Classification of Unlabeled observations using Mixture and Loop algorithms

Emy Guilbault and Ian Renner

01 Octobre 2020

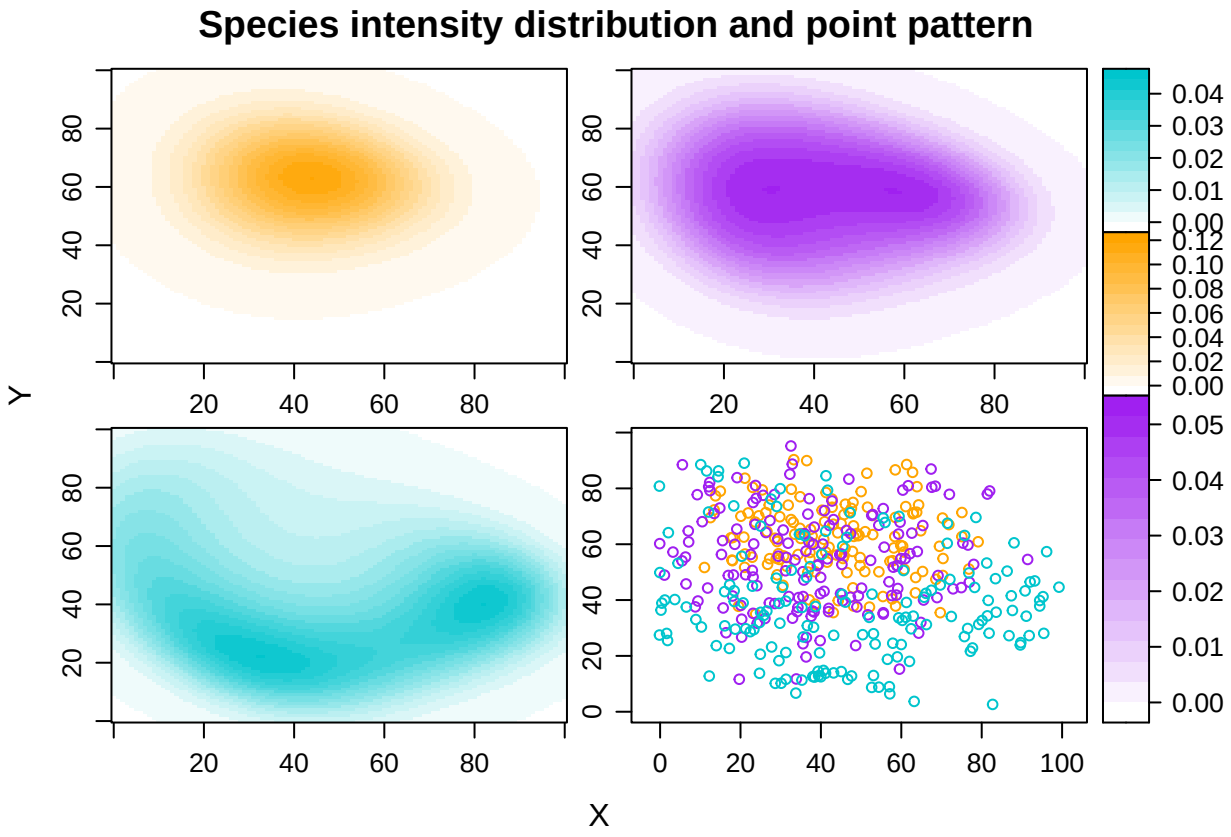
This document demonstrates use of the `ppmMixEngine` and `ppmLoopEngine` functions for data classification and model fitting when some observations have uncertain species identities. These functions and supporting functions are contained in the script `functionTestsim160420-SH.R`. First, we load the various functions and packages we will need.

```
source("functionTestsim160420-SH.r")

library(spatstat)
library(lattice)
library(latticeExtra)
library(gridExtra)
library(caret)
library(viridisLite)
```

Data and environmental covariates

Here, we will load simulated data points for three species and environmental covariates stored in the `PrepData.RData` file. Then, we will display the three true species intensity surfaces as well as the three point patterns generated from these surfaces.

```
load("PrepSimData.RDATA")

# Species intensities created
Lsp1 = levelplot(sp1_int ~ X + Y, main="Sp1",
                 col.regions=colorRampPalette(c("white", "orange"))(50))
Lsp2 = levelplot(sp2_int ~ X + Y, main="Sp2",
                 col.regions=colorRampPalette(c("white", "purple"))(50))
Lsp3 = levelplot(sp3_int ~ X + Y, main="Sp3",
                 col.regions=colorRampPalette(c("white", "turquoise3"))(50))

All_pts = ppp(x=c(sp1_sim$x, sp2_sim$x, sp3_sim$x),
              y=c(sp1_sim$y, sp2_sim$y, sp3_sim$y),
              window = win, marks=c(rep("sp1", sp1_sim$n),
              rep("sp2", sp2_sim$n), rep("sp3", sp3_sim$n)))

All.plot = xyplot(All_pts$y~All_pts$x, All_pts, groups = All_pts$marks,
                  cex = 0.6, col=c("orange", "purple", "turquoise3"))

comb_levObj <- c(Lsp3, All.plot, Lsp1, Lsp2, layout=c(2,2), merge.legends = T)
update(comb_levObj, main="Species intensity distribution and point pattern")
```



One simulation example

Mixture methods

Here, we use a simulated dataset where we hide some of the species label information. In the main article, we hid 20%, 50%, and 80%, but we only present an example with 50% of hidden observations here. The main function that supports the mixture methods is the `ppmMixEngine` function, which includes the following arguments:

- `Known.ppp`: a marked point pattern with the locations with known labels. This must be a `ppp` object as defined in the `spatstat` package.
- `Unknown.ppp`: a point pattern with the locations of the observations with hidden labels. This is also a `ppp` object as defined in the `spatstat` package.
- `quadsenv`: a data frame containing the quadrature point locations with coordinates `x` and `y` as well as the values of the environmental covariates at these locations.
- `ppmform`: a formula object describing the spatial trend in environmental covariates fit to the data
- `initweights`: an argument to specify the initialisation scheme, with options `"knn"` for k nearest neighbours, `"kmeans"` for k-means, `"random"` for random allocation of membership probabilities, `"kps"` for k nearest distances per species, or `"coinF"` for the coin flip allocation with random assignment. See the article for more details on these initialisation methods.
- `k`: the value of k as necessary for the `knn`, `kmeans`, and `kps` methods. Set to 1 by default.
- `classif`: the desired classification approach, with options `"hard"` and `"soft"` for hard and soft classification, respectively. Set to `"hard"` by default.

```
# models
simknn = ppmMixEngine(Known.ppp, Unknown.ppp, quadsenv = Quadmat, classif = "soft",
```

```
                      initweights = "knn", k=1, ppmform = ppmform)

simCF = ppmMixEngine(Known.ppp, Unknown.ppp, quadsenv = Quadmat, classif = "soft",
                     initweights = "CoinF", k=NULL, ppmform = ppmform)

simknn2 = ppmMixEngine(Known.ppp, Unknown.ppp, quadsenv = Quadmat, classif = "hard",

simCF2 = ppmMixEngine(Known.ppp, Unknown.ppp, quadsenv = Quadmat, classif = "hard",
                      initweights = "CoinF", k=NULL, ppmform = ppmform)
```

Loop methods

The `ppmLoopEngine` function applies the Loop methods discussed in the article. The arguments are as follows:

- `Known.ppp`: the same as the corresponding argument in the `ppmMixEngine` function
- `Unknown.ppp`: the same as the corresponding argument in the `ppmMixEngine` function
- `quadsenv`: the same as the corresponding argument in the `ppmMixEngine` function
- `ppmform`: the same as the corresponding argument in the `ppmMixEngine` function
- `addpt`: this argument allows the user to specify the implementation of the Loop algorithm: `"loopA"`, `"loopT"`, or `"loopE"`. See the article for a description of these methods.
- `delta_max`: The value $\delta_{\max}$ required by the `loopT` method. Set to 0.9 by default.
- `delta_min`: The value $\delta_{\min}$ required by the `loopT` method. Set to 0.5 by default.
- `delta_step`: The value δ_{step} required by the `loopT` method. Set to 0.1 by default.
- `num.add`: The initial value a_1 of points to be added if using the `loopE` method. Set to 1 by default.

```
# models
simLoopT = ppmLoopEngine(Known.ppp, Unknown.ppp, addpt = "LoopT",
                         quadsenv = Quadmat, ppmform= ppmform, delta_max=0.5,
                         delta_min=0.1, delta_step=0.1, num.add = NULL)

simLoopE = ppmLoopEngine(Known.ppp, Unknown.ppp, addpt = "LoopE",
                         quadsenv = Quadmat, ppmform= ppmform, delta_max=NULL,
                         delta_min=NULL, delta_step=NULL,  num.add = 1)
```

Model summaries

After fitting the model, we can access various features. Regardless of whether using the `ppmMixEngine` or `ppmLoopEngine` functions, the fitted model coefficients can be extracted with the function `coef_fit`.

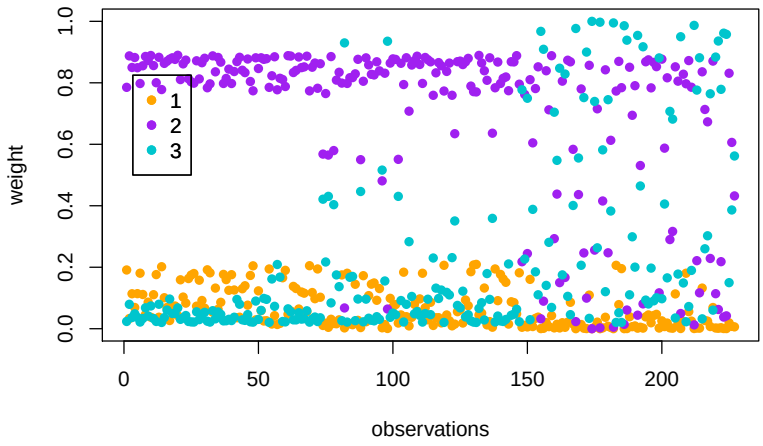
We can produce a map of predicted intensities for objects created with either the `ppmMixEngine` or `ppmLoopEngine` function with the `pred_int` function, which also requires the `quadsenv` data frame. The colour of the map can be controlled by the `colpred` argument, with numerals 1 to 3 representing different colour schemes from the `viridis` package.

Finally, the membership probabilities can be extracted from objects created with either the `ppmMixEngine` or `ppmLoopEngine` function with the `member_prob` function.

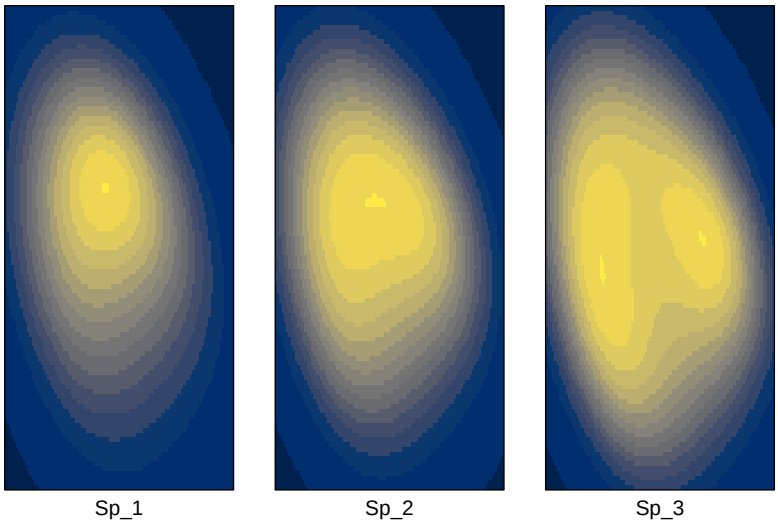
```
# Coefficients
Coef_fit(simknn2)
```

```
##                Sp_1       Sp_2       Sp_3
## Intercept -7.6956342  3.7436791  3.9748467
## v1         3.2153136  2.2274644 -0.9889899
## v1.2      -0.2234948 -1.5001532 -0.8037689
## v2         2.1411952  1.3145881  0.7783801
## v2.2      -0.1746103 -0.6114022 -0.9332154
```

```
# Membership probabilities
Member_prob(simknn2)
```




```
# Predictions
pred_int(simknn2, quadsenv = Quadmat, colpred="1")
```



Evaluating performance

If undertaking simulations as in the main article, we can evaluate performance against some assumed true set of species distributions with the `Perffun` function, which provides the accuracy and meanRSS metrics for evaluating classification performance as well as the IMSE and sumcor metrics for measuring alignment of the fitted intensity surfaces with the true surfaces. The `Perffun` function uses the following arguments:

- `fit`: the fitted model object from either the `ppmMixEngine` or `ppmLoopEngine` function
- `sp.int`: a list containing the true species intensities (as `img` objects)
- `Known.ppp.`: the marked point pattern of locations with known species labels.
- `Unknown_labels.`: the true species labels of the locations for which the labels were hidden.
- `pf`: which performance measure to output. The default `NULL` will output all measures.

- `method`: the type of correlation to be used when computing `sumcor`: (`"pearson"`, `"kendall"`, `"spearman"`). Set to `"pearson"` by default.
- `fun`: A function to be applied to the fitted intensities before evaluating performance. By default, `fun = "Else"`, which does not modify the intensities, but the user can specify `fun = "log"` or `fun = "sqrt"` to base calculation of `sumIMSE` and `sumcor`.

```
# for performance measures
knn.perf = Perffunc(fit = simknn, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))

CF.perf = Perffunc(fit = simCF, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))

knn2.perf = Perffunc(fit = simknn2, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))

CF2.perf = Perffunc(fit = simCF2, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))


LT.perf = Perffunc(fit = simLoopT, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))

LE.perf = Perffunc(fit = simLoopE, sp.int = sp_int.list, Known.ppp.=Known.ppp,
                    Unknown_labels.=Unknown_labels, pf = c(NULL),
                    method=c("pearson"))
```

Method comparison

We can now compare performance of the fitted models. If we have many simulations, we can compare boxplots of the various performance measures.

```
# Comparison between hard and soft classification
ACCvec2 = c(knn.perf$accmat, CF.perf$accmat, knn2.perf$accmat, CF2.perf$accmat)
meanRSSvec2 = c(knn.perf$meanRSS, CF.perf$meanRSS, knn2.perf$meanRSS, CF2.perf$meanRSS)
sumIMSEvec2 = c(knn.perf$sumIMSE, CF.perf$sumIMSE, knn2.perf$sumIMSE, CF2.perf$sumIMSE)
sumcorvec2 = c(knn.perf$sumcor1, CF.perf$sumcor1, knn2.perf$sumcor1, CF2.perf$sumcor1)

Perfmixt = cbind(ACCvec2, meanRSSvec2, sumIMSEvec2, sumcorvec2)
rownames(Perfmixt) = c("knn", "CoinF", "knn-hard", "CoinF-hard")
Perfmixt
```

```
##              ACCvec2 meanRSSvec2 sumIMSEvec2 sumcorvec2
## knn        0.3127753   0.3984819   3.7693657   2.522576
## CoinF      0.3127753   0.3984814   3.7693657   2.522576
## knn-hard   0.3436123   0.4115370   0.7987194   2.859153
## CoinF-hard 0.3436123   0.4027826   0.9042866   2.876304
```

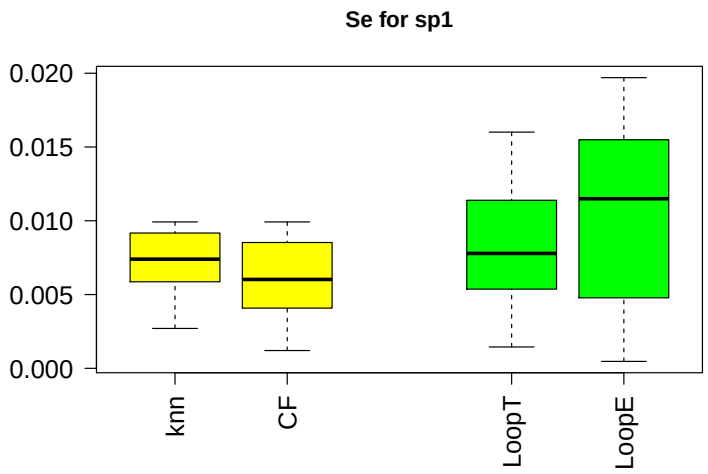
```
# Comparison between Mixture and Loop classifcation
ACCvec = c(knn.perf$accmat, CF.perf$accmat,LT.perf$accmat, LE.perf$accmat)
meanRSSvec = c(knn.perf$meanRSS, CF.perf$meanRSS, LT.perf$meanRSS, LE.perf$meanRSS)
```

```
sumIMSEvec = c(knn.perf$sumIMSE, CF.perf$sumIMSE, LT.perf$sumIMSE, LE.perf$sumIMSE)
sumcorvec = c(knn.perf$sumcor1, CF.perf$sumcor1, LT.perf$sumcor1, LE.perf$sumcor1)

Perfmat = cbind(ACCvec, meanRSSvec, sumIMSEvec, sumcorvec)
rownames(Perfmat) = c("knn", "CoinF", "LoopT", "LoopE")
Perfmat
```

```
##          ACCvec meanRSSvec sumIMSEvec sumcorvec
## knn   0.3127753  0.3984819  3.7693657  2.522576
## CoinF 0.3127753  0.3984814  3.7693657  2.522576
## LoopT 0.3127753  0.3212256  0.6698334  2.920188
## LoopE 0.3524229  0.3439102  3.8225718  2.664444
```

We can also calculate standard errors. We choose to display and compare the prediction standard errors for species 1 only and compare the standard error maps from the 4 methods: knn, CF, LoopT and LoopE using the `se_pred` function. This function only requires the user to input the `fit` argument, the fitted model object from either the `ppmMixEngine` or `ppmLoopEngine` function. The function returns the `se.simdat` object which corresponds to a list of standard errors vectors at the location for a species and the `seplot.vec` object which is the list of standard errors vectors at the quadrature points if the argument `ngrid` is given or at the default grid dimension otherwise.

```
# standard error values for species 1
se.knn = se_pred(fit=simknn)
se.CF  = se_pred(fit=simCF)
se.LT  = se_pred(fit=simLoopT)
se.LE  = se_pred(fit=simLoopE)

# boxplots
boxplot(se.knn$se.simdat[[1]], se.CF$se.simdat[[1]],
        se.LT$se.simdat[[1]], se.LE$se.simdat[[1]],
        col = c("yellow","yellow","green","green","green"),
        names = c("knn", "CF", "LoopT", "LoopE"), at = c(1,2, 4,5),
        ylab = "", main ="Se for sp1", las = 2, cex=1.3, cex.lab=1.3, cex.axis=1.3)
```




```
# standard error plots
l.knn = levelplot(se.knn$seplot.vec[[1]] ~ se.knn$Xplot + se.knn$Yplot,
                  col.regions=viridis(20),
                  main="knn method standard error", ylab=NULL, xlab=NULL)
```

```
l.CF = levelplot(se.CF$seplot.vec[[1]] ~ se.CF$Xplot + se.CF$Yplot,
                 col.regions=viridis(20),
                 main="CoinF method standard error", ylab=NULL, xlab=NULL)
l.LT = levelplot(se.LT$seplot.vec[[1]] ~ se.LT$Xplot + se.LT$Yplot,
                 col.regions=viridis(20),
                 main="LoopT method standard error", ylab=NULL, xlab=NULL)
l.LE = levelplot(se.LE$seplot.vec[[1]] ~ se.LE$Xplot + se.LE$Yplot,
                 col.regions=viridis(20),
                 main="LoopE method standard error", ylab=NULL, xlab=NULL)

grid.arrange(l.knn, l.CF, l.LT, l.LE, ncol=2)
```