Species Distribution Modelling II

MD

02/03/2021

In week 4 we looked at the use of envelope models, general linear models, maxent and point process models for estimating species distribution (*Bradypus variegatus*). According to the cross-validation approach taken (K-fold partitioning) We achieved a good level of prediction. However, as we saw in the lecture, ecological data tend to be spatially dependent, meaning that observations in close proximity will be more similar than those at a distance. In terms of model validation this means that, in the case of K-fold validation in which samples are selected at random, training and test samples may be quite similar for any given covariate. This can (and almost always does) lead to over-optimistic model validation statistics (e.g. AUC scores). A more reliable approach is to ensure that folds used in cross-validation are spatially disjoint i.e. that the geographic distance between training and testing data is maximized. This is particularly important for species distribution modeling where our goal is the prediction across space of our species of interest. Therefore the ability of our model to work across different geographical areas is critical.

In this practical you will explore how spatial dependence affects model validation and the options available for overcoming such problems. We will use statistical models and machine learning algorithms on data for *Bradypus variegatus*.

Using these data, you will then learn how, for machine learning algorithms, model hyperparameters can be tuned to improve performance. You can think of model hyperparameters as the “settings” for any given model. For example, hyperparameter “mtry” is the number of explanatory variables that are sampled for each tree in a random forest algorithm and “node size” denotes the minimum number of cases allowed at the end of each decision tree. “Parameters” on the other hand are the decisions that are made by the learner during training (so “hyper” refers to the fact that we set these properties BEFORE the learning process begins).

In part four of the practical we will adopt a point process modeling approach to understanding and making predictions on these same data.

Part One: creating learners in the MLR (“Machine Learning in R”) package

The MLR package provides a suite of tools to explore, create and test modelling strategies that are to some degree compatible with a machine learning approach.

A key concept in MLR and in machine learning in general is the concept of the “Learner”. These are the objects that generate information (i.e. they “learn”) about the data provided. The latter object (the data) are generally formulated as “tasks”. So the basic logic is that we “learn” how to carry out our “task” and then use this experience for optimal prediction. If that all sounds a little inaccessible, it may help to simply think of Learners as models (as in classical statistics e.g. regression models) and Tasks as data sets. Typically, creating a task involves the choosing of a data set and telling R which column holds the species observations.

Okay, let’s set up our first task. First we will get the same data as we used last week. If you have the code already from the Week 5 practical (SDM Part One) you can copy it to a new R script and run to create the **all.cov** data frame. For completeness, I’ve reproduced the code below.

As you will only be working with data extracted from inside R packages or downloaded from online sources it is not critical where you set your working directory. However, if you intend to save any of your outputs from this practical then create a folder/directory for the work you are about to do. Remember, you can find the code for saving spatial outputs to file at the end of Practical One of this course.

```
#Load packages

install.packages(c("dismo","maptools","raster","sp","spatstat","adehabitatHR"))

#We also need to load the "MLR" package for this practical

install.packages("mlr")

#in addition we will use machine leraning algorithms from the "kernlab" package

install.packages("kernlab") #install necessary package
```

Next we recreate the data from Week 5. The only difference that we will make in terms of creating the data for analysis is that this week we will trim down the study area. Both machine learning (due to number of tests/iterations) and point process modelling (due to modelling complex patterns) can be computationally intensive. We will use the minimum convex polygon approach that we used in week 3 to select a convex hull around 80% of our bradypus points and clip the study area to this polygon.

```
#access the dismo library

library(dismo)
```

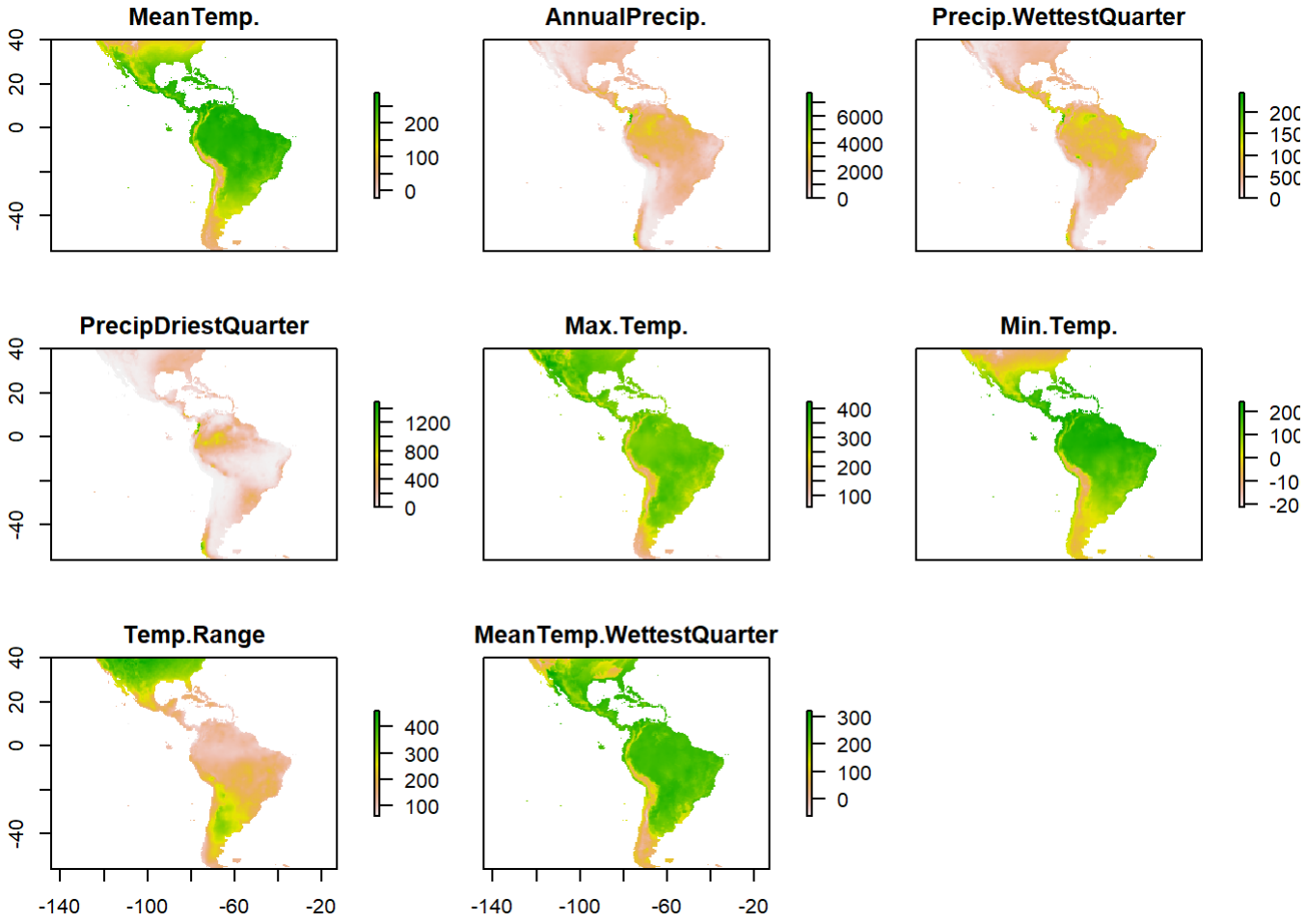
```
## Loading required package: raster
```

```
## Loading required package: sp
```

```
files <- list.files(path=paste(system.file(package="dismo"),
                               '/ex', sep=''), pattern='grd', full.names=TRUE )

files
```

```
## [1] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio1.grd"
## [2] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio12.grd"
## [3] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio16.grd"
## [4] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio17.grd"
## [5] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio5.grd"
## [6] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio6.grd"
## [7] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio7.grd"
## [8] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/bio8.grd"
## [9] "C:/Users/mlibxmda/Documents/R/win-library/4.0/dismo/ex/biome.grd"
```

```
env<-stack(files[1:8])


#use the names() function to label each covariate layer

names(env) <- c("MeanTemp.","AnnualPrecip.","Precip.WettestQuarter","PrecipDriestQuart
er","Max.Temp.","Min.Temp.","Temp.Range","MeanTemp.WettestQuarter")



plot(env)
```




```
SAproj<-"+init=epsg:31972"  # we want projected coordinate in order to calculate dista
nce accurately so we use  the EPSG code for South America

env<-projectRaster(env,crs=SAproj,res=10000) # project raster and set an appropriate r
esolution
```

```
## Warning in showSRID(uprojargs, format = "PROJ", multiline
## = "NO", prefer_proj = prefer_proj): Discarded datum
## Sistema_de_Referencia_Geocentrico_para_las_AmericaS_2000 in Proj4 definition
```

```
#And now plot:
 # first layer of the RasterStack
 plot(env, 1)


bradypus <- gbif("Bradypus","variegatus",sp=TRUE)  #use gbif() to download the data
```

```
## Loading required namespace: jsonlite
```

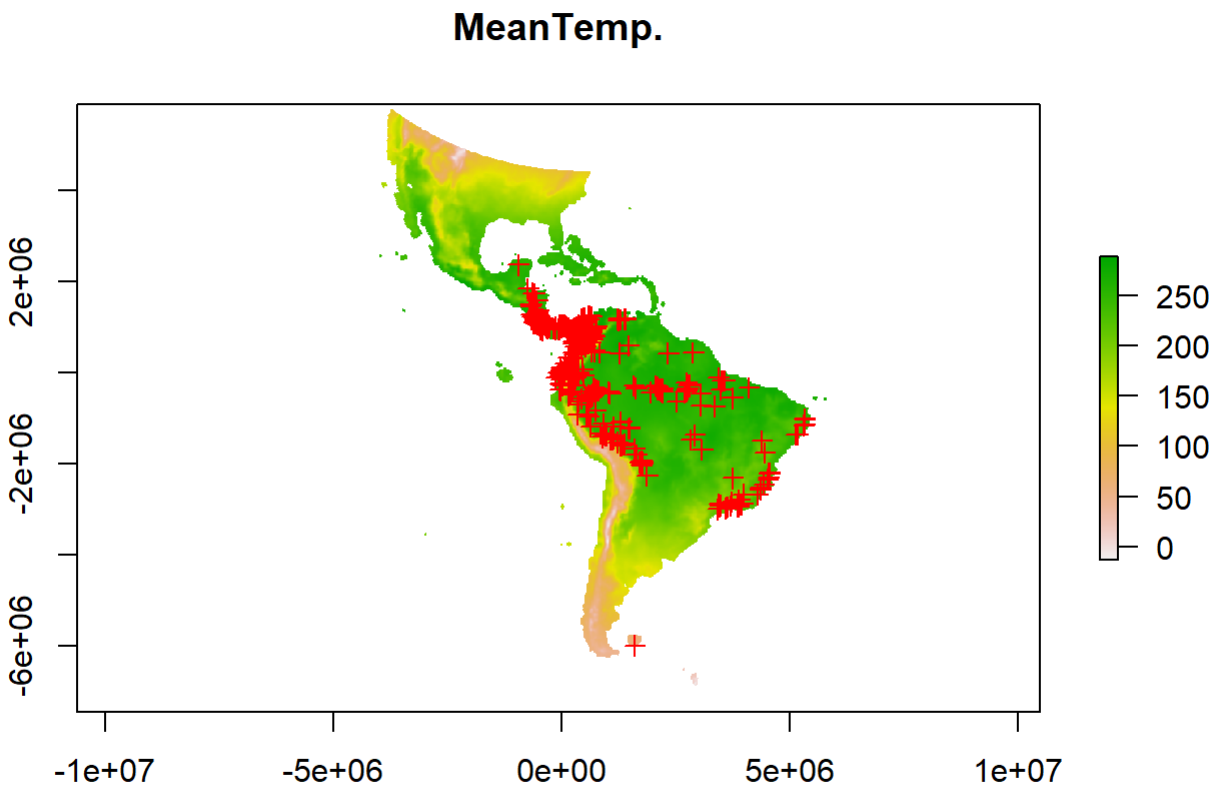
```
## 2691 records found
```

```
## 0-300-600-900-1200-1500-1800-2100-2400-2691 records downloaded
```

```
wgs84<-crs.latlong<-crs("+init=epsg:4326") # create CRS object for the points

crs(bradypus)<-wgs84 #set the CRS for bradypus

bradypus<-spTransform(bradypus,crs(env)) # transform to same projection as the environ
mental data

crs(bradypus)# check
```

```
## CRS arguments:
##  +proj=utm +zone=18 +ellps=GRS80 +units=m +no_defs
```

```
plot(env, 1)



plot(bradypus,add=TRUE,col='red')
```




```
library(adehabitatHR) #library for mcp() function
```

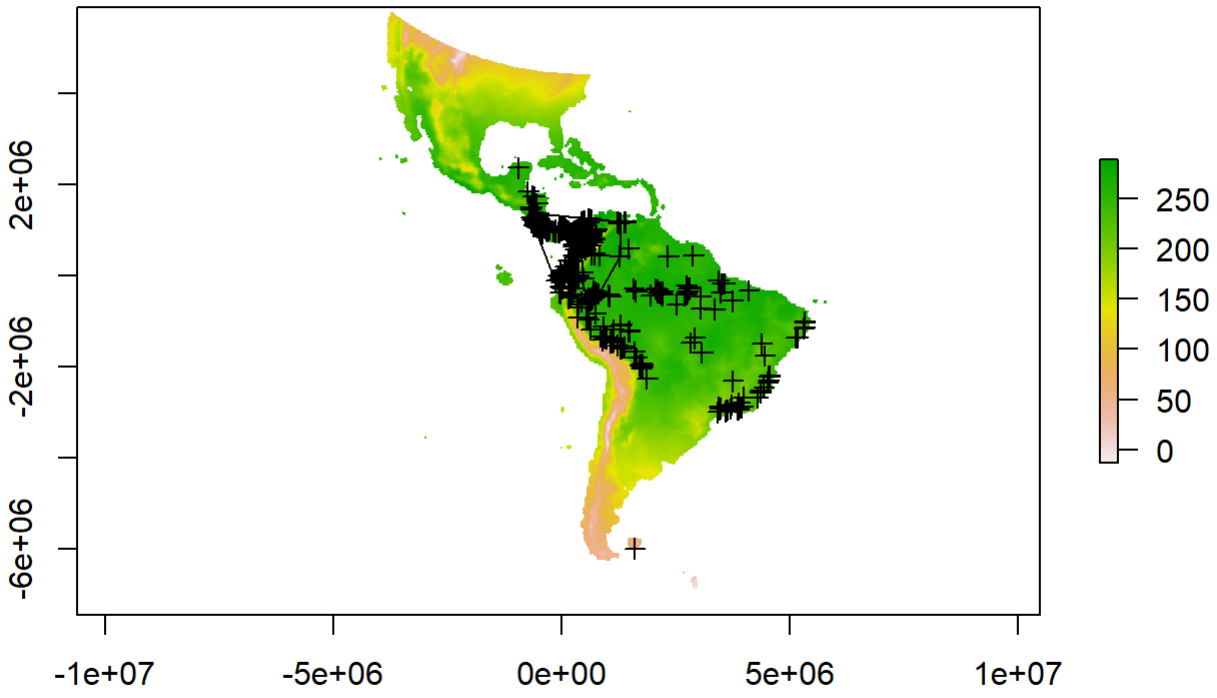
```
## Loading required package: deldir
## deldir 0.2-9      Nickname: "Morpheus and Euripides"
##
##      Note 1: As of version 0.2-1, error handling in this
##      package was amended to conform to the usual R protocol.
##      The deldir() function now actually throws an error
##      when one occurs, rather than displaying an error number
##      and returning a NULL.
##
##      Note 2:  As of version 0.1-29 the arguments "col"
##      and "lty" of plot.deldir() had their names changed to
##      "cmpnt_col" and "cmpnt_lty" respectively basically
##      to allow "col" and and "lty" to be passed as "..."
##      arguments.
##
##      Note 3: As of version 0.1-29 the "plotit" argument
##      of deldir() was changed to (simply) "plot".
##
##      See the help for deldir() and plot.deldir().
##
## Loading required package: ade4
## Loading required package: adehabitatMA
## Registered S3 methods overwritten by 'adehabitatMA':
##   method                       from
##   print.SpatialPixelsDataFrame sp
##   print.SpatialPixels          sp
##
## Attaching package: 'adehabitatMA'
##
## The following object is masked from 'package:raster':
##
##     buffer
##
## Loading required package: adehabitatLT
## Loading required package: CircStats
## Loading required package: MASS
##
## Attaching package: 'MASS'
##
## The following objects are masked from 'package:raster':
##
##     area, select
##
## Loading required package: boot
```

```
bradMCP<-mcp(bradypus,percent=80) #create minimum convex polygon
```

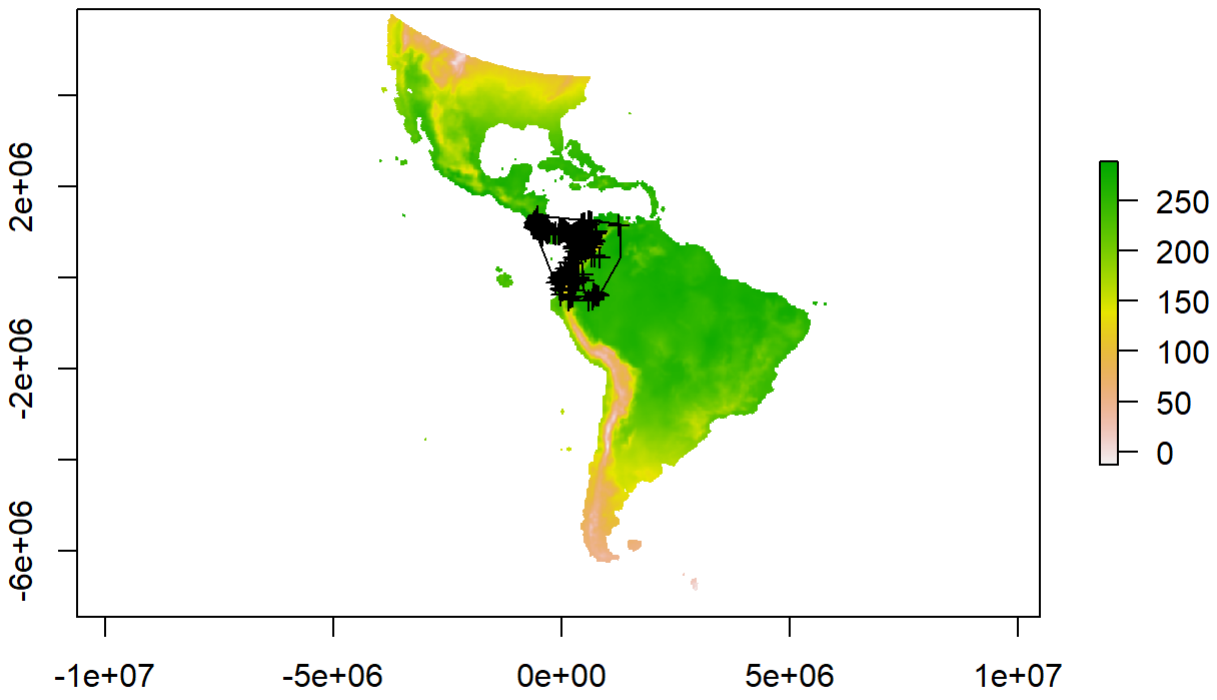
```
## Warning in proj4string(xy): CRS object has comment, which is lost in output
```

```
## Warning in showSRID(uprojargs, format = "PROJ", multiline = "NO", prefer_proj
## = prefer_proj): Discarded datum Unknown based on GRS80 ellipsoid in Proj4
## definition
```

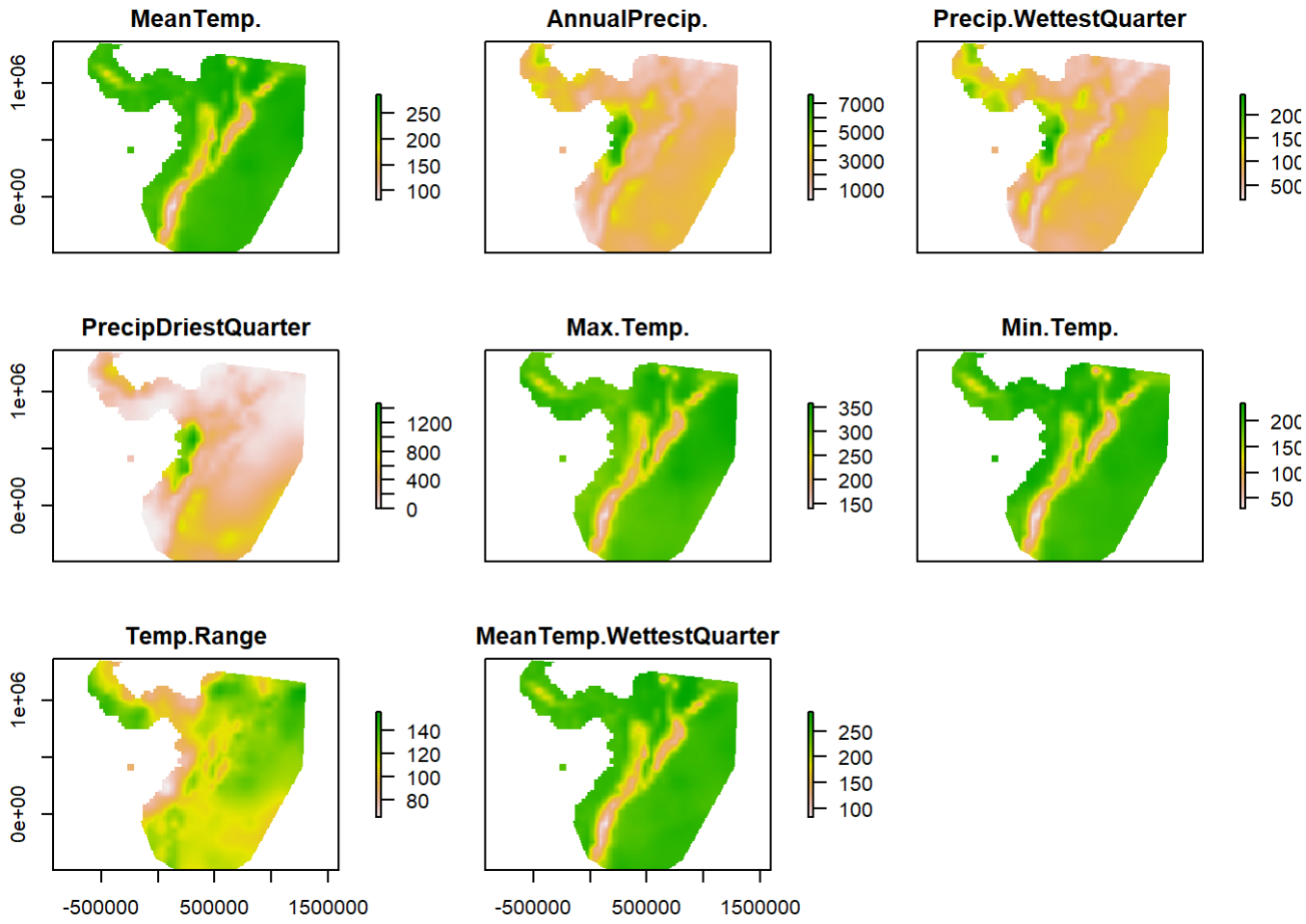
```
plot(env$MeanTemp.) #plot

plot(bradMCP,add=TRUE)

plot(bradypus,add=T)
```




```
bradSP<-crop(bradypus,bradMCP) # crop points to the mcp

#inspect
plot(env$MeanTemp.)
plot(bradSP,add=T)
plot(bradMCP,add=T)
```




```
###Crop the env object to this new polygon considering 80% of the point data

env<-crop(env,bradMCP)#crop
env<-mask(env,bradMCP)#mask
plot(env)#plot
```




```
res(env)
```

```
## [1] 10000 10000
```

```
set.seed(11)

#create random points on cells of the 'env' object within the extent of bradypus and a
voiding cells containing points from bradypus

back.xy <- randomPoints(env, n=1000,p= bradypus,ext = extent(bradypus))


#Create Spatial Points layer from back.xy

back<-SpatialPoints(back.xy,crs(bradypus))

env<-crop(env,back)


#absence values

eA<-extract(env,back)

#presence values
eP<-extract(env,bradypus)

#create data frames from values
Pres.cov<-data.frame(eP,Pres=1)


Back.cov<-data.frame(eA,Pres=0)


#combine both data sets using the rbind() function (binds datsets with same columns by
stacking one on top of the other - i.e. joining rows together)
all.cov<-rbind(Pres.cov,Back.cov)


#inspect
head(all.cov)
```

```
##   MeanTemp. AnnualPrecip. Precip.WettestQuarter PrecipDriestQuarter Max.Temp.
## 1        NA            NA                    NA                  NA        NA
## 2  257.3199      4153.812            1826.6311           236.89791  333.2949
## 3        NA            NA                    NA                  NA        NA
## 4  276.0819      1742.971             762.9147            62.17907  342.1479
## 5  244.3080      2628.179             861.1550           444.18504  299.9906
## 6        NA            NA                    NA                  NA        NA
##   Min.Temp. Temp.Range MeanTemp.WettestQuarter Pres
## 1        NA         NA                     NA    1
## 2  190.7434   142.0165               251.1439    1
## 3        NA         NA                     NA    1
## 4  220.1750   121.9726               274.0819    1
## 5  190.4570   109.5336               238.9543    1
## 6        NA         NA                     NA    1
```

```
tail(all.cov)
```

```
##      MeanTemp. AnnualPrecip. Precip.WettestQuarter PrecipDriestQuarter
## 3149  259.1182      6268.958             1878.8980          1068.36122
## 3150  241.2820      2579.362             1310.7665           188.86448
## 3151  261.0894      3109.171              913.7929           643.97156
## 3152  239.2296      2997.262             1029.8561           426.81113
## 3153  264.0721      3733.198             1296.8462           434.67820
## 3154  237.5877      1206.850              506.6119            68.82714
##      Max.Temp. Min.Temp. Temp.Range MeanTemp.WettestQuarter Pres
## 3149  314.0247  208.4511  105.76816                257.7708    0
## 3150  296.6353  191.4423  105.00765                249.0471    0
## 3151  314.0894  206.0894  108.00000                261.3381    0
## 3152  298.7351  187.2296  111.10070                235.7884    0
## 3153  315.4952  222.2163   93.27887                256.1442    0
## 3154  301.2938  168.1692  133.16816                236.6925    0
```

```
summary(all.cov)
```

```
##    MeanTemp.      AnnualPrecip.  Precip.WettestQuarter PrecipDriestQuarter
##  Min.   : 87.14   Min.   : 249   Min.   : 190.0        Min.   :   4.768
##  1st Qu.:241.95   1st Qu.:2298   1st Qu.: 858.1        1st Qu.: 122.135
##  Median :255.22   Median :2787   Median :1010.7        Median : 217.848
##  Mean   :247.10   Mean   :2792   Mean   :1044.0        Mean   : 296.923
##  3rd Qu.:262.59   3rd Qu.:3256   3rd Qu.:1216.1        3rd Qu.: 461.785
##  Max.   :284.57   Max.   :7056   Max.   :2408.8        Max.   :1369.858
##  NA's   :487      NA's   :487    NA's   :487           NA's   :487
##    Max.Temp.       Min.Temp.        Temp.Range     MeanTemp.WettestQuarter
##  Min.   :146.1   Min.   : 34.11   Min.   : 65.78   Min.   : 88.06
##  1st Qu.:300.0   1st Qu.:180.84   1st Qu.:104.12   1st Qu.:239.20
##  Median :312.2   Median :199.03   Median :112.58   Median :250.23
##  Mean   :306.6   Mean   :192.53   Mean   :113.96   Mean   :243.73
##  3rd Qu.:318.1   3rd Qu.:213.65   3rd Qu.:124.97   3rd Qu.:257.87
##  Max.   :357.3   Max.   :228.31   Max.   :153.66   Max.   :286.23
##  NA's   :487     NA's   :487      NA's   :487      NA's   :487
##       Pres
##  Min.   :0.0000
##  1st Qu.:0.0000
##  Median :1.0000
##  Mean   :0.6829
##  3rd Qu.:1.0000
##  Max.   :1.0000
##
```

Now we can use the data frame used in the week four class to create a task for our learners to carry out. As stated at the start of this practical, the spatial location of points is critical to our analysis and will be required at every step of model development, testing and tuning. So, first let’s create an object for the coordinates of our presence and background data for use in the analysis.

```
### create task for use with mlr classification algorithms

coordsBack<-coordinates(back)
coordsPres<-coordinates(bradypus)
coords<-rbind(coordsPres,coordsBack)
coords<-data.frame(coords)
colnames(coords)<-c("x","y")
all.cov<-cbind(all.cov,coords)
all.cov<-subset(all.cov,!is.na(all.cov$MeanTemp.))#remove any NA values

#Now we create our task, passing the data frame and specifying the species presence/ab
sence column (the "target") and tell R that 1 = present (i.e. a postive result)

library(mlr)
```

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

```
##
## Attaching package: 'ParamHelpers'
```

```
## The following object is masked from 'package:raster':
##
##     getValues
```

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

```
##
## Attaching package: 'mlr'
```

```
## The following object is masked from 'package:adehabitatLT':
##
##     subsample
```

```
## The following object is masked from 'package:raster':
##
##     resample
```

```
#For the makeClassifTask function to work, our target variable needs to be categorical
(a "factor" in R) so let's tidy that up first

all.cov$Pres<-factor(all.cov$Pres)

task = makeClassifTask(data = all.cov, target = "Pres",
                       positive = "1", coordinates = all.cov[,10:11])
```

Now that we have the task set up, we need to choose some learners that will help us carry out our task. If we already know which learner we prefer then we can use the makeLearner() function to specify our algorithm. If we don't then the mlr package has a really useful function listLearners() with which you can pass your existing task and get a list of machine learning algorithms which apply.

The listLearners function returns A LOT of information and most of this we do not want so we will not call it here but instead we will go straight to creating our learners which for today's practical are: 1. classif.binomial (an implementation of logistic regression), 2. classif.randomForest and 3. classif.ksvm (a support vector machine learning algorithm developed by the Kernlab group). We looked at binomial regression and Random Forest previously. Support Vector Machines (SVMs) are classifiers that attempt to create the largest possible margin between samples (such as groups of presence/absence points) that maximises their difference (distance between the groups) whilst reducing the number of misclassifications, thereby helping the classifer to make decisions about group membership. See here for an article giving a nice simple overview of how SVMs work: https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496 (https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496)

The next step is to create our individual learners that will do the work of solving our task. Let's start with the most familiar type - binomial regression:

```
################ Binomial (logistic regression)



lrnBinomial = makeLearner("classif.binomial",
                      predict.type = "prob",
                      fix.factors.prediction = TRUE)
```

In the above code we first tell the makeLearner function the type of classifier we want to work with, specify the type of prediction we want to make (probability) and if we want the algorithm to output precicted classes (i.e. 1 or 0) when used later for prediction.

Next we do the same for our Random Forest and Ksvm learners:

```
##########Randome Forest

lrnRF = makeLearner("classif.randomForest",
                    predict.type = "prob",
                    fix.factors.prediction = TRUE)


###### Support Vector Machine
lrn_ksvm = makeLearner("classif.ksvm",
                       predict.type = "prob",
                       fix.factors.prediction = TRUE,
                       kernel = "rbfdot") # this argument specifies a "Radial Basis ke
rnel function" which is basically a Gaussian kernel i.e. one that simulates a normal d
istribution. The algorithm uses the kernel type to decide how to differentiate groups
 (you can think of this as a means to estimate or weight distances between group 1 - p
resence and group 0 - absence).
```

Part Two: model evaluation

Now that we have both our task and learners ready we can apply each learner to the task and evaluate their performance.

To do this we have to set up a sampling scheme for cross validation. This is conceptually identical to the k-fold validation we did in Week 4 but here we don't have to go through the trouble of writing our own for loops - we can use the functionality within the mlr package to do this for us. First we will create a k-fold resampling strategy that DOES NOT take spatial location into account.

```
### Create resampling scheme (performance level) for cross-validation.

##This code creates 5 folds in the data and repeates the whole k-fold cross validation
process 10 times (so 50 models in total) and aggregates (averages) the AUC statistic.


perf_levelCV = makeResampleDesc(method = "RepCV", folds = 5, reps = 10)
```

In addition we can easily set up the same cross validation strategy but with spatially disjoint folds. All we have to do is change "RepCV" to "SpRepCV" like so:

```
perf_level_spCV = makeResampleDesc(method = "SpRepCV", folds = 5, reps = 10) #sampling
strategy to run five fold re-sampling ten times
```

Now we can compare, for each of our learners, their performance based on conventional k-fold validation and one where spatial location is considered (i.e. when test data are made to be geographically far from training data).

```
#Binomial conventional cross validation (K fold)

set.seed(11)


cvBinomial = resample(learner = lrnBinomial, task =task,
                resampling = perf_levelCV,
                measures = mlr::auc)
```

```
## Resampling: repeated cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.8816018
```

```
## [Resample] iter 2:    0.9227213
```

```
## [Resample] iter 3:    0.8962629
```

```
## [Resample] iter 4:    0.8825896
```

```
## [Resample] iter 5:    0.8976276
```

```
## [Resample] iter 6:    0.8987475
```

```
## [Resample] iter 7:    0.8814433
```

```
## [Resample] iter 8:    0.9149614
```

```
## [Resample] iter 9:    0.8980895
```

```
## [Resample] iter 10:   0.8854131
```

```
## [Resample] iter 11:   0.8923118
```

```
## [Resample] iter 12:   0.9072827
```

```
## [Resample] iter 13:   0.8974200
```

```
## [Resample] iter 14:   0.8825077
```

```
## [Resample] iter 15:   0.9000302
```

```
## [Resample] iter 16:   0.8869716
```

```
## [Resample] iter 17:   0.8933815
```

```
## [Resample] iter 18:   0.8764596
```

```
## [Resample] iter 19:   0.9125384
```

```
## [Resample] iter 20:   0.9028947
```

```
## [Resample] iter 21:   0.9122220
```

```
## [Resample] iter 22:   0.8935990
```

```
## [Resample] iter 23:   0.8922567
```

```
## [Resample] iter 24:   0.8896211
```

```
## [Resample] iter 25:   0.8914168
```

```
## [Resample] iter 26:   0.9020619
```

```
## [Resample] iter 27:   0.9053232
```

```
## [Resample] iter 28:   0.9080959
```

```
## [Resample] iter 29:   0.8740559
```

```
## [Resample] iter 30:   0.8902067
```

```
## [Resample] iter 31:   0.8878696
```

```
## [Resample] iter 32:   0.9088504
```

```
## [Resample] iter 33:   0.9036566
```

```
## [Resample] iter 34:   0.8924442
```

```
## [Resample] iter 35:   0.8895621
```

```
## [Resample] iter 36:   0.8849941
```

```
## [Resample] iter 37:   0.9033315
```

```
## [Resample] iter 38:   0.8997726
```

```
## [Resample] iter 39:   0.8967743
```

```
## [Resample] iter 40:   0.8985344
```

```
## [Resample] iter 41:   0.9050707
```

```
## [Resample] iter 42:   0.8634262
```

```
## [Resample] iter 43:   0.9057663
```

```
## [Resample] iter 44:   0.9069068
```

```
## [Resample] iter 45:   0.8924895
```

```
## [Resample] iter 46:   0.8935973
```

```
## [Resample] iter 47:   0.8989790
```

```
## [Resample] iter 48:   0.9129342
```

```
## [Resample] iter 49:   0.8918299
```

```
## [Resample] iter 50:   0.8810345
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.8957588
```

```
##
```

```
##Binomial spatial cross validation

sp_cvBinomial = resample(learner = lrnBinomial, task =task,
                   resampling = perf_level_spCV,
                   measures = mlr::auc)
```

```
## Resampling: repeated spatial cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.4487405
```

```
## [Resample] iter 2:    0.6575411
```

```
## [Resample] iter 3:    0.6179563
```

```
## [Resample] iter 4:    0.6015416
```

```
## [Resample] iter 5:    0.6629213
```

```
## [Resample] iter 6:    0.7024706
```

```
## [Resample] iter 7:    0.4772285
```

```
## [Resample] iter 8:    0.5354922
```

```
## [Resample] iter 9:    0.6158018
```

```
## [Resample] iter 10:   0.8482481
```

```
## [Resample] iter 11:   0.6015416
```

```
## [Resample] iter 12:   0.4487405
```

```
## [Resample] iter 13:   0.6179563
```

```
## [Resample] iter 14:   0.6629213
```

```
## [Resample] iter 15:   0.6575411
```

```
## [Resample] iter 16:   0.4402211
```

```
## [Resample] iter 17:   0.6641509
```

```
## [Resample] iter 18:   0.6496466
```

```
## [Resample] iter 19:   0.6015416
```

```
## [Resample] iter 20:   0.6179563
```

```
## [Resample] iter 21:   0.6496466
```

```
## [Resample] iter 22:   0.6015416
```

```
## [Resample] iter 23:   0.6641509
```

```
## [Resample] iter 24:   0.4402211
```

```
## [Resample] iter 25:   0.6179563
```

```
## [Resample] iter 26:   0.6540404
```

```
## [Resample] iter 27:   0.6321215
```

```
## [Resample] iter 28:   0.4373269
```

```
## [Resample] iter 29:   0.6106039
```

```
## [Resample] iter 30:   0.5770896
```

```
## [Resample] iter 31:   0.7711218
```

```
## [Resample] iter 32:   0.9213787
```

```
## [Resample] iter 33:   0.4521983
```

```
## [Resample] iter 34:   0.6078363
```

```
## [Resample] iter 35:   0.6214893
```

```
## [Resample] iter 36:   0.6179563
```

```
## [Resample] iter 37:   0.6575411
```

```
## [Resample] iter 38:   0.4487405
```

```
## [Resample] iter 39:   0.6629213
```

```
## [Resample] iter 40:   0.6015416
```

```
## [Resample] iter 41:   0.6214893
```

```
## [Resample] iter 42:   0.5993882
```

```
## [Resample] iter 43:   0.7393833
```

```
## [Resample] iter 44:   0.9015488
```

```
## [Resample] iter 45:   0.3835566
```

```
## [Resample] iter 46:   0.9213787
```

```
## [Resample] iter 47:   0.4521983
```

```
## [Resample] iter 48:   0.7711218
```

```
## [Resample] iter 49:   0.6214893
```

```
## [Resample] iter 50:   0.6078363
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.6199395
```

```
##
```

You should see that the result (aggregated AUC) for the spatial cross validation is noticeably lower than for the conventional cross validation. This proves that the latter is an over-optimistic assessment and that we should be cautious with non-spatial approaches to model evaluation.

Now let's look at our Random Forest learner.

```
###### Random Forest evaluation


##random sampling cross-validation

cvRF = resample(learner = lrnRF, task =task,
                   resampling = perf_levelCV,
                   measures = mlr::auc)
```

```
## Resampling: repeated cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.9725214
```

```
## [Resample] iter 2:    0.9798393
```

```
## [Resample] iter 3:    0.9795013
```

```
## [Resample] iter 4:    0.9756360
```

```
## [Resample] iter 5:    0.9751847
```

```
## [Resample] iter 6:    0.9689396
```

```
## [Resample] iter 7:    0.9719543
```

```
## [Resample] iter 8:    0.9767053
```

```
## [Resample] iter 9:    0.9883132
```

```
## [Resample] iter 10:   0.9783329
```

```
## [Resample] iter 11:   0.9829562
```

```
## [Resample] iter 12:   0.9820825
```

```
## [Resample] iter 13:   0.9787192
```

```
## [Resample] iter 14:   0.9720406
```

```
## [Resample] iter 15:   0.9688584
```

```
## [Resample] iter 16:   0.9726988
```

```
## [Resample] iter 17:   0.9736858
```

```
## [Resample] iter 18:   0.9834412
```

```
## [Resample] iter 19:   0.9722121
```

```
## [Resample] iter 20:   0.9698927
```

```
## [Resample] iter 21:   0.9807336
```

```
## [Resample] iter 22:   0.9751081
```

```
## [Resample] iter 23:   0.9816869
```

```
## [Resample] iter 24:   0.9808748
```

```
## [Resample] iter 25:   0.9703389
```

```
## [Resample] iter 26:   0.9737614
```

```
## [Resample] iter 27:   0.9689100
```

```
## [Resample] iter 28:   0.9824672
```

```
## [Resample] iter 29:   0.9799609
```

```
## [Resample] iter 30:   0.9768973
```

```
## [Resample] iter 31:   0.9789683
```

```
## [Resample] iter 32:   0.9753023
```

```
## [Resample] iter 33:   0.9813497
```

```
## [Resample] iter 34:   0.9759146
```

```
## [Resample] iter 35:   0.9722996
```

```
## [Resample] iter 36:   0.9723607
```

```
## [Resample] iter 37:   0.9825008
```

```
## [Resample] iter 38:   0.9808850
```

```
## [Resample] iter 39:   0.9622624
```

```
## [Resample] iter 40:   0.9793904
```

```
## [Resample] iter 41:   0.9825838
```

```
## [Resample] iter 42:   0.9708835
```

```
## [Resample] iter 43:   0.9694219
```

```
## [Resample] iter 44:   0.9757917
```

```
## [Resample] iter 45:   0.9794305
```

```
## [Resample] iter 46:   0.9845126
```

```
## [Resample] iter 47:   0.9649927
```

```
## [Resample] iter 48:   0.9705878
```

```
## [Resample] iter 49:   0.9818931
```

```
## [Resample] iter 50:   0.9817252
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.9763462
```

```
##
```

```
#spatial partitioning cross-validation

sp_cvRF = resample(learner = lrnRF, task =task,
                   resampling = perf_level_spCV,
                   measures = mlr::auc)
```

```
## Resampling: repeated spatial cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.6643496
```

```
## [Resample] iter 2:    0.5823031
```

```
## [Resample] iter 3:    0.6931257
```

```
## [Resample] iter 4:    0.4908170
```

```
## [Resample] iter 5:    0.4592884
```

```
## [Resample] iter 6:    0.4556906
```

```
## [Resample] iter 7:    0.5627555
```

```
## [Resample] iter 8:    0.7879360
```

```
## [Resample] iter 9:    0.4476190
```

```
## [Resample] iter 10:   0.4739879
```

```
## [Resample] iter 11:   0.6733775
```

```
## [Resample] iter 12:   0.5517337
```

```
## [Resample] iter 13:   0.7815949
```

```
## [Resample] iter 14:   0.4294202
```

```
## [Resample] iter 15:   0.6862492
```

```
## [Resample] iter 16:   0.5115313
```

```
## [Resample] iter 17:   0.6356248
```

```
## [Resample] iter 18:   0.4822978
```

```
## [Resample] iter 19:   0.5961453
```

```
## [Resample] iter 20:   0.6576425
```

```
## [Resample] iter 21:   0.5494234
```

```
## [Resample] iter 22:   0.6671632
```

```
## [Resample] iter 23:   0.4983471
```

```
## [Resample] iter 24:   0.5985127
```

```
## [Resample] iter 25:   0.6435977
```

```
## [Resample] iter 26:   0.4677732
```

```
## [Resample] iter 27:   0.4705602
```

```
## [Resample] iter 28:   0.4619048
```

```
## [Resample] iter 29:   0.7619675
```

```
## [Resample] iter 30:   0.5541601
```

```
## [Resample] iter 31:   0.4838622
```

```
## [Resample] iter 32:   0.6455311
```

```
## [Resample] iter 33:   0.6554544
```

```
## [Resample] iter 34:   0.5093326
```

```
## [Resample] iter 35:   0.6060049
```

```
## [Resample] iter 36:   0.5925071
```

```
## [Resample] iter 37:   0.3847854
```

```
## [Resample] iter 38:   0.4361916
```

```
## [Resample] iter 39:   0.6252052
```

```
## [Resample] iter 40:   0.4803816
```

```
## [Resample] iter 41:   0.4510330
```

```
## [Resample] iter 42:   0.5588296
```

```
## [Resample] iter 43:   0.6299813
```

```
## [Resample] iter 44:   0.3380682
```

```
## [Resample] iter 45:   0.4431324
```

```
## [Resample] iter 46:   0.4796857
```

```
## [Resample] iter 47:   0.6598002
```

```
## [Resample] iter 48:   0.5375867
```

```
## [Resample] iter 49:   0.6455311
```

```
## [Resample] iter 50:   0.5761127
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.5607183
```

```
##
```

You will notice that, although random forest out-performs the binomial regression learner when assessed from a conventional cross-validation point of view, the binomial learner achieves a higher AUC when spatial dependency is take into account. However, random forest has a set of hyperparameters that can be easily tuned to enhance model performance, a functionality not present with binomal learners (which are based on statistical methods rather than on machine learning algorithms). We will see later in the practical if model tuning changes the relative standing of these two learners. First though, let's perform the same evaluation on our support vector machine learner for comparison.

```
########### Support vector machine evaluation


cvksvm = resample(learner = lrn_ksvm, task =task,
                  resampling = perf_levelCV,
                  measures = mlr::auc)
```

```
## Resampling: repeated cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.9228632
```

```
## [Resample] iter 2:    0.9055110
```

```
## [Resample] iter 3:    0.9571741
```

```
## [Resample] iter 4:    0.9126148
```

```
## [Resample] iter 5:    0.9282079
```

```
## [Resample] iter 6:    0.9174112
```

```
## [Resample] iter 7:    0.8943111
```

```
## [Resample] iter 8:    0.9398358
```

```
## [Resample] iter 9:    0.9417261
```

```
## [Resample] iter 10:   0.9138103
```

```
## [Resample] iter 11:   0.9258964
```

```
## [Resample] iter 12:   0.9242981
```

```
## [Resample] iter 13:   0.9137197
```

```
## [Resample] iter 14:   0.9181580
```

```
## [Resample] iter 15:   0.9361565
```

```
## [Resample] iter 16:   0.9360048
```

```
## [Resample] iter 17:   0.8981988
```

```
## [Resample] iter 18:   0.9213733
```

```
## [Resample] iter 19:   0.9290944
```

```
## [Resample] iter 20:   0.9324368
```

```
## [Resample] iter 21:   0.9270166
```

```
## [Resample] iter 22:   0.9391124
```

```
## [Resample] iter 23:   0.8992238
```

```
## [Resample] iter 24:   0.9196806
```

```
## [Resample] iter 25:   0.9277439
```

```
## [Resample] iter 26:   0.9316977
```

```
## [Resample] iter 27:   0.9153459
```

```
## [Resample] iter 28:   0.9312180
```

```
## [Resample] iter 29:   0.8962782
```

```
## [Resample] iter 30:   0.9370317
```

```
## [Resample] iter 31:   0.9461727
```

```
## [Resample] iter 32:   0.9346010
```

```
## [Resample] iter 33:   0.9250000
```

```
## [Resample] iter 34:   0.9121450
```

```
## [Resample] iter 35:   0.9070109
```

```
## [Resample] iter 36:   0.8956963
```

```
## [Resample] iter 37:   0.9255803
```

```
## [Resample] iter 38:   0.9374128
```

```
## [Resample] iter 39:   0.9401106
```

```
## [Resample] iter 40:   0.9116008
```

```
## [Resample] iter 41:   0.9350318
```

```
## [Resample] iter 42:   0.8983129
```

```
## [Resample] iter 43:   0.9009409
```

```
## [Resample] iter 44:   0.9221552
```

```
## [Resample] iter 45:   0.9392384
```

```
## [Resample] iter 46:   0.9310479
```

```
## [Resample] iter 47:   0.9239980
```

```
## [Resample] iter 48:   0.9285693
```

```
## [Resample] iter 49:   0.9104927
```

```
## [Resample] iter 50:   0.9299040
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.9229634
```

```
##
```

```
sp_cvksvm = resample(learner = lrn_ksvm, task =task,
                     resampling = perf_level_spCV,
                     measures = mlr::auc)
```

```
## Resampling: repeated spatial cross-validation
```

```
## Measures:             auc
```

```
## [Resample] iter 1:    0.5515191
```

```
## [Resample] iter 2:    0.6165396
```

```
## [Resample] iter 3:    0.4338843
```

```
## [Resample] iter 4:    0.3594358
```

```
## [Resample] iter 5:    0.4410774
```

```
## [Resample] iter 6:    0.5848677
```

```
## [Resample] iter 7:    0.4201090
```

```
## [Resample] iter 8:    0.4254167
```

```
## [Resample] iter 9:    0.4866255
```

```
## [Resample] iter 10:   0.4757021
```

```
## [Resample] iter 11:   0.7612448
```

```
## [Resample] iter 12:   0.4213587
```

```
## [Resample] iter 13:   0.3983969
```

```
## [Resample] iter 14:   0.5101035
```

```
## [Resample] iter 15:   0.7717172
```

```
## [Resample] iter 16:   0.5084373
```

```
## [Resample] iter 17:   0.1363636
```

```
## [Resample] iter 18:   0.4313936
```

```
## [Resample] iter 19:   0.4968527
```

```
## [Resample] iter 20:   0.7661940
```

```
## [Resample] iter 21:   0.4255878
```

```
## [Resample] iter 22:   0.5266350
```

```
## [Resample] iter 23:   0.7769780
```

```
## [Resample] iter 24:   0.4815668
```

```
## [Resample] iter 25:   0.0682540
```

```
## [Resample] iter 26:   0.4150761
```

```
## [Resample] iter 27:   0.5139768
```

```
## [Resample] iter 28:   0.0539683
```

```
## [Resample] iter 29:   0.7828008
```

```
## [Resample] iter 30:   0.4449861
```

```
## [Resample] iter 31:   0.5197872
```

```
## [Resample] iter 32:   0.4411188
```

```
## [Resample] iter 33:   0.3942103
```

```
## [Resample] iter 34:   0.4249286
```

```
## [Resample] iter 35:   0.5873497
```

```
## [Resample] iter 36:   0.4269334
```

```
## [Resample] iter 37:   0.7701119
```

```
## [Resample] iter 38:   0.1205808
```

```
## [Resample] iter 39:   0.4873764
```

```
## [Resample] iter 40:   0.5110691
```

```
## [Resample] iter 41:   0.5265585
```

```
## [Resample] iter 42:   0.1292135
```

```
## [Resample] iter 43:   0.4922073
```

```
## [Resample] iter 44:   0.7700659
```

```
## [Resample] iter 45:   0.4338257
```

```
## [Resample] iter 46:   0.4891128
```

```
## [Resample] iter 47:   0.7355942
```

```
## [Resample] iter 48:   0.0804781
```

```
## [Resample] iter 49:   0.5318598
```

```
## [Resample] iter 50:   0.3933646
```

```
##
```

```
## Aggregated Result: auc.test.mean=0.4750563
```

```
##
```

So, both learners apparently suffer considerably when switching from a conventional to a spatially-oriented method of performance evaluation. especially in the case of the support vector machine. Support vector machines are the most readily improved through model tuning and this is a particular strength of this kind of learner. So, the winner in this contest is far from decided - not before we have taken a closer look at model tuning options for these different learners.

Part Three: Model tuning

For tuning of hyperparameters we follow a similar process initially as that taken when setting up cross validation resampling. We use the same resampling scheme so that the data are separated into five spatially disjoint folds and set the “ctrl” argument so that, for each fold, the tuning process generates 10 random combinations of model parameters. This is done via the makeTunerControlRandom() function with maximum iterations (“maxit=”) set to 10. We could (and should) try many more than this but this would become increasingly time consuming to do.

```
#hyperparameter tuning



# spatial partitioning
tune_level = makeResampleDesc(method = "SpCV", iters = 5)
# specifying random parameter value search
ctrl = makeTuneControlRandom(maxit = 10L)
```

Once the data resampling has been set up, we need to look at each individual learner to check which hyperparameters are available for tuning. These can be accessed using the getParamSet() function. Let’s do this for the randomForest learner first.

```
getParamSet(lrnRF)
```

```
##                     Type  len   Def   Constr Req Tunable Trafo
## ntree            integer    -   500 1 to Inf   -    TRUE     -
## mtry             integer    -     - 1 to Inf   -    TRUE     -
## replace          logical    -  TRUE        -   -    TRUE     -
## classwt     numericvector <NA>     - 0 to Inf   -    TRUE     -
## cutoff      numericvector <NA>     -   0 to 1   -    TRUE     -
## strata           untyped    -     -        -   -   FALSE     -
## sampsize    integervector <NA>     - 1 to Inf   -    TRUE     -
## nodesize         integer    -     1 1 to Inf   -    TRUE     -
## maxnodes         integer    -     - 1 to Inf   -    TRUE     -
## importance       logical    - FALSE        -   -    TRUE     -
## localImp         logical    - FALSE        -   -    TRUE     -
## proximity        logical    - FALSE        -   -   FALSE     -
## oob.prox         logical    -     -        -   Y   FALSE     -
## norm.votes       logical    -  TRUE        -   -   FALSE     -
## do.trace         logical    - FALSE        -   -   FALSE     -
## keep.forest      logical    -  TRUE        -   -   FALSE     -
## keep.inbag       logical    - FALSE        -   -   FALSE     -
```

For random forests the hyperparameters known to most closely affect model performance are “mtry” and “nodesize”. So, for these hyperparameters we can create a range of values to pass to the tuneParams() function for tuning our model. We create these hyperparameter ranges using the makeParamSet() function. For mtry the range of options is actually the number of predictor variables in our dataframe (eight) and a sensible range for nodesize starts from 1 with 10 being a common upper limit used in classification problems (any more than this reduces computing time but also precision).

```
paramsRF <- makeParamSet(
  makeIntegerParam("mtry",lower = 1,upper = 8),
  makeIntegerParam("nodesize",lower = 1,upper = 10)
)
```

Now we have the parameter set ready to go we use the tuneParams() function to get the result of our model tuning. Before we do that though, let’s break down exactly what is happening in the model tuning process. The code below is basically a list of information given to the tuneParams() function. We provide the name of the learner “lrnRF”, the task for the learner, the spatially-oriented resampling method “tune_level”, the range of parameter values to select from “paramsRF”, the number of random combinations of parameter values to use (“ctrl”) and the measure we wish to return “auc”.

The objects we pass to the tuneParams() function will ensure that for five spatially disjoint folds (subsets) of the data, 10 random combinations of hyperparameters will be used for model tuning with the optimum values returned (i.e. those that maximize the auc score). This means that a total of 50 models will be tested and each time the training and testing data will be spatially partitioned such that our tuning is based on maximizing the ability of our model to predict to new spatial locations. In reality we might run many more models but in the interests of time we will proceed with 10 combinations for tuning.

Please be patient as the following model tuning step can take a while.

```
set.seed(11)
tuneRF = tuneParams (learner = lrnRF,
                     task=task,
                     resampling = tune_level,
                     par.set = paramsRF,
                     control = ctrl,
                     measures = mlr::auc,
                     show.info = TRUE)
```

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

```
##             Type len Def  Constr Req Tunable Trafo
## mtry     integer   -   -  1 to 8   -    TRUE     -
## nodesize integer   -   - 1 to 10   -    TRUE     -
```

```
## With control class: TuneControlRandom
```

```
## Imputation value: -0
```

```
## [Tune-x] 1: mtry=1; nodesize=4
```

```
## [Tune-y] 1: auc.test.mean=0.6184476; time: 0.1 min
```

```
## [Tune-x] 2: mtry=8; nodesize=9
```

```
## [Tune-y] 2: auc.test.mean=0.5387819; time: 0.1 min
```

```
## [Tune-x] 3: mtry=2; nodesize=10
```

```
## [Tune-y] 3: auc.test.mean=0.5991324; time: 0.1 min
```

```
## [Tune-x] 4: mtry=3; nodesize=3
```

```
## [Tune-y] 4: auc.test.mean=0.5817708; time: 0.1 min
```

```
## [Tune-x] 5: mtry=6; nodesize=1
```

```
## [Tune-y] 5: auc.test.mean=0.5789081; time: 0.1 min
```

```
## [Tune-x] 6: mtry=2; nodesize=3
```

```
## [Tune-y] 6: auc.test.mean=0.5842723; time: 0.1 min
```

```
## [Tune-x] 7: mtry=3; nodesize=6
```

```
## [Tune-y] 7: auc.test.mean=0.5799153; time: 0.1 min
```

```
## [Tune-x] 8: mtry=6; nodesize=1
```

```
## [Tune-y] 8: auc.test.mean=0.5793146; time: 0.1 min
```

```
## [Tune-x] 9: mtry=3; nodesize=4
```

```
## [Tune-y] 9: auc.test.mean=0.5918177; time: 0.1 min
```

```
## [Tune-x] 10: mtry=3; nodesize=10
```

```
## [Tune-y] 10: auc.test.mean=0.5971708; time: 0.1 min
```

```
## [Tune] Result: mtry=1; nodesize=4 : auc.test.mean=0.6184476
```

```
##re-formulate the random forest learner based on these results (note we can access th
e tuning results directly with the '$' symbol)
```

We can see that the random forest in this case has responded quite well to the model tuning we have carried out. We can follow the same process for the support vector machine algorithm. Here we take some recommendation from the literature (Shratz et al., 2018) [https://arxiv.org/pdf/1803.11266.pdf (https://arxiv.org/pdf/1803.11266.pdf)] to set the values for the algorithm parameters “C” (this determines the emphasis placed on finding a large margin around the classification boundary - i.e. the distance between 1/0 groups - versus avoiding unwanted misclassifications) and “sigma” (the kernel shape determining the influence of cases close to the classification boundary).

Once we have the optimal model parameters we simply need to create a ksvm model (this works in much the same way that we have fitted glm() models previously). We pass the results from our model tuning step straight to the model formula (again we can access the tuning results directly with the ‘$’ symbol).This step can take a few minutes to run so please be patient.

Note in the below code, before we make our prediction, we have to provide a raster surface for each of the x and y coordinates (as these are variables in our model).

```
#support vector machine model tuning. Lower and upper limits are set to 1 and 2^15 for "C" and for "sigma" the range is 1 to 2^6. See Shratz et al (2018) above for more details on the choice of hyperparameter ranges.

paramsSVM = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 15, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -15, upper = 6, trafo = function(x) 2^x)
)



tuneSVM = tuneParams(learner = lrn_ksvm,
                  task=task,
                  resampling = tune_level,
                  par.set = paramsSVM,
                  control = ctrl,
                  measures = mlr::auc,
                  show.info = TRUE)
```

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

```
##          Type len Def    Constr Req Tunable Trafo
## C     numeric   -   - -12 to 15   -    TRUE     Y
## sigma numeric   -   -  -15 to 6   -    TRUE     Y
```

```
## With control class: TuneControlRandom
```

```
## Imputation value: -0
```

```
## [Tune-x] 1: C=375; sigma=0.481
```

```
## [Tune-y] 1: auc.test.mean=0.3832236; time: 0.0 min
```

```
## [Tune-x] 2: C=276; sigma=0.0826
```

```
## [Tune-y] 2: auc.test.mean=0.5072378; time: 0.0 min
```

```
## [Tune-x] 3: C=273; sigma=0.000191
```

```
## [Tune-y] 3: auc.test.mean=0.6193225; time: 0.0 min
```

```
## [Tune-x] 4: C=0.0229; sigma=19.8
```

```
## [Tune-y] 4: auc.test.mean=0.5367042; time: 0.1 min
```

```
## [Tune-x] 5: C=0.00196; sigma=0.00035
```

```
## maximum number of iterations reached 4.576096e-05 -4.577034e-05
```

```
## [Tune-y] 5: auc.test.mean=0.5709458; time: 0.1 min
```

```
## [Tune-x] 6: C=0.0303; sigma=0.00441
```

```
## [Tune-y] 6: auc.test.mean=0.5823815; time: 0.1 min
```

```
## [Tune-x] 7: C=0.0042; sigma=0.000159
```

```
## maximum number of iterations reached 5.871803e-05 -5.874694e-05
```

```
## [Tune-y] 7: auc.test.mean=0.5719476; time: 0.1 min
```

```
## [Tune-x] 8: C=135; sigma=4.86e-05
```

```
## [Tune-y] 8: auc.test.mean=0.6287810; time: 0.0 min
```

```
## [Tune-x] 9: C=0.00213; sigma=0.00113
```

```
## [Tune-y] 9: auc.test.mean=0.5778657; time: 0.1 min
```

```
## [Tune-x] 10: C=261; sigma=0.000579
```

```
## [Tune-y] 10: auc.test.mean=0.6116375; time: 0.0 min
```

```
## [Tune] Result: C=135; sigma=4.86e-05 : auc.test.mean=0.6287810
```

```
library(kernlab)
```

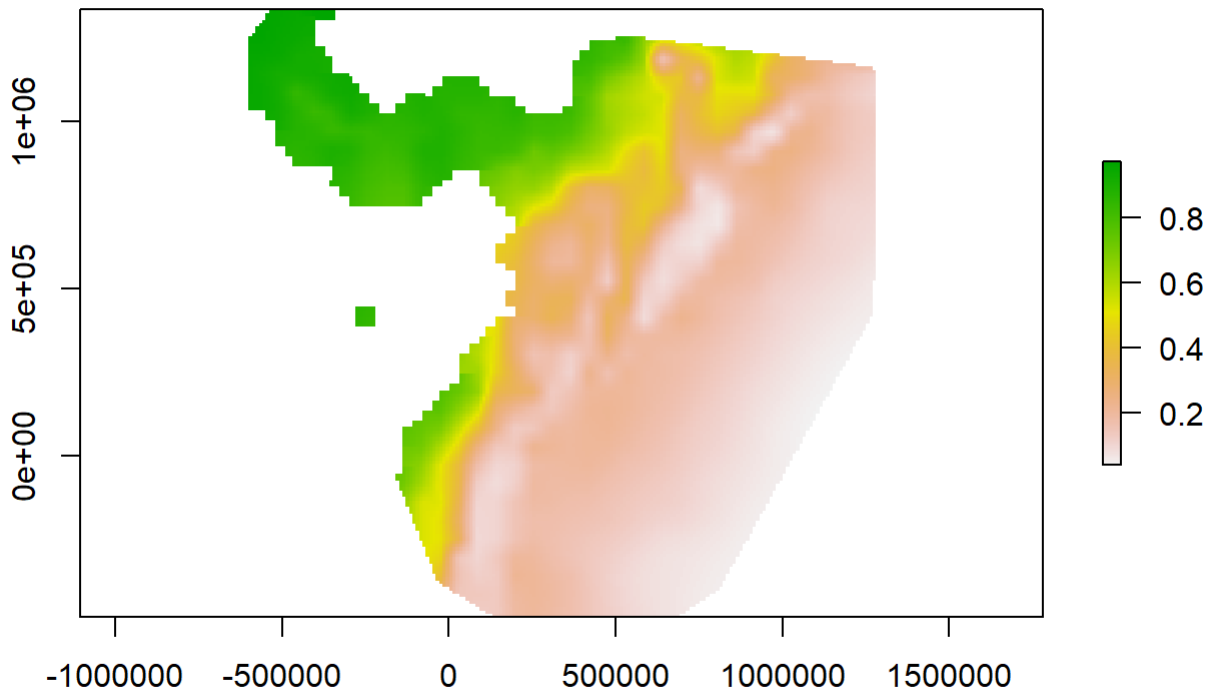
```
##
## Attaching package: 'kernlab'
```

```
## The following object is masked from 'package:CircStats':
##
##     rvm
```

```
## The following object is masked from 'package:adehabitatMA':
##
##     buffer
```

```
## The following objects are masked from 'package:raster':
##
##     buffer, rotated
```

```
#model with tuning information and set to allow probability predictions ("prob.model = TRUE"). "C-svc" is the default solver for predictions with binary outcomes in ksvm models and supports probability estimates so we use it here.

#before we predict we need to create rasters for the x and y coordinates. We can do this by creating copies of one of the layers from our env raster stack and using the init() function to fill the new rasters with x and y coordinates.


rna <- is.na(env$MeanTemp.) # returns NA raster

#Create two rasters whose values are coordinates of x and y for rna

# store coordinates in new raster:
na.x <- init(rna, 'x')
na.y <- init(rna, 'y')

XY<-stack(na.x,na.y) #create raster stack of coordinates
names(XY)<-c("x","y") # name the layers according to the headings in our dataframe
XY<-crop(XY,back)#crop
#XY<-mask(XY,bradMCP)#mask
env<-stack(env,XY)


#build the model

svmMOD<-ksvm(Pres~.,data=all.cov, type="C-svc",
             kernel=rbfdot,kpar=list(sigma=tuneSVM$x$sigma),C=tuneSVM$x$C,prob.model=TRUE)



predKSVM<-raster::predict(env,svmMOD,type="prob",index=2)

plot(predKSVM)
```



So, we can see that with only a small amount of model tuning (in reality we might run hundreds or even thousands of models with more time and computing power) we have been able to improve the support vector machine algorithm considerably, and we would now choose this algorithm to predict our species distribution on the original raster stack object. Random forest is outperformed here at this level of model tuning. However, considering the number of combinations of hyperparameters possible we might in practice achieve better results by specifying a greater number of models and parameter combinations.

Now we have made some headway in terms of improving our model prediction though tuning based on spatial partitioning of the data. Our AUC statistic is still not ideal and we have no real knowledge of the relationship between individual variables and species occurrence, much less the nature of point interactions. Let's take a look at the options provided by point pattern analysis through the Spatstat package to see if we can remedy some of these issues.

Part Four: Point pattern analysis

A key difference between the modeling approaches we have used till now and the point pattern analysis we will do next is the format of the environmental covariates which, in the Spatstat package that we will use to carry this out, take the form of Im (image) objects. So it is worth first converting our raster covariates to image objects. For this we can use the as.im.RasterLayer() function from the "maptools" library:

```
# load maptools and convert all environmental covariates to im objects
library(maptools)
```

```
## Checking rgeos availability: TRUE
```

```
tempIm<-as.im.RasterLayer(env$MeanTemp.)
precipIm<-as.im.RasterLayer(env$AnnualPrecip.)
maxIm<-as.im.RasterLayer(env$Max.Temp.)
minIm<-as.im.RasterLayer(env$Min.Temp.)
rangeIm<-as.im.RasterLayer(env$Temp.Range)
precipDry<-as.im.RasterLayer(env$PrecipDriestQuarter)
tempWettestIm<-as.im.RasterLayer(env$MeanTemp.WettestQuarter)
```

Next we call the spatstat library and set up our analysis window (necessary for point pattern analysis) and create our point pattern object.

```
library(spatstat)
```

```
## Loading required package: spatstat.data
```

```
## Loading required package: nlme
```

```
##
## Attaching package: 'nlme'
```

```
## The following object is masked from 'package:raster':
##
##     getData
```

```
## Loading required package: rpart
```

```
##
## spatstat 1.64-1       (nickname: 'Help you I can, yes!')
## For an introduction to spatstat, type 'beginner'
```

```
##
## Note: spatstat version 1.64-1 is out of date by more than 10 months; we recommend u
pgrading to the latest version.
```

```
##
## Attaching package: 'spatstat'
```

```
## The following object is masked from 'package:boot':
##
##     envelope
```

```
## The following object is masked from 'package:MASS':
##
##     area
```

```
## The following object is masked from 'package:ade4':
##
##     disc
```

```
## The following object is masked from 'package:dismo':
##
##     domain
```

```
## The following objects are masked from 'package:raster':
##
##     area, rotate, shift
```

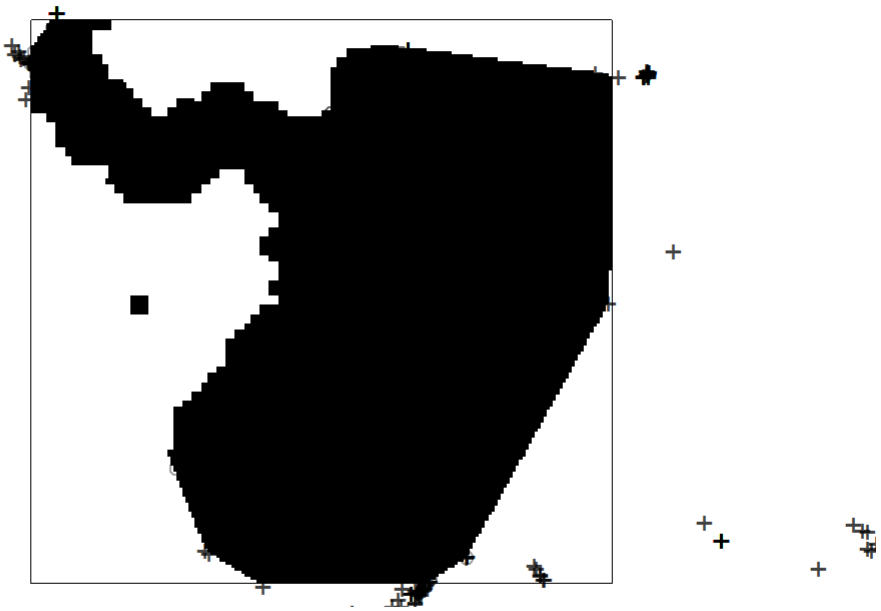
```
window.poly <- as.owin(tempIm)#create study window using the tempIm image object as a
 template

plot(window.poly)#inspect


bradCoords<-coordinates(bradypus) #get coordinates from our Erioyce object to create p
oint patter

pppBrad <- ppp(bradCoords[,1],bradCoords[,2], window = window.poly)#create point patte
r 'ppp' object
```

```
## Warning: 481 points were rejected as lying outside the specified window
```

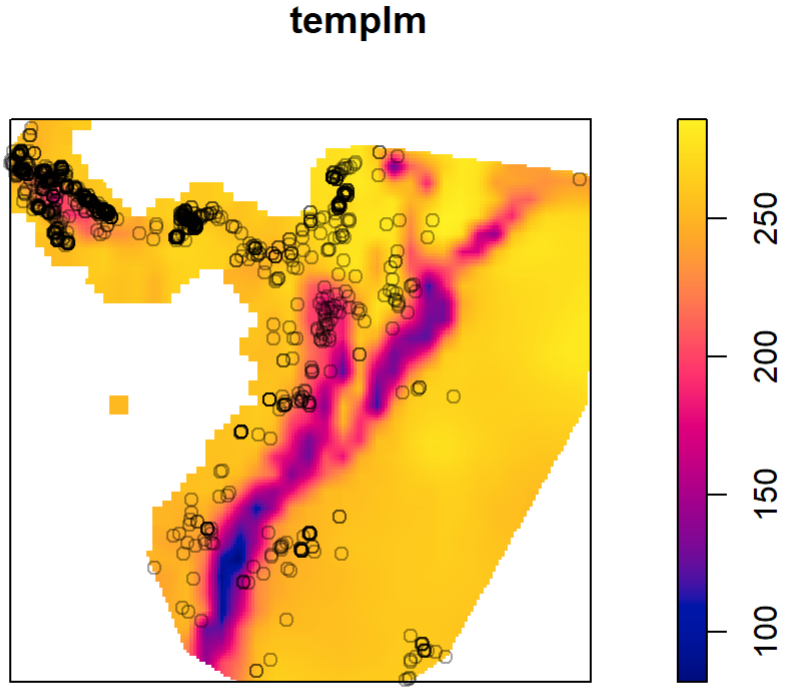
```
## Warning: data contain duplicated points
```

```
plot(pppBrad,add=T) #inspect
```

```
## Warning in plot.ppp(pppBrad, add = T): 481 illegal points also plotted
```

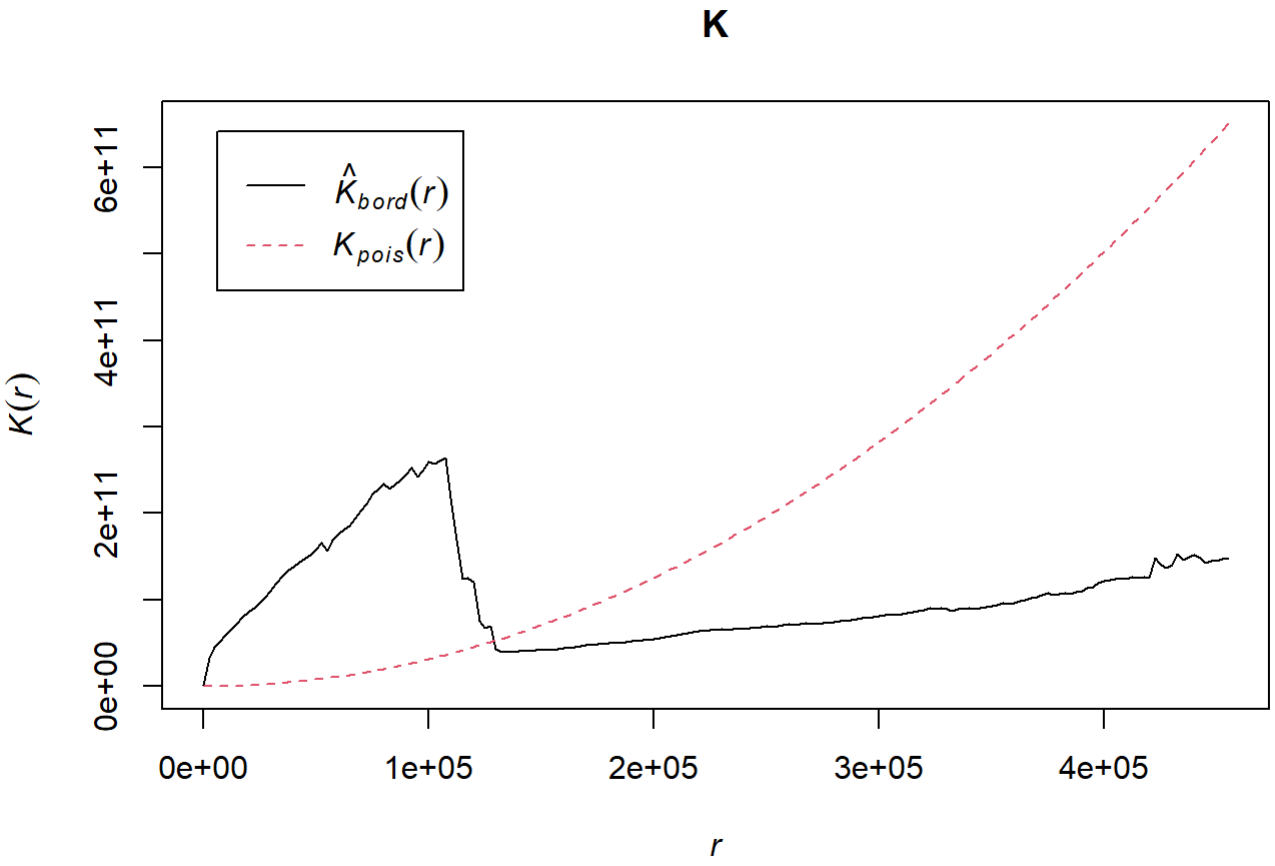



```
pppBrad<-as.ppp(pppBrad)#use as.ppp to remove points outside the window

#plot the points against covariate

plot(tempIm)

plot(pppBrad,add=TRUE)
```

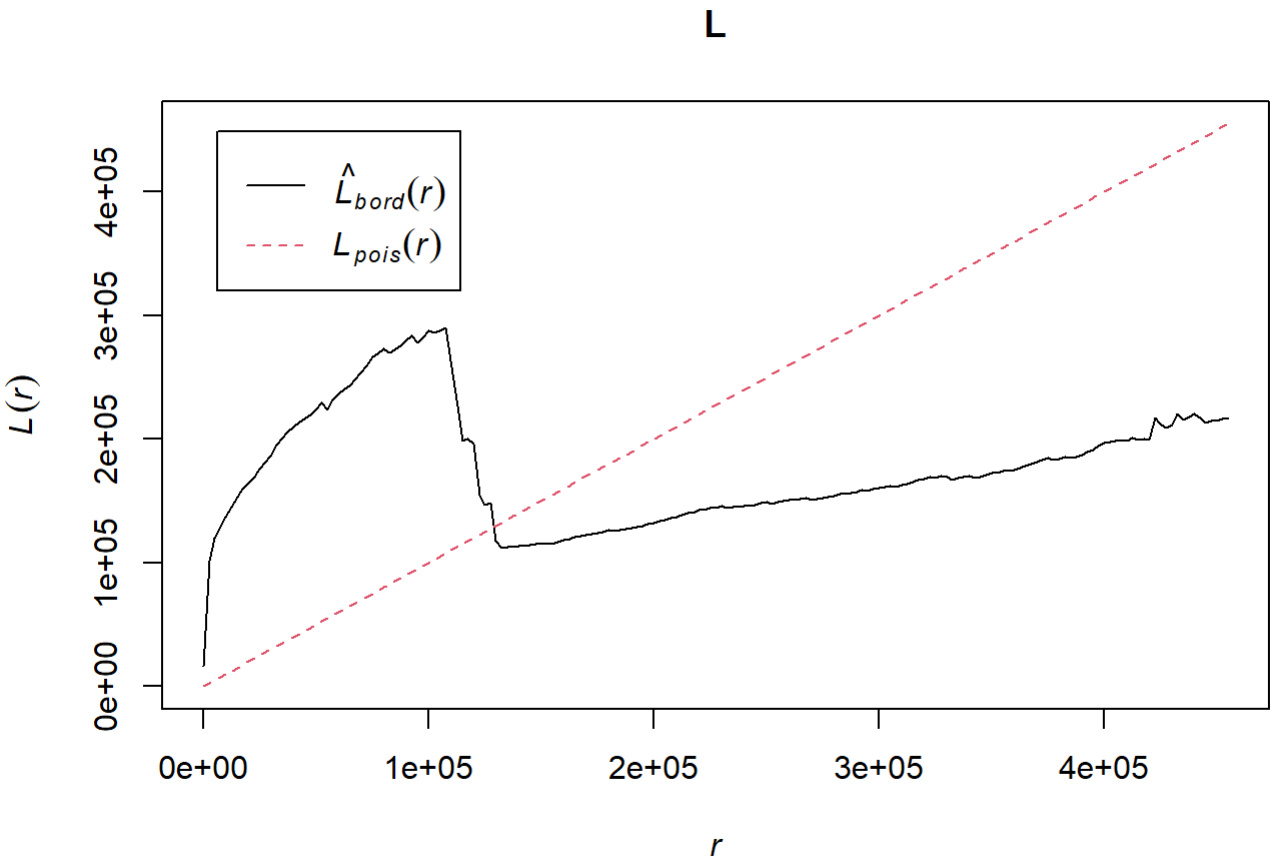


Now, as an exploratory step we can test our point pattern for evidence of spatial dependence using Ripley's K and the more intuitive L function:

```
K<-Kest(pppBrad,correction = "border") #test for spatial dependence and use the "borde
r" option to account for edge effects (points near the edge are ignored)

plot(K)#inspect the results
```



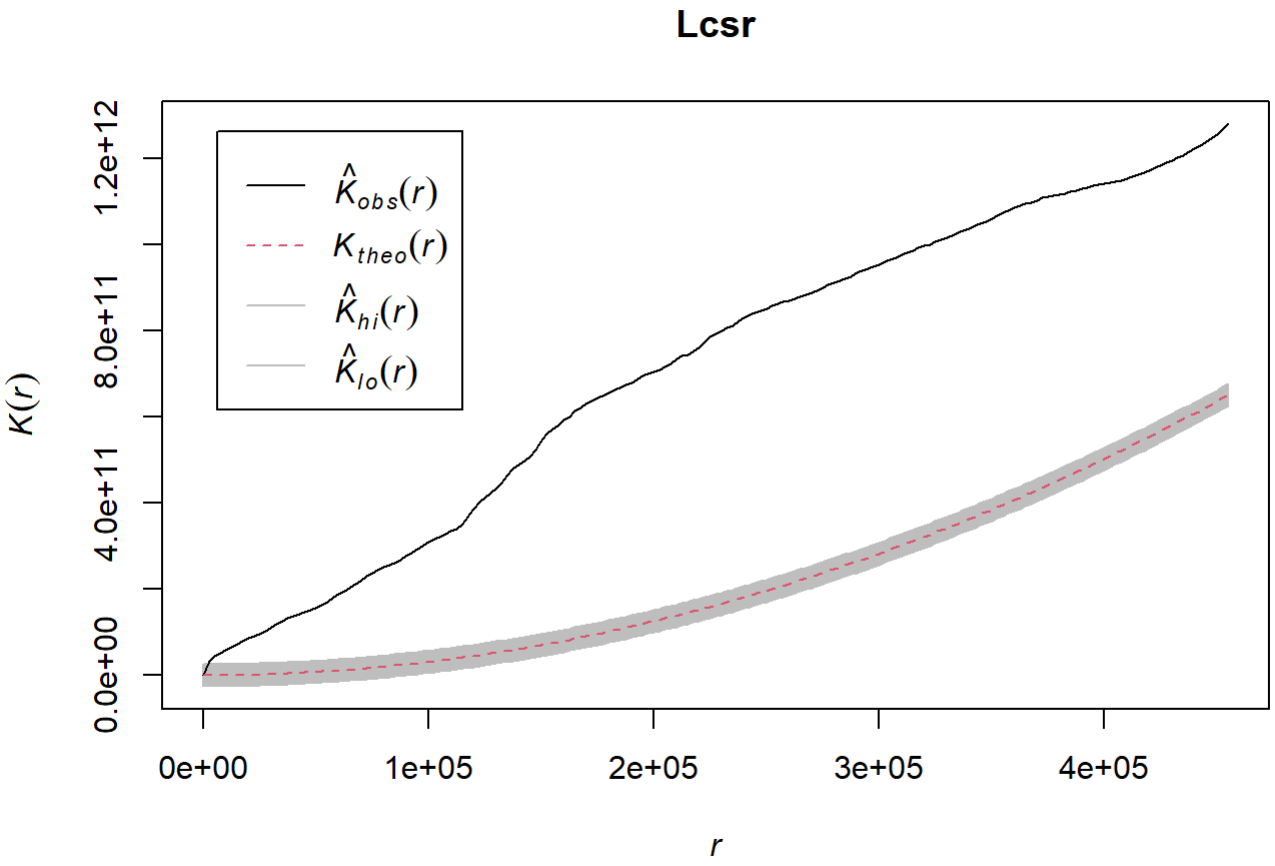

```
#check "linearized" Ripley's K for comparison
L<-Lest(pppBrad,correction = "border")

plot(L)
```




```
#generate envelope based on 39 random simulations of a homogeneous poisson process (rank = 1 means that we use minimum and maximum values to create the envelope, global=TRUE ensures that the estimate is made for all points simultaneously rather than point-wise).

Lcsr<-spatstat::envelope(pppBrad,Kest,nsim=39,rank=1,global =TRUE)
```

```
## Generating 39 simulations of CSR  ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,  39.
##
## Done.
```

```
plot(Lcsr,shade=c("hi","lo"),legend=T) #check whether our data fall within this envelope.
```



According to the Kest, Lest and envelope tests our data deviate from complete spatial randomness. We should therefore consider exploring inhomogeneous processes when developing our point process models, which we will do next.

For setting up a point process model, in addition to the study window established previously, we need to create a suitable quadrature scheme. The quadrature points used in point process modelling are analogous to the background (pseudo-absence) points we have used in previous weeks. In the case of point process modeling we generally use many more points than typically used in presence-absence modeling and adopt a uniform grid of points rather than a random distribution. This uniform distribution of quadrature points is important for establishing the “intensity” measurement (predicted points per unit area) that is the outcome of point process modeling.

We use the quadscheme() function from spatstat to create our quadrature points (also referred to as “dummy” points in some of the spatstat functions). The resolution (number) of these points is determined by the ‘nd’ argument which sets the number of grid points in the horizontal and vertical directions.

Below we test several values for nd using a for loop and a simple model using annual temperature and precipitation as predictors, each time checking the model selection criterion AIC (Akaike’s Information Criterion). Lower AIC values suggest a better fit to the data and we use this as a basis to select the number of quadrature points.

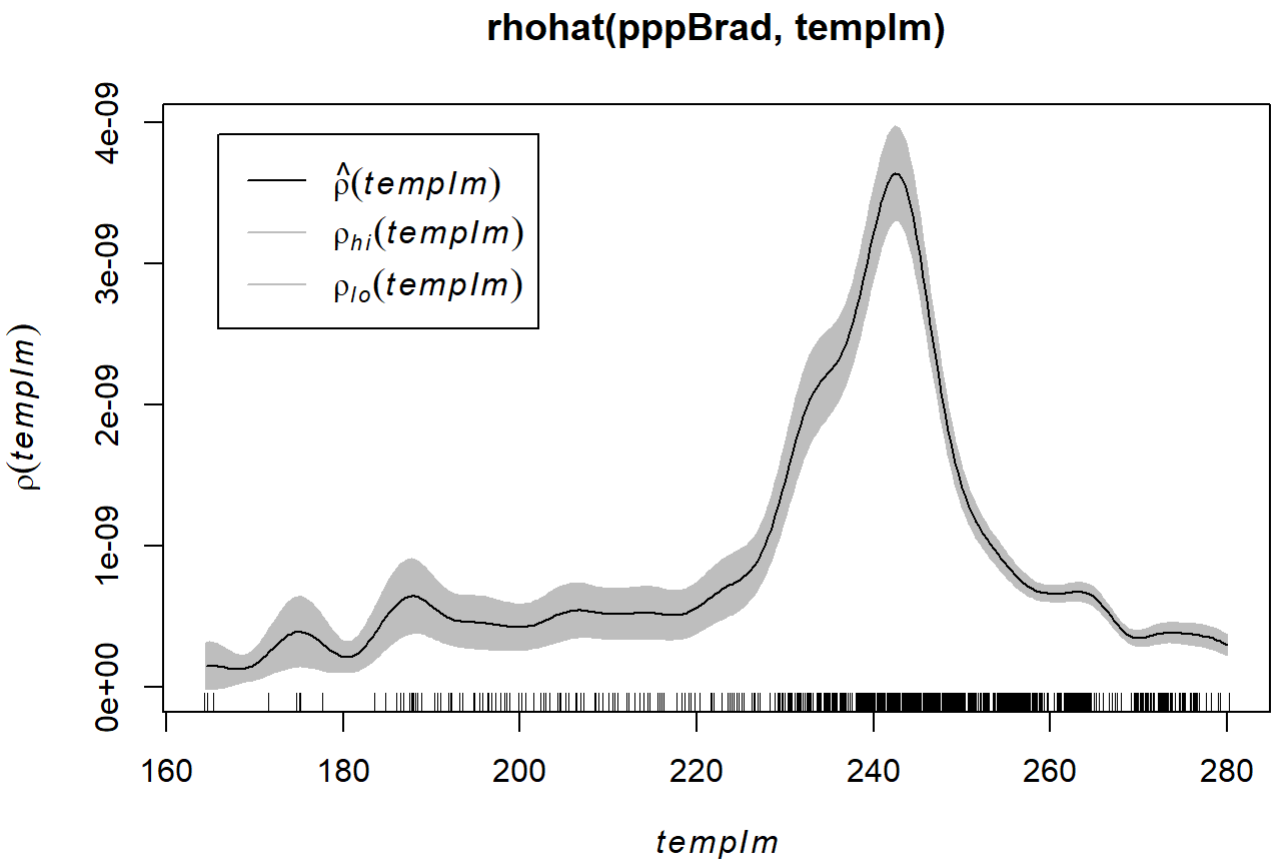
```
ndTry<-seq(100,500,by=100)


for(i in ndTry){
  Q.i<-quadscheme(pppBrad,method = "grid",nd=i)
  fit.i<-ppm(Q.i~tempIm+precipIm)
  print(i)
  print(AIC(fit.i))
}
```

```
## [1] 100
## [1] 73178.13
## [1] 200
## [1] 73177.95
## [1] 300
## [1] 73174.6
## [1] 400
## [1] 73179.78
## [1] 500
## [1] 73176.77
```

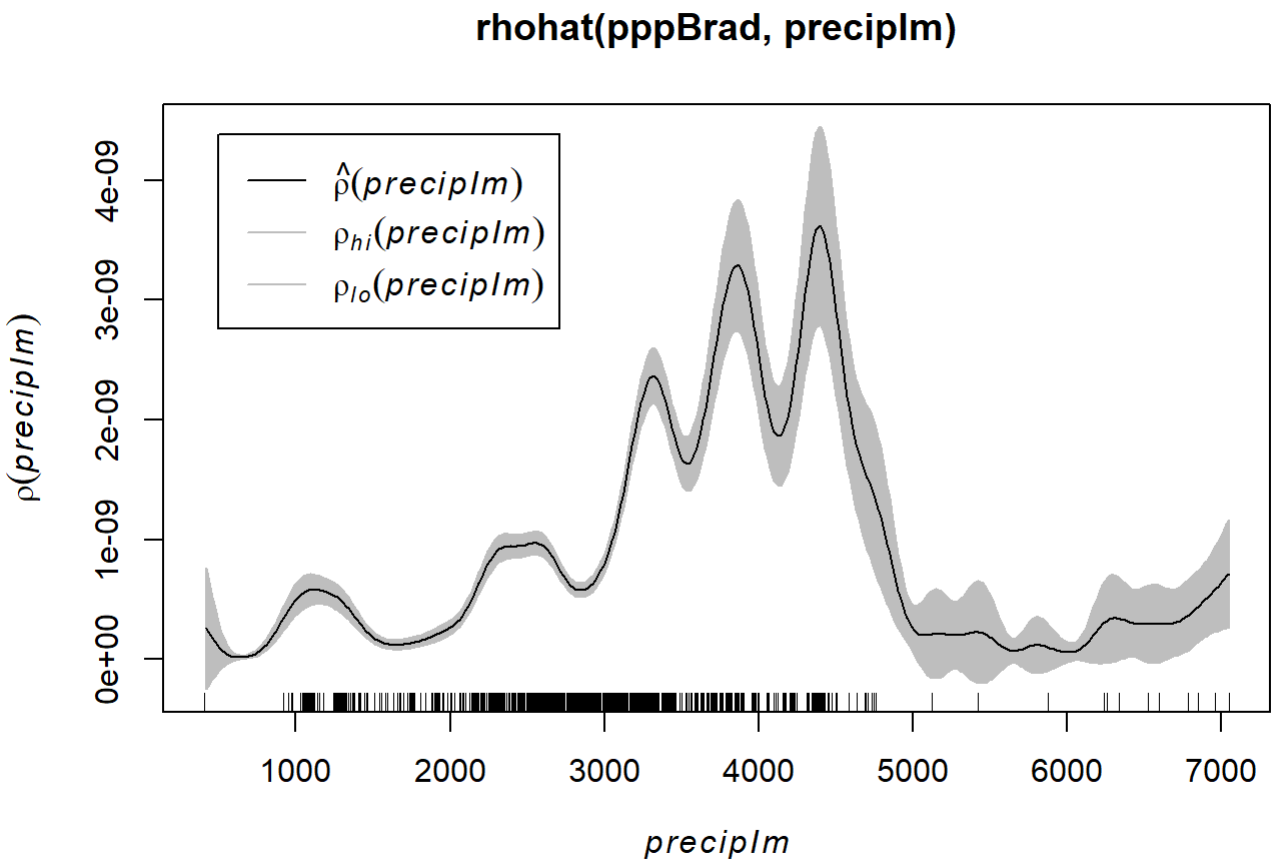
```
# select best performing value for nd to set quadrature scheme
Q<-quadscheme(pppBrad,method = "grid",nd=300)
```

Before we start our modelling in earnest, we can take advantage of some exploratory techniques in the spatstat library, in particular the rohat() function that allows us to plot values for individual covariates against predicted intensity values for our point pattern.

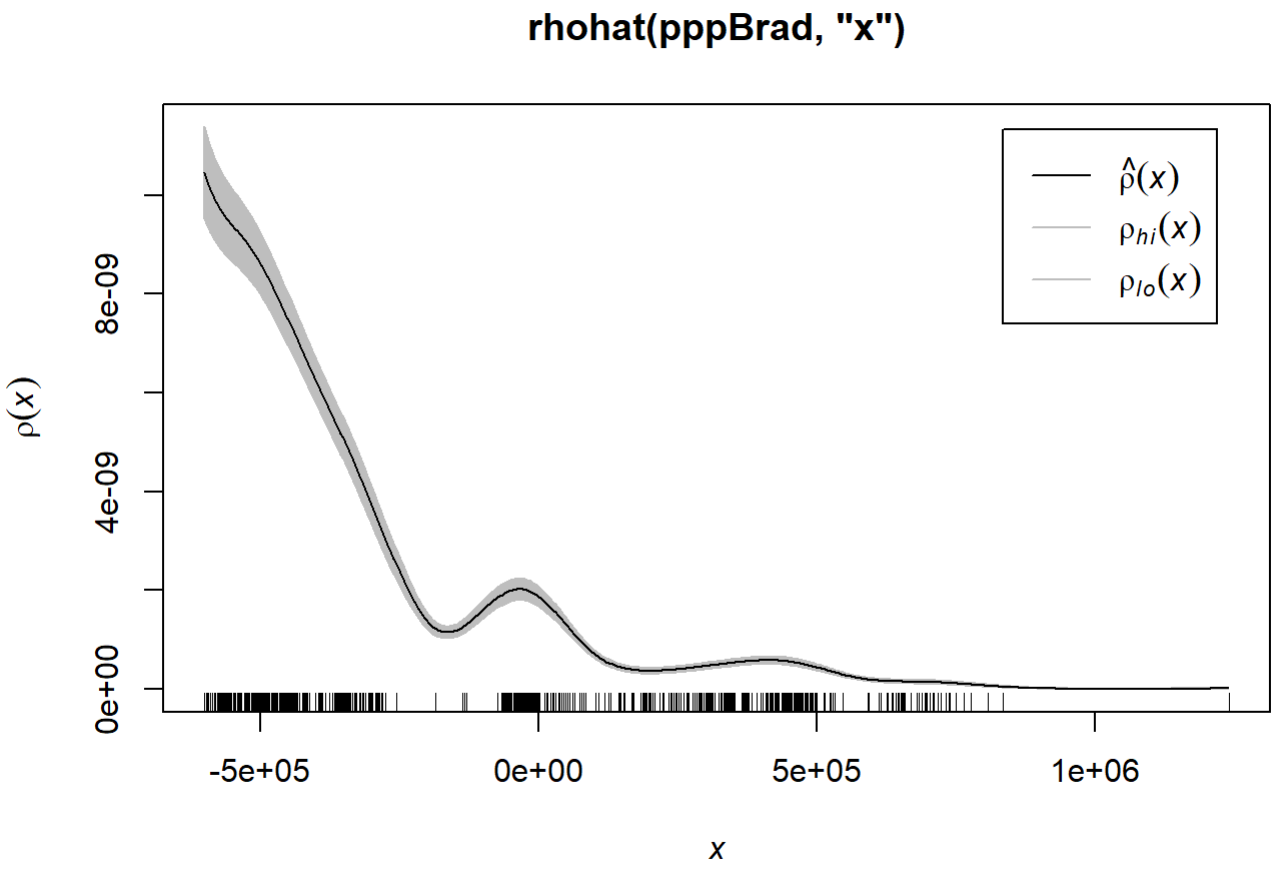
```
#plot increasing values for each environmental covariate against intensity (~ density) of the point pattern

plot(rhohat(pppBrad,tempIm)) # annual temperature
```



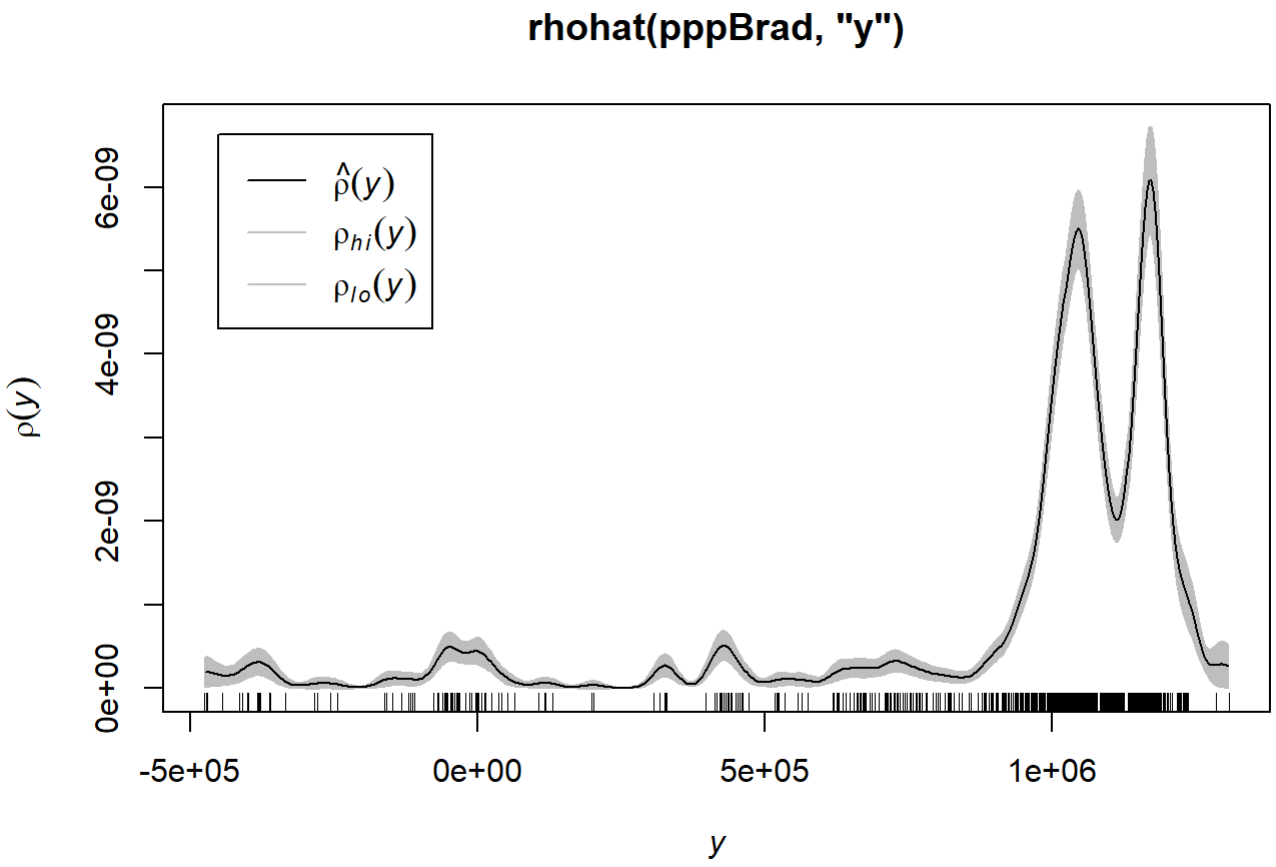

```
plot(rhohat(pppBrad,precipIm)) # annual precipitation
```




```
plot(rhohat(pppBrad,"x")) #x coordinate
```




```
plot(rhohat(pppBrad,"y")) #y coordinate
```



The rhohat() function suggests we have non-linear associations between our environmental covariates and intensity of points. For temperature and the y coordinate in particular this seems to roughly follow a quadratic trend. We can use this insight when fitting our model. For example we can use the poly() function to model these variables as quadratic curves. Below we fit a quadratic function to the tempIm object and y coordinate inside our model formula to reflect these trends.The ppm() function is used to create a point process model. In practice we would experiment will many more combinations of predictor variables and test their suitability through comparison of the model AIC statistics. This can be quite time consuming (and tedious) so for the sake of this class we will just consider the key variables of annual temperature and precipitation.

```
firstPPMod<-ppm(Q~precipIm+poly(tempIm,2)+x+poly(y,2)) #point process model based on p
olynomial terms for temperature and projected coordinates

AIC(firstPPMod) # get AIC
```

```
## [1] 68935.88
```

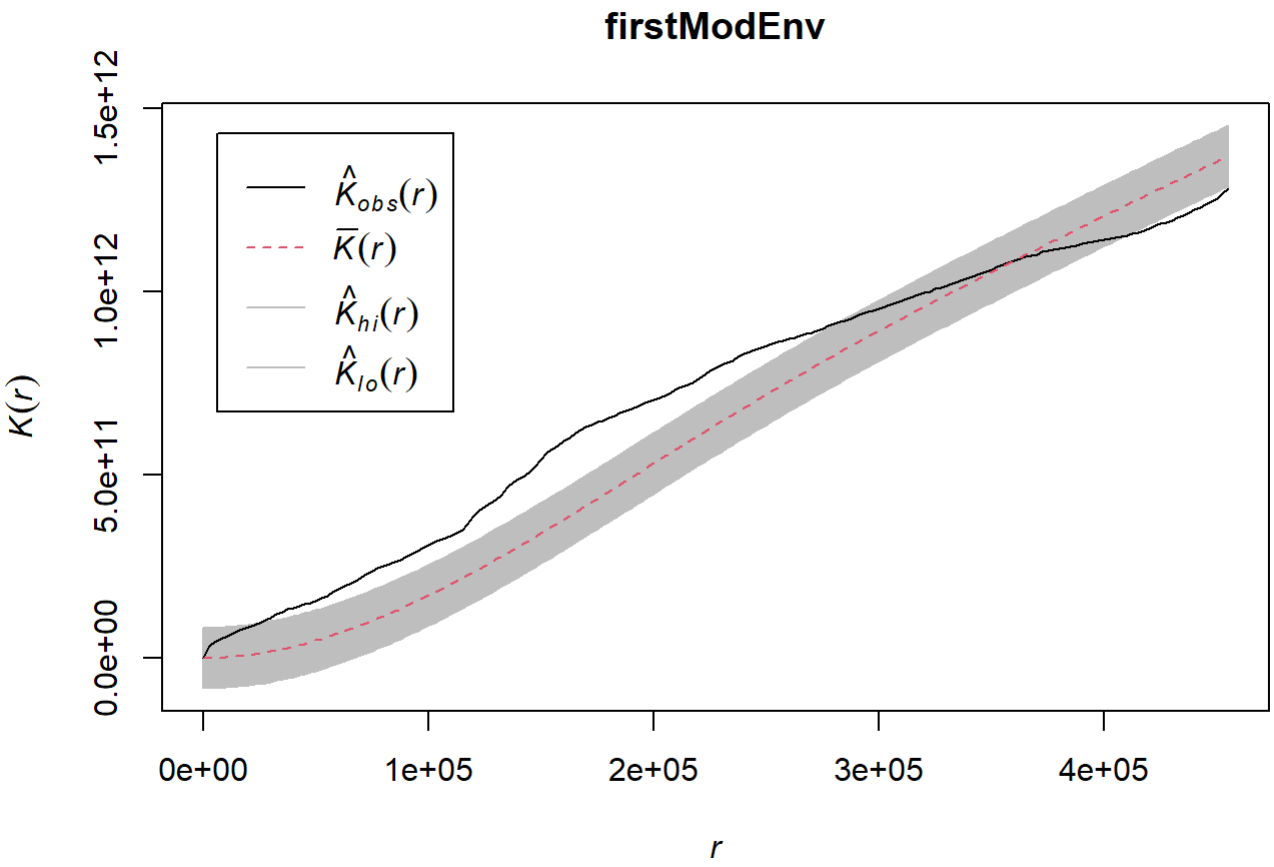
As we discovered in our earlier exploratory analysis, we have reason to believe that our data do not come from a distribution with complete spatial randomness and we may want to consider accounting for spatial dependence between points in our modeling.

This first ppm model is basically a standard poisson regression model and does not account for point interactions. It therefore assumes spatial randomness in the data as discussed in the lecture. We can test whether our model conforms to spatial randomness i.e. the data come from a random poisson distribution using the envelope function which can also accept fitted point process models as the input.

```
firstModEnv<-spatstat::envelope(firstPPMod,Kest,nsim=39,rank=1,global =TRUE)
```

```
## Generating 78 simulated realisations of fitted Poisson model (39 to estimate
## the mean and 39 to calculate envelopes) ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
## 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,  78.
##
## Done.
```

```
plot(firstModEnv)
```



As we can see our initial model results exhibit spatial clustering (values for K outside and higher than the theoretical CSR envelope) for most radii values.

We can attempt to account for this clustering inside our model using a Matern process, providing this option inside the kppm() function (for fitting clustered ppms).

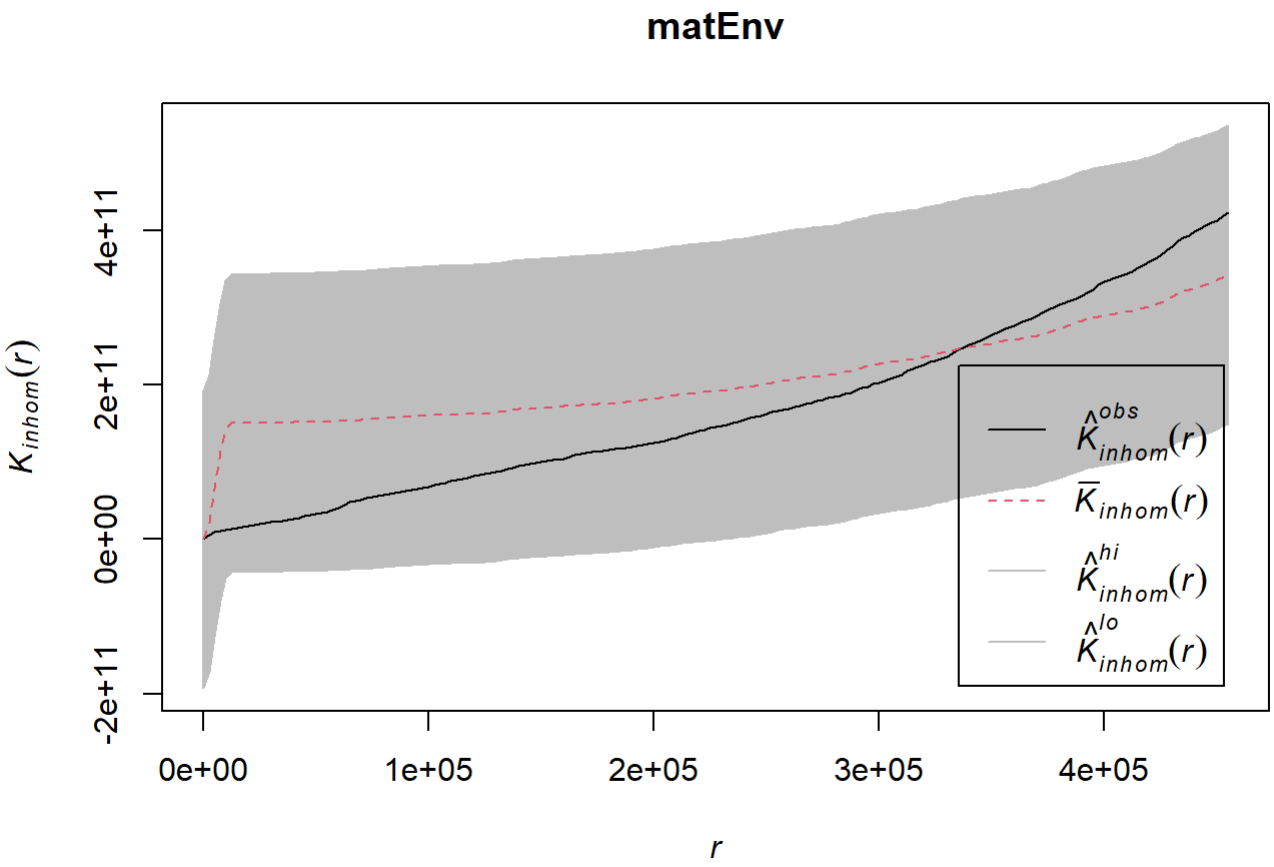
```
maternMod<-kppm(Q~poly(tempIm,2)+precipIm+x+poly(y,2),"MatClust") # apply matern clust
er process to the data.
```

Again, as for our first model we can perform a diagnostic test for our Matern process model using the envelope function. In this case, we have built our model based on the assumption that our points exhibit clustering.In other words, they represent an inhomogeneous point process. We therefore want to create an envelope based on simulations of inhomogeneous (clustering) point patterns rather than CSR. This can done by changing the second argument in the envelope() function from Kest (for Ripley's K test of CSR) to Kinhom (to simulate inhomogeneous patterns). We can then assess the goodness-of-fit of our model by seeing if our observed values for Ripley's K conform with an inhomogeneous distribution (i.e. if they lie within the simulation envelope).

```
matEnv<-spatstat::envelope(maternMod,Kinhom,nsim=39,rank=1,global=TRUE)
```

```
## Generating 78 simulated realisations of fitted cluster model (39 to estimate
## the mean and 39 to calculate envelopes) ...
## 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
## 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,  78.
##
## Done.
```

```
plot(matEnv)
```



Our model fits nicely within the simulation envelope suggesting that it fits well to an inhomogeneous point pattern and is therefore correctly specified - it is a good approximation of the behaviour of the data.

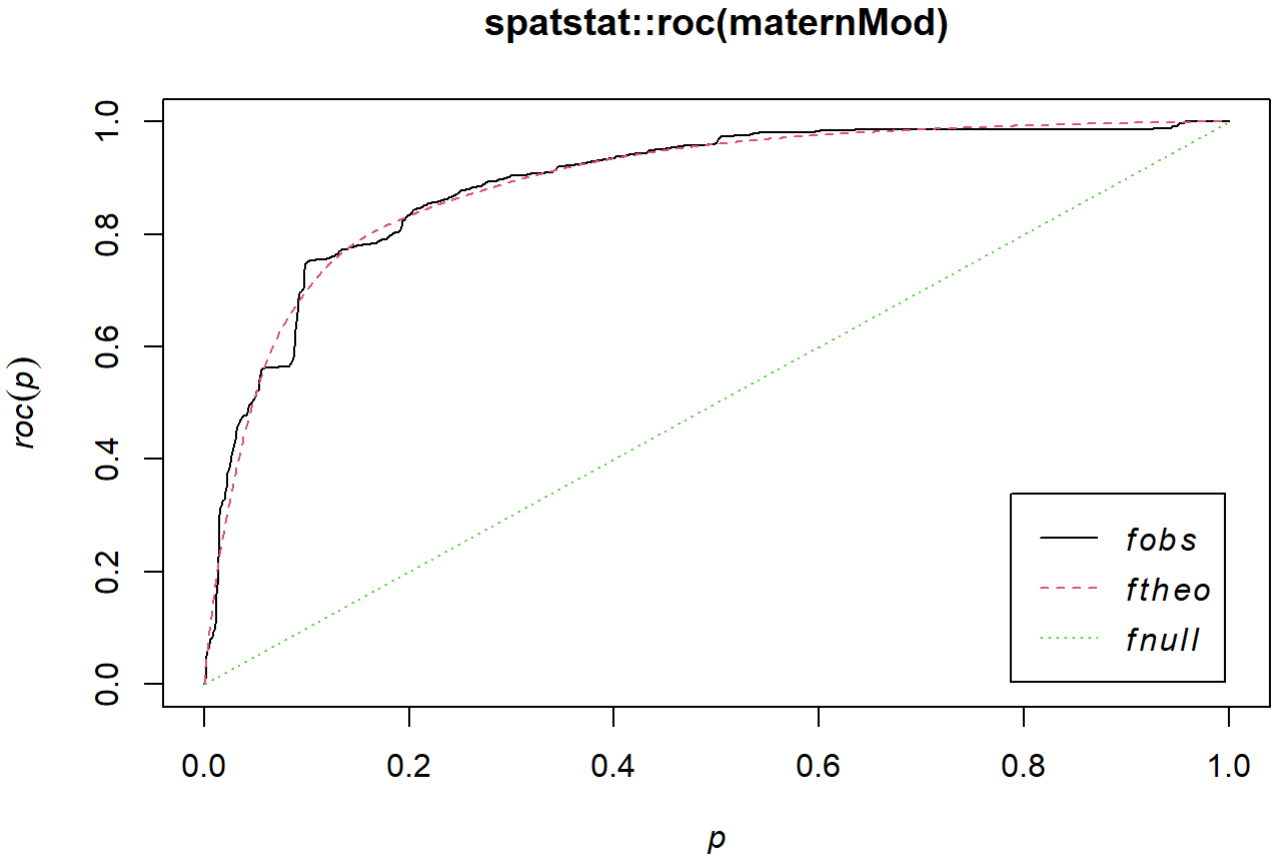
We can look at the model coefficients by calling coef(maternMod) to see the relationship between our predictor variables and the predicted intensity of points. Notice that for varibales fit with poly() you have two coefficients, the first describes the relationship with intensity for lower values of the covariate and the second coefficient describes that for higher values of the covariate. For example, in the case of temperature, intensity (species density) appears to heighten with increasing temperature before decreasing after a certain value for temperature is reached. Note that this information is not available in the machine learning approaches.

```
coef(maternMod)
```

```
##      (Intercept) poly(tempIm, 2)1 poly(tempIm, 2)2         precipIm
##    -2.078205e+01     1.685141e+02    -1.914034e+02    -1.673054e-04
##                x      poly(y, 2)1      poly(y, 2)2
##    -2.418236e-06     2.181943e+02    -1.891925e+01
```

The spatstat package contains a function for calculating the area under the curve for point process models. This differs slightly from binomial models (models with 2 outcomes) in that here it is the intensity rather than the success of correctly classifying a binary outcome that is being tested. The spatstat roc() and auc() functions offer an indication of the ability of the fitted model intensity to separate the spatial domain into areas of high and low density of points. The AUC here is the probability that a randomly-selected data point has higher predicted intensity than does a randomly-selected spatial location. So the test incorporates space into the assessment (i.e. by considering intensity across the entire study area rather than just our data points). We can plot the roc curve and return the auc as below:

```
plot(spatstat::roc(maternMod))
```



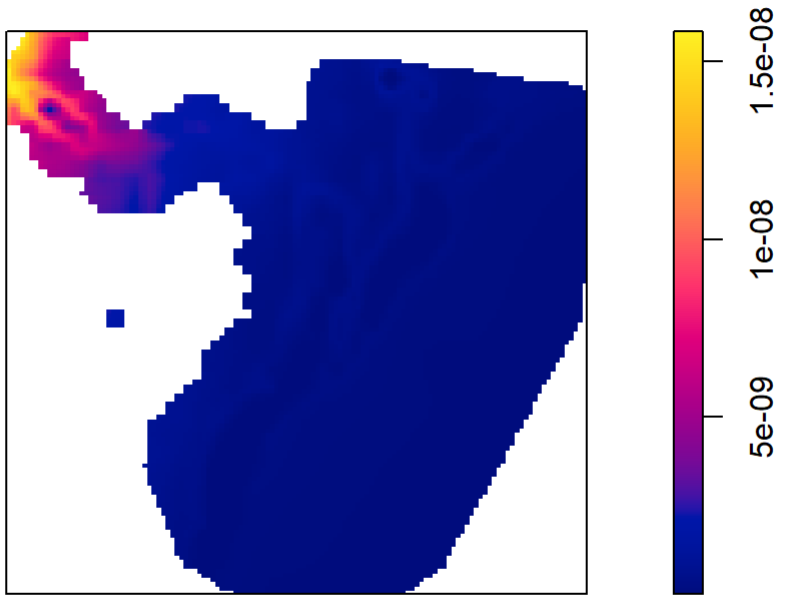

```
auc.kppm(maternMod)
```

```
##       obs      theo
## 0.8901111 0.8903418
```

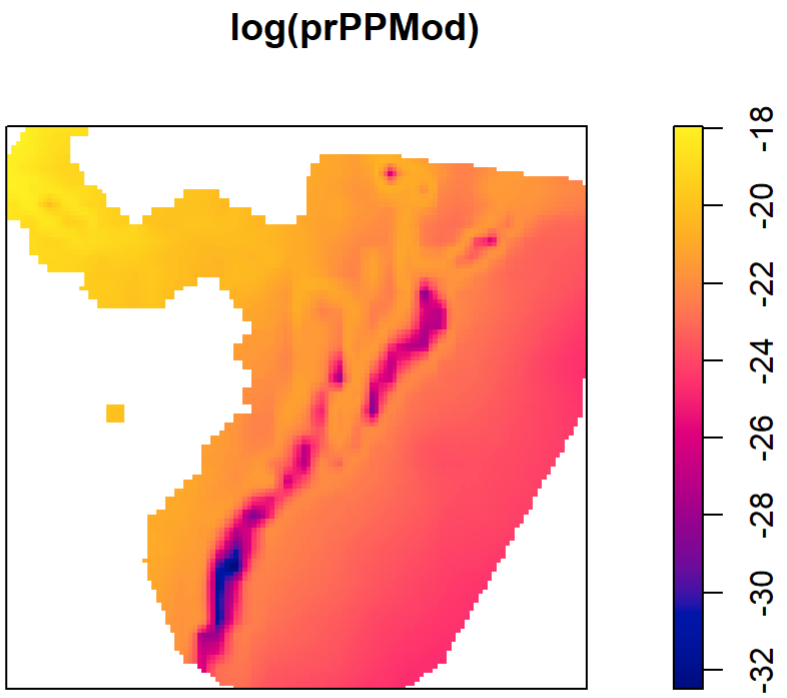
The roc curve and auc statistic for our model are high, following closely the output that would be expected for a well specified inhomogeneous point process model ("ftheo" here) and well above the null line (a 50:50 line then represents no ability to discriminate high from low intensity - "fnull" here). So we can see that with a well specified model that incorporates spatial dependence we have achieved a better representation of the data with the point process approach compared to our machine learning techniques. In reality we could carry out much more extensive model tuning for our machine learning (increasing number of iterations in the tuning step) but it's perhaps unlikely that an auc score comparable to that of the Matern model would be achieved.

We can create a mapped output of our results using the predict function. Remember our point process model output is one of intensity (points per unit area) and has very low values given that our units are in metres. We can log transform the raster values to better differentiate the distribution of values.

```
prPPMod<-predict(maternMod)

plot(prPPMod)
```

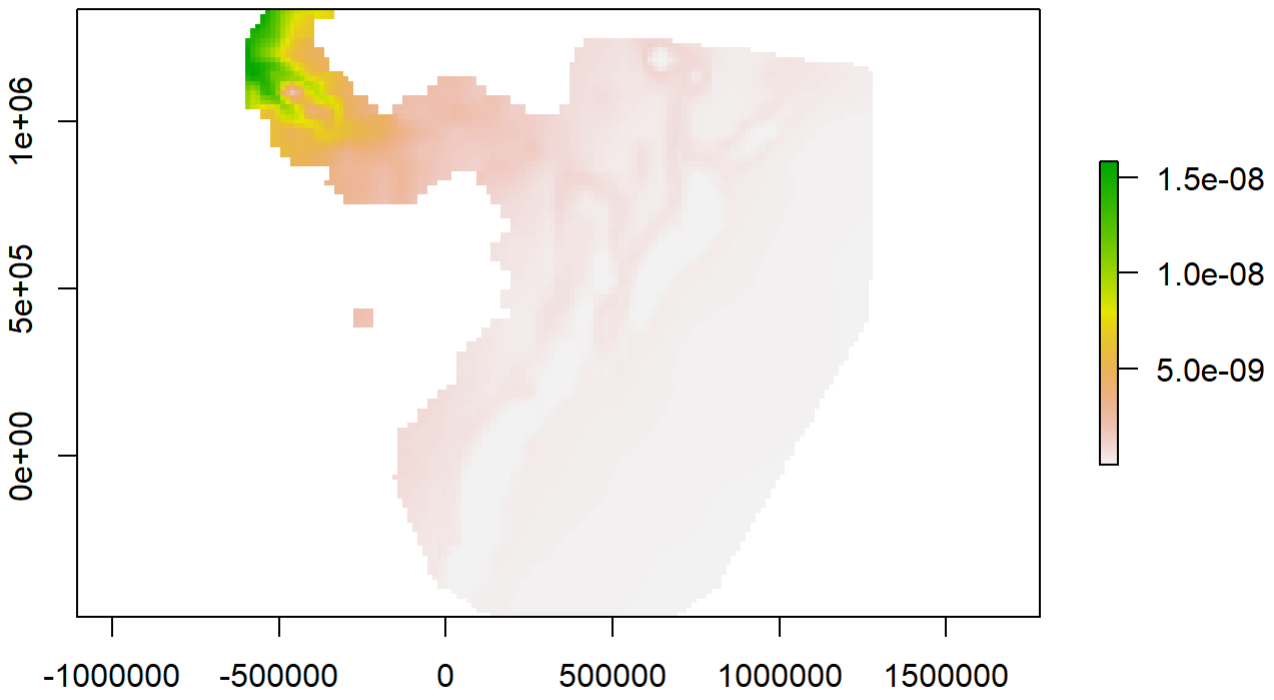



```
#log transform the result to highlight differences in value distribution
plot(log(prPPMod))
```



Finally we can convert our output to raster (remember spatstat deals in image objects):

```
maternRas<-raster(prPPMod)

plot(maternRas)
```



You've reached the end of this week's practical. You've come a long way on your species distribution modeling journey. There is no exercise associated with this week's class you but might want to practice these techniques by applying them to other species records for this study area. iNaturalist (https://www.inaturalist.org (https://www.inaturalist.org)) is a good place to start (you can download records from this website using the gbif() function as most if not all records are also hosted by the GBIF)