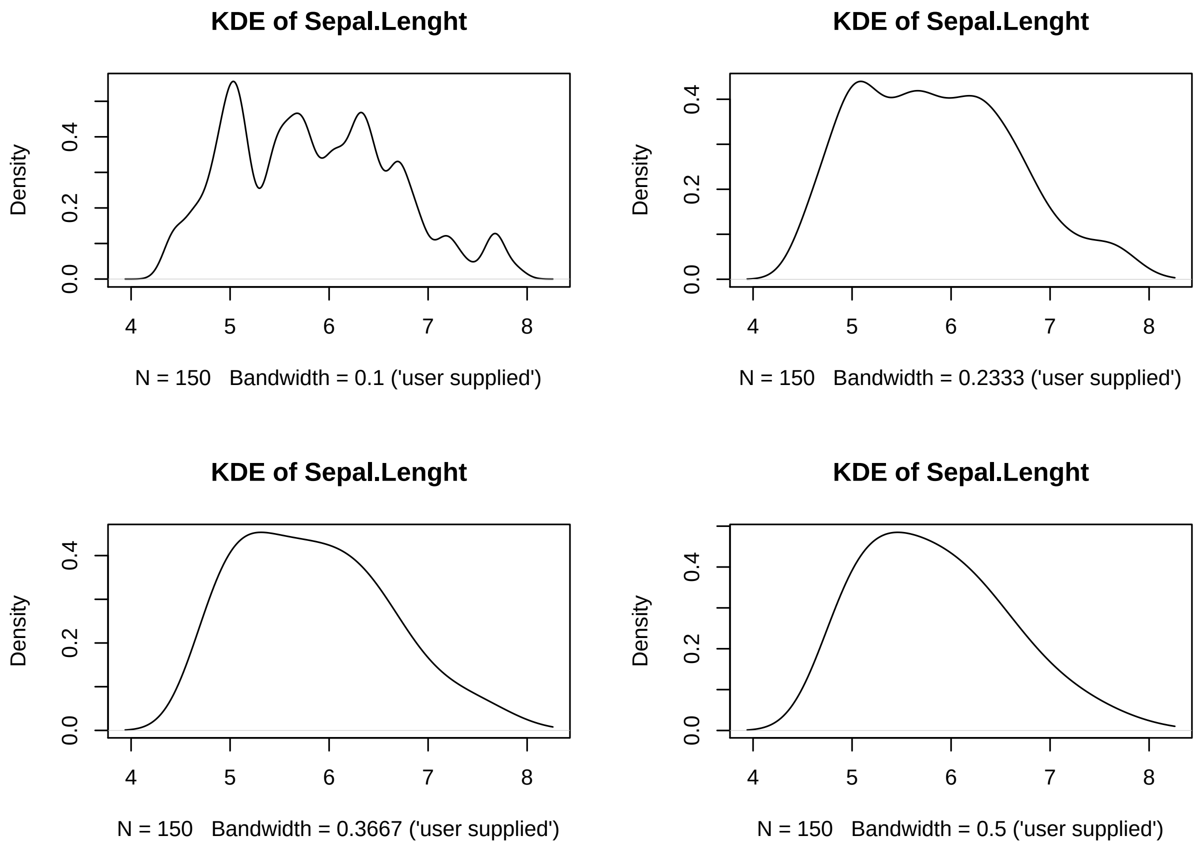
One way to do so is to use a kernel density estimate of the empirical class-conditional distributions of the features as likelihoods. Let's begin by checking out what the kernel density estimate of the *Sepal.Width* feature looks like with the Gaussian and the Triangular kernel functions:

```r
library("kdensity")
library("EQL")
par(mfrow = c(1, 2))
for (kernel in c("gaussian", "triangular")) {
  plot(kdensity(iris_train[, "Sepal.Width"], start = "gumbel", kernel = kernel),
    main = paste0("Kernel: ", kernel)
  )
}
```



Other commonly used kernels can be found here. The kernel width has an even bigger influence on the density estimate:

```r
par(mfrow = c(2, 2))
for (bw in seq(0.1, 0.5, length.out = 4)) {
  plot(kdensity(iris_train[, "Sepal.Length"],
    start = "gumbel",
    kernel = "gaussian", bw = bw
  ),
  main = paste0("KDE of Sepal.Lenght")
  )
}
```

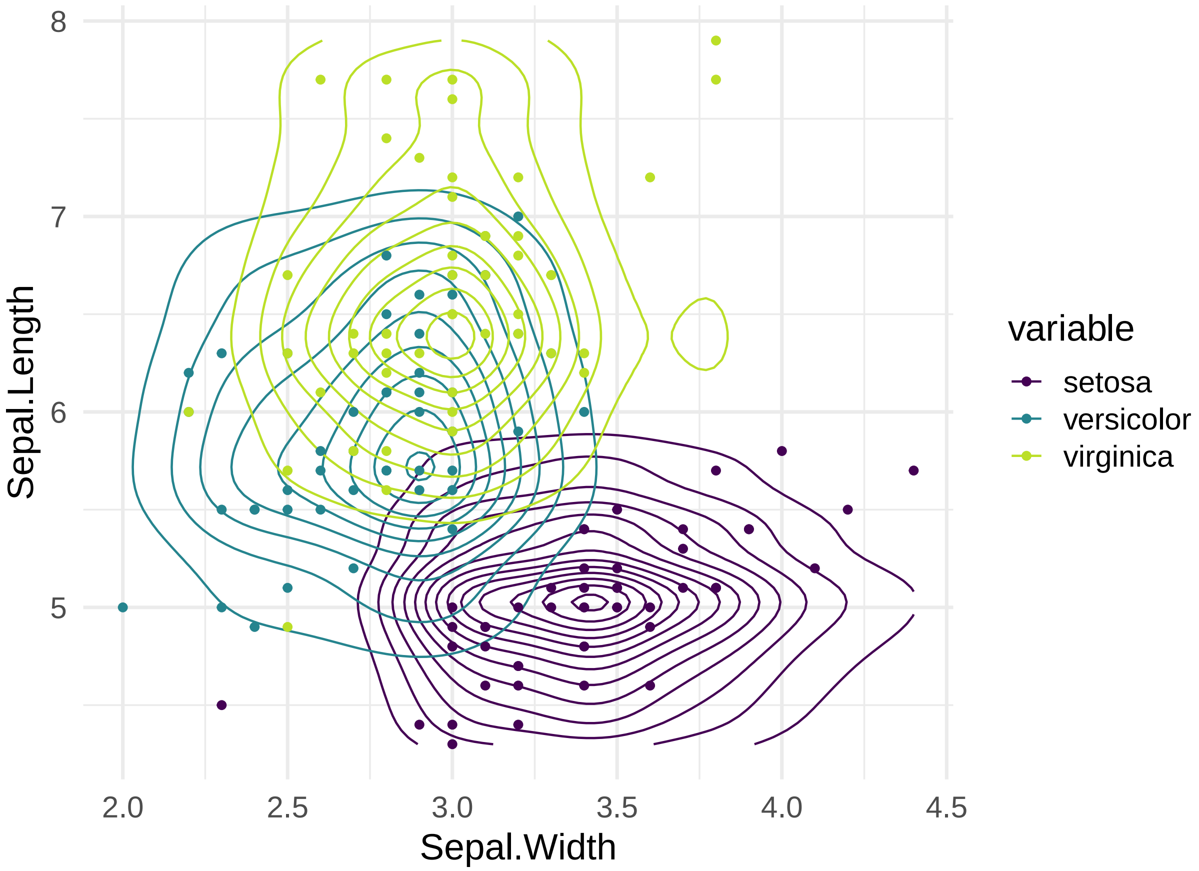


We see that with a broader kernel width (i.e., higher bandwidth `bw`) the density estimate becomes smoother, but smaller features are lost.

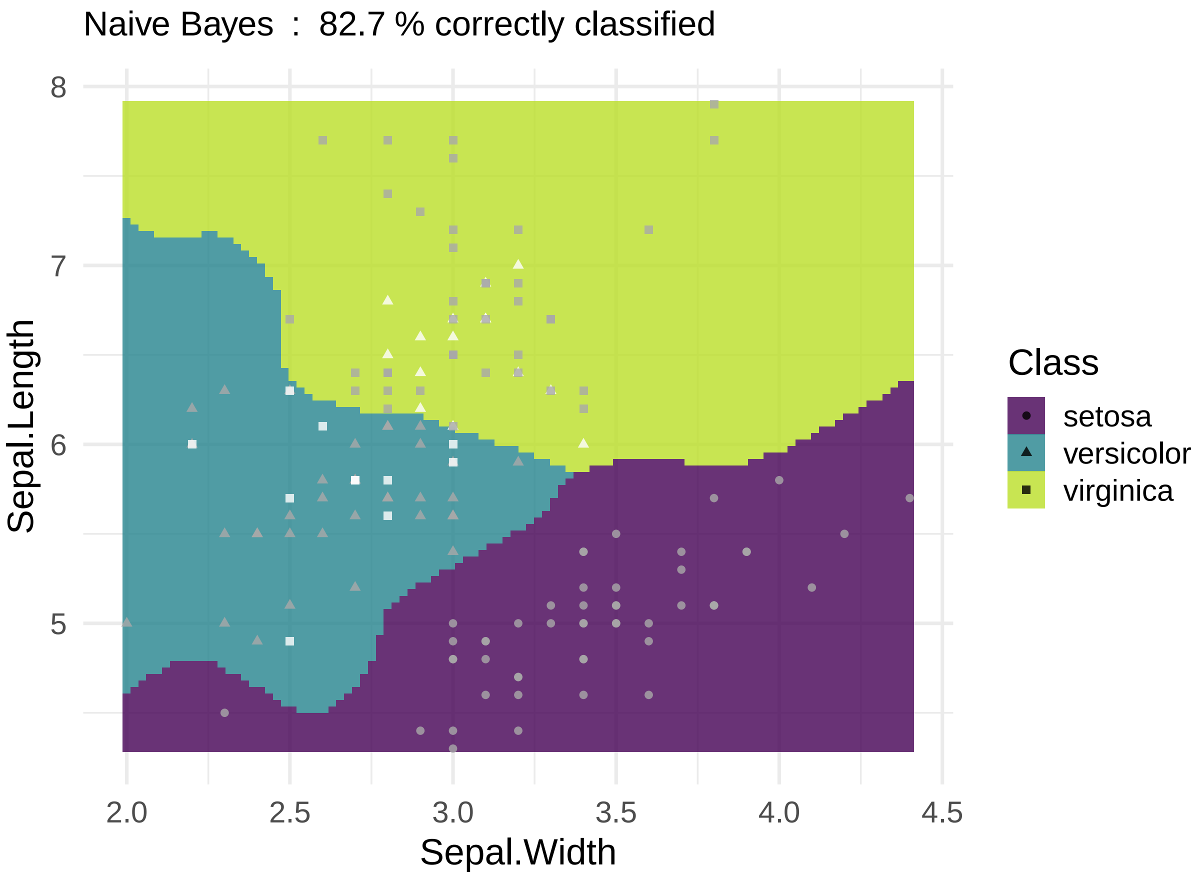
We extend our Naive Bayes framework with the ability to use kernel density estimates and see how it performs on our training data:

```r
likelihood <- function(likelihood_def, data) {
  switch(likelihood_def$type,
    mvgaussian_lda = get_mvgaussian_lda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    ),
    mvgaussian_qda = get_mvgaussian_qda(
      data, likelihood_def$target,
      likelihood_def$level,
      likelihood_def$features
    ),
    gaussian = list(
      mean = mean(data[, likelihood_def$features]), sd =
        sd(data[, likelihood_def$features]),
      features = likelihood_def$features, type = "gaussian"
    ),
    categorical = list(
      freq_table = prop.table(table(data[, likelihood_def$features])),
      features = likelihood_def$features, type = "categorical"
    ),
    kde = list(
      kde =
        # the result here will be (something like) a density function,
        # that we can simply evaluate by doing "lik$kde(x)"
        do.call(kdensity, append(likelihood_def$args,
                                 structure(list(data[, likelihood_def$features]),
                                           names = "x"))),
      type = "kde", features = likelihood_def$features
    ),
    naivebayes = list(
      likelihoods = sapply(likelihood_def$likelihood_defs,
        function(likelihood_def)
          likelihood(likelihood_def, data),
        simplify = FALSE
      ),
      features = likelihood_def$features, type = "naivebayes"
    )
  )
}

predict_likelihood <- function(likelihood, x) {
  if (is.data.frame(x)) x <- x[, likelihood$features]
  switch(likelihood$type,
    mv_gaussian = dmvnorm(x,
      mean = likelihood$mean,
      sigma = likelihood$sigma
    ),
    gaussian = dnorm(x, mean = likelihood$mean, sd = likelihood$sd),
    categorical = {
      which_level <- match(x, names(likelihood$freq_table))
      likelihood$freq_table[which_level]
    },
    kde = likelihood$kde(x),
    naivebayes = exp(rowSums(log(sapply(likelihood$likelihoods,
                                        function(likelihood)
      predict_likelihood(likelihood, x)))))
  )
}
```

```r
liks_nb_kde <- get_naivebayes_likelihoods(
  iris_train, target,
  sapply(features, function(feature)
    list(
      args = list(),
      features = feature,
      type = "kde"
    ),
  simplify = FALSE,
  USE.NAMES = TRUE
  )
)

plot_2D_likelihood(liks_nb_kde, iris_train, "Sepal.Width", "Sepal.Length", target)
```



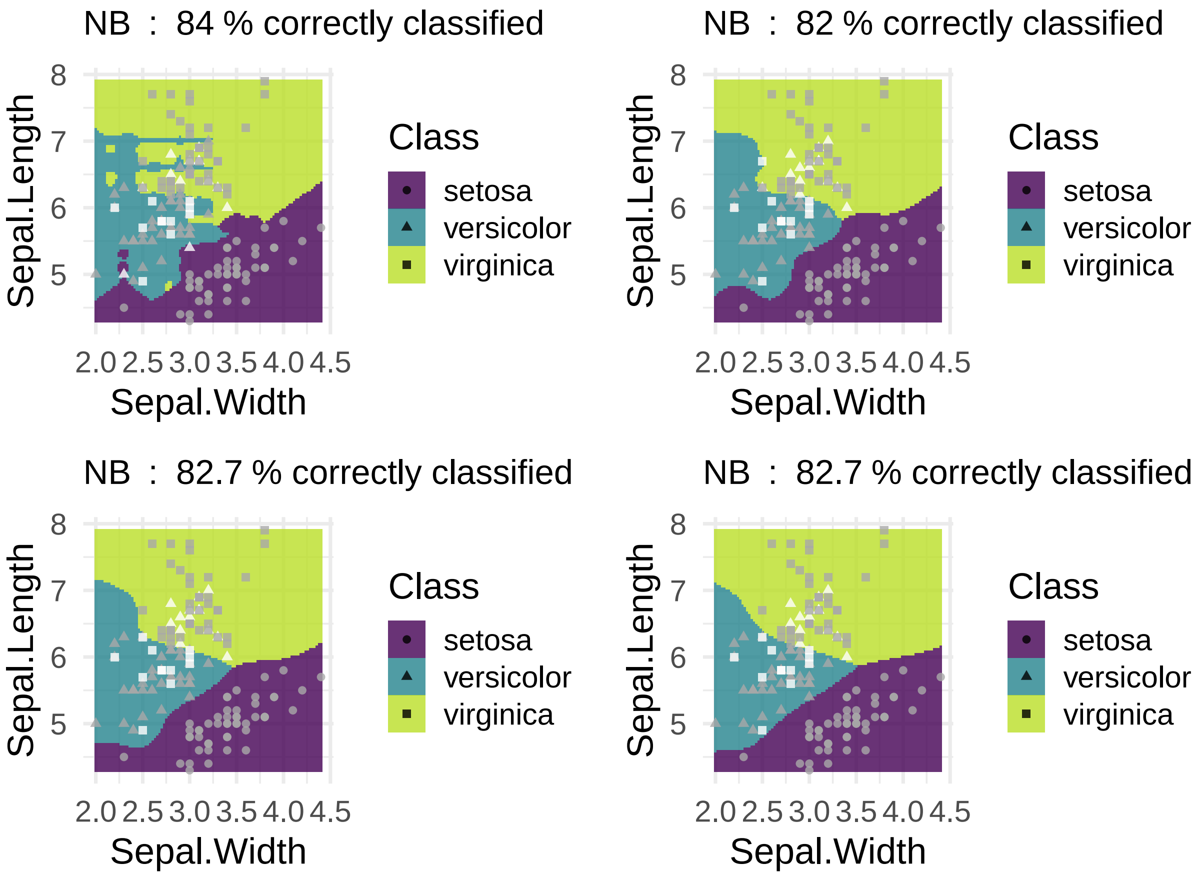
```
plot_2D_naivebayes(priors, liks_nb_kde, "Sepal.Width", "Sepal.Length",
                   iris_train$Species, X)
```



We see that we are now able to model our data more flexibly, with clearly nonlinear decision boundaries.

Now we can take a look how different kernel widths influence the classification on our data:

```r
library(gridExtra)

bw_values <- seq(0.05, 0.3, length.out = 4)
ggplot_list <- lapply(
  bw_values,
  function(bw) {
    liks_kde_bw <- get_naivebayes_likelihoods(
      iris_train, target,
      sapply(features, function(feature)
        list(
          args = list(bw = bw),
          features = feature,
          type = "kde"
        ),
      simplify = FALSE,
      USE.NAMES = TRUE
      )
    )
    plot_2D_naivebayes(priors, liks_kde_bw, "Sepal.Width", "Sepal.Length",
      iris_train$Species, X,
      title = '"NB" ~'
    )
  }
)
do.call(grid.arrange, ggplot_list)
```



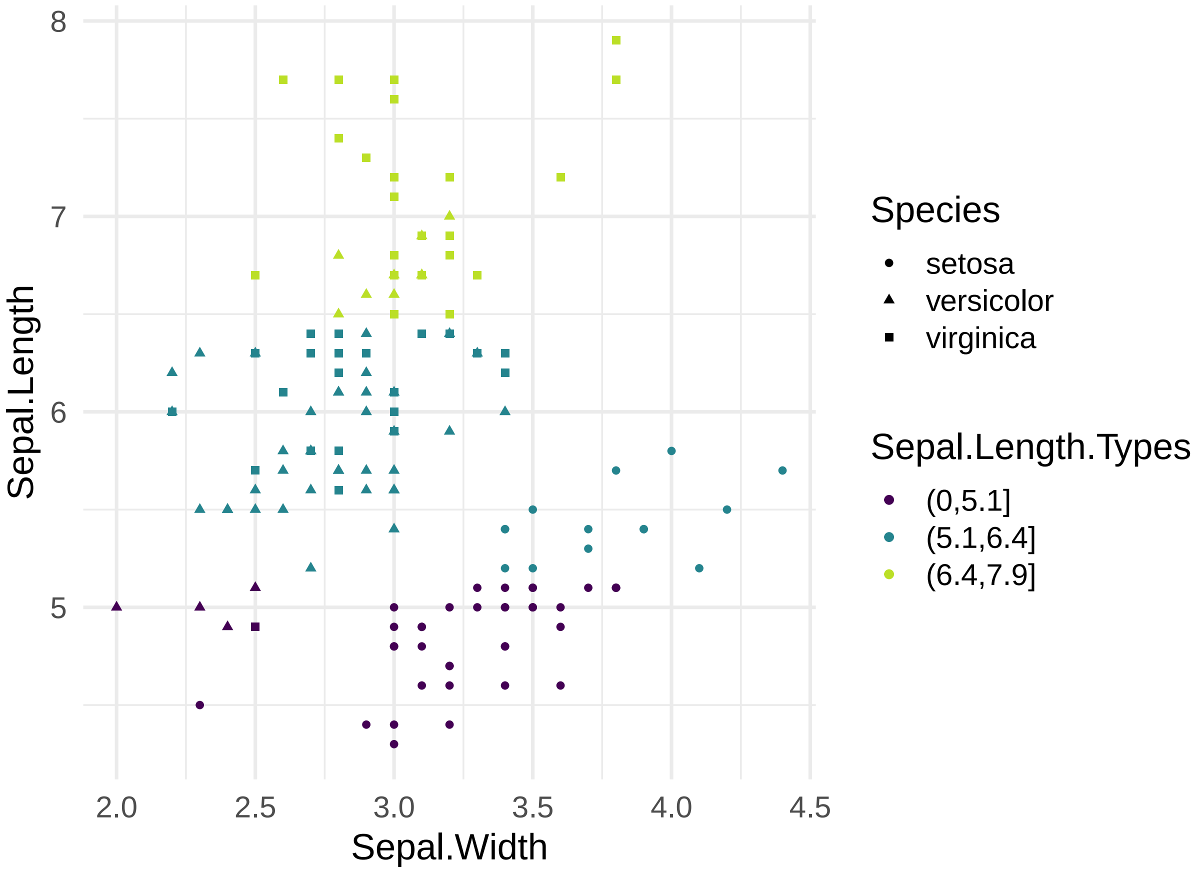
We see that we a smaller kernel width, we can even get a better classification result on our training data. But we actually observe is a phenomenon called overfitting (take a look at the decision boundaries), where the model fits the observed data very well, but is unlikely to generalize well to new, unseen data. Overfitting and how to avoid it will be discussed in depth in subsequent chapters.

### Additional material: Naive Bayes with categorical predictors

If a task includes categorical features, the assumption that the feature vector comes from Gaussian distributions that LDA and QDA and standard Naive Bayes rely on don't make any sense anymore. For the Naive Bayes classifier, however, we can simply use multinomial densities for such categorical features.

To see how well this can work, we turn *Sepal.Length* into a categorical feature and use this as a new feature instead:

```
iris_train$Sepal.Length.Types <-
  cut(iris_train$Sepal.Length,
      breaks = c(0, quantile(iris_train$Sepal.Length, c(.25, .75, 1))))

# visualize discretization
ggplot(iris_train) +
  geom_point(aes(y = Sepal.Length, x = Sepal.Width,
                 color = Sepal.Length.Types, shape = Species))
```



```
features <- c("Sepal.Width", "Sepal.Length.Types")
X <- iris_train[, features]
liks_nb_cat <- get_naivebayes_likelihoods(iris_train[, c(features, target)], target)
head(naivebayes(X, priors, liks_nb_cat, normalize = TRUE))
```

```
##      setosa versicolor virginica
## [1,]  0.982    0.00897   0.00864
## [2,]  0.796    0.15457   0.04903
## [3,]  0.914    0.05752   0.02799
## [4,]  0.865    0.09710   0.03804
## [5,]  0.990    0.00446   0.00545
## [6,]  0.927    0.00982   0.06300
```

```
nb_cat_pred <- classify_naivebayes(X, priors, liks_nb_cat)$prediction

sum(nb_cat_pred == as.character(iris_train$Species)) / nrow(X)
```

```
## [1] 0.76
```

Estimating class-conditional multinomial densities over just 3 categories for `Sepal.Length` works about as well as using univariate Gaussian distributions for the original `Sepal.Length` measurements – this is similar to the classification performance for the original Gaussian Naive Bayes classifier above.