

Evaluating environmental predictors of breeding waterfowl population distribution and abundance in the Central Interior of British Columbia (2007-2017)

ABSTRACT

Conservation biology has seen an explosion of species distribution models (SDMs) in the recent published literature and growing numbers of governments and organizations responsible for small-scale regional to global conservation activities are actively utilizing them or have embarked on their own predictive studies (Franklin & Miller, 2009; Guisan, Wilfried, & Zimmermann, 2017). Waterfowl conservation planning in British Columbia (BC) however currently does not have the benefit of breeding waterfowl SDMs. To address this gap I have created SDMs using survey data from BC's annual Breeding Waterfowl Survey. Models utilizing a random forest-based approach were used to produce predictive maps for ten species and five species groups within the boundaries of the central interior representing core provincial breeding waterfowl habitat.

<insert brief discussion of general distribution trends>

The results of the models can be used to inform future modeling studies and guide conservation planning.

Keywords: waterfowl, machine learning, random forest, species distribution mode

INTRODUCTION

While the uses and practical applications of species distribution models in conservation decision-making are diverse (Guillera-Aroita et al., 2015; Guisan & Thuiller, 2005; Guisan et al., 2013) the overarching aim is to conserve biodiversity and support better land use planning and environmentally sustainable management practices. The breadth of practical applications include predicting climate change adaptation, invasive species management, critical habitat identification, reserve selection, impact assessment and ecological restoration (Elith & Graham, 2009; Franklin & Miller, 2009).

Recognizing the lack of waterfowl species distribution models (SDMs) to support conservation actions in response to the urgency of conservation challenges in Canada's boreal forests, Barker et al (2014) published the first national-scale waterfowl SDMs in 2014 based on the traditional Waterfowl Breeding Population and Habitat Survey (WBPHS) database. The WBPHS has been described as "arguably the largest and best-designed population survey in the world" (Murray, Anderson, & Steury, 2010), continental in scale and running on a continuous annual basis since 1955, the survey captures what is recognized to be "core waterfowl breeding habitat" in North America. Designed primarily to provide annual breeding population estimates and inform hunting regulations in the U.S. and Canada, these data additionally provide a long-term population monitoring dataset that has since informed countless studies on species-habitat relationships in support of waterfowl conservation. While the models performed well over all, the results of extrapolation to out-of-sample areas were variable for a number of reasons as outlined by the authors. British Columbia (BC) is not within the traditional survey and no survey data from BC informed the models which predicted low total densities for the Western Cordillera between the Pacific coast and Rocky Mountains—a result that conflicts with anecdotal expert opinion (A. Breault, personal communication 2018). As such there are currently no adopted waterfowl breeding population distribution models in use for the province. This study aims to fill this gap and help support the mandate of the Canadian Wildlife Service (CWS) to conserve biodiversity (Environment and Climate Change Canada, 2019; Environment Canada - Biodiversity Convention Office, 1995).

In the early 2000s after exploratory pilot surveys in BC's central and sub-boreal highlands determined waterfowl population abundances to be significant enough to justify a regional breeding survey program, the British Columbia Breeding Waterfowl Survey began in earnest in 2006. BC's May surveys, run jointly by the CWS and the U.S. Fish and Wildlife Service and conducted in partnership with Ducks Unlimited Canada, inform the annual population status of migratory game birds in the Central Interior Plateau of BC and contributes to adaptive harvest strategies for mallards in the Pacific Flyway (Zimpfer, Breault, & Sanders, 2019).

After consolidating and standardizing the annual survey data, I developed methods to create SDMs for the top ten most abundant species as well as five groups based on nesting and feeding guilds (Baldassarre, 2014) (Table #). Guisan et al (2013) have observed that despite the oft-cited assumption of the utility of SDMs in conservation decision-making there is little evidence demonstrating their application in real-world conservation management. Following their recommendation that modellers make explicit the objectives of the framework within which models were developed, I provide guidance on how the models may be improved and implemented as a decision-making platform for conservation planning by the Canadian Wildlife Service and partners in conservation.

Table 1. Species included in the study and their nesting and feeding guilds based on Baldassarre (2014) and Cornell (2015). Species-specific models for only the top ten most common species were created however less common species were accounted for in the groups outlined in Table 2.

Species Code	Common Name	Scientific Name	Species Group	Foraging Guild	Nesting Guild
AMWI	American Wigeon*	<i>Mareca americana</i>	Dabblers	Dabbler - Plants	Ground
BAGO	Barrow's Goldeneye*	<i>Bucephala islandica</i>	Divers	Surface - Dive - Insects	Cavity
BUFF	Bufflehead*	<i>Bucephala albeola</i>	Divers	Aerial Dive - Insects	Floating
BWTE	Blue-winged Teal*	<i>Spatula discors</i>	Dabblers	Dabbler - Seeds	Ground
CAGO	Canada Goose*	<i>Branta canadensis</i>	Geese	Ground Forager - Seeds	Ground
CANV	Canvasback	<i>Aythya valisineria</i>	Divers	Surface - Dive - Plants	Floating
CITE	Cinnamon Teal	<i>Spatula cyanoptera</i>	Dabblers	Dabbler - Seeds	Ground
COGO	Common Goldeneye	<i>Bucephala clangula</i>	Divers	Surface - Dive - Insects	Cavity
COME	Common Merganser	<i>Mergus merganser</i>	Mergansers	Surface - Dive - Fish	Cavity
GADW	Gadwall	<i>Mareca strepera</i>	Dabblers	Dabbler - Plants	Ground
GWTE	Green-winged Teal*	<i>Anas crecca</i>	Dabblers	Dabbler - Seeds	Ground
HADU	Harlequin Duck	<i>Histrionicus histrionicus</i>	Divers	Surface - Dive - Insects	Ground
HOME	Hooded Merganser	<i>Lophodytes cucullatus</i>	Mergansers	Surface - Dive - Fish	Cavity
LTDU	Long-tailed Duck	<i>Clangula hyemalis</i>	Divers	Surface - Dive - Insects	Ground
MALL	Mallard*	<i>Anas platyrhynchos</i>	Dabblers	Dabbler - Seeds	Ground
NOPI	Northern Pintail	<i>Anas acuta</i>	Dabblers	Dabbler - Seeds	Ground
NSHO	Northern Shoveler*	<i>Spatula clypeata</i>	Dabblers	Dabbler - Plants	Ground
RBME	Red-breasted Merganser	<i>Mergus serrator</i>	Mergansers	Surface - Dive - Fish	Ground
REDH	Redhead	<i>Aythya americana</i>	Divers	Surface - Dive - Plants	Floating
RNDU	Ring-necked Duck*	<i>Aythya collaris</i>	Divers	Surface - Dive - Plants	Floating
RUDU	Ruddy Duck	<i>Oxyura jamaicensis</i>	Divers	Surface - Dive - Insects	Ground
SCAU**	Lesser Scaup*	<i>Aythya affinis</i>	Divers	Surface - Dive - Insects	Ground
SNGO	Snow Goose	<i>Anser caerulescens</i>	Geese	Ground Forager - Plants	Ground
TRUS	Trumpeter Swan	<i>Cygnus buccinator</i>	Swans	Dabbler - Plants	Ground
WFGO	White-fronted Goose	<i>Anser albifrons</i>	Geese	Dabbler - Plants	Ground
WODU	Wood Duck	<i>Aix sponsa</i>	Dabblers	Dabbler - Plants	Ground

* Species within the top ten most commonly observed for which a species-specific model was created.

** The waterfowl population survey does not distinguish scaup species however only lesser scaup are observed in the area.

Table 2. Species groups for which group-specific species distribution models were built (refer to Table 1 for specific species included within the groups).

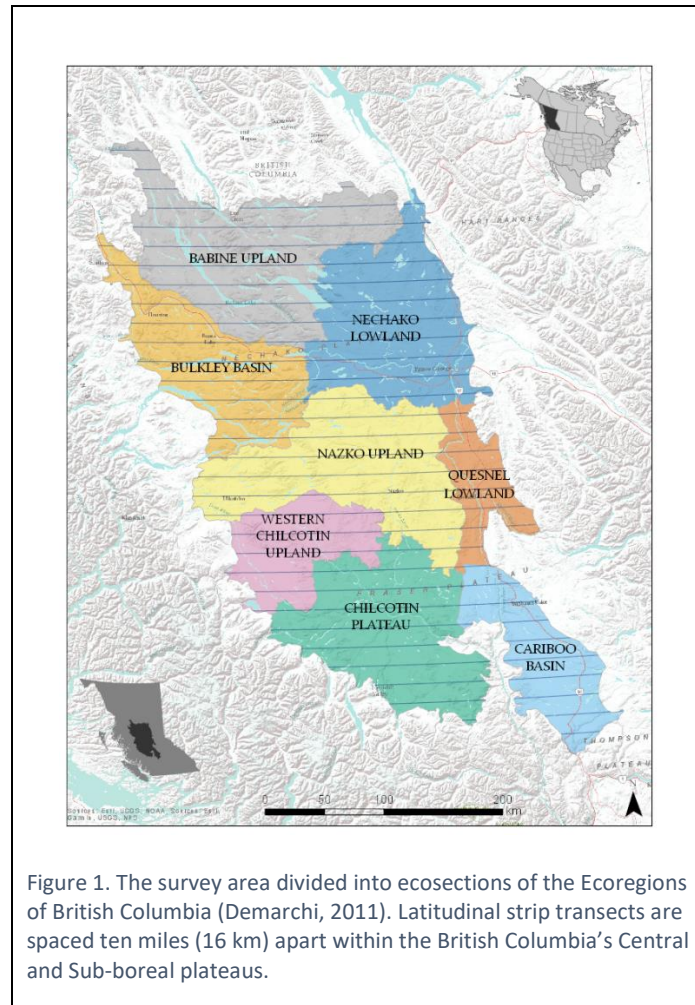
Abbreviation	Explanation
sp_div	Count of total indicated breeding population of distinct species
sp_tot	Count of total indicated breeding population of all waterfowl
dabblers	Count of total indicated breeding population of dabbling ducks
divers	Count of total indicated breeding population of diving ducks
cavity	Count of total indicated breeding population of cavity nesting species

MATERIALS AND METHODS

Survey Methods

The study area was designed to capture BC's prime waterfowl breeding habitat of the humid, continental plateaus of the central and sub-boreal Interior between the Coast mountains to the west and Rocky Mountains to the east. The region of interlocking highlands and valleys are mainly forested and sparsely populated. The main industries include forestry, agriculture within lowland valleys, and mining (Demarchi, 2011).

The survey design consists of latitudinal strip transects 400 metres wide, spaced ten miles apart (16.09 km) and delineated by the boundaries of eight ecosections which represent the finest scale, sub-regional stratum within the Ecoregion Classification System of British Columbia—a small-scale ecosystem management framework stratifying regions of “similar climate, physiography, oceanography, hydrology, vegetation and wildlife potential” (Demarchi, 2011) (Figure 1).



The study area falls within two main ecoprovinces with the majority contained within the Central Interior and the northern and northeastern-most tip stretching into the Sub-boreal. Six of the ecosections are within the Central Interior Plateau—Cariboo Basin, Chilcotin Plateau, Western Chilcotin Upland, Quesnel Lowland, Nazko Upland and Bulkley Basin. The flat or rolling hills of the ecoprovince contain many depressions through which meandering streams and depressions create an abundance of wetland habitat (Demarchi, 2011). The area supports 65% of all species known to occur and 61% of all bird species known to breed in the province with the greatest total abundances of waterfowl occurring in the Cariboo Basin ecosection within the Riske Creek area (Demarchi, 2011; Savard, Sean Boyd, & John Smith, 1994). The Nechako Lowland and Babine Upland ecosections form the southernmost ecosections of the Sub-boreal Interior Ecoprovince which supports 57% of all bird species known to occur and 45% known to breed in the province (Demarchi, 2011).

The surveys are conducted by helicopter and surveyed by a consistent survey crew of two to three experienced observers (Zimpfer et al., 2019). The transects when originally conceived were designed within the NAD 1983 geographic coordinate system and were based on the mapped Ecoregion Classification System version 2.0 circa 1995. The annual surveys continue to follow the extent of the original design however the ecosection boundaries have since been updated with finer scale vegetation zonation data in version 2.1 (2006). All analyses were based on the areal overlap between the two versions.

The survey does not follow the design methodology of the traditional WBPHS, however the survey techniques are consistent with the methods outlined in Smith (1995) and the Standard Operating Procedures (US Fish and Wildlife Service (USFWS) and Canadian Wildlife Service (CWS), 1987). Prior to 2010, the survey technologies employed consisted of paper maps, field notes and GPS units with navigation determined by piloting along bearings to GPS waypoints. In 2010, the mobile GIS software PC-Mapper with Airborne Inspection (version 4.0,

Corvallis Microtechnology Inc, 2015) was adopted for both survey navigation and data collection. The software is run on Panasonic Toughbooks (CF-19 and CF-31) with the screen in view of both the pilot and observer. Real-time navigation is guided by the GIS with reference base data containing strip transect boundaries, freshwater polygons and stream segments, ecosection boundaries and fuel waypoints. Digital data collection via georeferenced voice recordings transcribed by the observer post-survey collection have replaced paper analogue methods.

The survey is designed to capture the primary breeding period beginning in early May but is weather and climate dependent and has taken place as early as late April (April 28, 2015) and as late as mid-May (May 13, 2011). Heavy snowpack and/or cold spring temperatures can delay accessibility to open water and wetlands. The study area extent is almost 11 million hectares and takes an average 23 days (~105 flying hours) to survey.

Population estimates

The surveys are conducted by helicopter along meandering paths at altitudes and speeds lower than fixed-wing aircraft (30-50 m and 40-80 km/h, respectively) therefore no complementary ground surveys are conducted and no visibility correction factor is applied as visibility is assumed to be complete. The total indicated breeding population estimates were derived from raw counts based on species, sex and grouping as outlined in Smith (1995). The temporal subset of the analysis was from 2007 to 2017 inclusive (data from 2006 was excluded due to data inconsistencies and 2018 was excluded due to the absence of interpreted climate data). Survey observation points record the location of the observer within the helicopter and not the bird on the ground. Efforts to correct locations guided by reference data and field observation notes were discontinued from 2012 onward due to resource constraints and inconsistent observational record-keeping. In order to account for this spatial uncertainty, observation location points were collated to the nearest 400m interval along the center line of the strip transect.

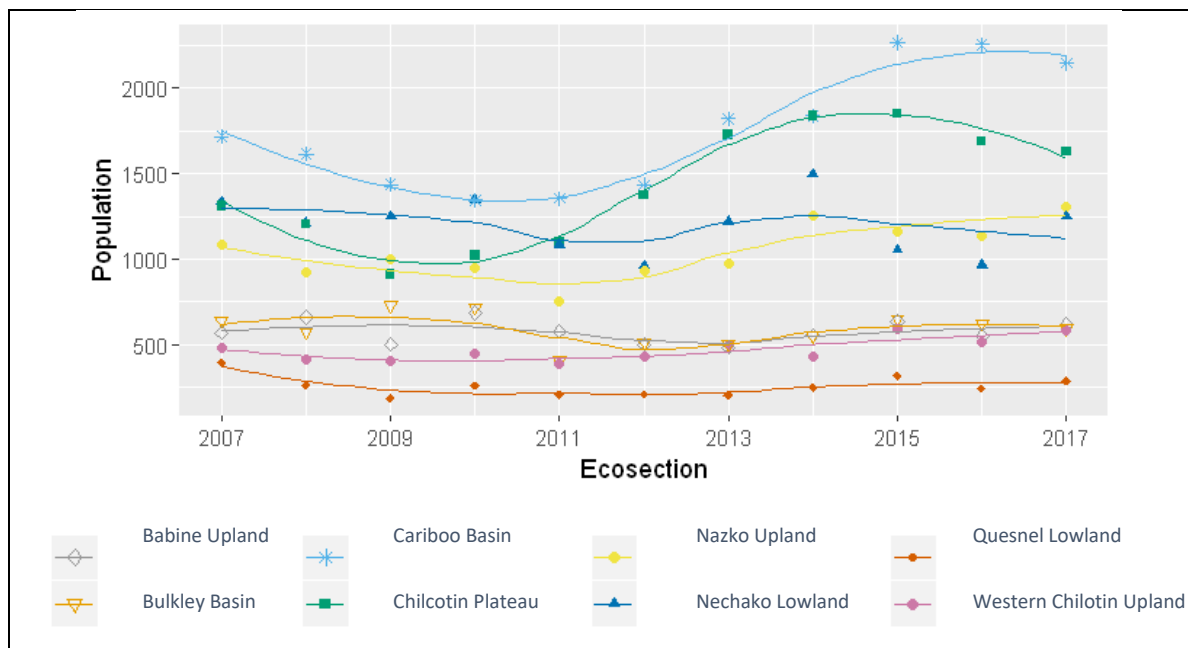


Figure 2. Total indicated breeding population of the top ten most common species observed within the transect (2007-2017).

The Cariboo Basin is by far the most populous ecosection with the greatest total count (Figure 1) and population density (Figure 2) followed by the Chilcotin Plateau, Nechako Lowland and Nazko Upland. The remaining ecosections—Bulkley Basin, Quesnel Lowland, Western Chilcotin Upland and Babine Upland—have similar population densities and share similarly stable year-to-year trends while the densities in the two most populous ecosections, Cariboo Basin and Chilcotin Plateau, display concordant fluctuations and appear to be increasing.

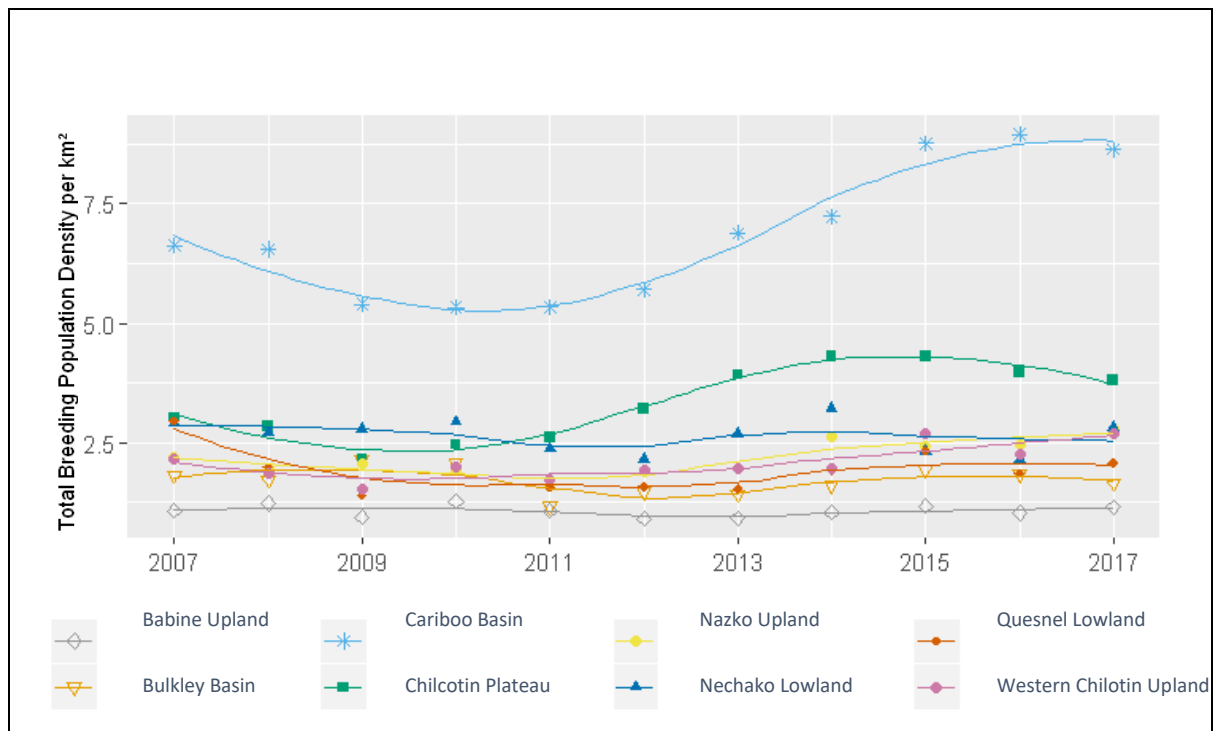


Figure 3. Total annual indicated breeding population density per square kilometre of the top ten most common species by ecosection observed within the transect.

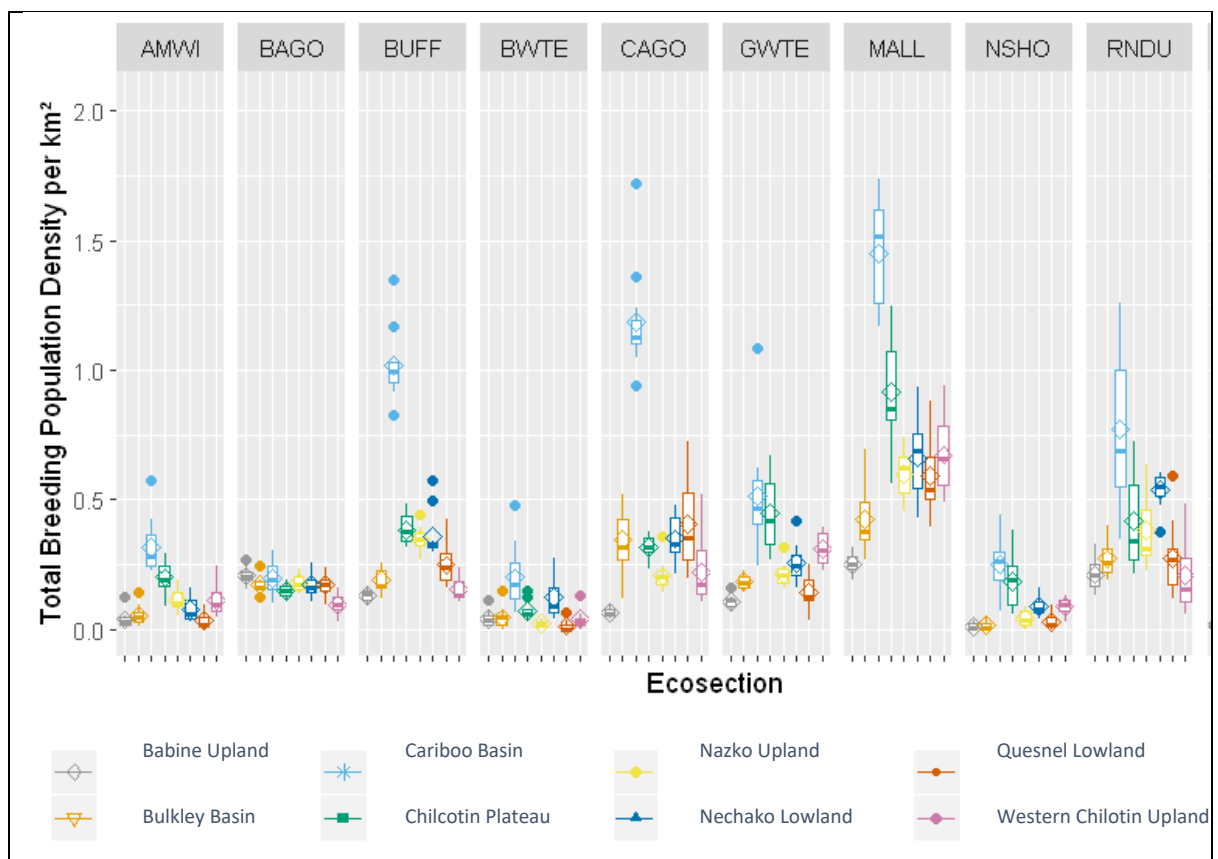


Figure 3. Boxplots of total annual indicated breeding population density per square kilometre of the top ten most common species by ecosection observed within the transect.

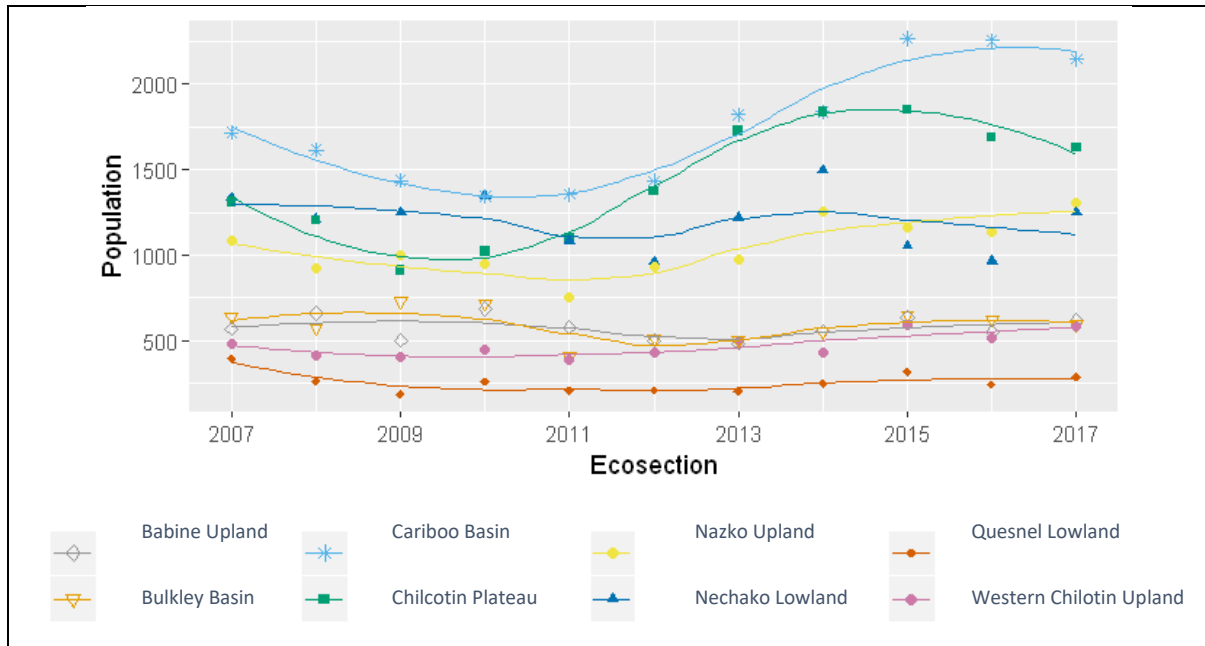
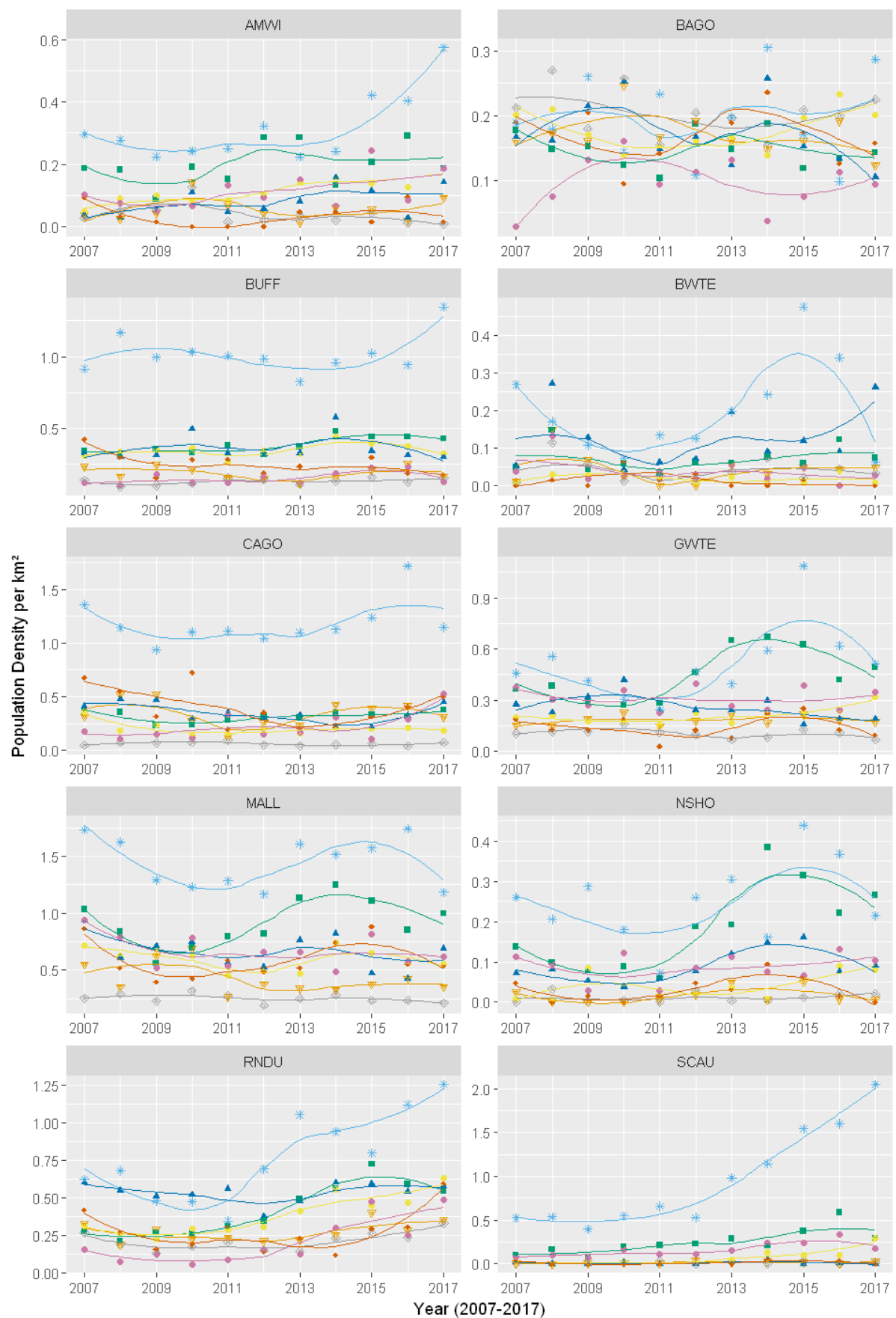


Figure 4. Total indicated breeding population of the top ten most common species observed within the transect (2007-2017).

Of the top ten most commonly observed species, the species demonstrating the greatest year-to-year variability in abundance include Lesser Scaup* (LESC), Ring-necked Duck (RNDU), Canada Goose (CAGO), and Mallard (MALL). The distributions suggest it is LESC and RNDU species driving the increasing population trends in the Cariboo Basin and the Chilcotin Plateau. Mallard (MALL), Canada Goose (CAGO), Green-Winged Teal (GWTE), Blue-Winged Teal (BWTE), and Northern Shoveler (NOSH) reflect concordant annual fluctuations—decreasing from 2007 to 2010, increasing between 2010 and 2015, and decreasing again to 2017. American Wigeon (AMWI) to a lesser degree follows the main trend but appears to be slightly increasing in the Cariboo Basin. Barrow's Goldeneye (BAGO) distributions remained steady throughout their distributions in all

ecosections during the study period.



Environmental data

Environmental predictor variables were pre-selected based on ecological theory of life history traits and physiological processes, hydrological features associated with wetlands, and land use practices and disturbances that can potentially influence waterfowl habitat. Where possible direct measures of predictor variables rather than proxy data was selected. For example, measures of temperature and precipitation rather than the indirect measure of elevation were included in the analyses. A number of datasets were collated and evaluated but the final data inputs were constrained by availability, resolution, coverage, and currency, and complicated by interpretation and relevance (Appendix 1).

Annual, seasonal, monthly and 30-year normal climatic variables were extracted from ClimateBC software (version 6.10, Wang, Hamann, Spittlehouse, & Carroll, 2016) based on the coordinates and elevation at 400m interval points along the transect center line. Growing degree days and average April temperature were included to capture primary productivity. To represent and characterize hydrological regimes which are driven mainly by snowpack accumulation in winter within the study area (Demarchi, 2011; Islam et al., 2017; R. Pike, Redding, Moore, Winkler, & Bladon, 2010) precipitation as snow and the climatic moisture deficit (precipitation minus potential evapotranspiration) were selected. The Biogeoclimatic Zones of British Columbia (BEC) is a standardized provincial dataset that classifies ecosystems within nested classes of regional, site and chronological levels of apex vegetation (Ministry of Forests, Lands, Natural Resource Operations and Rural Development (FLNRO), 2018) and was included to represent the combined influence of soil chemistry, vegetation, topography and climate. Additionally, the BEC Zone classifications provide an alternative regional classification system to the Ecoregions of BC ecoregions for supplemental model development. Moreover, predicted changes to BEC Zone boundaries due to predicted climate change scenarios can be extracted from ClimateBC for future climate impact studies.

Land cover variables to characterize habitat were derived from 2010 Landsat imagery published by the North American Land Change Monitoring System (CCRS/CCMEO/NRCAN, 2017). Cover classes were aggregated and reclassified to account for imbalanced classes (Appendix #). Topographic data included slope and aspect derived from 1:20,000 DEM (FLNRO, 2014).

Hydrological variables were derived from the provincial reference dataset for standardized hydrological features, and included polygons of lakes, rivers, man-made waterbodies and wetlands as well as stream lines (FLNRO, 2011). Lakes were classified by size class and streams were aggregated by stream order values (Appendix #). To account for shoreline complexity and avoid correlation between lake area and perimeter, a lake perimeter:area index was derived.

The line network of unpaved roads was included to represent anthropogenic disturbance of resource extraction activities (Ministry of Forests, Lands, Natural Resource Operations and Rural Development, 2013).

Land management practices were represented by the Agricultural Land Reserve (Agricultural Land Commission, 2018)—a provincial designation designed to identify and conserve agricultural productive lands, and by the Protected Areas Database of lands under federal, provincial and municipal protection and land conservancy trusts (CWS, 2018).

Geoprocessing Methods

Spatial analyses were performed and mapping products were produced in ArcGIS Pro (versions 2.2 to 2.4, ESRI, 2019) and Python (version 3.6.5, Python Software Foundation, 2018). Predictor variables were extracted, projected, rasterized and generalized as required in BC Albers equal area coordinate system within a 1 km buffer of the study area to eliminate edge effects. The location uncertainty of point observations determined the finest scale of the analysis was a resolution of 400m. Topographic values of slope and aspect were generalized to the mean average within 16 ha (400m x 400m) cells determined by the boundaries of the strip transect and 400m interval breaks. BEC zone classifications were generalized by majority area within the cell and remained categorical. All remaining predictors were represented by continuous values and generalized to

a 1.2 km radius of the interval centroid to capture landscape level effects (Figure #). Predicted response values were projected to a fishnet grid of 400m cell centroid points with attributed predictor values and rasterized for output and display. Many of the steps required in the pre-processing of spatial data and map production were automated in a Python Script Toolbox (Appendix #).

The line network of unpaved roads was included to represent anthropogenic disturbance of resource extraction activities (Ministry of Forests, Lands, Natural Resource Operations and Rural Development, 2013).

Land management practices were represented by the Agricultural Land Reserve (Agricultural Land Commission, 2018)—a provincial designation designed to identify and conserve agricultural productive lands, and by the Protected Areas Database of lands under federal, provincial and municipal protection and land conservancy trusts (CWS, 2018).

Geoprocessing Methods

Spatial analyses were performed and mapping products were produced in ArcGIS Pro (versions 2.2 to 2.4, ESRI, 2019) and Python (version 3.6.5, Python Software Foundation, 2018). Predictor variables were extracted, projected, rasterized and generalized as required in BC Albers equal area coordinate system within a 1 km buffer of the study area to eliminate edge effects. The location uncertainty of point observations determined the finest scale of the analysis was a resolution of 400m. Topographic values of slope and aspect were generalized to the mean average within 16 ha (400m x 400m) cells determined by the boundaries of the strip transect and 400m interval breaks. BEC zone classifications were generalized by majority area within the cell and remained categorical. All remaining predictors were represented by continuous values and generalized to a 1.2 km radius of the interval centroid to capture landscape level effects (Figure #). Predicted response values were projected to a fishnet grid of 400m cell centroid points with attributed predictor values and rasterized for output and display. Many of the steps required in the pre-processing of spatial data and map production were automated in a Python Script Toolbox (Appendix #).

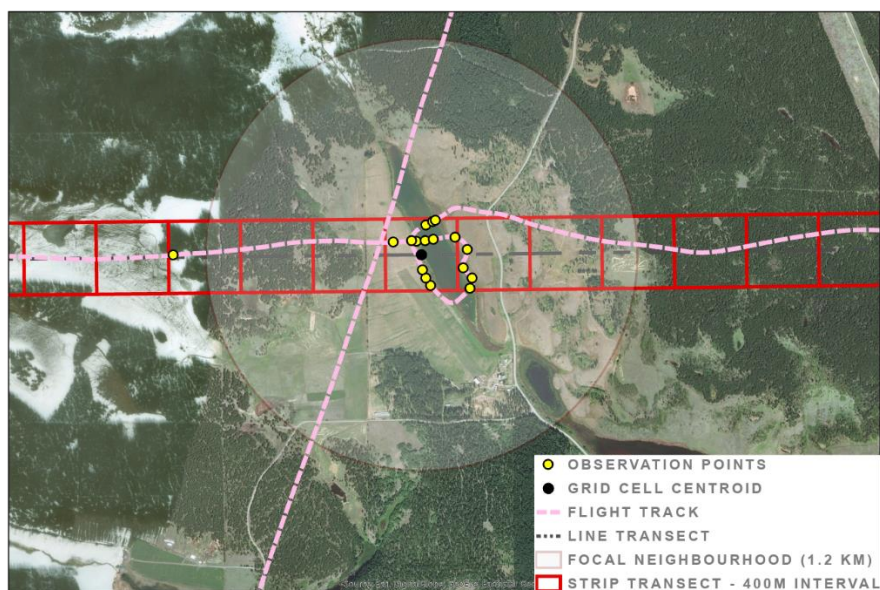
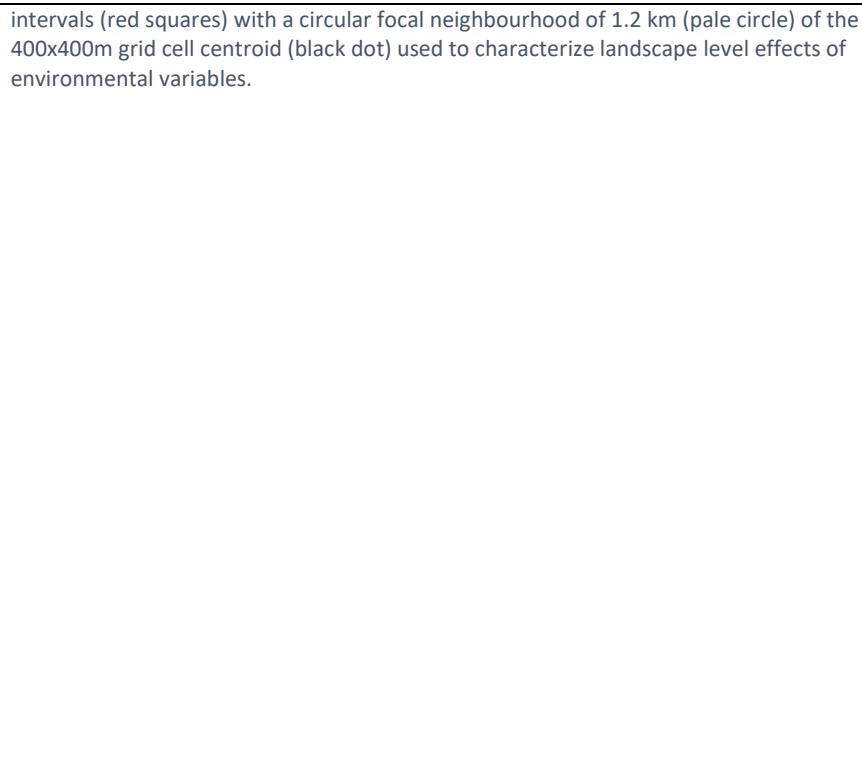


Figure 5. Representative schematic of the survey methods incorporating survey data from 2017. Observation points (yellow dots) are collected along the meandering flight tracklog (dashed pink line) within the transect. The strip transect was spliced into 400m wide



intervals (red squares) with a circular focal neighbourhood of 1.2 km (pale circle) of the 400x400m grid cell centroid (black dot) used to characterize landscape level effects of environmental variables.

Statistical Methods

Random forest is an ensemble machine learning algorithm that creates a series of decision trees based on the principles of bagging (or bootstrapped aggregation—the drawing of a large number of data samples by random sampling with replacement) and random permutation. Each individual decision tree is based on a random subset of the data and at each node of the tree the data is partitioned into two branches based on a randomly selected but predetermined number of predictor variables evaluated by the algorithm to produce the best split. As individual trees are highly susceptible to noise in the data and sensitive to local optima, each tree is considered a ‘weak learner’ (Guisan et al., 2017). But by combining and averaging the results of several regression trees, a ‘strong learner’ is created in the final ensembled prediction.

The random forest approach for building the SDMs was chosen for its predictive accuracy, resistance to overfitting, ability to account for imbalanced classes, ability to handle both continuous and categorical variables, and its independence from requirements of feature scaling and centering as well as assumptions of normality (Cutler, Cutler, & Stevens, 2012; Guisan et al., 2017). The observation frequency distribution of bird counts reflected a zero-inflated negative binomial distribution typical of ecological count data (Qian, 2010) but not amenable to standard regression-based approaches.

All statistical modeling and data manipulation was performed in R (version 3.5.3, R Core Team, 2018). The R ‘party’ package (version 1.3-3) was selected for its implementation of the random forest and bagging algorithm ‘cforest_unbiased’ function which utilizes conditional inference trees that account for correlation structures between variables in the permutation scheme of variable selection; additionally, the function implements a permutation importance measure, function ‘varimp’, that is immune to erroneous calculations due to correlated responses and is not biased towards continuous data or categorical variables with many classifications unlike the Gini accuracy of traditional random forest implementations (Strobl, Boulesteix, Kneib, Augustin, & Zeileis, 2008; Strobl, Hothorn, & Zeileis, 2009). Separate models for each species or species group within each ecosection were developed due to regional differences in predictor variables and for computational efficiency. Slight differences in selected predictor variables between ecosection models were due to zero or near-zero variance, correlated data structures and/or mismatched factor variables in ecosection-based data inputs for model training and prediction.

Preliminary results of model inputs of annual survey data indicated low variable importance rankings for the dynamic climate variables. As all other environmental data was static the model input values were based on mean averaged counts and climate measures. Due to high zero-inflation (average of 93.6% frequency of zeroes for the top ten most common species) the data was not subset into training, validation, and test sets to evaluate model performance, instead the internal out-of-bag (OOB) error measures were determined. As each tree is based on a random subset of the training data, a collection of datasets which do not contain a particular record can be ensembled for each record. This collection forms the out-of-bag examples which are used as an unbiased test set to assess prediction accuracy by averaging the error rate. The predetermined number of candidate variables for each node split ('mtry' parameter of 'cforest') was left at the default value of 5 which resulted in better OOB estimates than 8 which was tested using Briemann's rule of thumb to divide the number of candidate variables by three for regression trees (Guisan et al., 2017).

The conditional inference trees utilized by 'party' package are insensitive to highly correlated data structures but in order to reduce processing time, data preparation included addressing correlation and multicollinearity by step-wise elimination of correlated variables with threshold measures of $r = 0.8$ followed by elimination of variance inflation factors of 5 and greater corresponding to general thresholds recommended by Guisan et al (2017). The 'varimp' function to derive variable importance can be parameterized to produce unbiased importance measures of highly correlated data however preliminary trial results in Cariboo Basin reflected exaggerated processing times that restricted the application in this study. Neither correlation nor multicollinearity in predictor variables reduces predictive accuracy of random forests however the candidate predictors removed in the preprocessing step for the generation of 'varimp' values were not re-incorporated in the final prediction models. It is recommended these be included in future modeling exercises. The predictive models were re-run setting different seed values in order to confirm the stability of importance measures.

Random forest methods are widely recognized to be fast and able to handle large amounts of data and model variables but the unbiased algorithm of the 'cforest' implementation is more computationally intensive than standard approaches. For example model training took approximately 10 minutes to process ~3,000 records and 27 variable inputs for mallards in Babine Upland, while model forecasting to the ecosection, an area ~40 times greater, took over 18 hours on a 64-bit OS workstation with 48.0 GB RAM and an Intel Xeon(R) CPU 3.60GHz.

The R scripts developed for statistical analyses as well as links to a Github repository of latest project updates as well as an interactive Jupyter Notebook documenting major processing steps are provided in Appendix #

RESULTS AND DISCUSSION

Species Abundance and Distribution

<insert discussion of predicted distributions and abundance of mapped outputs>

Model Performance

The training models and predicted outputs are representations of the complex ecological interactions and associations between the predictor variables and the species or species group under study. Predictive modelling algorithms like random forest forsake the theoretical hypotheses of explanatory causal models for accurate forecasting (Shmueli, 2011). The variable importance ('varimp') measures derived from recursive partitioning methods are unlike regression coefficients in that they do not provide a linear measure of the relationship between the predictor and response variables, rather the reported measure reflects the drop in prediction accuracy of the model by random permutation of the variable (Guisan et al., 2017; Strobl et al., 2008). Their most unambiguous application is in variable selection for model building. In summary, 'varimp' values are not by themselves interpretive however they may reveal causal mechanisms to inform the development of explanatory models (Shmueli, 2011) and estimated values are provided to aid in variable selection for future

studies (Table #). Preliminary explorations of zero-inflated generalized mixed models suggest they are a promising approach for explanatory insight.

Model performance was assessed with the out-of-bag (OOB) error.

<insert discussion of rsquared, RMSE and MAE>

Model Limitations

We know that “[d]ucks like water” (Pimm, 1994), but wetlands have yet to be mapped: Canada, unlike most industrialized nations, does not have a national wetland dataset (Canadian Wetland Inventory — Ducks Unlimited Canada, 2019). The delineation of highly productive wetlands, including ephemeral wetlands due to increased nutrient mineralization by aerobic microbes (Schlesinger & Bernhardt, 2013) and smaller wetlands due to increased light penetration at shallower depths and increased shoreline vegetation, are difficult to capture with freely available, coarser-scale remote sensing imagery (e.g. Landsat imagery at 30m resolution). Techniques for finer-scale wetland feature extraction based on Landsat have been developed for open wetlands but have not yet been fully developed and tested on forested wetlands (Halabisky et al., 2018).

Relative to other provinces, BC has an abundance of fine-scale (1:20,000) base reference spatial data however the creation of much of these data has been driven by the forestry sector which is a significant component of the BC economy. Consequently, much of the mapping of environmental features has been focused on supporting forestry management practices and the delineation of wetlands (areas unsuited to logging) especially at higher elevations is poor (D. Filatow, personal communication April 28, 2018). Ongoing efforts to develop methodologies for wetland mapping coincident with the May surveys utilizing Radarsat-2 technologies in a collaborative effort between the CWS and the Science and Technology branches of Environment and Climate Change Canada are promising but results are still preliminary and the project is far from complete (K. Moore, personal communication, September 30, 2018). Additionally, the Canadian Wetland Inventory project spearheaded by Ducks Unlimited Canada aims to provide a comprehensive national wetland inventory but BC has yet to be mapped (Canadian Wetland Inventory — Ducks Unlimited Canada, 2019). Model performance and predictive accuracy can be expected to improve with the incorporation of improved wetland cover products, but proactive management measures should not be delayed while waiting for these data.

Hydrological regimes that create and maintain wetland ecosystems are influenced by a host of factors such as the depth of snowpack, the rate of snowmelt, levels of groundwater, precipitation, glacial retreat, land management practices, and anthropogenic and natural disturbance (Adamus, 2014; R. G. Pike et al., 2010). Future studies may be improved by incorporating snow basin, drainage and/or watershed data to better model hydrology. BC's forested ecosystems have been challenged by the mountain pine beetle outbreak from 1999 to 2015 followed by the extreme fire seasons of 2017 and 2018. The impacts of these regional disturbances on forest hydrology are difficult to predict (R. G. Pike et al., 2010; R. Pike et al., 2010) but are worthy of monitoring and future investigation (Bunnell, Fred L, Wells, Ralph, 2010; MacKenzie & Moran, 2004; Snauffer, Hsieh, & Cannon, 2016).

The population estimates reflect sampled observations over a limited period of time and therefore reflect only a snapshot of the species-habitat relationship assumed to be in pseudo-equilibrium (Guisan et al., 2017). Moreover, the models did not account for density-dependence, breeding philopatry, site fidelity, territoriality, influence of breeding phenology, species nesting chronologies or lagged response to environmental shifts. A recent study on the breeding phenology of cavity-nesting birds identified impacts on nesting activities with critical temperature periods of local temperature as short as 4 days (Drake & Martin, 2018). Daily temperature datasets from Natural Resources Canada at 1 km resolution (McKenney et al., 2011) were collated but not included in the generation of the models. Future efforts are encouraged to explore the incorporation of these and other datasets as identified in Appendix #.

Model Applications

This modeling study had two main, related objectives: first, to develop the methods and techniques required to standardize and consolidate the BC May Surveys for analysis and model generation and secondly, to create SDMs of relative abundance and distribution in support of breeding waterfowl conservation and habitat management. The results of the models can be used to highlight hotspot areas of high abundance and diversity to evaluate and assess protection or mitigation measures, provide finer-scale population estimates for environmental impact assessments and improve the calculation of offsets, and help guide adaptive management measures to prepare for climate change. The most immediate application of the workflows developed to generate the SDMs is to use them as a basis for further studies of ecological relationships, and to serve as guidance for future i address the models' limitations.

Changes in BC's hydrological regimes due to increased warming and drying trends are predicted to lead to wetland losses at low to mid-elevations, however, as higher elevations are less sensitive to temperature changes that can affect snowpack accumulation, BC's higher elevation wetlands may buffer the impacts for waterfowl and other wetland species (Bunnell, Fred L, Wells, Ralph, 2010; R. G. Pike et al., 2010; R. Pike et al., 2010). BC's central interior waterfowl breeding habitat is nowhere near as productive as the Prairie Pothole Region (PPR), however the conservation value of our wetlands, especially at elevations greater than 1200 metres (Bunnell, Fred L, Wells, Ralph, 2010), may rise in rank with the present and predicted climate-related shifts in wetland productivity of the PPR (N. D. Niemuth, Fleming, & Reynolds, 2014; N. Niemuth, Wangler, & Reynolds, 2010; Zhao, Silverman, Fleming, & Boomer, 2016). Future studies of climate impacts should take into account the wide ranges in variability of projected climate between the global models available in ClimateBC and employ ensemble methods that encompass this range of variability for impact assessments and adaptive management (R. G. Pike et al., 2010; Spittlehouse & Wang, 2016).

CONCLUSION

"The single story creates stereotypes, and the problem with stereotypes is not that they are untrue, but that they are incomplete. They make one story become the only story" (Adichie, 2009).

With the quote above, the Nigerian author Chimamanda Ngozi Adichie was referring to the dangers of racial stereotyping. But similarly, a species distribution model is a single story: a simplified human construct of a complex phenomenon shared to help reveal patterns and better our understanding of the conditions around us. SDMs can help us to assess our conservation efforts and assist us in conservation action. But models, like stories, must be developed and communicated in meaningful ways. Future modeling efforts and applications are encouraged to validate and build upon these models by integrating best available methods and data, address and account for uncertainty, and to support continued systematic monitoring and ecological research to better our understanding of fundamental biological and ecological theory (Sinclair et al, 2010). This study is only one story but it is a story worth telling in support of waterfowl conservation in British Columbia.

ACKNOWLEDGEMENTS

Throughout this project I have been provided with a great deal of support and assistance. Foremost, I sincerely thank Dr. Julian Avery, Beth King, and Dr. Justine Blanford at Penn State for their invaluable guidance and unflagging support. I express gratitude to the Canadian Wildlife Service and Fisheries and Oceans Canada who as my employers provided the time and resources to carry out and complete the study. My CWS colleagues—Andre Breault, Kathleen Moore, Dr. Rhonda Millikin, Dr. Andrea Norris, Jeffrey Thomas and Dr. Caroline Fox—who provided indispensable guidance and encouragement and generously shared their time and wealth of knowledge. Finally, I thank my partner Brady Ciel Marks for her enduring patience, humour, and awesome dance moves.

REFERENCES

- Adamus, P. (2014). Effects of Forest Roads and Tree Removal In or Near Wetlands of the Pacific Northwest: A Literature Synthesis, (December).
- Adichie, C. N. (2009). The Danger of a Single Story. Ted Global. Retrieved from https://www.ted.com/talks/chimamanda_adichie_the_danger_of_a_single_story?language=en
- Baldassarre, G. (2014). *Ducks, Geese, and Swans of North America* (Revised an). John Hopkins University Press.
- Bunnell, Fred L, Wells, Ralph, M. A. (2010). Vulnerability of wetlands to climate change in the Southern Interior Ecoprovince : a preliminary assessment 1 Final Report. *Centre for Applied Conservation Research, University of British Columbia*, (March).
- Canadian Wetland Inventory — Ducks Unlimited Canada. (n.d.). Retrieved July 27, 2019, from <https://www.ducks.ca/initiatives/canadian-wetland-inventory/>
- CCRS/CCMEO/NRCan. (2017). 2010 Land Cover of North America at 30 meters. Retrieved from <http://www.cec.org/tools-and-resources/map-files/land-cover-2010-landsat-30m>
- Commission, A. L. (2018). Agricultural Land Reserve. Agicultural Land Commission. Retrieved from <https://catalogue.data.gov.bc.ca/dataset/alc-alr-boundary>
- Corvallis Microtechnology Inc. (2015). PC-Mapper with Airborne Inspection. Corvallis, Oregon.
- Cutler, A., Cutler, D., & Stevens, J. (2012). Random Forests. In C. Zhang & Y. Ma (Eds.), *Ensemble Machine Learning*.
- Demarchi, D. A. (2011). *An Introduction to the Ecoregions of British Columbia*. Victoria. Retrieved from <http://www.env.gov.bc.ca/wld/documents/techpub/rn324.pdf>
- Drake, A., & Martin, K. (2018). Local temperatures predict breeding phenology but do not result in breeding synchrony among a community of resident cavity-nesting birds. *Scientific Reports*, 8(1), 2756. <https://doi.org/10.1038/s41598-018-20977-y>
- Elith, J., & Graham, C. H. (2009). Do they? How do they? WHY do they differ? On finding reasons for differing performances of species distribution models. *Ecography*, 32(1), 66–77. <https://doi.org/10.1111/j.1600-0587.2008.05505.x>
- Environment and Climate Change Canada. (2019). *Summary of Canada's 6th National Report to the Convention on Biological Diversity*. Gatineau, Quebec.
- Environment Canada - Biodiversity Convention Office. (1995). *Canadian Biodiversity Strategy--Canada's Response to the Convention on Biological Diversity*. Hull, Quebec. Retrieved from http://www.biodivcanada.ca/560ED58E-0A7A-43D8-8754-C7DD12761EFA/CBS_e.pdf
- ESRI. (2019). ArcGIS Pro. Redlands.
- Franklin, J., & Miller, J. (2009). *Mapping Species Distributions*. Cambridge: Cambridge University Press.
- Guillera-Arroita, G., Lahoz-Monfort, J. J., Elith, J., Gordon, A., Kujala, H., Lentini, P. E., ... Wintle, B. A. (2015). Is my species distribution model fit for purpose? Matching data and models to applications. *Global Ecology and Biogeography*, 24(3), 276–292. <https://doi.org/10.1111/geb.12268>
- Guisan, A., & Thuiller, W. (2005). Predicting species distribution: offering more than simple habitat models. *Ecology Letters*, 8(9), 993–1009. <https://doi.org/10.1111/j.1461-0248.2005.00792.x>
- Guisan, A., Tingley, R., Baumgartner, J. B., Naujokaitis-Lewis, I., Sutcliffe, P. R., Tulloch, A. I. T., ... Buckley, Y. M. (2013). Predicting species distributions for conservation decisions. *Ecology Letters*, 16(12), 1424–1435. <https://doi.org/10.1111/ele.12189>
- Guisan, A., Wilfried, T., & Zimmermann, N. E. (2017). *Habitat Suitability and Distribution Models with Applications in R*. Cambridge University Press. <https://doi.org/10.1017/9781139028271>
- Halabisky, M., Babcock, C., Moskal, L., Halabisky, M., Babcock, C., & Moskal, L. M. (2018). Harnessing the Temporal Dimension to Improve Object-Based Image Analysis Classification of Wetlands. *Remote Sensing*, 10(9), 1467. <https://doi.org/10.3390/rs10091467>
- Islam, S. ul, Déry, S. J., Werner, A. T., Islam, S. ul, Déry, S. J., & Werner, A. T. (2017). Future Climate Change Impacts on Snow and Water Resources of the Fraser River Basin, British Columbia. *Journal of Hydrometeorology*, 18(2), 473–496. <https://doi.org/10.1175/JHM-D-16-0012.1>
- MacKenzie, W. H., & Moran, J. R. (2004). *Wetlands of British Columbia: a guide to identification*. *Land Management Handbook* 52. <https://doi.org/Land Management Handbook No. 52>
- McKenney, D. W., Hutchinson, M. F., Papadopol, P., Lawrence, K., Pedlar, J., Campbell, K., ... Owen, T. (2011). Customized Spatial Climate Models for North America. *Bulletin of the American Meteorological Society*, (December), 1611–1622. <https://doi.org/10.1175/BAMS-D-10-3132.1>
- Ministry of Forests, Lands, Natural Resource Operations, and R. D. (Organization/Institution). (2011). Freshwater Atlas. Government of British Columbia. Retrieved from ftp://ftp.geobc.gov.bc.ca/sections/outgoing/bmgs/FWA_Public
- Ministry of Forests, Lands, Natural Resource Operations, and R. D. (Organization/Institution). (2018). Biogeoclimatic Zones of British Columbia. Victoria: Government of British Columbia. Retrieved from <https://catalogue.data.gov.bc.ca/dataset/bec-map>
- Ministry of Forests, Lands, N. R. O. and R. D. (Organization/Institution). (2013). Digital Road Atlas - Master Partially Attributed Roads. Victoria: Government of British Columbia. Retrieved from <https://catalogue.data.gov.bc.ca/dataset/digital-road-atlas-dra-master-partially-attributed-roads/resource/a06a2e11-a0b1-41d4-b857-cb2770e34fb0>
- Ministry of Forests, Lands, N. R. O. and R. D. (Organization/Institution). (2014). TRIM Contours. Victoria: Government of British Columbia.
- Murray, D. L., Anderson, M. G., & Steury, T. D. (2010). Temporal shift in density dependence among North American

- breeding duck populations. *Ecology*, 91(2), 571–581. <https://doi.org/10.1890/MS08-1032.1>
- Niemuth, N. D., Fleming, K. K., & Reynolds, R. E. (2014). Waterfowl Conservation in the US Prairie Pothole Region: Confronting the Complexities of Climate Change. *PLoS ONE*, 9(6), e100034. <https://doi.org/10.1371/journal.pone.0100034>
- Niemuth, N., Wangler, B., & Reynolds, R. (2010). Spatial and temporal variation in wet area of wetlands in the Prairie Pothole Region of North Dakota and South Dakota. *Wetlands*, 30.
- Pike, R. G., Bennett, K. E., Redding, T. E., Werner, A. T., Spittlehouse, D. L., Moore, R. D. D., ... Tschaplinski, P. J. (2010). Climate Change Effects on Watershed Processes in British Columbia. In *Compendium of Forest Hydrology and Geomorphology in British Columbia* (Vol. 3). <https://doi.org/10.5194/nhess-12-1-2012>
- Pike, R., Redding, T., Moore, R., Winkler, R., & Bladon, K. (2010). *Compendium of forest hydrology and geomorphology in British Columbia*. Retrieved from <https://www.for.gov.bc.ca/hfd/pubs/docs/lmh/Lmh66.htm>
- Pimm, S. L. (1994). The importance of watching birds from airplanes. *Trends in Ecology & Evolution*, 9(2), 41–43. [https://doi.org/10.1016/0169-5347\(94\)90264-X](https://doi.org/10.1016/0169-5347(94)90264-X)
- Python Software Foundation. (2018). Python Language Reference. Retrieved from <https://www.python.org/downloads/>
- Qian, S. S. (2010). *Environmental and Ecological Statistics with R*. CRC Press.
- R Core Team. (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from <http://www.r-project.org/>
- Savard, J.-P. L., Sean Boyd, W., & John Smith, G. E. (1994). Waterfowl-wetland relationships in the Aspen Parkland of British Columbia: comparison of analytical methods. *Hydrobiologia*, 279–280(1), 309–325. <https://doi.org/10.1007/BF00027864>
- Schlesinger, W. H., & Bernhardt, E. S. (2013). *Biogeochemistry* (3rd ed.). Elsevier. <https://doi.org/https://doi.org/10.1016/C2010-0-66291-2>
- Shmueli, G. (2011). To Explain or to Predict? *Statistical Science*, 25(3), 289–310. <https://doi.org/10.1214/10-sts330>
- Sinclair, S., White, M., & Newell, G. (2010). How useful are species distribution models for managing biodiversity under future climates? *Ecology and Society*, 15(1). Retrieved from <http://www.ecologyandsociety.org/vol15/iss1/art8/>
- Smith, G. (1995). A Critical Review of the Aerial and Ground Surveys of Breeding Waterfowl in North America. *Biological Science Report* (Vol. 268). <https://doi.org/10.1126/science.268.5215.1262>
- Snauffer, A. M., Hsieh, W. W., & Cannon, A. J. (2016). Comparison of gridded snow water equivalent products with in situ measurements in British Columbia, Canada. *Journal of Hydrology*, 541, 714–726. <https://doi.org/10.1016/J.JHYDROL.2016.07.027>
- Spittlehouse, D., & Wang, T. (2016). Comparison of Climate Change Projections in ClimateBC v5 . 30, 1–6.
- Strobl, C., Boulesteix, A.-L., Kneib, T., Augustin, T., & Zeileis, A. (2008). Conditional variable importance for random forests. *BMC Bioinformatics*, 9(1), 307. <https://doi.org/10.1186/1471-2105-9-307>
- Strobl, C., Hothorn, T., & Zeileis, A. (2009). Party on! A new, conditional variable-importance measure for random forests available in the party package. *R Journal*, 1(2), 14–17.
- US Fish and Wildlife Service (USFWS) and Canadian Wildlife Service (CWS). (1987). *Standard operating procedures for aerial waterfowl breeding ground population and habitat surveys in North America*.
- Wang, T., Hamann, A., Spittlehouse, D., & Carroll, C. (2016). Locally Downscaled and Spatially Customizable Climate Data for Historical and Future Periods for North America, 1–17. <https://doi.org/10.1371/journal.pone.0156720>
- Zhao, Q., Silverman, E., Fleming, K., & Boomer, G. S. (2016). Forecasting waterfowl population dynamics under climate change — Does the spatial variation of density dependence and environmental effects matter? *Biological Conservation*, 194, 80–88. <https://doi.org/10.1016/J.BIOCON.2015.12.006>
- Zimpfer, N. L., Breault, A., & Sanders, T. (2019). *British Columbia Breeding Waterfowl Survey Report, 2019*.

APPENDIX 1 – Dataset Sources

Environmental datasets collated and evaluated for the modelling exercise. The table describes each dataset and provides a brief description of variables and their assessment.

APPENDIX 2—Reclassification

APPENDIX 3 – Python Script Toolbox

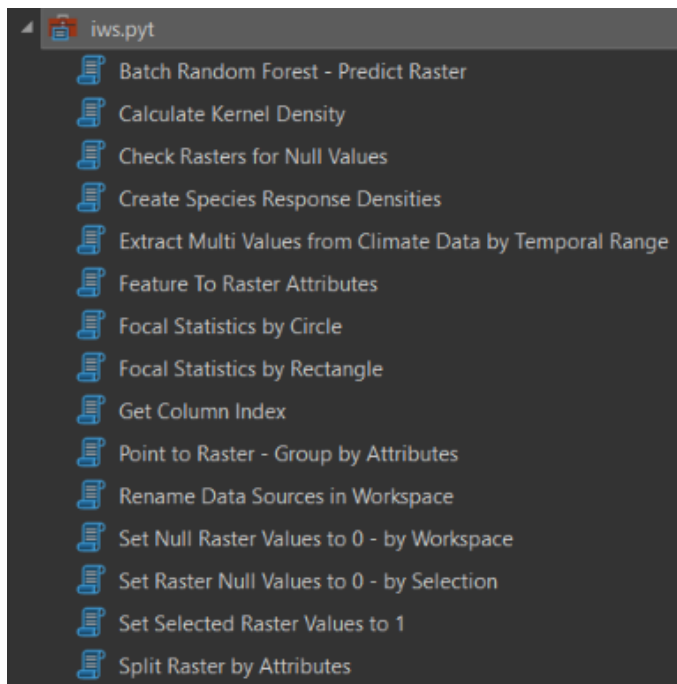


Figure 6. Python Script Toolbox as displayed in ArcGis Pro.

The code for the Python script toolbox is provided below.

```
# -*- coding: utf-8 -*-

import arcpy, os
from arcpy import env
arcpy.env.overwriteOutput = True
from arcpy.sa import *
arcpy.CheckOutExtension("Spatial")
arcpy.env.qualifiedFieldNames = False
# scratch_ws = arcpy.CreateScratchName(workspace=arcpy.env.scratchGDB)
arcpy.env.snapRaster = r"C:\Users\hashimotoy\Desktop\ws\base_alb.gdb\dem"
arcpy.env.mask = r"C:\Users\hashimotoy\Desktop\ws\base_alb.gdb\survey_mask"
scratch_ws = r"C:\Users\hashimotoy\Desktop\ws_prep\scratch.gdb"

def checkFieldExists(in_tbl, field_nm, field_type):
    fields = arcpy.ListFields(in_tbl)
    x = False
    for field in fields:
        if field.name == field_nm:
            x = True
    if x == False:
        arcpy.AddField_management(in_tbl, field_nm, field_type)

def getBaseName(in_fc):
    desc = arcpy.Describe(in_fc)
    basename = desc.basename
    return basename

def unique_values(table, field):
    with arcpy.da.SearchCursor(table, [field]) as cursor:
        return sorted({row[0] for row in cursor})

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of the
        .pyt file)."""
        self.label = "IWS Data Tools"
        self.alias = ""

        # List of tool classes associated with this toolbox
        self.tools = [featureToRasterAttributes,
                      setRasterNullValuesTo0,
                      setRasterNullValuesTo0Workspace,
                      renameDataInWorkspace,
                      kernelDensityCalculations,
                      splitRasterByAttributes,
                      focalStatsByCircle,
                      focalStatsByRectangle,
                      setRasterValuesTo1,
                      pointToRasterGroupByAttributes,
                      getColumnIndex,
                      extractClimateValuesForRange,
                      checkRasterForNullValues,
                      batchRandomForest,
                      createPredictedDensitySurfaces]
```

```

class Tool(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Tool"
        self.description = ""
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Features",
            name="in_features",
            datatype="GPFeatureLayer",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Sinuosity Field",
            name="sinuosity_field",
            datatype="Field",
            parameterType="Optional",
            direction="Input")

        param1.value = "sinuosity"

        # Third parameter
        param2 = arcpy.Parameter(
            displayName="Output Features",
            name="out_features",
            datatype="GPFeatureLayer",
            parameterType="Derived",
            direction="Output")

        param2.parameterDependencies = [param0.name]
        param2.schema.clone = True

        parameters = [param0, param1, param2]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        """The source code of the tool."""
        return

class batchRandomForest(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Batch Random Forest - Predict Raster"
        self.description = "Tool uses explanatory rasters to predict to raster. Specify the output workspace" \
            "for the four resulting outputs: prediction raster, variable importance table, trained " \
            "features and validation r2. The outputs will be named according to the name of the input feature class" \
            "and the species selected for prediction."
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""

        param0 = arcpy.Parameter(
            displayName="Input Training Features",
            name="in_fc",
            datatype="DEFeatureClass",
            parameterType="Required",
            direction="Input")

        param1 = arcpy.Parameter(
            displayName="Variables to Predict",
            name="sp_field",
            datatype="Field",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        param1.parameterDependencies = [param0.name]

        param2 = arcpy.Parameter(
            displayName="Output Workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        param3 = arcpy.Parameter(
            displayName="Output Files Prefix",
            name="prefix",
            datatype="GPString",
            parameterType="Required",
            direction="Input")

        param4 = arcpy.Parameter(
            displayName="Number of Trees",

```

```

        name="n_tree",
        datatype="GPLong",
        parameterType="Required",
        direction="Input")

param5 = arcpy.Parameter(
    displayName="# of Random Variables - mtry",
    name="m_try",
    datatype="GPLong",
    parameterType="Required",
    direction="Input")

param6 = arcpy.Parameter(
    displayName="Compensate for Sparse Categories",
    name="sparse",
    datatype="GPBoolean",
    parameterType="Required",
    direction="Input")
param6.value = True

param7 = arcpy.Parameter(
    displayName="Percent Excluded for Validation",
    name="pct_train",
    datatype="GPLong",
    parameterType="Required",
    direction="Input")
param7.filter.type = "ValueList"
param7.filter.list = [10, 20, 30]

param8 = arcpy.Parameter(
    displayName="Number of Validation Runs",
    name="n_validation",
    datatype="GPLong",
    parameterType="Required",
    direction="Input")

# Fifth parameter
param9 = arcpy.Parameter(
    displayName="Select Explanatory Rasters",
    name="in_rasters",
    datatype="DERasterDataset",
    parameterType="Required",
    direction="Input",
    multiValue=True)

parameters = [param0, param1, param2, param3, param4, param5,
               param6, param7, param8, param9]

return parameters

def isLicensed(self):
    """Set whether tool is licensed to execute."""
    return True

def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""
    return

def updateMessages(self, parameters):
    return

def execute(self, parameters, messages):
    ws_fishnet = r"C:\Users\hashimotoy\Desktop\ws_prep\fishnet.gdb"

    in_fc = parameters[0].valueAsText
    lst_sp = parameters[1].valueAsText
    out_ws = parameters[2].valueAsText
    prefix = parameters[3].valueAsText
    n_tree = parameters[4].valueAsText
    m_try = parameters[5].valueAsText
    sparse = parameters[6].valueAsText
    pct_train = parameters[7].valueAsText
    n_validation = parameters[8].valueAsText
    in_rasters = parameters[9].valueAsText

    prediction_type = "PREDICT_RASTER"
    nm = getBaseName(in_fc)
    in_features = in_fc
    arcpy.env.mask = r"C:\Users\hashimotoy\Desktop\ws_prep\raster_attributes.gdb\\" + "eco_" + nm

    lst_rasters = []
    lst_matching = [] # Explanatory raster matching
    for raster in in_rasters.split(';'):
        lst_rasters.append(raster)
        sub_lst = [raster, raster]
        lst_matching.append(sub_lst)

    treat_variable_as_categorical = None
    explanatory_variables = None
    distance_features = None
    explanatory_rasters = lst_rasters
    features_to_predict = None
    explanatory_variable_matching = None
    explanatory_distance_matching = None
    explanatory_rasters_matching = lst_matching
    use_raster_values = True
    number_of_trees = n_tree
    minimum_leaf_size = None
    maximum_level = None
    sample_size = None
    random_sample = m_try

```

```

percentage_for_training = pct_train
output_classification_table = None
compensate_sparse_categories = True
number_validation_runs = n_validation

for sp in lst_sp.split(';'):
    arcpy.AddMessage("Running forest for " + sp)
    variable_predict = sp
    file_prefix = prefix + "_" + nm + "_" + sp + "_"
    output_features = os.path.join(out_ws, file_prefix + "_output_features")
    output_raster = os.path.join(out_ws, file_prefix + "_rtr")
    output_trained_features = os.path.join(out_ws, file_prefix + "_trained")
    output_importance_table = os.path.join(out_ws, file_prefix + "_varimp")
    output_validation_table = os.path.join(out_ws, file_prefix + "_validation")

    arcpy.stats.Forest(prediction_type,
                        in_features,
                        variable_predict,
                        treat_variable_as_categorical,
                        explanatory_variables,
                        distance_features,
                        explanatory_rasters,
                        features_to_predict,
                        output_features, output_raster,
                        explanatory_variable_matching,
                        explanatory_distance_matching,
                        explanatory_rasters_matching,
                        output_trained_features,
                        output_importance_table,
                        use_raster_values,
                        number_of_trees,
                        minimum_leaf_size,
                        maximum_level,
                        sample_size,
                        random_sample,
                        percentage_for_training,
                        output_classification_table,
                        output_validation_table,
                        compensate_sparse_categories,
                        number_validation_runs)

    return

class checkRasterForNullValues(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Check Rasters for Null Values"
        self.description = "Interrogates selected rasters for NULL values"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Select Rasters",
            name="in_rasters",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Output Workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        lst_rasters = parameters[0].valueAsText
        out_ws = parameters[1].valueAsText

        for raster in lst_rasters.split(';'):
            nm = getBaseName(raster)
            arcpy.AddMessage("Reading {}".format(raster))
            outras = IsNull(raster)
            outras.save("{}{}/{}".format(out_ws, nm))
            arcpy.AddMessage("Saving {}".format(raster))

        return

class setRasterNullValuesTo0Workspace(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Set Null Raster Values to 0 - by Workspace"

```

```

        self.description = "Interrogates rasters in a workspace and checks for null values. If present, NA values are set" \
            "to 0"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Raster Workspace",
            name="in_rasters",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Output Workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        in_ws = parameters[0].valueAsText
        out_ws = parameters[1].valueAsText
        arcpy.env.workspace = in_ws
        rasters = arcpy.ListRasters()

        for raster in rasters:
            nm = getBaseName(raster)
            arcpy.AddMessage("Reading {0}".format(raster))
            out_ras = Con(IsNull(raster), 0, raster)
            arcpy.AddMessage("Setting null for {0}".format(raster))
            out_ras.save("{0}/{1}".format(out_ws, nm))
            arcpy.AddMessage("Saving {0}".format(raster))
        return

class setRasterNullValuesTo0(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Set Raster Null Values to 0 - by Selection"
        self.description = "Sets all null raster values to 0 in new raster"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Rasters",
            name="in_rasters",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        param1 = arcpy.Parameter(
            displayName="Target workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        lst_rasters = parameters[0].valueAsText
        out_ws = parameters[1].valueAsText

        for raster in lst_rasters.split(';'):
            nm = getBaseName(raster)
            arcpy.AddMessage("Reading {0}".format(raster))

```

```

        out_raster = Con(IsNull(raster), 0, raster)
        arcpy.AddMessage("Setting null for {0}".format(raster))
        out_raster.save("{0}/{1}".format(out_ws, nm))
        arcpy.AddMessage("Saving {0}".format(raster))

    return

class featureToRasterAttributes(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Feature To Raster Attributes"
        self.description = ""
        self.canRunInBackground = True

    def getParameterInfo(self):
        # Define parameter definitions

        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Features",
            name="in_features",
            datatype="DEFeatureClass",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Attribute Field",
            name="field",
            datatype="Field",
            parameterType="Required",
            direction="Input")

        param1.filter.list = ['TEXT', 'LONG']
        param1.parameterDependencies = [param0.name]

        param2 = arcpy.Parameter(
            displayName="Output workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        param2.defaultEnvironmentName = "workspace"

        parameters = [param0, param1, param2]
        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        import os

        in_fc = parameters[0].valueAsText
        in_field = parameters[1].valueAsText
        out_ws = r"C:\Users\hashimoto\Desktop\ws_prep\raster.gdb"
        attributes_ws = r"C:\Users\hashimoto\Desktop\ws_prep\raster_attributes.gdb"
        out_raster = os.path.join(out_ws, os.path.basename(in_fc))
        # Converting to Raster
        arcpy.AddMessage("Converting to raster: " + out_raster + "...")
        raster = arcpy.FeatureToRaster_conversion(in_fc, in_field, os.path.join(out_ws, out_raster))
        # raster = r"C:\Users\hashimoto\Desktop\ws_prep\raster.gdb\{0}"
        values = unique_values(raster, in_field)
        # Extract By Attributes each unique value
        for value in values:
            if value == "":
                pass
            else:
                print("    Extracting " + value)
                where_clause = in_field + " = '" + value + "'"
                arcpy.AddMessage("    " + where_clause)
                extract = ExtractByAttributes(raster, where_clause)
                out_nm = (in_field + "_" + value).lower()
                extract.save(os.path.join(attributes_ws, out_nm))

        return

class renameDataInWorkspace(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Rename Data Sources in Workspace"
        self.description = "The tool will batch rename data within a workspace by one of three options: " \
            "replace, prefix or add suffix. if "
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Workspace",
            name="in_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")
        #param0.filter.type = "Workspace"

        param1 = arcpy.Parameter(

```



```

        displayName="Select String Parsing Function",
        name="parse_function",
        datatype="GPString",
        parameterType="Required",
        direction="Input")

param1.filter.list = ["REPLACE SUBSTRING", "ADD PREFIX", "ADD SUFFIX"]

# Second parameter
param2 = arcpy.Parameter(
    displayName="Input string",
    name="in_string",
    datatype="GPString",
    parameterType="Required",
    direction="Input")

# Third parameter
param3 = arcpy.Parameter(
    displayName="Output string",
    name="out_string",
    datatype="GPString",
    parameterType="Optional",
    direction="Input")

parameters = [param0, param1, param2, param3]

return parameters

def isLicensed(self):
    """Set whether tool is licensed to execute."""
    return True

def updateParameters(self, parameters):
    parameters[3].enabled = False
    if parameters[1].value:
        if parameters[1].valueAsText == "REPLACE SUBSTRING":
            parameters[3].enabled = True
        else:
            parameters[3].enabled = False
    return

def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter. This method is called after internal validation."""
    return

def execute(self, parameters, messages):
    in_ws = parameters[0].valueAsText
    parse_function = parameters[1].valueAsText
    in_string = parameters[2].valueAsText
    out_string = parameters[3].valueAsText

    env.workspace = in_ws
    fcs = arcpy.ListFeatureClasses()
    rasters = arcpy.ListRasters()

    lst = fcs + rasters

    for data in lst:
        nm = getBaseName(data)
        if in_string in nm:
            pass
        elif parse_function == "ADD PREFIX":
            out_string = in_string + nm
        elif parse_function == "ADD SUFFIX":
            out_string = nm + out_string
        elif parse_function == "REPLACE SUBSTRING":
            if in_string not in nm:
                arcpy.AddMessage("\nThe substring " + in_string + " is not within data source names")
                pass
            else:
                arcpy.Rename_management(data, nm.replace(in_string, out_string))
                arcpy.AddMessage("\nRenamed " + nm + " to " + nm.replace(in_string, out_string))
                arcpy.Rename_management(data, out_string)
                arcpy.AddMessage("\nRenamed " + nm + " to " + out_string)
    return

class kernelDensityCalculations(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Calculate Kernel Density"
        self.description = "Tool to generalize vector point and line inputs (use the Focal Statistics tool for raster inputs)"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Feature Classes",
            name="lst_in_features",
            datatype="DEFeatureClass",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        param0.filter.list = ["Point", "Multipoint", "Polyline"]

        param1 = arcpy.Parameter(
            displayName="Search radius",
            name="search_radius",
            datatype="GPIlong",
            parameterType="Required",
            direction="Input")
        param1.value = 564 # pi 'r' squared ~ 1sqkm

        param2 = arcpy.Parameter(

```

```

        displayName="Cell size",
        name="cell_size",
        datatype="GPIong",
        parameterType="Required",
        direction="Input")

    param2.value = 20 # Default

    param3 = arcpy.Parameter(
        displayName="Density unit value",
        name="area_unit_scale_factor",
        datatype="GPString",
        parameterType="Required",
        direction="Input")
    param3.filter.list = ["SQUARE_MAP_UNITS", "SQUARE_KILOMETERS", "HECTARES", "SQUARE_METERS"]

    param3.value = "SQUARE_METERS"

    param4 = arcpy.Parameter(
        displayName="Output workspace",
        name="out_ws",
        datatype="DEWorkspace",
        parameterType="Required",
        direction="Input")

    parameters = [param0, param1, param2, param3, param4]
    # parameters = [param0, param1, param2, param3]
    return parameters

def isLicensed(self):
    """Set whether tool is licensed to execute."""
    return True

def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""
    return

def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter. This method is called after internal validation."""
    return

def execute(self, parameters, messages):
    1st_inputs = parameters[0].valueAsText
    search_radius = parameters[1].valueAsText
    cell_size = parameters[2].valueAsText
    area_unit_scale_factor = parameters[3].valueAsText
    out_ws = parameters[4].valueAsText

    for in_fc in 1st_inputs.split(';'):
        nm = getBaseName(in_fc)
        arcpy.AddMessage("\n\nProcessing " + nm)
        kd = KernelDensity(in_features=in_fc,
                           population_field="",
                           cell_size=cell_size,
                           search_radius=search_radius,
                           area_unit_scale_factor=area_unit_scale_factor,
                           out_cell_values="DENSITIES",
                           method="PLANAR")
        kd.save(os.path.join(out_ws, nm))

    return

class splitRasterByAttributes(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Split Raster by Attributes"
        self.description = "Output rasters are named according to the attribute field and value"
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Raster",
            name="in_raster",
            datatype=["DERasterDataset", "GPRasterLayer"],
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Attribute Field",
            name="field",
            datatype="Field",
            parameterType="Required",
            direction="Input")

        param1.filter.list = ['TEXT', 'LONG']
        param1.parameterDependencies = [param0.name]

        param2 = arcpy.Parameter(
            displayName="Output workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        param2.defaultEnvironmentName = "workspace"

        parameters = [param0, param1, param2]
        return parameters

    def isLicensed(self):

```

```

        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        in_raster = parameters[0].valueAsText
        in_field = parameters[1].valueAsText
        out_ws = parameters[2].valueAsText

        values = unique_values(in_raster, in_field)
        # Extract By Attributes each unique value
        for value in values:
            if value == "":
                pass
            else:
                print("    Extracting " + value)
                where_clause = in_field + " = '" + value + "'"
                arcpy.AddMessage("    " + where_clause)
                extract = ExtractByAttributes(in_raster, where_clause)
                out_nm = (in_field + " " + value).lower()
                extract.save(os.path.join(out_ws, out_nm))

        return

class focalStatsByCircle(object):
    def __init__(self):
        self.label = "Focal Statistics by Circle"
        self.description = "Tool to generalize raster inputs (use the Kernel Density tool for vector points and lines)"
        self.canRunInBackground = True

    def getParameterInfo(self):

        param0 = arcpy.Parameter(
            displayName="Input Rasters",
            name="in_rasters",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        param1 = arcpy.Parameter(
            displayName="Neighbourhood in map units",
            name="neighbourhood",
            datatype="GPLong",
            parameterType="Required",
            direction="Input")
        param1.value = 1200 # Circular neighbourhood of 1.2km to account for center of 400m cell grid

        param2 = arcpy.Parameter(
            displayName="Statistics type",
            name="stats_type",
            datatype="GPString",
            parameterType="Required",
            direction="Input",
            multiValue=False)
        param2.filter.list = ["MEAN", "MAJORITY", "MAXIMUM", "MEDIAN",
                              "MINIMUM", "MINORITY", "RANGE", "STD",
                              "SUM", "VARIETY"]
        param2.value = "SUM"

        param3 = arcpy.Parameter(
            displayName="Output workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1, param2, param3]
        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):

        rasters = parameters[0].valueAsText
        neighbourhood = parameters[1].valueAsText
        stats_type = parameters[2].valueAsText
        out_ws = parameters[3].valueAsText

        for in_raster in rasters.split(';'):
            arcpy.AddMessage(in_raster)
            nm = getBaseName(in_raster)
            arcpy.AddMessage("\nProcessing focal statistics for " + nm)
            fs = FocalStatistics(in_raster, NbrCircle(neighbourhood, "MAP"),

```

```

        stats_type,"DATA")
        fs.save(os.path.join(out_ws, nm))

    return

class focalStatsByRectangle(object):
    def __init__(self):
        self.label = 'Focal Statistics by Rectangle'
        self.description = "Tool to generalize raster inputs (use the Kernel Density tool for vector points and lines)"
        self.canRunInBackground = True

    def getParameterInfo(self):

        param0 = arcpy.Parameter(
            displayName="Input Rasters",
            name="in_rasters",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input",
            multiValue=True)

        param1 = arcpy.Parameter(
            displayName="Neighbourhood in map units",
            name="neighbourhood",
            datatype="GPLong",
            parameterType="Required",
            direction="Input")
        param1.value = 1000 # Rectangular neighbourhood 1km x 1km

        param2 = arcpy.Parameter(
            displayName="Statistics type",
            name="stats_type",
            datatype="GPString",
            parameterType="Required",
            direction="Input",
            multiValue=True)
        param2.filter.list = ["MEAN", "MAJORITY", "MAXIMUM", "MEDIAN",
                              "MINIMUM", "MINORITY", "RANGE", "STD",
                              "SUM", "VARIETY"]
        param2.value = "SUM"

        param3 = arcpy.Parameter(
            displayName="Output workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1, param2, param3]
        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):

        rasters = parameters[0].valueAsText
        neighbourhood = parameters[1].valueAsText
        stats_type = parameters[2].valueAsText
        out_ws = parameters[3].valueAsText

        for in_raster in rasters.split(';'):
            arcpy.AddMessage(in_raster)
            nm = getBaseName(in_raster)
            arcpy.AddMessage("\nProcessing focal statistics for " + nm)
            fs = FocalStatistics(in_raster, NbrRectangle(neighbourhood, neighbourhood, "MAP"),
                                stats_type,"DATA") #
            fs.save(os.path.join(out_ws, nm))

        return

class setRasterValuesTo1(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Set Selected Raster Values to 1"
        self.description = "Sets all raster values to 1 within a specified workspace. For categorical (factor) variables."
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Rasters",
            name="in_rasters",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input",
            multiValue=False)

        param1 = arcpy.Parameter(
            displayName="Output workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",

```

```

        direction="Input")

    parameters = [param0, param1]
    return parameters

def isLicensed(self):
    """Set whether tool is licensed to execute."""
    return True

def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""
    return

def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter. This method is called after internal validation."""
    return

def execute(self, parameters, messages):
    lst_rasters = parameters[0].valueAsText
    out_ws = parameters[1].valueAsText

    for in_raster in lst_rasters.split(';'):
        nm = getBaseName(in_raster)
        arcpy.AddMessage("\n" + nm)
        out_con = Con(~IsNull(in_raster), 1, 0)
        out_con.save(os.path.join(out_ws, nm))

    return

class pointToRasterGroupByAttributes(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Point to Raster - Group by Attributes"
        self.description = "Create raster surfaces from one or more attributes in an input point feature class." \
            " Output rasters will be prefixed by user defined string" \
            "and appended with the field value. The optional 'group by' parameter will output individual rasters" \
            "based on attribute."
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Point Feature Class",
            name="in_pts",
            datatype="GPFeatureLayer",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Attribute fields",
            name="fields",
            datatype="Field",
            parameterType="Required",
            direction="Input",
            multiValue=True)
        param1.parameterDependencies = [param0.name]

        # Third parameter
        param2 = arcpy.Parameter(
            displayName="Output Workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")
        param2.defaultEnvironmentName = "workspace"
        # param2.filter.list = ["Local Database"]

        # Fourth parameter
        param3 = arcpy.Parameter(
            displayName="Prefix for out raster name",
            name="prefix",
            datatype="GPString",
            parameterType="Required",
            direction="Input")

        # Fifth parameter
        param4 = arcpy.Parameter(
            displayName="Cell Assignment",
            name="assignment_type",
            datatype="GPString",
            parameterType="Required",
            direction="Input")
        param4.filter.type = 'ValueList'
        param4.filter.list = ["MEAN", "MAXIMUM", "MOST_FREQUENT",
            "MINIMUM", "RANGE", "STANDARD_DEVIATION",
            "SUM", "COUNT"]
        param4.value = "MEAN"

        # Sixth parameter
        param5 = arcpy.Parameter(
            displayName="Cell Size",
            name="cell_size",
            datatype="GPIInteger",
            parameterType="Input",
            direction="Input")
        param5.value = 400

        # Seventh Optional parameter
        param6 = arcpy.Parameter(
            displayName="Optional - Group By attribute",
            name="groupby",

```

```

        datatype="Field",
        parameterType="Optional",
        direction="Input",
        multiValue = False)

    param6.parameterDependencies = [param0.name]

    parameters = [param0, param1, param2, param3, param4, param5, param6]

    return parameters

def isLicensed(self):
    """Set whether tool is licensed to execute."""
    return True

def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""
    return

def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter. This method is called after internal validation."""
    return

def execute(self, parameters, messages):
    # Set environment settings

    in_pts = parameters[0].valueAsText
    fields = parameters[1].valueAsText
    out_ws = parameters[2].valueAsText
    prefix = parameters[3].valueAsText
    assignment_type = parameters[4].valueAsText
    cell_size = parameters[5].valueAsText
    groupby = parameters[6].valueAsText

    env.workspace = out_ws

    arcpy.AddMessage("Running Point to Raster on: " + in_pts)

    for field in fields.split(';'):
        valField = field

        if groupby is None:
            arcpy.AddMessage("\n    Processing " + field)
            inFeatures = in_pts
            outRaster = (prefix + "_" + field).lower()
            arcpy.PointToRaster_conversion(inFeatures, valField, outRaster,
                                           assignment_type, "", cell_size)

        else:
            values = unique_values(in_pts, groupby)
            for value in values:
                arcpy.AddMessage("\n    Processing " + field + " for " + str(value))
                nm = getBaseName(in_pts)
                out_nm = nm + "_" + str(value)
                where_clause = groupby + " = '" + str(value) + "'"
                inFeatures = arcpy.Select_analysis(in_pts, out_nm, where_clause)
                outRaster = (prefix + "_" + field + "_" + str(value)).lower()
                arcpy.PointToRaster_conversion(inFeatures, valField, outRaster,
                                               assignment_type, "", cell_size)

    return

class getColumnIndex(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Get Column Index"
        self.description = "Using pandas module, read a csv file and get the index of selected columns"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input CSV",
            name="csv",
            datatype="DEFile",
            parameterType="Required",
            direction="Input")
        param0.filter.list = ['txt', 'csv']

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Fields",
            name="fields",
            datatype="Field",
            parameterType="Required",
            direction="Input",
            multiValue=True)
        param1.parameterDependencies = [param0.name]

        parameters = [param0, param1]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

```

```

def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter. This method is called after internal validation."""
    return

def execute(self, parameters, messages):
    import pandas as pd

    in_csv = parameters[0].valueAsText
    fields = parameters[1].valueAsText

    arcpy.AddMessage("\n\nGetting indices for selected columns...\n\n")
    csv = pd.read_csv(in_csv)

    for field in fields.split(';'):
        index = csv.columns.get_loc(field)
        arcpy.AddMessage("\n" + field + " : " + str(index) + "\n")

    return

class extractClimateValuesForRange(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Extract Multi Values from Climate Data by Temporal Range"
        self.description = "The tool will extract the values of selected climate variables for" \
            "the input points for each year in the study range 2007-2017. Basename of filename" \
            "will be appended with _annual_clim. Note intermediate files will be output to the 'Output" \
            "workspace"
        self.canRunInBackground = True

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Features",
            name="in_features",
            datatype="GPFeatureLayer",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Output Workspace",
            name="out_ws",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        # Third parameter
        param2 = arcpy.Parameter(
            displayName="Select climate variables",
            name="clim_var",
            datatype="GPString",
            parameterType="Input",
            direction="Input",
            multiValue=True)
        param2.filter.type = 'ValueList'
        param2.filter.list = ["ahm", "bfff", "cmd", "dd5", "dd18", "effp", "mat", "map",
            "ppt", "pas_sp", "pas_wt", "shm", "tave_sp", "tave_wt", "tave04"]

        # Fourth parameter
        param3 = arcpy.Parameter(
            displayName="Workspace containing climate rasters",
            name="clim_ws",
            datatype="DEWorkspace",
            parameterType="Input",
            direction="Input")

        parameters = [param0, param1, param2, param3]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        pts = parameters[0].valueAsText
        out_ws = parameters[1].valueAsText
        clim_var = parameters[2].valueAsText
        ws_clim = parameters[3].valueAsText

        base_nm = getBaseName(pts)

        lst_merge = []
        for i in range(2007, 2018):
            out_fc = os.path.join(out_ws, base_nm + "clim_" + str(i))
            arcpy.AddMessage("\nRunning value extraction for " + out_fc)
            in_pts = arcpy.CopyFeatures_management(pts, out_fc)
            checkFieldExists(in_pts, "year_txt", "TEXT")
            lst_clim = []
            for var in clim_var.split(';'):
                clim_raster = os.path.join(ws_clim, "clim_" + var + "_" + str(i))
                arcpy.AddMessage("\n" + clim_raster)
                lst_clim.append([clim_raster, var])

```

```

        arcpy.CalculateField_management(out_fc, "year_txt", "" + str(i) + "")
        ExtractMultiValuesToPoints(out_fc, lst_clim)
        lst_merge.append(out_fc)
    arcpy.AddMessage("\n\nCompleted extractions by year. \n\nMerging into final dataset...")
    arcpy.Merge_management(lst_merge, os.path.join(out_ws, base_nm + "_annual_clim"))
    return

class createPredictedDensitySurfaces(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Create Species Response Densities"
        self.description = "This tool will create density surfaces for predicted SDMs output from 'cforest'"
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Directory of CSV response files",
            name="in_csv",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Geodatabase of fishnets for each ecosection",
            name="ws_fishnet",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        # Third parameter
        param2 = arcpy.Parameter(
            displayName="Output Workspace",
            name="ws_out",
            datatype="DEWorkspace",
            parameterType="Required",
            direction="Input")

        # Fourth parameter
        param3 = arcpy.Parameter(
            displayName="Snap Raster",
            name="snap_raster",
            datatype="DERasterDataset",
            parameterType="Required",
            direction="Input")

        parameters = [param0, param1, param2, param3]

        return parameters

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        ws_response = parameters[0].valueAsText
        ws_fishnet = parameters[1].valueAsText
        ws_output = parameters[2].valueAsText
        snap_raster = parameters[3].valueAsText

        # Loop through the CSV files of predicted densities
        # For each CSV file, get the basename, join it to the appropriate ecosection fishnet
        # Output a raster
        env.workspace = ws_response
        arcpy.env.snapRaster = snap_raster
        files = arcpy.ListFiles("*.csv")
        for f in files:
            f_nm = getBaseName(f)
            arcpy.AddMessage(f_nm)
            eco = f_nm[7:10]
            sp = f_nm[11:len(f_nm)]
            arcpy.AddMessage("Preparing " + sp.upper() + " for " + eco + "...")
            fishnet_lyr = arcpy.MakeFeatureLayer_management(os.path.join(ws_fishnet, eco), "fishnet_lyr")
            joined = arcpy.AddJoin_management(fishnet_lyr, "id_fishnet", f, "id_fishnet")
            arcpy.FeatureToRaster_conversion(joined, sp, os.path.join(ws_output, eco + "_" + sp), cell_size=400)
            arcpy.AddMessage("\n\nCompleted rasterized densities. Check results in " + ws_output)
        return

```


APPENDIX 4 – R Scripts – Project Repository

A cloned Github repository of the project can be found at <https://github.com/Yuhash/iws>. The repository contains the base survey data inputs as well as an interactive [Jupyter notebook](#) that documents the major R script processes in the workflow. For optimal viewing display without interactive code functionality, the Notebook viewer's static view is at https://nbviewer.jupyter.org/github/Yuhash/iws/blob/master/IWS_Notebook.ipynb.

```
# Create a data frame from a list of csv files converted to a list of data frames merged
by common column values
path <-
"C:/Users/hashimotoy/Documents/MGIS/Analysis/Models/grid_cell/cab/glmm_2007to2017"
extension = "*.csv"
batch_read <- function(path, extension) {
  file_names <- list.files(path, pattern = extension)
  data_list <- lapply(paste(path, file_names, sep = "/"), read.csv)
  data_frame <- bind_rows(data_list)
  data_frame
}

#-----
# From a list of CSV files, select columns and merge

# Helper function
readdata <- function(filename) {
  df <- read.csv(filename)
  vec <- df[, 4]# Identify the columns
  names(vec) <- df[, 1]
  return(vec)
}
result <- do.call(rbind, lapply(filenamees, readdata))

# =====
# Merge all files in a folder
wd = "C:/Users/hashimotoy/Documents/MGIS/Data/climateBC/climatebc_normals1980-2010"
setwd(wd)
files <- list.files(pattern = "\\*.csv")
all_files <- lapply(files, function(x){
  read.csv(x, header=TRUE)
})
df <- do.call(rbind.data.frame, all_files)

write.csv(df, "climatebc_normals_1980-2010_merged.csv", row.names = FALSE)

# Another way
library(dplyr)
library(readr)

df <- do.call(rbind(lapply(files, function(x)read.table(x, header=TRUE, sep = ","))))

#=====
# Flat correlation matrix with P and r values

library(tidyr)
library(tibble)
```

```

library(Hmisc)
flat_cor_mat <- function(cor_r, cor_p){
  #This function provides a simple formatting of a correlation matrix
  #into a table with 4 columns containing :
  # Column 1 : row names (variable 1 for the correlation test)
  # Column 2 : column names (variable 2 for the correlation test)
  # Column 3 : the correlation coefficients
  # Column 4 : the p-values of the correlations
  library(tidyr)
  library(tibble)
  cor_r <- rownames_to_column(as.data.frame(cor_r), var = "row")
  cor_r <- gather(cor_r, column, cor, -1)
  cor_p <- rownames_to_column(as.data.frame(cor_p), var = "row")
  cor_p <- gather(cor_p, column, p, -1)
  cor_p_matrix <- left_join(cor_r, cor_p, by = c("row", "column"))
  cor_p_matrix
}

library(corrplot)
cor <- rcorr(as.matrix(mtcars[, 1:7]))

my_cor_matrix <- flat_cor_mat(cor$r, cor$p)
head(my_cor_matrix)

corr <- rcorr(as.matrix(df))
corrplot(corr$r, method='number', number.cex= 12/ncol(df_sub))#To alter font size

#=====
# Check for NA values
df %>%
  select_if(function(x) any(is.na(x))) %>%
  summarise_each(funs(sum(is.na(.)))) -> extra_NA

#=====
# Set NA values to 0
df[is.na(df)] <- 0

#=====
# The function complete.cases() returns a logical vector indicating which cases are
complete.
# list rows of data that have missing values
df[!complete.cases(df),]

#=====

# Check which version of R is running
Sys.getenv("R_ARCH")
# "/i386" or "/x64"

# =====
# UPDATE PACKAGES
update.packages(checkBuilt=TRUE, ask=FALSE)

#=====
# Find column names with NA values
colnames(df)[colSums(is.na(df)) > 0]

```

```

mypath <- path

multmerge = function(mypath){
  filenames=list.files(path=mypath, full.names=TRUE)
  datalist = lapply(filenames, function(x){read.csv(file=x,header=T)})
  Reduce(function(x,y) {merge(x,y)}, datalist)}

mymergeddata = multmerge(path)

#####
# UPDATE R Version
# installing/loading the package:
if(!require(installr)) {
  install.packages("installr"); require(installr)} #load / install+load installr

library(installr)
# using the package:
updateR()

#####
# # Rename a column in R
colnames(data)[colnames(data)=="old_name"] <- "new_name"

sp_start <- grep("dabblers", colnames(df))
sp_end <- grep("tot_sp", colnames(df))

print(sp_start)
print(sp_end)

sp <- colnames(df)[sp_start:sp_end]

var_start <- sp_end + 1
var_end <- ncol(df)
print(var_start)
print(var_end)

int_eco <- grep("eco", colnames(df))
int_trans <- grep("trans_id", colnames(df))
int_bec_zn <- grep("bec_zone", colnames(df))
int_bec_sz <- grep("bec_sz", colnames(df))

for (i in 1:length(sp)){
  nm <- sp[i]
  print(nm)
  int = grep(nm, colnames(df))
  data <- df[c(int,int_eco,int_trans,int_bec_zn, int_bec_sz,var_start:var_end)]
  colnames(data)[1]<- "sp"
  write.csv(data, paste(outwd, paste("rf_",scale, "_",nm,".csv", sep = ""), sep = "/"),
row.names = FALSE)
}

```


APPENDIX 5 – Predicted Distribution Maps