Document S1: Source code used in (a) Sentinel-1 and Sentinel-2 data preparation and input variable generation; and (b) boosted regression tree modeling, classification, and accuracy assessment.

(a) Satellite data preparation and processing

The following source codes perform processing (e.g., cloud-masking, compositing, etc.) of Sentinel-1 and Sentinel-2 data, and derivation of model input variables for use in wetland probability modeling. They are written using the JavaScript programming language, within the Google Earth Engine code editing environment [33] (see References in main article). They are meant for use within that particular environment. We have included additional angle-based masking, angle correction and speckle filtering within the provided code for processing Sentinel-1 data; these were not used in the original processing of these data for the experiment described in the corresponding *Remote Sensing* article, but were added later by the authors, and we believe are of interest for future applications. It was decided that including these additional processing steps in the supplied code would be of value to readers of the article.

```
////////////////////////////////////////////////////////////////////////////////
// CODE: Sentinel-1 Data Processing and Model Input Variable Derivation       //
//                                                                            //
// AUTHOR: Evan R. DeLancey, GIS Land-use Analyst, Alberta Biodiversity       //
//         Monitoring Institute (Edmonton, AB, Canada)                        //
//         email: edelance@ualberta.ca                                        //
//                                                                            //
// INPUTS/ARGUMENTS: - Geometry of area of interest                           //
//                                                                            //
// NOTES/COMMENTS: The angle-based masking, wind filtering, angle correction  //
//                 and speckle filtering included here are all later additions //
//                 to the original S-1 data processing codes                  //
//                                                                            //
// LAST UPDATED: 2017-08-24                                                   //
////////////////////////////////////////////////////////////////////////////////

//Create geometry object encompassing area of interest
var geometry = ee.Geometry.Polygon(
        [[[-114.00, 56.00],
          [-114.00, 57.00],
          [-112.00, 57.00],
          [-112.00, 56.00]]]);



// 1) Proccess and derive normalized porlarization (Pol) input variable
// ----------------------------------------------------------------------
// ----------------------------------------------------------------------

// Create image collection of S-1 images filtered by date, polarization,
// resolution, and orbit
var s1pol = ee.ImageCollection('COPERNICUS/S1_GRD')
  .filterBounds(geometry)
  .filterDate('2016-05-15', '2016-08-31')
  .filterMetadata('transmitterReceiverPolarisation', 'equals', ['VV', 'VH'])
  .filterMetadata('resolution_meters', 'equals' , 10);
```

```
// Function to mask out edges of images using angle
// (mask out angles <= 30.63993)
var maskAngGT30 = function(image) {
  var ang = image.select(['angle']);
  return image.updateMask(ang.gt(30.63993));
};

// Function to mask out edges of images using using angle
// (mask out angles >= 44.73993)
var maskAngLT45 = function(image) {
  var ang = image.select(['angle']);
  return image.updateMask(ang.lt(45.53993));
};

// Apply angle masking functions to image collection
var s1pol = s1pol.map(maskAngGT30);
var s1pol = s1pol.map(maskAngLT45);

// Function to filter out windy days using climate forecasts
var pctWat = function(image){
  var d = image.date().format('Y-M-d');
  var wx = ee.ImageCollection('NOAA/CFSV2/FOR6H')
    .filterDate(d);
  var vWind = wx.select(['v-component_of_wind_height_above_ground']);
  var a = vWind.max();
  var uWind = wx.select(['u-component_of_wind_height_above_ground']);
  var b = uWind.max();
  var a = a.pow(2);
  var b = b.pow(2);
  var ab = a.add(b);
  var ws = ab.sqrt();
  var ws = ws.multiply(3.6);
  return image.updateMask(ws.lt(12));
};

// Apply windy day filter to image collection
var s1pol = s1pol.map(pctWat);

// Function to perform angle correction
function toGamma0(image) {
  var vh = image.select('VH').subtract(image.select('angle')
  .multiply(Math.PI/180.0).cos().log10().multiply(10.0));
  return vh.addBands(image.select('VV').subtract(image.select('angle')
  .multiply(Math.PI/180.0).cos().log10().multiply(10.0)));
}

// Apply angle correction
var s1pol = s1pol.map(toGamma0);

// Function to add band containing normalized difference
// between VH and VV
var adddiff = function(image) {
  return image.addBands(image.expression(
      '(VH - VV) / (VH + VV)', {
        'VH': image.select(['VH']),
        'VV': image.select(['VV'])
      }
    ));
};
```

```
// Add normalized difference band to image collection and
// create new variable of this band
var s1pol = s1pol.map(adddiff);
var NPOL = s1pol.select(['VH_1']);

// Function to create boxcar 3 x 3 pixel filter
var boxcar = ee.Kernel.circle({
  radius: 3, units: 'pixels', normalize: true
});

// Function to apply boxcar filter
var fltr = function(image) {
  return image.convolve(boxcar);
};

// Apply 3 x 3 pixel mean filter to Pol image
var NPOL = NPOL.map(fltr);
var NPOL = NPOL.mean();

// // Map Npol
// Map.addLayer(NPOL, {min:0, max:0.5}, "Filtered NPol");

// Clip to area of interest (with 500 m buffer)
var NPOLout = NPOL.clip(geometry.buffer(500));

// Display results in map window
Map.centerObject(geometry,8);
Map.addLayer(geometry, {}, "AOI");
Map.addLayer(NPOLout, {min:0, max:0.5}, "NPOL clipped");



// 2) Proccess and derive VV and VVsd input variables
// ----------------------------------------------------------------------
// ----------------------------------------------------------------------

// Create image collection of S-1 images, filtered by geometry, year,
// day of year, polarization, and resolution
var s1vv = ee.ImageCollection('COPERNICUS/S1_GRD')
  .filterBounds(geometry)
  .filterDate('2014-10-03', '2017-08-31')
  .filter(ee.Filter.dayOfYear(91,304)) //April 1 - Oct 31
  .filter(ee.Filter.listContains('transmitterReceiverPolarisation', 'VV'))
  .filterMetadata('resolution_meters', 'equals' , 10)


// Function to mask out edges of images using using angle
// (mask out angles >= 45.23993)
var maskAngLT452 = function(image) {
  var ang = image.select(['angle']);
  return image.updateMask(ang.lt(45.23993));
};

// Apply angle masking functions to image collection
var s1vv = s1vv.map(maskAngGT30);
var s1vv = s1vv.map(maskAngLT452);
```

```
// Function to apply angle correction (for VV)
function toGamma01(image) {
  return image.select('VV').subtract(image.select('angle')
  .multiply(Math.PI/180.0).cos().log10().multiply(10.0));
}

// Apply angle correciton to image collection
var s1vv = s1vv.map(toGamma01);

//------------------------------------------------------------------------------//
// Sigma Lee speckle filtering
//------------------------------------------------------------------------------//
function toNatural(img) {
  return ee.Image(10.0).pow(img.select(0).divide(10.0));
}

function toDB(img) {
  return ee.Image(img).log10().multiply(10.0);
}

// The RL speckle filter from
// https://code.earthengine.google.com/2ef38463ebaf5ae133a478f173fd0ab5
// by Guido Lemoine
function RefinedLee(img) {
  // img must be in natural units, i.e. not in dB!
  // Set up 3x3 kernels
  var weights3 = ee.List.repeat(ee.List.repeat(1,3),3);
  var kernel3 = ee.Kernel.fixed(3,3, weights3, 1, 1, false);

  var mean3 = img.reduceNeighborhood(ee.Reducer.mean(), kernel3);
  var variance3 = img.reduceNeighborhood(ee.Reducer.variance(), kernel3);

  // Use a sample of the 3x3 windows inside a 7x7 windows to determine gradients
  // and directions
  var sample_weights = ee.List([[0,0,0,0,0,0,0], [0,1,0,1,0,1,0],
     [0,0,0,0,0,0,0], [0,1,0,1,0,1,0], [0,0,0,0,0,0,0], [0,1,0,1,0,1,0],
     [0,0,0,0,0,0,0]]);

  var sample_kernel = ee.Kernel.fixed(7,7, sample_weights, 3,3, false);

  // Calculate mean and variance for the sampled windows and store as 9 bands
  var sample_mean = mean3.neighborhoodToBands(sample_kernel);
  var sample_var = variance3.neighborhoodToBands(sample_kernel);

  // Determine the 4 gradients for the sampled windows
  var gradients = sample_mean.select(1).subtract(sample_mean.select(7)).abs();
  gradients = gradients.addBands(sample_mean.select(6).subtract(sample_mean
  .select(2)).abs());
  gradients = gradients.addBands(sample_mean.select(3).subtract(sample_mean
  .select(5)).abs());
  gradients = gradients.addBands(sample_mean.select(0).subtract(sample_mean
  .select(8)).abs());

  // And find the maximum gradient amongst gradient bands
  var max_gradient = gradients.reduce(ee.Reducer.max());

  // Create a mask for band pixels that are the maximum gradient
  var gradmask = gradients.eq(max_gradient);
```

```
// duplicate gradmask bands: each gradient represents 2 directions
  gradmask = gradmask.addBands(gradmask);

  // Determine the 8 directions
  var directions = sample_mean.select(1).subtract(sample_mean.select(4))
     .gt(sample_mean.select(4).subtract(sample_mean.select(7))).multiply(1);
  directions = directions.addBands(sample_mean.select(6).subtract(sample_mean
     .select(4)).gt(sample_mean.select(4).subtract(sample_mean.select(2)))
     .multiply(2));
  directions = directions.addBands(sample_mean.select(3).subtract(sample_mean
     .select(4)).gt(sample_mean.select(4).subtract(sample_mean.select(5)))
     .multiply(3));
  directions = directions.addBands(sample_mean.select(0).subtract(sample_mean
     .select(4)).gt(sample_mean.select(4).subtract(sample_mean.select(8)))
     .multiply(4));
  // The next 4 are the not() of the previous 4
  directions = directions.addBands(directions.select(0).not().multiply(5));
  directions = directions.addBands(directions.select(1).not().multiply(6));
  directions = directions.addBands(directions.select(2).not().multiply(7));
  directions = directions.addBands(directions.select(3).not().multiply(8));

  // Mask all values that are not 1-8
  directions = directions.updateMask(gradmask);

  // "collapse" the stack into a singe band image (due to masking, each pixel
  // has just one value (1-8) in it's directional band, and is otherwise masked)
  directions = directions.reduce(ee.Reducer.sum());

  // Generate stats
  var sample_stats = sample_var.divide(sample_mean.multiply(sample_mean));

  // Calculate localNoiseVariance
  var sigmaV = sample_stats.toArray().arraySort().arraySlice(0,0,5)
     .arrayReduce(ee.Reducer.mean(), [0]);

  // Set up the 7*7 kernels for directional statistics
  var rect_weights = ee.List.repeat(ee.List.repeat(0,7),3)
     .cat(ee.List.repeat(ee.List.repeat(1,7),4));

  // Set weights
  var diag_weights = ee.List([[1,0,0,0,0,0,0], [1,1,0,0,0,0,0], [1,1,1,0,0,0,0],
    [1,1,1,1,0,0,0], [1,1,1,1,1,0,0], [1,1,1,1,1,1,0], [1,1,1,1,1,1,1]]);
  var rect_kernel = ee.Kernel.fixed(7,7, rect_weights, 3, 3, false);
  var diag_kernel = ee.Kernel.fixed(7,7, diag_weights, 3, 3, false);

  // Create stacks for mean and variance using the original kernels.
  // Mask with relevant direction.
  var dir_mean = img.reduceNeighborhood(ee.Reducer.mean(), rect_kernel)
     .updateMask(directions.eq(1));
  var dir_var = img.reduceNeighborhood(ee.Reducer.variance(), rect_kernel)
     .updateMask(directions.eq(1));

  dir_mean = dir_mean.addBands(img.reduceNeighborhood(ee.Reducer.mean(),
     diag_kernel).updateMask(directions.eq(2)));
  dir_var = dir_var.addBands(img.reduceNeighborhood(ee.Reducer.variance(),
     diag_kernel).updateMask(directions.eq(2)));
```

```
// and add the bands for rotated kernels
  for (var i=1; i<4; i++) {
    dir_mean = dir_mean.addBands(img.reduceNeighborhood(ee.Reducer.mean(),
       rect_kernel.rotate(i)).updateMask(directions.eq(2*i+1)));
    dir_var = dir_var.addBands(img.reduceNeighborhood(ee.Reducer.variance(),
       rect_kernel.rotate(i)).updateMask(directions.eq(2*i+1)));
    dir_mean = dir_mean.addBands(img.reduceNeighborhood(ee.Reducer.mean(),
       diag_kernel.rotate(i)).updateMask(directions.eq(2*i+2)));
    dir_var = dir_var.addBands(img.reduceNeighborhood(ee.Reducer.variance(),
       diag_kernel.rotate(i)).updateMask(directions.eq(2*i+2)));
  }

  // "collapse" the stack into a single band image (due to masking, each pixel
  // has just one value in it's directional band, and is otherwise masked)
  dir_mean = dir_mean.reduce(ee.Reducer.sum());
  dir_var = dir_var.reduce(ee.Reducer.sum());

  // A finally generate the filtered value
  var varX = dir_var.subtract(dir_mean.multiply(dir_mean).multiply(sigmaV))
     .divide(sigmaV.add(1.0));

  var b = varX.divide(dir_var);

  var result = dir_mean.add(b.multiply(img.subtract(dir_mean)));
  return(result.arrayFlatten([['sum']]));
}
//----------------------------End Sigma Lee Filtering-------------------------//

// Apply three functions as part of Sigma Lee filtering
var s1vv = s1vv.map(toNatural);
var s1vv = s1vv.map(RefinedLee);
var s1vv = s1vv.map(toDB);

// Extract mean VV and VVsd input variables
var VV = s1vv.mean();
var VVsd = s1vv.reduce(ee.Reducer.stdDev());

// Apply 3 x 3 pixel mean filter to VV and VVsd images
//var VVfltrd = VV.map(fltr);
var VVfltrd = fltr(VV);
var VVsdfltrd = fltr(VVsd);

// // Map orig mean VV and VV sd
// Map.addLayer(VVfltrd,{min:-16.0, max:-0.53},'Filtered VV');
// Map.addLayer(VVsdfltrd,{min:1, max:5},'Filtered VVsd');

// Clip to area of interest (with 500 m buffer)
var VVout = VVfltrd.clip(geometry.buffer(500));
var VVsdout = VVsdfltrd.clip(geometry.buffer(500));

// Display results in map window
Map.addLayer(geometry, {}, "AOI");
Map.addLayer(VVout, {min:-10, max:-5}, 'VV filtered, clip');
Map.addLayer(VVsdout, {min:1, max:5}, 'VVsd filtered, clip');
```

```
// 3) Export image input variables created above
// ---------------------------------------------------------------------

Export.image.toDrive({
  image: NPOLout,
  description: 'NPOL',
  scale: 10,
  region: geometry,
  maxPixels: 3E10

});

Export.image.toDrive({
  image: VVout,
  description: 'VV',
  scale: 10,
  region: geometry,
  maxPixels: 3E10

});

Export.image.toDrive({
  image: VVsdout,
  description: 'VVsd',
  scale: 10,
  region: geometry,
  maxPixels: 3E10

});
```

```
////////////////////////////////////////////////////////////////////////////////
// CODE: Sentinel-2 Data Processing and Model Input Variable Derivation        //
//                                                                            //
// AUTHOR: Evan R. DeLancey, GIS Land-use Analyst, Alberta Biodiversity        //
//         Monitoring Institute (Edmonton, AB, Canada)                        //
//         email: edelance@ualberta.ca                                        //
//                                                                            //
// INPUTS/ARGUMENTS: - Geometry of area of interest                           //
//                                                                            //
// NOTES/COMMENTS:                                                            //
//                                                                            //
// LAST UPDATED: 2017-08-24                                                   //
////////////////////////////////////////////////////////////////////////////////

//Create geometry object encompassing area of interest
var geometry = ee.Geometry.Polygon(
        [[[-114.00, 56.00],
          [-114.00, 57.00],
          [-112.00, 57.00],
          [-112.00, 56.00]]]);

// Create image collection of S-2 imagery for 2016
var S2 = ee.ImageCollection('COPERNICUS/S2')
  //filter start and end date
  .filterDate('2016-05-15', '2016-08-31')
  //filter according to drawn boundary
  .filterBounds(geometry);

// Create image collection of S-2 imagery for 2017
var S2_1 = ee.ImageCollection('COPERNICUS/S2')
  //filter start and end date
  .filterDate('2017-05-15', '2017-08-31')
  //filter according to drawn boundary
  .filterBounds(geometry);

// Merge two image collections
var S2 = ee.ImageCollection(S2.merge(S2_1));

// Function to mask cloud from built-in quality band
// information on cloud
var maskcloud1 = function(image) {
  var QA60 = image.select(['QA60']);
  return image.updateMask(QA60.lt(1));
};

// Apply cloud-masking to image collection
var S2 = S2.map(maskcloud1);

// Function to mask further cloud using B1 (cirrus cloud) threshold
var maskcloud2 = function(image) {
  var B1 = image.select(['B1']);
  var bin = B1.gt(1500);
  return image.updateMask(bin.lt(1));
};

// Apply 2nd cloud-masking to image collection
var S2 = S2.map(maskcloud2);
```

```
// Function to calculate and add an NDVI band
var addNDVI = function(image) {
  return image.addBands(image.normalizedDifference(['B8', 'B4']));
};

// Add NDVI band to image collection
var S2 = S2.map(addNDVI);

// Extract NDVI band and create NDVI median composite image
var NDVI = S2.select(['nd']);
var NDVI = NDVI.median();

// Function to calculate and add an NDWI band
var addNDWI = function(image) {
  return image.addBands(image.normalizedDifference(['B3', 'B8']));
};

// Add NDWI band to image collection
var S2 = S2.map(addNDWI);

// Extract NDWI band and creat NDWI median composite image
var NDWI = S2.select(['nd_1']);
var NDWI = NDWI.median();

// Extract bands B2 (blue), B3 (green), B4 (red), and B8 (NIR); create
// median composite images of each
var B2 = S2.select(['B2']);
var B2 = B2.median();
var B3 = S2.select(['B3']);
var B3 = B3.median();
var B4 = S2.select(['B4']);
var B4 = B4.median();
var B8 = S2.select(['B8']);
var B8 = B8.median();

// Clip input variable images
var NDVI = NDVI.clip(geometry.buffer(500));
var NDWI = NDWI.clip(geometry.buffer(500));
var B2 = B2.clip(geometry.buffer(500));
var B3 = B3.clip(geometry.buffer(500));
var B4 = B4.clip(geometry.buffer(500));
var B8 = B8.clip(geometry.buffer(500));

// Create palettes for display of NDVI and NDWI
var ndvi_pal = ['#d73027', '#f46d43', '#fdae61', '#fee08b', '#d9ef8b',
   '#a6d96a', '#66bd63', '#1a9850'];
var ndwi_pal = ['#ece7f2', '#d0d1e6', '#a6bddb', '#74a9cf', '#3690c0',
   '#0570b0', '#045a8d', '#023858'];

// Display NDVI and NDWI results on map
Map.addLayer(NDVI, {min:-0.5, max:0.9, palette: ndvi_pal}, 'NDVI');
Map.addLayer(NDWI, {min:-1, max:1, palette: ndwi_pal}, 'NDWI');
```

```
// Export results to Google Drive

Export.image.toDrive({
  image: NDWI,
  description: 'NDWI',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});

Export.image.toDrive({
  image: NDVI,
  description: 'NDVI',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});

Export.image.toDrive({
  image: B2,
  description: 'B2',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});

Export.image.toDrive({
  image: B3,
  description: 'B3',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});

Export.image.toDrive({
  image: B4,
  description: 'B4',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});

Export.image.toDrive({
  image: B8,
  description: 'B8',
  scale: 10,
  region: geometry,
  maxPixels: 10E10

});
```

(b) Boosted regression tree modeling

The following source code was used to perform training and reference data sample selection, boosted regression tree modeling, classification, and accuracy assessment. It is written using the R programming language [49] (see References list in main article).

```
#################################################################################
## CODE: Boosted regression tree modeling of wetland probability of occurrence, ##
##       including training and reference data sampling, modeling,             ##
##       classification, and accuracy assessment.                              ##
##                                                                             ##
## AUTHORS: Evan R. DeLancey, Alberta Biodiversity Monitoring Institute        ##
##                            Edmonton, Alberta, Canada                        ##
##                            email: edelance@ualberta.ca                      ##
##                                                                             ##
##          Jennifer N. Hird, Dept. of Geography, University of Calgary        ##
##                            Calgary, Alberta, Canada                         ##
##                            email: jennifer.hird@ucalgary.ca                 ##
##                                                                             ##
## ACKNOWLEDGEMENTS: We gratefully acknowledge the input and advice of Marc-   ##
##                   André Parisien, of the Canadian Forest Service            ##
##                                                                             ##
## INPUTS/ARGUMENTS: - Model input rasters (tiff files)                        ##
##                   - Water mask files (tiff and shapefile)                   ##
##                   - Training and reference data set (tiff file)             ##
##                   - Locations of input raster files (optical, radar and     ##
##                     topographic input variable surfaces)                    ##
##                   - Locations of wet/dry data used for training and         ##
##                     reference                                               ##
##                   - Location of output folder for model outputs, etc.       ##
##                   - Location of a temp directory for use while modeling     ##
##                   - Location of ArcGIS python program files                 ##
##                                                                             ##
## NOTES/COMMENTS: The current code relies on the Create Random Points tool in ##
##                 ESRI's ArcGIS software program when creating training and   ##
##                 reference sample locations. A license for this software is  ##
##                 needed in order to run the code in its current form.        ##
##                                                                             ##
## LAST UPDATED: May 1, 2017                                                   ##
#################################################################################

# Load code libraries
library(raster)
library(rgdal)
library(ggplot2)
library(plyr)
library(dplyr)
library(caret)
library(snow)
library(rgeos)
library(RPyGeo)
library(dismo)
library(gbm)
library(ggthemes)

# Display time at beginning of code run
t1 <- Sys.time()
t1
```

```
## SET UP ENVIRONMENT FOR MODELING
##################################################################################

# Location of input rasters, wet/dry training and reference data set,
# output folder, and python location (this is used in generating training and
# reference sample locations using ESRI ArcGIS's Create Random Points tool)
location <- "C:/BRTmodeling/InputRasters"
location.pts <- "C:/BRTmodeling/InputPts"
outputs <- "C:/BRTmodeling//Outputs_TOSmodel"
py.loc <- "C:/Python27/ArcGIS10.3"
# set location of a temporary raster dump; this can take up to 100-300 GB of
# memory per model run, but can easily be deleted afterward using a file manager
rasterOptions(tmpdir = "C:/BRTmodeling/Rtemp", tmptime = 6)

# Model version, for naming outputs
OutNum <- "BRT_versionTOS"

# Number of iterations of subsampling
iter <- 50

# Create lists to hold names of input variables, for use later
lowland.tifs <- c("B2.tif", "B3.tif", "B4.tif", "B8.tif", "ndvi.tif", "ndwi.tif",
                  "pol.tif", "tpi.tif", "twi.tif", "vv.tif", "vvsd.tif")
lowland.colnames <- c("b2", "b3", "b4", "b8", "ndvi", "ndwi", "pol", "tpi", "twi",
                      "vv", "vvsd", "wetland")
lowland.bricknames <- c("b2", "b3", "b4", "b8", "ndvi", "ndwi", "pol", "tpi",
                        "twi", "vv", "vvsd")
model_1 <- "as.factor(wetland) ~ b2 + b3 + b4 + b8 + "
model_2 <- "ndvi + ndwi + pol + tpi + twi + vv + vvsd"
lowland.model <- paste(model_1, model_2, sep="")

# Function to generate random points; must provide location land areas
# shaefile (i.e., areas not masked by water mask), located in the points input
# folder, and the minimum distance to be used between individual random points
pts.gen2 <- function(directory, mindist){
       # Set up ArcGIS environment
       w <- directory
       env <- rpygeo.build.env(python.path = py.loc, overwrite=1,
                               workspace = directory)
       rpygeo.geoprocessor("CreateRandomPoints_management",
                           c("", "ptsLand1",
                           constraining_feature_class = "Land.shp",
                           constraining_extent = "",
                           number_of_points_or_field = 20000,
                           minimum_allowed_distance = mindist),
                           env = env, detect.required.extensions = T)
       pts <- readOGR(w, "ptsLand1")
       return(pts)
}

# Run pts.gen2 function to generate points
pts.model <- pts.gen2(location.pts, 3500)

## MODEL PROBABILITY OF WETLAND
##################################################################################

# Set working directory (to where input rasters stored)
setwd(location)
# Set 'fls' (short for 'files') variable to character vector of raster input names
fls <- lowland.tifs
```

```
# Create a RasterLayer object from training/reference data layer
wetland <- raster("wetland.tif")

# Use water mask input to create mask object to mask out water areas
waterMask <- raster("New_GoAwatermask.tif")
is <- c(1,0)
becomes <- c(NA,1)
reclass2 <- cbind(is, becomes)
landClass <- reclassify(waterMask, reclass2)

# Mask out the non-land (i.e., water) pixels
wetland <- mask(wetland, landClass)

# Create data frame for containing points and associated data for modeling
dat <- data.frame(row=1:length(pts.model))

# For each file in input raster list, mask out water, extract values from raster
# under points locations, and add those values to the data frame
for (i in 1:length(fls)){
       r <- raster(fls[i])
       r <- mask(r,landClass)
       ext <- extract(r,pts.model)
       dat <- cbind(dat,ext)
       print(paste0("done ", i, " of ", length(fls)))
}

# Extract reference value for each point from 'wetland.tif' - for training, and
# combine data from raster inputs and from training data into one data frame;
# then remove first column (extra column, not needed)
ext <- extract(wetland,pts.model)
dat <- cbind(dat,ext)
dat <- dat[,-1]

# Add column names to 'dat' data frame (input raster names), and remove cases
# containg 'NA' values (i.e., incomplete cases)
colnames(dat) <- lowland.colnames
dat <- na.omit(dat)

# Perform Spearman's correlation calculation between input variables,
# and print to screen
corrTable = cor(dat, method="spearman")
corrTable

# Bring in input variable rasters, mask by water mask, and extract
# min and max stats
r1 <- raster(fls[1])
r1 <- mask(r1, landClass)
min1 <- cellStats(r1, 'min')
max1 <- cellStats(r1, 'max')
r2 <- raster(fls[2])
r2 <- mask(r2, landClass)
min2 <- cellStats(r2, 'min')
max2 <- cellStats(r2, 'max')
r3 <- raster(fls[3])
r3 <- mask(r3, landClass)
min3 <- cellStats(r3, 'min')
max3 <- cellStats(r3, 'max')
```

```
r4 <- raster(fls[4])
r4 <- mask(r4, landClass)
min4 <- cellStats(r4, 'min')
max4 <- cellStats(r4, 'max')
r5 <- raster(fls[5])
r5 <- mask(r5, landClass)
min5 <- cellStats(r5, 'min')
max5 <- cellStats(r5, 'max')
r6 <- raster(fls[6])
r6 <- mask(r6, landClass)
min6 <- cellStats(r6, 'min')
max6 <- cellStats(r6, 'max')
r7 <- raster(fls[7])
r7 <- mask(r7, landClass)
min7 <- cellStats(r7, 'min')
max7 <- cellStats(r7, 'max')
r8 <- raster(fls[8])
r8 <- mask(r8, landClass)
min8 <- cellStats(r8, 'min')
max8 <- cellStats(r8, 'max')
r9 <- raster(fls[9])
r9 <- mask(r9, landClass)
min9 <- cellStats(r9, 'min')
max9 <- cellStats(r9, 'max')
r10 <- raster(fls[10])
r10 <- mask(r10, landClass)
min10 <- cellStats(r10, 'min')
max10 <- cellStats(r10, 'max')
r11 <- raster(fls[11])
r11 <- mask(r11, landClass)
min11 <- cellStats(r11, 'min')
max11 <- cellStats(r11, 'max')

# Create raster brick for modeling, and set names of rasters in brick
r.b <- brick(r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11)
names(r.b) <- lowland.bricknames

# Add per-raster min and max cell values to a data frame
minval <- rbind(min1, min2, min3, min4, min5, min6, min7, min8, min9,
                min10, min11)
maxval <- rbind(max1, max2, max3, max4, max5, max6, max7, max8, max9,
                max10, max11)
mm.df <- data.frame(lowland.bricknames, minval, maxval)


# Generate BRT model test (to look at output, number of trees, etc.)
# Use gbm.step() to assess the optimal number of boosting trees, using k-fold
# cross-validation
fit <- gbm.step(dat, 1:11, 12, family = "bernoulli", tree.complexity = 5,
                learning.rate = 0.005, bag.fraction = 0.5)

# Create data fram object from summary of 'fit' variable, reorder data by
# variable, subset data frame to extract second column, and plot gbm.step()
# results (e.g., deviance vs. number of trees, variable influence, and variable
# response curves)
df.importance <- data.frame(summary(fit))
df.importance <- arrange(df.importance, var)
df.importance <- df.importance[,2]
gbm.plot(fit)
```

```
# Create response curve dataframe from above model, for use in plotting later
response.df <- data.frame(dummy=c(1:1001))
for (n in lowland.bricknames){
  d <- plot.gbm(fit, i.var = n, return.grid=TRUE, type="response")
  get.min.max <- filter(mm.df, lowland.bricknames == n)
  mn <- get.min.max[,2]
  mx <- get.min.max[,3]
  xout <- seq(mn, mx, (mx-mn)/1000)
  d <- approx(d[,1], d[,2], xout = xout, rule=2)
  d <- as.data.frame(d)
  response.df <- cbind(response.df,d)
}

# Begin multi-core cluster; set up modeling function within cluster
beginCluster()
r.b.p <- clusterR(r.b, raster::predict,
                  args = list(model = fit, type = "response",
                              n.trees = fit$gbm.call$best.trees))
endCluster()

# Create variables to hold AUC and deviance measures, total and mean deviations
# per iteration, number of trees per iteration
AUC <- vector()
dev<- vector()
devTot <- vector()
devRes <- vector()
nTrees <- vector()

# Create data frame to hold variable relative influences (per variable), and
# set row names to bricknames, sorted asphabetically
varInfl <- data.frame(r=1:length(lowland.bricknames))
rownames(varInfl) <- sort(lowland.bricknames)

# Boosted Regression Tree modeling - multiple iterations
for (i in 1:iter){
  pts.model <- pts.gen2(location.pts, 3500)
  dat <- data.frame(row=1:length(pts.model))
  ext <- extract(r1, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r2, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r3, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r4, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r5, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r6, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r7, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r8, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r9, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r10, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(r11, pts.model)
  dat <- cbind(dat,ext)
  ext <- extract(wetland,pts.model)
  dat <- cbind(dat,ext)
  dat <- dat[,-1]
  colnames(dat) <- lowland.colnames
  dat <- na.omit(dat)
  fit <- gbm.step(dat, 1:11, 12, family = "bernoulli",
                  tree.complexity = 5, learning.rate = 0.01,
                  bag.fraction = 0.5)
  v.importance <- data.frame(summary(fit))
  v.importance <- arrange(v.importance, var)
  v.importance <- v.importance[,2]
  df.importance <- cbind(df.importance, v.importance)
  gbm.plot(fit)
  beginCluster()
  prediction <- clusterR(r.b, raster::predict,
                         args = list(model = fit, type = "response",
                                     n.trees = fit$gbm.call$best.trees))
  endCluster()
  r.b.p <- stack(r.b.p, prediction)

  # Generate response curve dataframe for model of current iteration
  for (n in lowland.bricknames){
    d <- plot.gbm(fit, i.var = n, return.grid=TRUE, type="response")
    get.min.max <- filter(mm.df, lowland.bricknames == n)
    mn <- get.min.max[,2]
    mx <- get.min.max[,3]
    xout <- seq(mn, mx, (mx-mn)/1000)
    d <- approx(d[,1], d[,2], xout = xout, rule=2)
    d <- as.data.frame(d)
    response.df <- cbind(response.df,d)
  }

  # Store model stats for current iteration in previously defined vectors
  AUC[i] <- fit$cv.statistics$discrimination.mean
  dev[i] <- (fit$self.statistics$mean.null - fit$self.statistics$mean.resid) /
    fit$self.statistics$mean.null
  devTot[i] <- fit$self.statistics$mean.null
  devRes[i] <- fit$self.statistics$mean.resid
  nTrees[i] <- fit$n.trees
  # Extract variable relative influences, sort by variable name, and add to
  # varInfl data frame
  relInfl <- fit$contributions
  relInfl <- relInfl[order(relInfl$var),]
  varInfl <- cbind(varInfl,relInfl$rel.inf)

  # Print message to console indicating which iteration has been completed
  print(paste0("done ", i, " of ", iter))
}


# Create data frame of variable importance information
importance <- rowMeans(df.importance)
imp.names <- c("b2", "b3", "b4", "b8", "ndvi", "ndwi", "pol", "tpi",
               "twi", "vv", "vvsd")
importance <- data.frame(imp.names, importance)

# Calculate mean and standard deviation of model iterations
wet <- calc(r.b.p, fun = mean)
wet.sd <- calc(r.b.p, fun = sd)
```

```
## CLASSIFY WETLAND AND CONDUCT ACCURACY ASSESSMENT
##################################################################################

# Calculate wetland classification image from modeled probability image using a
# threshold of 0.5
from <- c(0,0.50000000001)
to <- c(0.5, 1)
becomes <- c(0, 1)
wet.reclass <- cbind(from,to,becomes)
wet.class <- reclassify(wet, wet.reclass)

# Calculate overall classification accuracy by generateing a set of validation
# points
pts.validation1 <- pts.gen2(location.pts, 10)
pts.validation2 <- pts.gen2(location.pts, 10)
pts.validation <- rbind(pts.validation1, pts.validation2)
model <- extract(wet.class, pts.validation)
reference <- extract(wetland, pts.validation)
acc <- cbind(model, reference)
acc <- na.omit(acc)
correct <- abs(acc[,1] - acc[,2])
accuracy <- length(correct[correct==0])/length(correct)

# Below: create error matrix, in which the columns are
# 'model' values and the rows are the equivalent 'reference' values

# Calculate number of correctly classified 'wet' points
mWetTwet <- acc[,1]==1 & acc[,2]==1
noCorrWet <- length(mWetTwet[mWetTwet==TRUE])
# Calculate number of correctly classified 'dry' points
mDryTdry <- acc[,1]==0 & acc[,2]==0
noCorrDry <- length(mDryTdry[mDryTdry==TRUE])
# Calculate number of false positive 'wet' points
mWetTdry <- acc[,1]==1 & acc[,2]==0
noIncorrWet <- length(mWetTdry[mWetTdry==TRUE])
# Calculate number of false positive 'dry' points
mDryTwet <- acc[,1]==0 & acc[,2]==1
noIncorrDry <- length(mDryTwet[mDryTwet==TRUE])
# Calculate number of Model 'wet' and 'dry' points
noModWet <- length(model[model==1])
noModDry <- length(model[model==0])
# Calculate nubmer of Reference 'wet' and 'dry' points
noRefWet <- length(reference[reference==1])
noRefDry <- length(reference[reference==0])
# Calculate Producer's Accuracies
PAwet <- noCorrWet / noRefWet
PAdry <- noCorrDry / noRefDry
# Calculate User's Accuracies
UAwet <- noCorrWet / noModWet
UAdry <- noCorrDry / noModDry
# Calculate Overall Accuracy
totalCorrect <- noCorrWet + noCorrDry
totalSamples <- length(model)
overallAcc <- totalCorrect / totalSamples
# Build the error matrix table
emRow1 <- c(noCorrWet, noIncorrDry, noRefWet, PAwet)
emRow2 <- c(noIncorrWet, noCorrDry, noRefDry, PAdry)
emRow3 <- c(noModWet, noModDry, totalCorrect, 0)
emRow4 <- c(UAwet, UAdry, totalSamples, overallAcc)
```

```
errMatrix <- rbind(emRow1, emRow2, emRow3, emRow4)
emCols <- c("Wet (Predicted)", "Dry (Predicted", "Total (Predicted)",
            "Prod. Accuracy")
emRows <- c("Wet (Reference)", "Dry (Reference)", "Total (Reference)",
            "User's Accuracy")
colnames(errMatrix) <- emCols
rownames(errMatrix) <- emRows

# Below: calculate Kappa (k-hat) statistic

# Calculate products of matrix totals
prodR1 <- c((noRefWet*noModWet), (noRefWet*noModDry))
prodR2 <- c((noRefDry*noModWet), (noRefDry*noModDry))
prodMatrix <- rbind(prodR1, prodR2)
# Calculate sum of diagonal, and all products
sumDiag <- prodMatrix[[1,1]] + prodMatrix[[2,2]]
sumAll <- sum(prodMatrix)
expectAcc <- sumDiag / sumAll
# Calculate kappa
kappaAcc <- (overallAcc - expectAcc) / (1 - expectAcc)

# Smooth borders of transition between wet/dry, for purposes of display,
# and re-run water mask
wet.class <- focal(wet.class,w=matrix(1,9,9),fun=modal, na.rm=TRUE)
wet.class <- mask(wet.class, landClass)

# Set working directory to output folder and create output raster files of
# the mean wetland probabiliy surface, the wetland probability multi-iteration
# standard deviation, thresholded wetland classification, and various other
# model and classification statistics (including plots of variable relative
# importance)
setwd(outputs)
writeRaster(wet, paste0("WetProbs", OutNum, ".tif"), datatype= "FLT4S")
writeRaster(wet.sd, paste0("WetSD", OutNum, ".tif"), datatype= "FLT4S")
writeRaster(wet.class, paste0("WetClass", OutNum, ".tif"), datatype= "INT2S")
write.csv(accuracy, paste0("WetAcc", OutNum, ".csv"), row.names=FALSE)
write.csv(corrTable, paste0("WetCorrTable", OutNum, ".csv"), row.names=TRUE)
write.csv(errMatrix, paste0("WetErrMatrix", OutNum, ".csv"), row.names=TRUE)
write.csv(kappaAcc, paste0("WetKappa", OutNum, ".csv"), row.names=FALSE)
write.csv(dat, paste0("WetDF", OutNum, ".csv"), row.names=FALSE)
write.csv(importance, paste0("WetVarImportance", OutNum, ".csv"),
          row.names=FALSE)
png(paste0("WetVarImportance", OutNum, ".png"), width = 1000, height = 900)
ggplot(importance,aes(x=reorder(imp.names,-importance),
                      y=importance, fill=importance)) +
                      geom_bar(stat="identity") +
                           theme(axis.text = element_text(size=18),
                                 axis.title=element_text(size=18),
                                 legend.text=element_text(size=16),
                                 legend.title=element_text(size=16),
                                 legend.position="none") +
                           labs(x = "Variables") + labs(y = "Importance" )
dev.off()

# Generate mean and standar deviations for model variable response curves, from
# the multiple iterations
response.df <- response.df[,-1]
response.df.x <- response.df[,seq(1,length(response.df),2)]
response.df.x <- response.df.x[,1:length(lowland.bricknames)]
response.df.y <- response.df[,seq(2,length(response.df),2)]
```

```
for (i in 1:length(lowland.bricknames)){
  varname <- lowland.bricknames[i]
  collumns <- seq(i, length(response.df.y), length(lowland.bricknames))
  yvals <- response.df.y[,collumns]
  xvals <- response.df.x[,i]
  yvals.mean <- rowMeans(yvals)
  yvals.std <- apply(yvals,1,sd)
  yvals.neg.std <- yvals.mean - yvals.std
  yvals.add.std <- yvals.mean + yvals.std
  yvals.df <- cbind(xvals, yvals.mean, yvals.neg.std, yvals.add.std)
  yvals.df <- as.data.frame(yvals.df)
  if(i>3){
    xlim <- c(min(xvals), max(xvals))
  } else{
    xlim <- c(min(xvals), 1200)
  }
  g <- ggplot(yvals.df, aes(x=xvals, y=yvals.mean)) +
    theme_pander()+
    geom_ribbon(aes(ymin=yvals.neg.std, ymax=yvals.add.std), fill="gray80") +
    geom_line(colour="limegreen", size=1.6) +
    xlab(varname) + ylab("predicted probability") + xlim(xlim) +
    theme(axis.title.x = element_text(size=14),
          axis.title.y = element_text(size=12),axis.text = element_text(size=10))
  assign(paste0(varname,".plot"), g)
}

# Calculate AUC and deviance mean and standard deviation measures, and number of
# trees from multiple model iterations
AUC.mean <- mean(AUC)
dev.mean <- mean(dev) # percent explained deviance
AUC.sd <- sd(AUC)
dev.sd <- sd(dev)
devTot.mean <- mean(devTot) # total deviance
devTot.sd <- sd(devTot)
devRes.mean <- mean(devRes) # residual deviance
devRes.sd <- sd(devRes)
nTrees.mean <- mean(nTrees)
nTrees.sd <- sd(nTrees)

# Combine above into one vector for output
stats <- cbind(AUC.mean, AUC.sd, dev.mean, dev.sd, devTot.mean, devTot.sd,
               devRes.mean, devRes.sd, nTrees.mean, nTrees.sd)

# Calculate mean and standard deviation of variable influence, per variable and
# combine into single vector for output; also add names to rows
varInfl <- varInfl[,-1]
varInfl.mean <- vector()
varInfl.sd <- vector()
for (j in 1:length(lowland.bricknames)){
  varInfl.mean[j] <- mean(as.vector(as.matrix(varInfl[j,])))
  varInfl.sd[j] <- sd(as.vector(as.matrix(varInfl[j,])))
}
statsInfl <- cbind(varInfl.mean, varInfl.sd)
rownames(statsInfl) <- sort(lowland.bricknames)

# Output statistics to csv files
write.csv(stats, paste0("WetModelStats_AUCDev", OutNum, ".csv"),
          row.names=FALSE)
write.csv(statsInfl, paste0("WetModelStats_RelInfl", OutNum, ".csv"),
          row.names=TRUE)
```

```
# -----------------------------------------------------------------------
# The following code is taken from an existing code
# SOURCE URL for this function:
# http://stackoverflow.com/questions/24387376/r-weird-error-could-
# not-find-function-multiplot
#
# Multiple plot function
#
# ggplot objects can be passed in ..., or to plotlist (as a list of ggplot
# objects)
# - cols:   Number of columns in layout
# - layout: A matrix specifying the layout. If present, 'cols' is ignored.
#
# If the layout is something like matrix(c(1,2,3,3), nrow=2, byrow=TRUE),
# then plot 1 will go in the upper left, 2 will go in the upper right, and
# 3 will go all the way across the bottom.

# Multiplot function
multiplot <- function(..., plotlist=NULL, file, cols=1, layout=NULL) {
  require(grid)

  # Make a list from the ... arguments and plotlist
  plots <- c(list(...), plotlist)

  numPlots = length(plots)

  # If layout is NULL, then use 'cols' to determine layout
  if (is.null(layout)) {
    # Make the panel
    # ncol: Number of columns of plots
    # nrow: Number of rows needed, calculated from # of cols
    layout <- matrix(seq(1, cols * ceiling(numPlots/cols)),
                     ncol = cols, nrow = ceiling(numPlots/cols))
  }

  if (numPlots==1) {
    print(plots[[1]])

  } else {
    # Set up the page
    grid.newpage()
    pushViewport(viewport(layout = grid.layout(nrow(layout), ncol(layout))))

    # Make each plot, in the correct location
    for (i in 1:numPlots) {
      # Get the i,j matrix positions of the regions that contain this subplot
      matchidx <- as.data.frame(which(layout == i, arr.ind = TRUE))

      print(plots[[i]], vp = viewport(layout.pos.row = matchidx$row,
                                      layout.pos.col = matchidx$col))
    }
  }
}
# -----------------------------------------------------------------------
# Use Multiplot function to plot variable response curves into a png document
png(paste0("WetResponseCurves", OutNum, ".png"), width = 1300, height = 1466)
multiplot(b2.plot, b3.plot, b4.plot, b8.plot, ndvi.plot, ndwi.plot, pol.plot,
          tpi.plot, twi.plot, vv.plot, vvsd.plot, cols=3)
dev.off()

# Use Multiplot function to plot variable response curves into a pdf document
pdf(paste0("WetResponseCurves2", OutNum, ".pdf"), width =15, height = 17)
multiplot(b2.plot, b3.plot, b4.plot, b8.plot, ndvi.plot, ndwi.plot, pol.plot,
          tpi.plot, twi.plot, vv.plot, vvsd.plot, cols=3)
dev.off()

# Display time at end of code run, and time since t1 at beginning
t2 = Sys.time()
t2
t2-t1

#################################################################################
##                                  END OF CODE                                ##
#################################################################################
```