2015

Automating the Classification of Thematic Rasters for Weighted Overlay Analysis in GeoPlanner for ArcGIS

Charles J. Mayfield
Univeristy of Redlands

University of Redlands

Automating the Classification of Thematic Rasters for Weighted Overlay Analysis in GeoPlanner for ArcGIS

A Major Individual Project submitted in partial satisfaction of the requirements for the degree of Master of Science in Geographic Information Systems

by

C. Joseph Mayfield

Mark Kumler, Ph.D., Committee Chair

Nader Afzalan, Ph.D.

February 2016

Automating the Classification of Thematic Rasters for Weighted Raster Overlay Analysis in GeoPlanner for ArcGIS

by

C. Joseph Mayfield

The report of C. Joseph Mayfield is approved.

Nader Afzalan, Ph.D.

Mark Kumler, Ph.D., Committee Chair

February 2016

Acknowledgements

The successful completion of this project was facilitated by the support of a number of individuals who assisted in a variety of ways. As my academic advisor, Dr. Mark Kumler provided invaluable guidance and feedback throughout the course of the project. Dr. Nader Afzalan lent his perspective as both a geodesign and GIS expert. Nate Strout helped me get off the ground with Python and software development and then provided invaluable troubleshooting assistance in the later stages of the project. I'm also grateful for the efforts of the other faculty staff of the University of Redlands MS GIS program, Dr. Flewelling, Dr. Ma, Dr. Ren, Andrea Barrios, and adjunct faculty. It was a privilege to be a member of Cohort 25 – The Golden Cohort. The opportunity to participate in the program and undertake this project was made possible through the Esri Fellows scholarship, and I am grateful in particular for the support of my manager, Vin Thomas, and the project client, Rob Stauder. This project represents a significant effort that required sacrifices of time availability and would not have been possible without the loving support of my wife, Emily. I am grateful to her and the rest of my family for their support and providing the motivation to complete this work.

Abstract

Automating the Classification of Thematic Rasters for Weighted Overlay Analysis in GeoPlanner for ArcGIS

by

C. Joseph Mayfield

Esri's GeoPlanner for ArcGIS application provides powerful analysis capabilities through the weighted overlay analysis modeler. This modeler consumes weighted overlay services composed of pre-processed raster layers. Creating custom weighted overlay services for GeoPlanner is a difficult and complex process that requires both domain-specific and GIS expertise. This challenge was addressed by simplifying the weighted overlay service creation workflow and developing two new custom Python tools that guide GeoPlanner users through the process of preparing input datasets and then classifying the raster datasets. Where possible these tools automate the required steps and where user input is needed, the tools provide default recommendations based on the input datasets properties and characteristics. As a result, the weighted overlay services creation workflow has been significantly improved and more GeoPlanner users can include their own data in weighted overlay analyses.

Table of Contents

Chapter 1 – Introduction 1

1.1 Client 2

1.2 Problem Statement 2

1.3 Proposed Solution 3

1.3.1 Goals and Objectives 4

1.3.2 Scope 4

1.3.3 Methods 5

1.4 Audience 6

1.5 Overview of the Rest of this Report 6

Chapter 2 – Background and Literature Review 7

2.1 Land-Use Planning in the Context of Geodesign 7

2.2 Weighted Overlay Analysis 8

2.2.1 History of Weighted Overlay in GIS 8

2.2.2 Using Weighted Overlay in Multiple Criteria Decision Making 9

2.3 Automating the Classification Process 10

2.3.1 Classification Methods 10

2.3.2 Determining the Best Grid Resolution 11

2.4 Summary 13

Chapter 3 – Systems Analysis and Design 15

3.1 Problem Statement 15

3.2 Requirements Analysis 17

3.3 System Design 19

3.3.1 Prepare the Input Dataset 21
3.3.2 Classify the Raster Dataset 22
3.3.3 Create the Mosaic Dataset 23
3.3.4 Create the Weighted Overlay Service 23
3.4 Project Plan 24
3.4.1 Design Phase 24
3.4.2 Develop Phase 25
3.4.3 Deploy Phase 26
3.4.4 Work Breakdown Structure 27
3.4.5 Project Plan Analysis 28
3.5 Summary 28

Chapter 4 – Database Design 31

4.1 Conceptual and Logical Data Models 31
4.2 Data Sources 34
4.3 Data Collection Methods 35
4.4 Data Scrubbing and Loading 36
4.5 Summary 36

Chapter 5 – Implementation 37

5.1 Prepare the Input Dataset 37
5.1.1 Step 1 – Input Dataset 38
5.1.2 Step 2 – Project the Input Dataset 40
5.1.3 Step 3 – Convert Vector to Raster 41
5.1.4 Step 4 – Clip the Area of Interest 43

5.1.5 Step 5 – Output the Prepared Raster 44

5.2 Classify the Raster Dataset 45

5.2.1 Step 1 – Input Prepared Raster Dataset and Determine Raster Type 46

5.2.2 Step 2 – Classify the Raster Dataset 48

5.2.3 Step 3 – Add Related Fields..... 51

5.2.4 Step 4 – Write Data to an XML File..... 54

Chapter 6 – Results and Analysis..... 55

Chapter 7 – Conclusions and Future Work 59

Works Cited 61

Appendix A. Python Toolbox Code..... 63

Table of Figures

Figure 2.1 Steps in the weighted overlay analysis process 8

Figure 3.1 Weighted Overlay Service Creation Workflow 20

Figure 3.2 Process A: Prepare the Input Dataset Tool 21

Figure 3.3 Process B: Classify the Raster Dataset Tool 22

Figure 3.4 Process C: Create the Mosaic Dataset 23

Figure 3.5 Process D: Create the Weighted Overlay Service 24

Figure 4.1 Prepare the Input Dataset Tool - Conceptual Data Model 32

Figure 4.2 Classify the Raster Dataset Tool - Conceptual Data Model 33

Figure 5.1 Input Dataset Processing through the Project Tools 37

Figure 5.2 Prepare the Input Dataset Tool 38

Figure 5.3 Parameter 1: Input Dataset 39

Figure 5.4 Parameter 4: Select a Resampling Method 41

Figure 5.5 Parameter 2: Select a Cell Size 42

Figure 5.6 Parameter 3: Select a Raster Value Field 43

Figure 5.7 Parameter 4: Select a Mask for the Design Scenario 44

Figure 5.8 Parameter 6: Output Raster Name and Location 45

Figure 5.9 Classify the Raster Dataset Tool 46

Figure 5.10 Step 1: Input the Prepared Raster 47

Figure 5.11 Select the Classification Method and Number of Classes 49

Figure 5.12 Python Code for Calculating the Break Points 49

Figure 5.13 Python Code for Defining the Class Ranges 50

Figure 5.14 Select the Field for Class Labels 52

Figure 5.15 Default class labels for continuous datasets in GeoPlanner 53

Figure 6.1 Weighted overlay service and composite raster in GeoPlanner 56

List of Tables

Table 2.1 Cell Size Values 13

Table 3.1 Requirements for Process A: Prepare the Input Dataset 18

Table 3.2 Requirements for Process B: Classify the Raster Dataset 19

Table 3.3 Work Breakdown Structure 27

Table 4.1 Project Datasets and Characteristics 34

List of Acronyms and Definitions

geoTIFF	Raster file format that associates spatial information with a TIFF image
GIS	Geographic Information System
IDE	Integrated Development Environment
MCDM	Multiple Criteria Decision-Making
MCE	Multicriteria Evaluation
MLA	Maximum Location Accuracy
MS GIS	University of Redlands Masters of Science in Geographic Information Systems
OWA	Ordered Weighted Averaging
WLC	Weighted Linear Combination
WOA	Weighted Overlay Analysis
WROS	Weighted Raster Overlay Service

Chapter 1 – Introduction

The emerging field of geodesign promises to improve the current design and planning paradigm by providing a number of spatially-enabled tools, services, and processes that bring greater understanding and transparency to the decision making process. These geodesign assets leverage a multi-disciplinary approach and incorporate the strengths of each on a common spatial platform. In 2014, Esri launched GeoPlanner for ArcGIS, a web-hosted geodesign application that guides users through a tailored geodesign workflow. This application has been designed to make geodesign accessible to a wide variety of users, from GIS professionals to knowledge workers in other fields with little to no GIS experience.

GeoPlanner includes a number of analysis tools that can be used to assess and evaluate datasets. One of these core tools is the weighted overlay modeler tool that uses weighted overlay analysis to perform complex multi-criteria evaluations. Creating new weighted overlay services for GeoPlanner is a difficult and complex process that requires both domain-specific and GIS expertise. These requirements prevent many GeoPlanner users from being able to include their own datasets in the application's weighted overlay analysis tool. Without this information, these users are unable to realize the full value of the GeoPlanner application in their design and development workflows.

One of the primary goals of the GeoPlanner for ArcGIS application is to democratize the geodesign process by making it easy for non-GIS experts to use. However, these kinds of users currently face a number of barriers that prevent them from uploading their own datasets, as weighted overlay services, into the application's weighted overlay

modeler tool. The client needs a better way for GeoPlanner users to create weighted overlay services and then upload them into the application. The purpose of this project is to introduce a solution that addresses these needs in a context that supports and assists all of the application's end users.

1.1 Client

The client for this project is Mr. Robert Stauder, Product Engineer on the GeoPlanner for ArcGIS team, from Esri Professional Services. The client needs a solution that enables GeoPlanner users to create and upload new weighted overlay services from their own datasets. This solution needs to be developed in Python, the scripting language used in the development of the GeoPlanner application.

1.2 Problem Statement

The problem addressed in this project is the high level of GIS expertise that is required to create and upload weighted overlay services for the GeoPlanner for ArcGIS application. In the GeoPlanner application, the current process for preparing a dataset, classifying the data, and then uploading the layers to a weighted overlay service is quite involved, requiring access to ArcGIS for Desktop and Server, a high level of GIS expertise, and domain-specific knowledge. The process for creating a weighted overlay service includes four general steps. First, the user must prepare the dataset by converting it to the required raster format, clipping it to the area of interest, addressing any No Data cells, and more. Second, the dataset must classified and new raster fields must be added and populated with the classification information. This step is particularly challenging for users because it requires that they create new raster fields and populate those fields with specific parameters. If the parameters are entered incorrectly, the final weighted overlay service

will fail to deliver accurate analysis results. Third, multiple raster datasets are added to a mosaic dataset, a collection of raster datasets that are stored together in a single mosaicked image. Fourth, the mosaic must be published to ArcGIS Online as a weighted overlay service with specific tags for the GeoPlanner application.

These steps require access to ArcGIS 10.3 for Desktop and Server as well as the Server Image Extension. The added software requirements and GIS expertise needed to complete this complex process create a usability barrier for users of the application. This is a problem because in many cases, users of the GeoPlanner application do not have this access or the necessary expertise. Developing a solution to this problem will expand the usefulness of the GeoPlanner application and allow more users to take full advantage of the tool.

1.3 Proposed Solution

Addressing the problems related to the creation of weighted overlay services for GeoPlanner for ArcGIS is a significant challenge. This challenge was addressed by simplifying the weighted overlay service creation workflow and developing two new tools that guide GeoPlanner users through the process of preparing input datasets and then classifying the raster datasets. All of the workflow steps are associated with four main processes: Prepare the Input Dataset, Classify the Raster Dataset, Create the Mosaic Dataset, and Create the Weighted Overlay Service. These processes pass the user's dataset from step to step and perform all of the geoprocessing, formatting, and publishing tasks with minimal user input. This workflow resolves the client's problem by making the GeoPlanner application easier to use by removing the barriers users currently face when they try to upload their own datasets as weighted overlay services.

1.3.1 Goals and Objectives

The overall goal for this project is to simplify the processes of creating weighted overlay services and classifying thematic raster datasets for use in the GeoPlanner for ArcGIS application. This goal has been accomplished by meeting the following objectives:

- Develop a tool that automates the initial preparation and processing of user-input datasets.
- Develop a tool that that classifies datasets and automatically updates the raster fields.
- Create an intuitive user interface that minimizes user inputs, clearly explains each step of the process, and reduces the required level of GIS- and domain-specific expertise for the end user.

1.3.2 Scope

The scope of this project extends from the planning and research stage to the creation of an automated workflow design, followed by the development of a working solution, to the testing and delivery of the final product. The project's initial research included topics such as the history of land suitability assessment, the application of multi-criteria evaluations and weighted overlay analysis, the theory and methodology behind thematic raster classification, and Python development. A thorough understanding of these topics was applied to the design and development of an automated workflow that produces weighted overlay services. This workflow provides an enhanced user experience by automatically recommending appropriate classifications for user-supplied datasets, reducing the chance of user error by automating the configuration of the raster dataset, and by making the whole process faster and more efficient.

Testing this workflow with synthetic data and with client supplied sample data confirmed that the final solution successfully met the project's objectives and the client's expectations. The final product will be used by the client and the GeoPlanner for ArcGIS team to inform future product updates and enhancements. This project did not include any work related to developing other aspects of the GeoPlanner app, raster analysis, or data collection.

Working on this project required access to a number of applications, tools, services, and data sources. The major technical components of the solution included the following software requirements: ArcGIS 10.3 for Desktop with the Spatial Analyst Extension, ArcGIS 10.3 for Server with the Image Extension, and an ArcGIS Online organizational account with access to GeoPlanner for ArcGIS. The Weighted Raster Overlay Services toolbox provided access to many of the basic geoprocessing tools that were needed for the new workflow. Additionally, PyScripter, a free open source IDE for Python, was used to develop the final solution. The services and data for GeoPlanner are hosted through ArcGIS Online and ArcGIS for Server. Having this clearly defined scope for the project's goals and objectives informed many of the project's major decisions and helped to prevent scope creep.

1.3.3 Methods

The final workflow was developed by first mapping out each of the processes in a conceptual diagram. That conceptual diagram was used to create a series of models in ModelBuilder and standalone Python scripts that replicated each of the steps and processes included in the final workflow. ModelBuilder is an Esri tool that is included in the ArcGIS for Desktop software package. All of the Python scripting was done in the

PyScripter, a common Python IDE. The final workflow utilizes a custom set of tools, called the Weighted Raster Overlay Service toolbox, which is provided on GitHub by the GeoPlanner for ArcGIS team.

1.4 Audience

The primary audience for this project report is the project client, Rob Stauder and his GeoPlanner for ArcGIS team. The secondary audience includes other planners, designers, and researchers exploring solutions for automating the classification of thematic rasters, as well as students learning more about multicriteria evaluation methods in suitability analyses.

1.5 Overview of the Rest of this Report

The following chapters and sections of this report follow the project's framework. Chapter 2 provides a summary of the background research that was conducted and a literature review of project-related books and articles. Chapter 3 describes the system analysis and design. The system analysis explains the process of designing and developing the weighted overlay services creation workflow. Chapter 4 discusses the decisions that were made about data processing defaults and presets in the workflow. Chapter 5 details the implementation of the project – how the final workflow was developed and completed. Chapter 6 reviews the results of the project and the quality of the weighted overlay services produced by the automated workflow. The final section, Chapter 7, concludes this project by reviewing the overall project's successes and shortcomings. Potential future work that furthers the goals and objectives of this project is also listed.

Chapter 2 – Background and Literature Review

The challenge of automating the classification of thematic datasets for a web-based geodesign tool touches on a number of application domains, including land-use planning and suitability analysis, various multicriteria decision-making methodologies, and the specific techniques and approaches used in the automation of raster classification. Each of these domains is supported by a wealth of academic research and professional literature. In the following sections the concepts of land-use planning and multicriteria analysis are introduced, the weighted overlay analysis methodology is explained, and the specific techniques related to automating data classification workflows are explored.

2.1 Land-Use Planning in the Context of Geodesign

The practice of geodesign has its roots in the discipline of land-use planning. Land-use planning is the process of determining the ideal locations, in a predefined area of interest, for a specific design scenario based on a number of factors or characteristics (Collins, Steiner, & Rushman, 2001; Steiner, McSherry, & Cohen, 2000). Land-use planning methods have traditionally been employed by local and regional government planners, but today they have been adopted by others such as conservationists and commercial developers (Van der Merwe, 1997). As part of land-use suitability planning, geodesigners regularly create suitability assessments that employ multiple criteria decision-making (MCDM) methodologies to ensure that both environmental and socio-economic factors are considered in the final evaluation (Carver, 1991). A popular technique included in this multicriteria evaluation (MCE) approach is the weighted overlay analysis.

2.2 Weighted Overlay Analysis

As a geodesign tool, GeoPlanner is largely built on the theories and practices used by land-use planners who have developed a variety of methods to evaluate the suitability of various land-uses. One of these methods is weighted overlay analysis (WOA). WOA is performed by overlaying classified datasets, such as soil type, land cover, or topography, for a defined area, assigning a weight to each dataset, summing the values of each vertical cell stack, and then evaluating the resulting composite map (Collins, Steiner, & Rushman, 2001). This method of suitability analysis, as shown in Figure 2.1, plays a key role in many geodesign workflows and is included in GeoPlanner for ArcGIS.

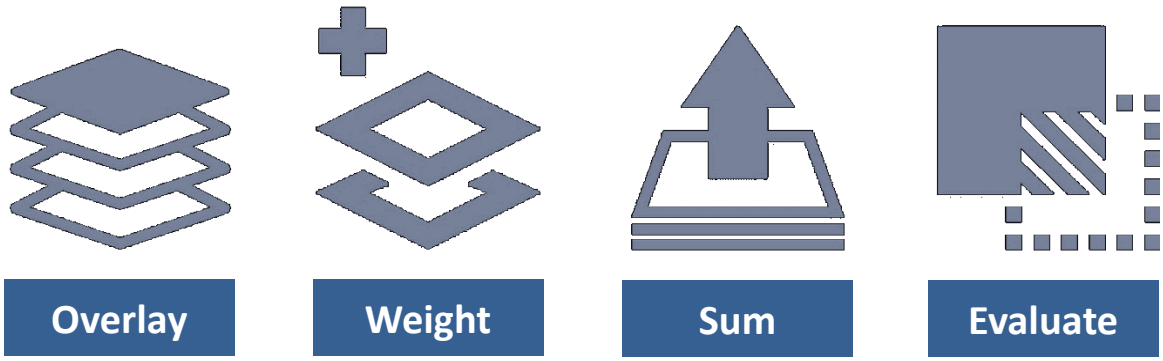

Figure 2.1 Steps in the weighted overlay analysis process

2.2.1 History of Weighted Overlay in GIS

The concept of using weighted overlay in the service of land-use suitability analysis was described at length by Ian McHarg in his seminal work, Design with Nature (1969). In this book, McHarg describes the process as a means of conducting multicriteria analyses by categorizing and ranking values from a variety of thematic datasets, creating a transparency for each dataset, and then overlaying the transparencies together to create a composite image. This final composite image was then used to evaluate the suitable land-uses in the design scenario.

The value of this weighted overlay approach to land-use suitability analysis was understood and adopted by designers, researchers, and others who saw great potential from the early GIS programs that were being pioneered and developed in the 1960s and 1970s. With significant advancements in the field of computer science, some of the very first geographic information system programs, such as Harvard's SYMAP, were designed to assess land-use in scenarios such as site selection for reservoirs (Chrisman, 2004). Roger Tomlinson states that he first proposed computerizing the overlay method while working at Spartan Air in 1962 (Tomlinson, 1999). As the availability and processing power of computers improved, modern GIS software has incorporated the practices and analytical processes of early land-use planners while adopting more complex and sophisticated decision making methodologies.

2.2.2 Using Weighted Overlay in Multiple Criteria Decision Making

One of the first decision making methodologies was the Weighted Linear Combination (WLC) technique, which improved on the map overlay approach by allowing planners to create composite maps. These composite maps made it possible for decision makers to consider multiple attributes in a single map (Hopkins, 1977). However, according to Malczewski (2000), WLC maps "are often used without full understanding of the assumptions underlying this approach" (p. 5). This issue has been addressed in some geographic information systems by the incorporation of robust Multiple Criteria Decision Making (MCDM) methods designed to help stakeholders make well-informed decisions based on various attributes (Jankowski, 1995). Additionally, Ordered Weighted Averaging (OWA) can be employed to assign importance and order to attribute values as weights in a dataset that extend and generalize the other methods used in creating land-

use suitability maps (Malczewski, 2004). These various methods and approaches provide important context and considerations for the design of the GeoPlanner application and the way it classifies data and presents that data to the user.

2.3 Automating the Classification Process

Automating the classification of thematic rasters has been the subject of significant research and development efforts for several decades as organizations have undertaken the task of processing large datasets of remotely sensed imagery. Much of this effort has been directed at classifying physical features, such as topography and landform elements, through a combination of image processing procedures that consider variables such as location, elevation, slope, and surface texture (Iwahashi & Pike, 2007; Dragut & Blaschke, 2006). Other projects have explored using the shared characteristics of similar features to categorize man-made structures, such as building complexes and schools (Wilson, 2007). This previous work provides an extensive body of knowledge that informed the direction and approaches used in automating the classification of thematic rasters for the GeoPlanner application.

2.3.1 Classification Methods

Fuzzy classification of datasets, which contain multiple classes, is preferred over Boolean classification methods because it provides geodesigners a more realistic perspective by being able to consider all of an area's characteristics, define the extent of suitability, and differentiate between 'somewhat suitable' and 'highly suitable' locations (Hall, Wang, & Subaryono, 1992).

2.3.2 Determining the Best Grid Resolution

The grid resolution, which is determined by the raster's cell size, plays an important role in the ability of the weighted overlay tool to produce accurate and useful composite maps. As a dataset is being processed in preparation for inclusion in a weighted overlay service, the size of the raster's cells can have a significant impact on the required processing power and disk space. Smaller cell sizes are associated with high resolution rasters, and they tend to demand more computation power to process and take up more disk space with larger file sizes. Conversely, a raster with the same extent but a larger cell size will require less computing power and have a smaller file size. There is no ideal cell size or grid resolution; instead, when all factors are considered, a range of suitable resolutions, or cell sizes, can be employed.

High resolution rasters with small cell sizes typically represent more data for a given area than a coarser raster with large cells covering the same extent. Aggregating data by using a large pixel size leads to a coarser grid and the potential for critical data loss. When high resolution rasters are resampled to a larger cell size, there is some level of data loss, and details are obscured by combining and averaging, in one way or another, the values that were previously being represented in the finer raster grid. Conversely, downscaling to a smaller pixel size in a finer grid can increase file size and the required computational power without necessarily improving the quality of the final composite map (Hengl, 2006). However a consideration of just the processing and storage demands ignores the purpose of using rasters in the first place. The benefits and shortcomings of both small and large pixel sizes make selecting an appropriate grid resolution a challenge for an automated workflow where multiple input datasets with different grid resolutions will need to be converted to the same pixel size.

In this project, multiple strategies were considered to fully automate the task of determining the most suitable cell size, but due to the inherent complexities and compromises associated with the decision, it was determined that this selection should be made by the user. To simplify this choice, a number of cell size calculation and selection guidelines were evaluated. For example, Esri's recommended method of calculating cell size by dividing the shortest extent of the raster by 250 was determined to be an overly simplistic equation that failed to produce meaningful cell sizes. Other guidelines recommended selecting a cell size as a function of the map scale or data resolution. Waldo Tobler, for instance, proposed a simple formula based the smallest detectable size – "Divide the denominator of the map scale by 1,000" and then divide that number by 2 to reach the resolution (Tobler, 1988). The idea of referencing the map scale of the raster is a sound principle, but the cell sizes produced by this formula are too coarse for many geodesign scenarios.

A variation of this concept that considers both map scale and the maximum location accuracy was found to produce a more suitable range of cell sizes. To address this issue, Hengl (2006) suggests using the scale of the dataset and the maximum location accuracy (MLA) to calculate grid resolution. The MLA for typical datasets ranges from 0.1 mm for digitally produced maps to 0.25 mm for maps produced by analog methods (Rossiter & Hengl, 2002; Vink, 1975). By taking the product of the dataset's scale and the MLA, a reasonable pixel size for the dataset can be calculated

$P \geq SN \bullet MLA = SN \bullet 0.00025$ or $P \geq SN \bullet MLA = SN \bullet 0.0001$

where P is the grid resolution (pixel size), SN is the scale number, and MLA is the maximum location accuracy (Hengl, 2006). This method of calculating grid resolution

offers an acceptable compromise that limits information loss from aggregation while ensuring the output dataset will not be needlessly large in terms of file size. The Maximum Location Accuracy method was determined to provide the best cell size recommendations for geodesign applications, and the values from Table 2.1 were added as dropdown selections for the tool's second input parameter.

Table 2.1 Cell Size Values

Scale Name	Map Scale	Cell Size (meters)	Description
State (Large)	6,000,000	600	600 (Large State - 1:6,000,000)
State (Medium)	3,000,000	300	300 (Medium State - 1:3,000,000)
Counties	1,500,000	150	150 (Counties - 1:1,500,000)
County	750,000	75	75 (County - 1:750,000)
Metro Area	320,000	32	32 (Metro Area - 1:320,000)
Cities	160,000	16	16 (Cities - 1:160,000)
City	80,000	8	8 (City - 1:80,000)
Town	40,000	4	4 (Town - 1:40,000)
Neighborhood	20,000	2	2 (Neighborhood - 1:20,000)
Block Group	10,000	1	1 (Block Group - 1:10,000)
Street	5,000	0.5	0.5 (Street - 1:5,000)

2.4 Summary

Many of the tools and practices used by modern day geodesigners were pioneered and refined by land use planners and other designers. These academics and professionals recognized early on the value of leveraging computers to assist in complex MCDM processes. In particular, weighted overlay analysis has been incorporated into geographic information systems to assist in land-use suitability assessments. Recent research related to classification methods and determining grid resolutions informs the way an automated raster classification workflow could produce useful output datasets. In the next chapter these concepts are applied in the system analysis and design of an automated raster classification workflow.

Chapter 3 – Systems Analysis and Design

This chapter describes the approach that was taken to address the issues with the weighted overlay service creation workflow. Section 3.1 details the problems with the original weighted overlay service creation workflow. Section 3.2 describes the functional and nonfunctional requirements that shaped the final tools. Section 3.3 outlines the conceptual system design. Section 3.4 introduces the project plan that was initially implemented at the beginning of this project.

3.1 Problem Statement

The GeoPlanner for ArcGIS application has been designed with a focus on typical geodesign and land-use planning workflows, emphasizing the analysis, assessment, and collaborative aspects of the process. One of the key features for assessing land-use suitability in GeoPlanner is the weighted overlay analysis modeler. Weighted overlay analysis includes reclassifying the rasters to a common scale, ranking the class values in each raster, assigning each raster a respective weight as a percentage, and then overlaying the rasters on top of each other and calculating the total summed value for each cell in an output raster. This output raster is then used to determine each raster cell's suitability service, hosted on ArcGIS Online as "weighted overlay services," to produce suitability models.

Esri has created and maintains a number of these weighted overlay services for general usage. And for specific design scenarios, users also have the ability to build their own weighted overlay services with their own data. However, the process of creating a custom weighted overlay service is complex, requiring access to a suite of advanced

geoprocessing tools, a high level of GIS expertise, and domain-specific knowledge, all of which introduce significant barriers to the typical GeoPlanner user. This is a problem because it conflicts with one of the core purposes of the GeoPlanner application – "GeoPlanner is designed with the intent of being quick to learn and easy to use by a wide range of non-GIS users" (Esri, 2016). This project addresses this problem by automating and simplifying the most complex steps in the weighted overlay services creation process.

The weighted overlay service creation workflow is composed of four processes: A – Prepare the Input Dataset; B – Classify the Raster Dataset; C – Create the Mosaic Dataset; and D – Create the Weighted Overlay Service. Of these four processes, the first two, preparing the input dataset and classifying the resulting raster dataset, present the greatest challenges to GeoPlanner users and represents the majority of the effort required to complete the workflow.

Process A – Prepare the Input Dataset includes a number of steps requiring the user to transform all datasets into the same projection, convert any vector data into rasters, standardize all cells to an appropriate and uniform size, select a resampling method, define the extent of their project, and address any NoData values or irregularities. The resulting rasters must all have the same cell size, projection, extent, and be saved to the same file type, geoTIFF.

Process B – Classify the Raster Dataset requires users to classify their data by first determining a suitable classification method and number of classes for their analysis, then calculating the value ranges for each class. When the value ranges have been calculated, the user must define the minimum and maximum values for each class by manually

entering a comma-delimited string in a custom Python tool dialog to configure the raster fields. Output values, range labels, and NoData ranges must also be determined and entered by hand. At the end of this process, all of the user-entered parameters are written to an XML file that is associated with the input raster dataset. Any errors in this process are typically difficult to identify and prevent the weighted overlay service from being created at the end of the workflow.

3.2 Requirements Analysis

The technical requirements of this project are framed in part by the geoprocessing tools used to prepare and classify the datasets and their system environments. To start with, as a web application, GeoPlanner for ArcGIS is designed to be accessed through a web browser, such as Internet Explorer, Chrome, Firefox, or Safari, on a computer with a high-speed internet connection. In addition, the Weighted Raster Overlay Service (WROS) toolset and the two new processing tools, as custom Python tools built with the ArcPy site package, require ArcGIS 10.3 for Desktop with the Spatial Analyst extension. Publishing the mosaic dataset as an image service and then hosting it as a weighted overlay service on ArcGIS Online requires access to ArcGIS 10.3 for Server with the Image Extension for Server and an ArcGIS Online organizational account. These software and system platform requirements together make up the environment necessary to build and run the new tools that have been developed for this project.

At the request of the project's client, these two new tools were developed in a custom Python toolbox with Esri's ArcPy site-package for compatibility with the existing WROS toolset. The primary requirement for the first tool, A – Prepare the Dataset, was to process a variety of input datasets in preparation to be classified, mosaicked, and the

shared as an image service via ArcGIS for Server. The requirements for this tool are documented in Table 3.1.

Table 3.1 Requirements for Process A: Prepare the Input Dataset

Process Requirement	Functional/Nonfunctional
Allow the input of vector and raster datasets of various file types.	Functional
Determine the input data type.	Functional
Convert any input vector datasets into raster datasets.	Functional
Repair input vector datasets by deleting any null geometry.	Functional
Assign a standardized cell size to output datasets.	Functional
Assist users in selecting an appropriate raster cell size based on the scale of their design projects.	Non-functional
Assist users in selecting an appropriate resampling method based on the input data type.	Non-functional
Provide an option to clip or mask the input dataset to the extent or boundary of the user's design project.	Functional
Project the input dataset to Web Mercator (as required by the GeoPlanner for ArcGIS application).	Functional
Save the output raster as a geoTIFF.	Functional
Provide well documented tool help for the user.	Non-functional
Run validation rules to confirm that required parameters have been filled out correctly.	Functional

The primary requirement for the second tool, Classify the Raster Dataset, was to help the user configure the input raster fields with the classification information needed to group, label, and rank the dataset's value ranges. The requirements for this tool are described in Table 3.2.

Table 3.2 Requirements for Process B: Classify the Raster Dataset

Process Requirement	Functional/Non-functional
Restrict tool input to only accept geoTIFF rasters.	Functional
Provide the ability to classify the input rasters by equal intervals, quantiles, or unique values.	Functional
Assist users in selecting an appropriate classification method based on their dataset and design project.	Non-functional
Provide the ability to select the number of classes that are created in the classification process.	Functional
Assist the user in selected the appropriate number of classes based on their dataset and design project.	Non-functional
Automatically label the new classes through recommended defaults and field mapping.	Functional
Provide a way for users to add optional information such as layer description, metadata, and source URL.	Functional
Produce an output XML with all the required raster field configurations.	Functional
Associate the output XML with the original raster input.	Functional
Provide well documented tool help for the user.	Non-functional
Run validation rules to confirm that required parameters have been filled out correctly.	Functional

3.3 System Design

The objective of this project was to improve the custom weighted overlay service creation workflow by developing two new tools to simplify and automate the classification of thematic rasters. These two new tools integrate with the existing WROS tools in the context of the full weighted overlay service creation workflow. A conceptual model of the overall workflow is depicted in Figure 3.1. The solid containers in the left

column represent the various datasets (input, intermediate, and output) that are pushed through the workflow. The outlined containers in the right column represent the four processes, or steps, that make up the workflow. The tools developed for this project replace the existing tools for the first of these two processes, preparing and then classifying the input dataset. The final row on the bottom shows the final step where the workflow produces a weighted overlay service that is ready to be consumed in the GeoPlanner application. Following the workflow's conceptual model diagram, each of the processes is modeled and described in greater detail.

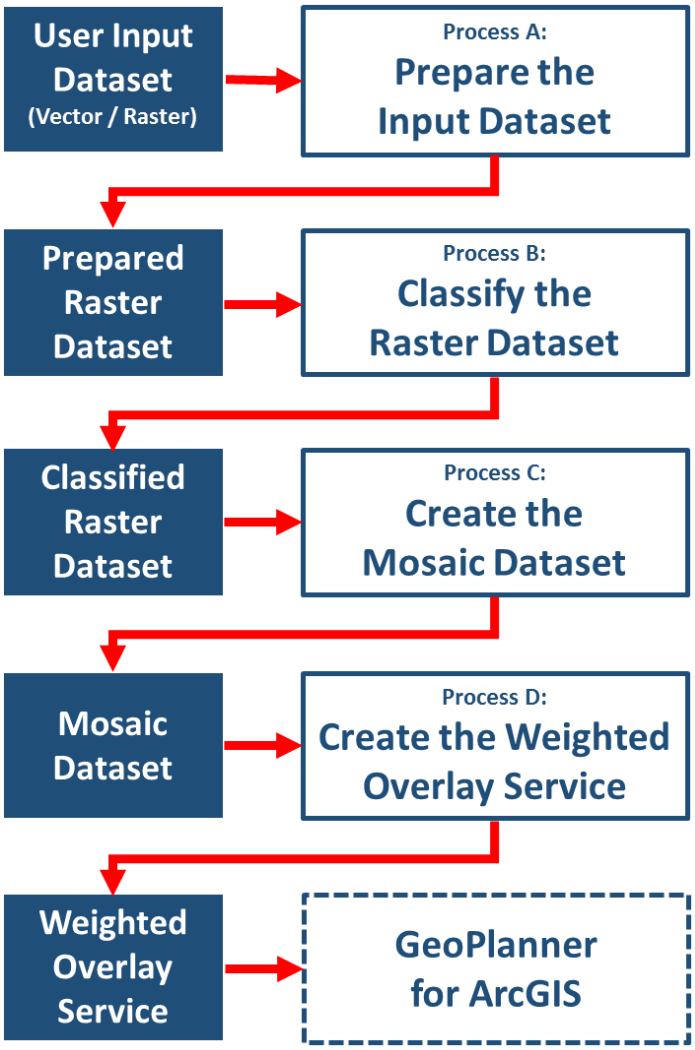

Figure 3.1 Weighted Overlay Service Creation Workflow

3.3.1 Prepare the Input Dataset

The first process in the workflow prepares the input dataset to be classified and ensures that it meets the data format, metadata, and raster field configuration requirements for a weighted overlay service. The individual steps in this process utilize ArcGIS for Desktop's geoprocessing capabilities and include projecting the dataset to a common projection, clipping the it to the extent of the design project to reduce file size, converting any vectors into a raster, resizing raster cells if needed, addressing any NoData values or data gaps, and saving the output as a geoTIFF. This process was simplified by combining the individual steps in the new Prepare the Input Dataset tool. The process and its steps are mapped out in Figure 3.2.

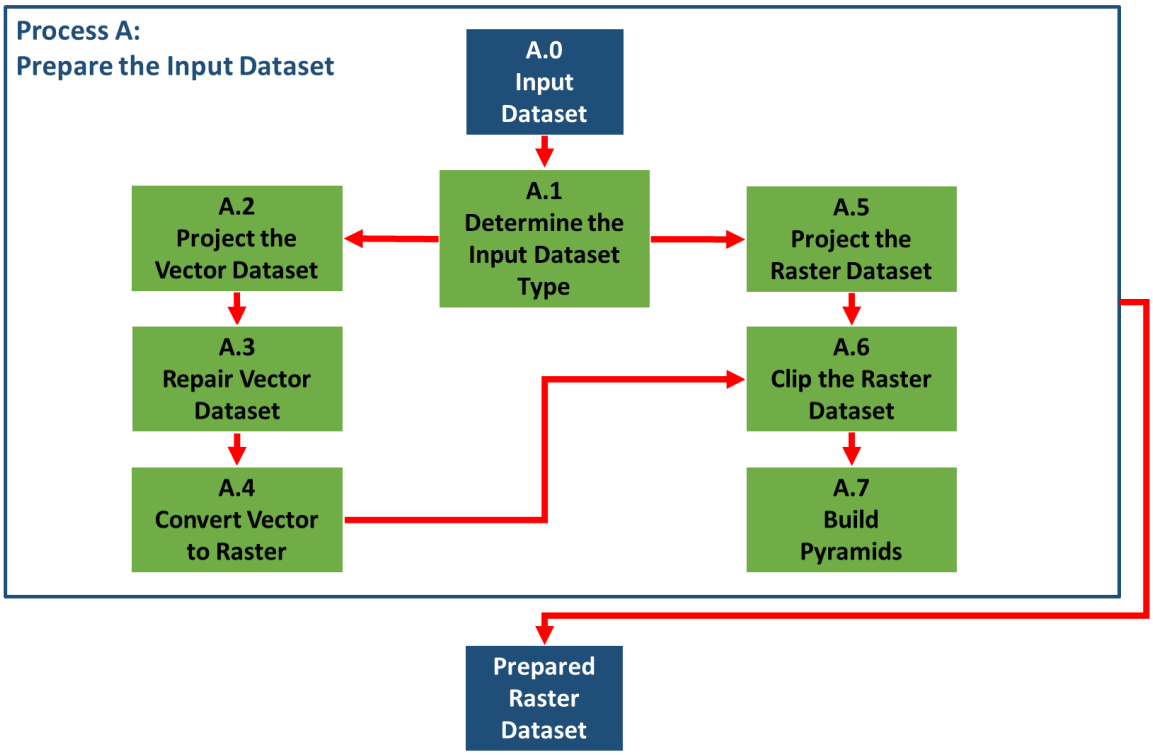

Figure 3.2 Process A: Prepare the Input Dataset Tool

3.3.2 Classify the Raster Dataset

When the source dataset has been cleaned up, the prepared raster needs to be classified with ranks and labels assigned to each new class in preparation for it to be converted into a weighted overlay service. In the second workflow process, the first step is to determine whether the input raster's values are nominal or continuous. Based on this data type, a classification method is recommended – Equal Intervals, Quantiles, or Unique Values. After the classification method has been chosen, the number of classes to be created is selected. With the classification method and number of classes defined, the raster is classified by calculating the raster statistics, break values, and class ranges. The next step adds the class labels and default ranks to these new classes, and in the final step all of this information is written to an XML file that is associated with the input raster. This process was simplified and automated by combining and refining the individual steps in the new Classify the Raster Dataset tool. The process and its steps are mapped out in Figure 3.3.

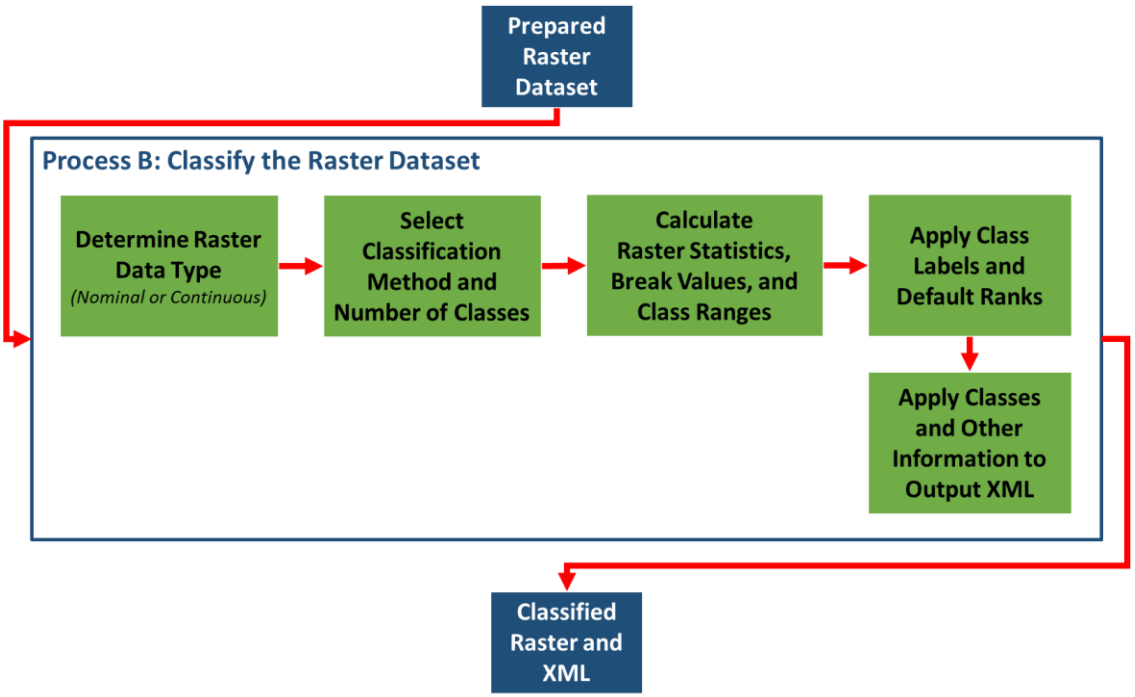

Figure 3.3 Process B: Classify the Raster Dataset Tool

3.3.3 Create the Mosaic Dataset

The next process in the weighted overlay service creation workflow is to build the mosaic dataset. All of the prepared rasters, outputs from the first process, to be included in the weighted overlay service are input into the Convert Raster to a Mosaic Dataset tool, an existing tool in the WROS toolset. This process combines the rasters, ensures that the dataset metadata has been formatted correctly, and produces a copy of the data for backup storage on the user's local machine or network (Figure 3.4).

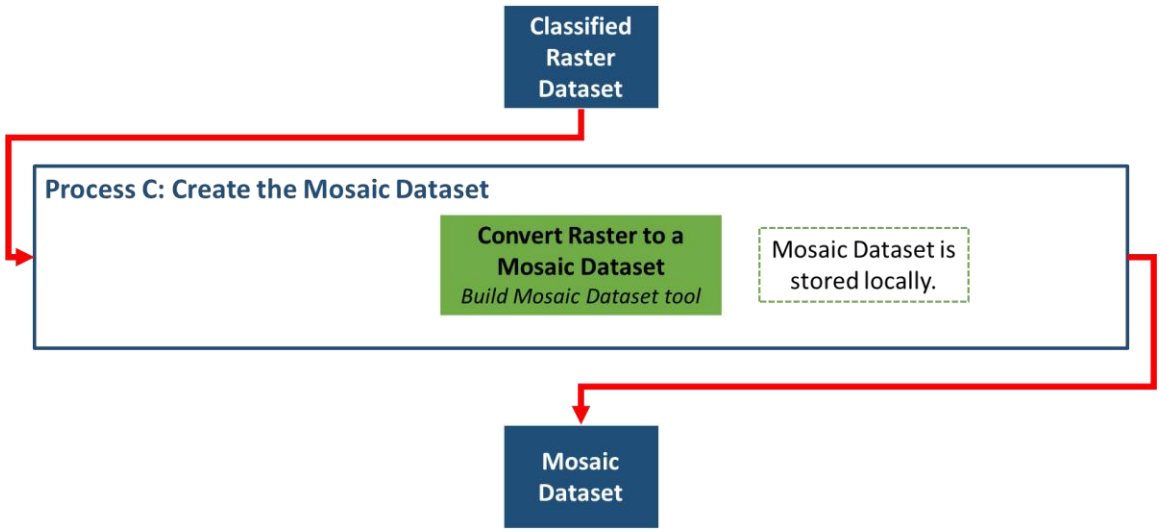

Figure 3.4 Process C: Create the Mosaic Dataset

3.3.4 Create the Weighted Overlay Service

The final process in the workflow is to make the mosaic dataset available on ArcGIS Online as a weighted overlay service. The mosaic dataset is first published as an image service through ArcGIS for Server. This image service is then added to an ArcGIS Online for Organizations account. Metadata tags added in the classification and mosaic dataset creation processes allow this image service to be consumed as a weighted overlay service in GeoPlanner for ArcGIS. Figure 3.5 shows the final steps in the workflow – the mosaic

dataset is uploaded to the ArcGIS Online platform and made available inside the user's organization for creating new weighted raster overlay models in GeoPlanner.

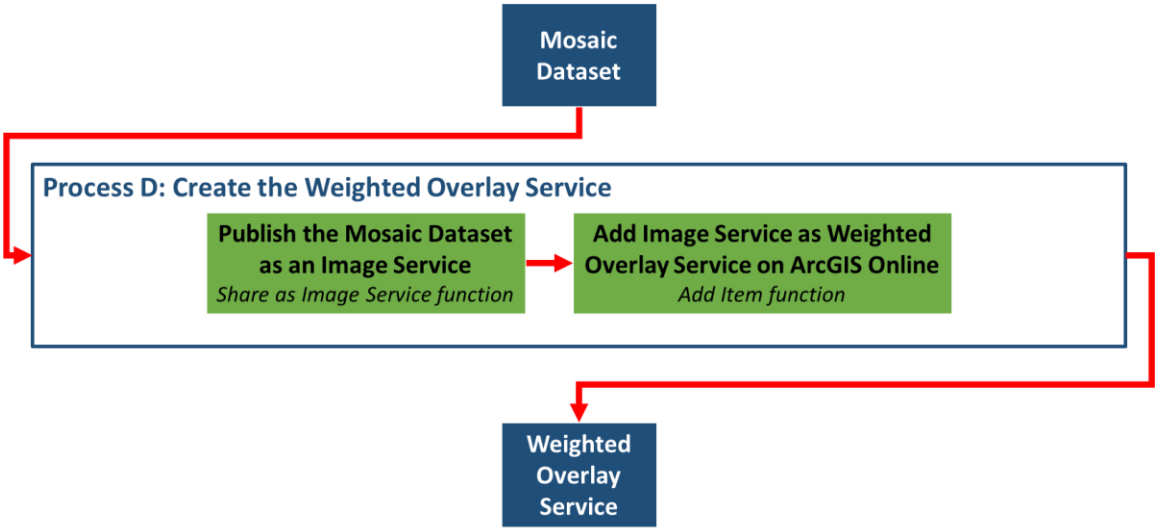

Figure 3.5 Process D: Create the Weighted Overlay Service

3.4 Project Plan

The central challenge of this project was the development of two new custom Python tools. Based on this goal, a software development approach was adopted, and the project plan followed a general framework focused on three distinct phases – design, develop, and deploy.

3.4.1 Design Phase

The first phase of this project was the design phase. During this phase the general outline of the work was defined and the work breakdown structure table was created. The design phase included the following tasks:

Task 1 - Conduct background research on the history of land suitability assessment, the application of weighted overlay analysis, and the theory and methodology behind thematic raster classification.

Task 2 – Gather and define the project requirements through client meetings and feedback from GeoPlanner users. Map out the current GeoPlanner workflows and become more familiar with the Weighted Raster Overlay Services toolset.

Task 3 - Design an automated workflow and user interface that includes all of the requirements and the new thematic raster classification recommendations.

3.4.2 Develop Phase

The next phase of the project was the develop phase. This phase was used to build the new custom Python tools for the first two processes of the weighted overlay service creation workflow. This included all of the coding and integration into both the existing WROS toolset and the ArcGIS platform. Specifically, the develop phase included the following tasks:

Task 1 - Develop a web-based workflow in Python that processes input datasets by converting them into a geoTIFF format and then adds the geoTIFF raster to a mosaic dataset.

Task 2 – Build and integrate a mosaic dataset configuration script into the workflow to configure the required parameters and add them to the output.

Task 3 - Automate the publication of newly configured mosaic datasets to ArcGIS Online as a weighted overlay service.

Task 4 - Develop a solution that recommends thematic raster classifications based on the data type and geodesign objectives.

3.4.3 Deploy Phase

The deploy phase was the final phase in the project plan. In this phase, the final solution required extensive testing before it could be approved by the client and integrated into the GeoPlanner application's WROS toolset. The new tools were initially be tested with synthetic data created for the purpose of verifying the functionality of the solution. After testing with synthetic data, further testing with real-world sample data from the project client and other sources confirmed that the final solution successfully met the project's objectives and the client's expectations. The results of this testing was documented to record any errors, bugs, or needed enhancements. Based on this documentation, revisions and fixes were be developed and added to the tools. The following tasks are associated with the deploy phase:

Task 1 – Test each tool independently; then test the tools together; and at the end, test the complete workflow with synthetic data.

Task 2 – Test each tool independently; then test the tools together; and at the end, test the complete workflow with sample data provided by the client.

Task 3 – Document bugs and enhancement requests, prioritize these tasks, and develop fixes and resolutions them.

Task 4 – Deliver the tools, as the final solution, to the client and then integrate it into the existing GeoPlanner code as a functionality update.

Task 5 – Create and deliver product documentation for both the end users and the client's team.

3.4.4 Work Breakdown Structure

As part of the design phase, a work breakdown structure was created to document the phases of the project plan, the tasks associated with each phase, and the estimated time each task would require (Table 3.3).

Table 3.3 Work Breakdown Structure

	Task Title	Start Month	End Month	Labor Hours
1	**Design Phase**			
1.1	Research land suitability assessment, weighted overlay analysis, and classification methods.	1	2	40
1.2	Define project requirements and map the weighted overlay service creation workflow.	1	2	20
1.3	Design an automated workflow and user interface that includes raster classification tools.	2	3	20
2	**Develop Phase**			
2.1	Develop a web-based workflow in Python that processes and classifies input datasets.	3	4	40
2.2	Build and integrate a mosaic dataset configuration script into the workflow.	4	5	60
2.3	Automatically publish new mosaic datasets to ArcGIS Online as weighted overlay services.	4	5	20
2.4	Build a solution that recommends thematic raster classifications.	4	6	60
2.5	Develop a user interface for the entire workflow.	5	6	20
3	**Deploy Phase**			
3.1	Test the solution and workflow with synthetic data.	6	6	8
3.2	Test the solution and workflow with client supplied “real life” data.	6	6	8
3.3	Deliver the solution to the client and integrate the solution into the existing GeoPlanner code as a functionality update.	6	7	20

3.4	Create and deliver product documentation for the end user that describes how to use the new functionality.	4	7	20
3.5	Create and deliver backend documentation for the client that describes how the solution was designed and how code is organized.	4	7	20

3.4.5 Project Plan Analysis

Over the course of the project, the project plan and work breakdown structure were useful in providing a framework for the tasks that needed to be completed and the general order and priority these tasks should hold in relation to each other. However, from the beginning, the project did not proceed as initially expected. Some of the changes were predictable; for example, under estimations of the time and effort required to finish certain tasks led to compounding tasks as new work was started before previous tasks were fully completed. The most significant impacts on the original project plan came from developing a better understanding of the client's needs, the limitations of the ArcGIS and GeoPlanner platforms, and the experiences of learning to code in a new scripting language. This added experience, insight, and information required changes to the original project plan – the timeline was extended, the tool requirements were modified, and the expectations for the final deliverable changed. While this presented a number of challenges, it resulted in the delivery of a final solution that works with the existing workflow and better meets the client's and end-users' needs.

3.5 Summary

The objective of the project was to improve the existing workflow for creating new weighted overlay services to be used in the GeoPlanner for ArcGIS application. An assessment of the workflow found that two of the four processes in the workflow were

too difficult and complex for the average end-user. A solution was proposed to address these issues, and a project plan was created that provided a framework for the development of two new tools that would simplify and automate the most challenging aspects of the weighted overlay service creation workflow. This project plan followed the general guidelines of software development and was divided into design, development, and deployment phases. Throughout the course of the project, the project plan evolved as new information and experience necessitated changes to the originally proposed solution. At the conclusion of the project, two new tools that met the client's needs and expectations were delivered.

Chapter 4 – Database Design

In this chapter, the database design of the project is reviewed. Section 4.1 describes the conceptual data model. Section 4.2 covers the logical data model. Section 4.3 describes the data sources. Section 4.4 reviews the data collection sources. Section 4.5 describes the data clean up processes. And the chapter concludes with a summary of these efforts in Section 4.6.

4.1 Conceptual and Logical Data Models

The efforts of this project were primarily engaged in the development of two new custom Python tools. These two tools were designed to integrate into a weighted overlay service creation workflow as part of a complete geographic information system As such, there was no data component in the final solution or included in the client deliverables. However, as the two tools were required to prepare and classify input datasets, a data model was required for the synthetic and real-world data that was used in the development and testing of the tools' functionality.

The Prepare the Input Dataset tool's conceptual data model defines the input dataset, its influence on the processing steps, the changes to the dataset at each step, the output raster dataset, and the required characteristics at each step in the process. The tool's input accepts both vector and raster datasets. Depending on the data type, vector or raster, the input dataset is processed through a series of steps that transform it into an output raster with uniform characteristics that match the other raster datasets to be included in the mosaic dataset (Figure 4.1).

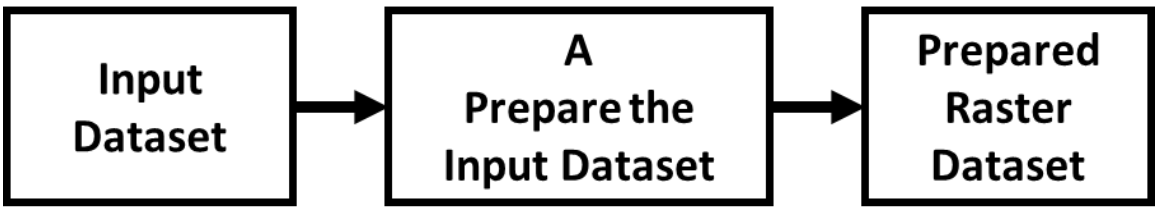

Figure 4.1 Prepare the Input Dataset Tool - Conceptual Data Model

The output of the process is a prepared raster dataset with the following characteristics:

- Saved in a geoTIFF format.
- Projected to Web Mercator (Auxiliary Sphere).
- Clipped to the design project's boundaries or extent.
- Resampled to a uniform cell size.
- Optimized for performance by building pyramids and removing features with no associated spatial data.

These characteristics are required for each raster that is added to the weighted overlay service because each raster layer in the service must align with every other raster layer to produce accurate results. The tool does not modify the original input dataset. The tool creates a new dataset as its output.

The Classify the Raster Dataset tool's conceptual data model defines the input datasets, the assessment and classification process, and the relationship between the output file and the input dataset. The raster prepared by the first tool is the input for this tool. The cell values of this raster are used to calculate new classes. For the output, these classes, and other information, are written to a new file that is associated with the input raster. The conceptual data model for this process is displayed in Figure 4.2.

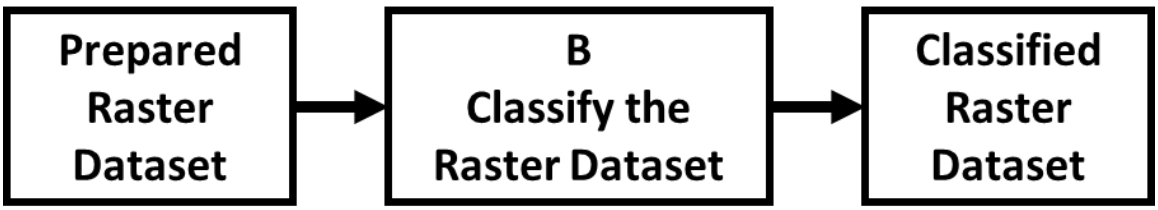

Figure 4.2 Classify the Raster Dataset Tool - Conceptual Data Model

The output file must include the following information:

- File name of the associated raster dataset
- Input ranges of the new classes
- Output values, or weights, for each of the new classes
- Range labels for each of the new classes
- Description of the dataset

The prepared raster dataset from the first tool and the associated classification information file created by the second tool are stored together on the user's system. Both files must be kept together and be stored in a location accessible to the mosaicking tool used in the third process.

The logical data model for this project was designed to accommodate a wide range of file types for testing data. The top level file folder, labeled "Data", contains file geodatabases, various raster image files, shapefiles, and other data types. Each geodatabase represents datasets collected for specific design scenarios, such as the South Campus Train Station Land Planning project, a new water treatment facility suitability study, landform classification sample data, and so forth. These geodatabases include both vector and raster datasets as well as other sample data, such as annotation, networks, and tables. The raster images and shapefiles represent many different file formats, pixel depths, cell sizes, projections, data types, extents, and NoData value ranges.

4.2 Data Sources

A wide variety of synthetic sample datasets and real-world datasets were used in the development and testing of the Prepare the Input Dataset and Classify the Raster Dataset tools. These datasets were acquired from a number of sources including the project client, the University of Redlands Geodesign Studio, the City of Redlands, USGS image services, and ArcGIS Online. The objective was to collect a representative sampling of the various file formats, data types, and raster themes that are typically used in weighted raster overlay analyses. The datasets included in Table 4.1 are a sample of the inputs used in the development and final acceptance testing of these new tools. Note the variety of data types, formats, pixel depths and pixel types.

Table 4.1 Project Datasets and Characteristics

ID	Name	Data Type	Format	Nominal/ Continuous	Pixel Depth and Type
1	soiltype.tif	raster	File System Raster -TIFF	Nominal	8-bit Signed integer
2	inyo_landcvr	raster	File System Raster - GRID	Nominal	8-bit Unsigned integer
3	LandformClassification	raster	File Geodatabase Raster	Continuous	16-bit Signed integer
4	whitelev_20	raster	File System Raster - GRID	Nominal	8-bit Unsigned integer
5	whitelev_int	raster	File System Raster - GRID	Continuous	16-bit Signed integer
6	RedlandsTerrain.tif	raster	File System Raster -TIFF	Continuous	32-bit Floating point
7	Zoning.tif	raster	File System Raster -TIFF	Nominal	32-bit Signed integer
8	SoilTypes	vector	File Geodatabase Feature Class	Nominal	N/A
9	SouthCampus_Trees	vector	File Geodatabase Feature Class	Nominal	N/A

10	GeneralPlanLines	vector	File Geodatabase Feature Class	Nominal	N/A
11	RedlandsLandUse.shp	vector	Shapefile	Nominal	N/A
12	AnalysisArea.shp	vector	Shapefile	Nominal	N/A

The data sources for GeoPlanner users varies depending on the design scenario, project scope, and organizational resources. Dataset characteristics such as scale, projection, and format are subject to each user's needs. Because a weighted overlay service is composed of multiple dataset layers, it is important that every dataset included in the model shares the same projection and that scale, resolution, and format have been considered in the workflow. The weighted overlay service creation workflow has been designed to accept a wide range of input dataset. To accommodate this flexibility, these two new tools also have to able to accept a wide range of input datasets. While there are data type restrictions in place, the most commonly used data formats can be input and processed by the Prepare the Input Dataset tool.

4.3 Data Collection Methods

Collecting a large and diverse assortment of datasets was key to satisfying some of the project's core objectives. These datasets were collected from a variety of sources. The project client provided a number of sample datasets that represent typical inputs for the weighted overlay creation workflow. To further expand the collection, datasets were sourced from geodesign related projects undertaken by the faculty and students of the University of Redlands Masters of Science in GIS program. This effort was completed by identifying, searching, and obtaining a subset of more obscure datasets with uncommon characteristics and formats. The full complement of datasets was used extensively in the

development and testing of the Prepare the Input Dataset and Classify the Raster Dataset tools.

4.4 Data Scrubbing and Loading

In many cases, data obtained from external sources require some level of modification or clean up before they can be included in the primary database. For the purposes of this project, however, dissimilar datasets were preferred. The design of the new tools included a requirement that they be able to process a wide range of dataset types. Datasets that returned exceptions or errors when run through the tools were not cleaned up; they were used as a standard to meet. In a few cases, datasets that initially worked perfectly with the tools were modified in ways that were intended to break or crash the tools. The process of assessing the properties and attributes of each dataset played a significant role in the latter stages of the tool development as enhancements were added to make the tools more robust and flexible.

4.5 Summary

The success of this project was related to the collection of various datasets that were used in the development and testing of the Prepare the Input Dataset and Classify the Raster Dataset tools. The project's database was designed to meet the needs of the workflow as described and modeled in this chapter. The datasets were collected from the project client, resources in the Masters of Science in GIS program, and external sources related to geodesign and land-use planning. Modeling the workflow processes and creating a project database informed many of the design decisions made in the software development phase of this project and ensured that the final tools met the client's needs.

Chapter 5 – Implementation

Chapter 5 describes the implementation of the project and the development of two custom Python tools. These tools were built to facilitate the first two processes in the weighted overlay service creation workflow - Prepare the Input Dataset and Classify the Raster Dataset (Figure 5.1). The two tools were developed in Python 2.7 with the ArcPy site package. To improve the workflow processes, a number of individual steps were simplified by adding validation rules, default best practices, and expert recommendations to automate the new tools. The process for developing each of these tools and their functionality is covered in detail. The decisions and rational behind the automated components of these tools is also presented.

Figure 5.1 Input Dataset Processing through the Project Tools

5.1 Prepare the Input Dataset

The Prepare the Input Dataset tool prepares the user's input dataset for the raster classification process. This tool produces an output raster that is correctly projected and repaired and then clipped to the extent of the design scenario with a defined cell size and pyramids to improve performance. The Prepare the Input Dataset tool was designed to

simplify the processing of multiple datasets by making it easy to standardize the output dataset's characteristics and attributes.

There are six parameters for the tool: Input Dataset, Select a Cell Size, Select a Raster Value Field, Select a Resampling Method, Select a Mask for the Design Scenario, and Output Raster Name and Location (Figure 5.2). The input, output, and cell size parameters are required, and the clipping mask parameter is optional. The Select a Raster Value Field and Select a Resampling Method parameters are enabled and disabled depending on the input data type.

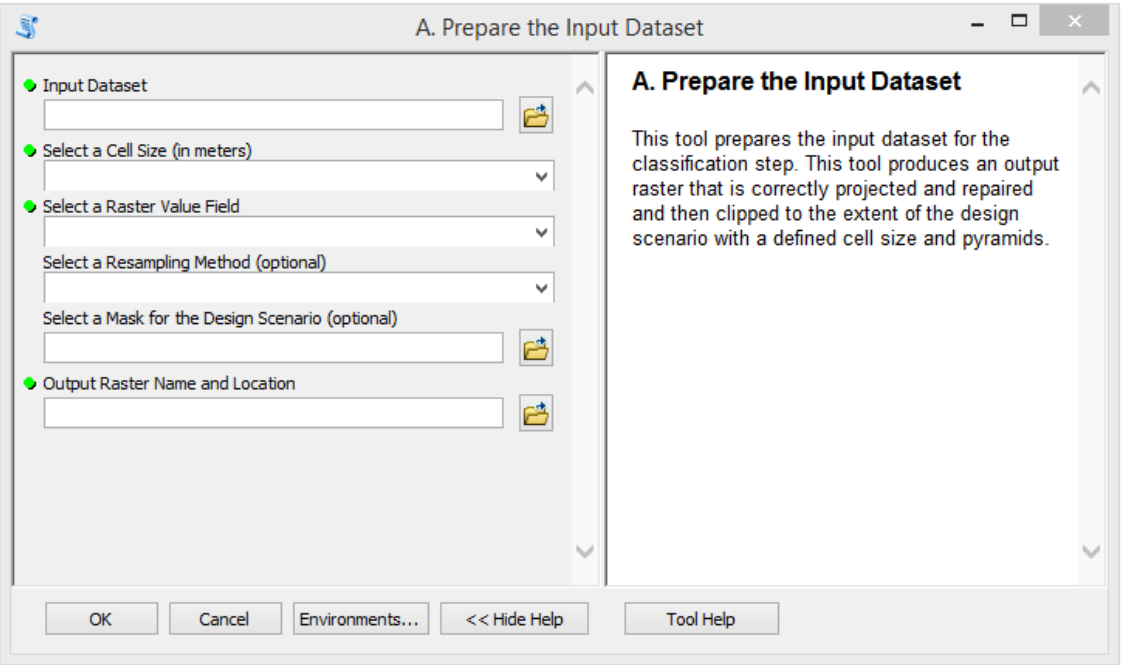

Figure 5.2 Prepare the Input Dataset Tool

5.1.1 Step 1 – Input Dataset

The Prepare the Input Dataset tool has been configured to accept a range of inputs. These inputs include raster datasets and feature datasets. Because this tool is built on the ArcGIS platform, the supported raster data formats are dependent on Esri's extensive raster list. These raster formats include: Bitmap (*.bmp), Standard Raster Product

(*.img), File Geodatabase (*.gdb), Graphic Interchange Format (*.gif), Joint Photographic Experts Group (*.jpg or *.jpeg), Portable Network Graphics (*.png), and many more. For the vector type inputs, the supported feature datasets include shapefiles and feature classes with polygon and line geometry.

When the tool runs, the first step is to determine if the input is a vector or raster by using the arcpy.Describe() function. If the input is a feature class, the same function is used to verify that the specific data type is supported (polygon or line data). For example, if a vector with point geometry is input, the validation step returns an error message, "Input feature classes must have line or polygon geometry." It then instructs the user to modify the input dataset. Depending on the input data type, the Select a Raster Value Field and Select a Resampling Method parameters are either enabled or disabled (Figure 5.3). The Select a Raster Value Field parameter is only enabled when the input is a vector dataset. And the Select a Resampling Method parameter is only enabled when the input is a raster dataset.

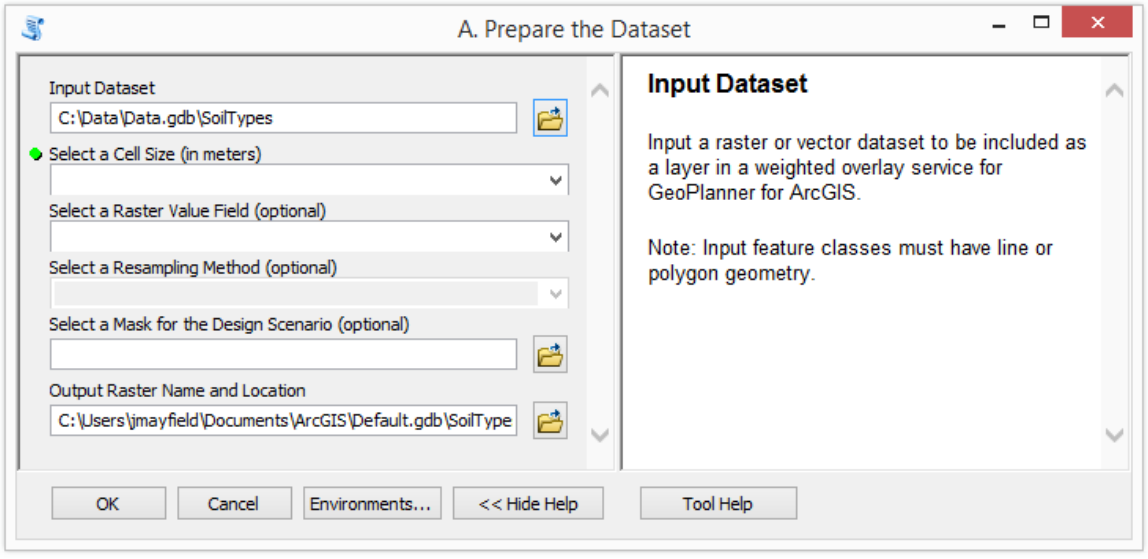

Figure 5.3 Parameter 1: Input Dataset

5.1.2 Step 2 – Project the Input Dataset

After the input dataset from the first parameter has been described and validated, the tool then projects the dataset. This task is performed for both vector and raster input datasets using the arcpy.Project_management() and arcpy.ProjectRaster_management() functions respectively. This task is run before any other geoprocessing tasks to minimize distortion and errors that could be introduced by running it later in the dataset preparation process.

As an application on the ArcGIS Online platform, the GeoPlanner for ArcGIS requires all datasets, including weighted overlay services, to be projected in WGS 1984 Web Mercator (Auxiliary Sphere). Requiring any one projection to be used in all cases is problematic, and the Web Mercator projection, in particular, presents a unique set of challenges including increasingly exaggerated distortions moving from the equator toward the poles. This is a known limitation of the GeoPlanner application and the results of the application's weighted raster overlay analysis models. The client has deemed this as an acceptable limitation because GeoPlanner has been positioned and designed as a tool for rapid and iterative design, analysis, and evaluation. Additionally, weighted raster overlay analysis, by its nature, is not a precise tool. It incorporates many subjective value judgments and produces naturally fuzzy results. Initial ideas and designs created and proposed in GeoPlanner are typically recreated by a GIS professional in ArcMap or ArcGIS Pro with more exacting standards.

For input raster datasets, the projecting process step also requires resampling the raster grid cells to match the selected cell size. The Prepare the Dataset tool provides two resampling options, Nearest Neighbor and bilinear, as the fourth parameter in the tool's user interface (UI). When a raster is entered as the input for parameter 1, the Select a Resampling Method parameter is enabled, and the Nearest Neighbor option is selected by

default (Figure5.4). The nearest neighbor resampling method assigns cell values to the output raster based on the value of the corresponding closest cell from the input raster. This method does not create any new values. It is selected by default because it is suitable for both nominal and continuous datasets. The bilinear resampling method interpolates cell values by averaging the values of the four nearest cell centers. The Bilinear resampling method can be used for continuous datasets.

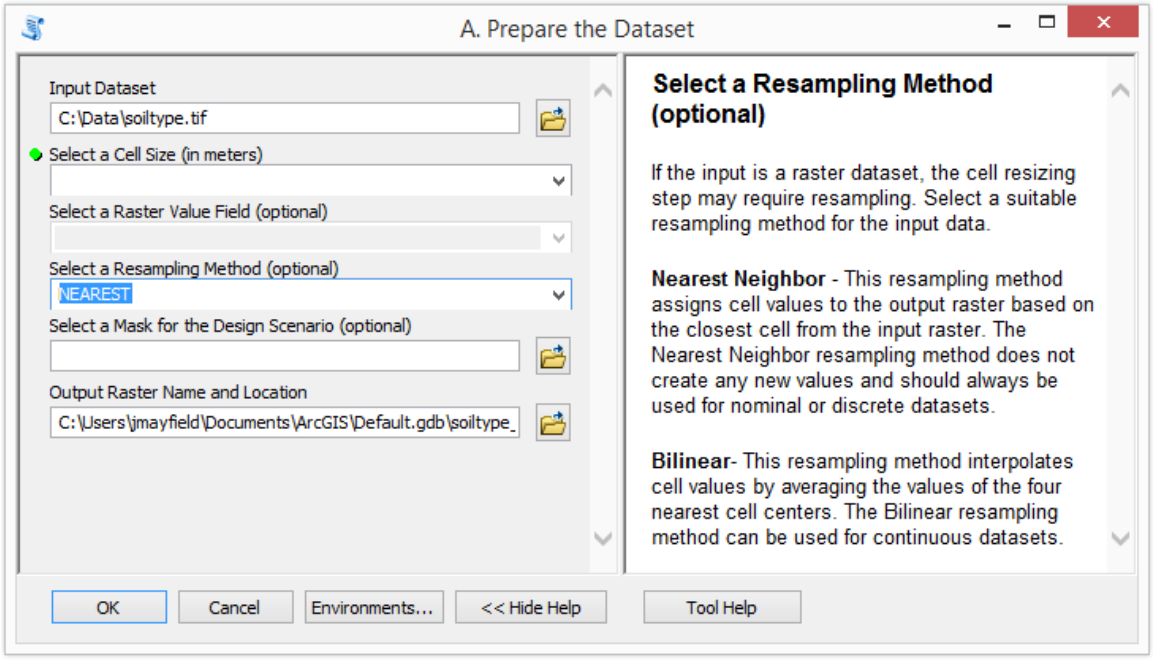

Figure 5.4 Parameter 4: Select a Resampling Method

5.1.3 Step 3 – Convert Vector to Raster

After the input vector dataset has been projected, it is almost ready to be converted to a raster. Prior to rasterization, the tool cleans up the vector dataset by deleting any features with null geometry through the arcpy.RepairGeometry_management() function. When the vector dataset has been prepared, the vector is converted into a raster. This step requires two parameter inputs from the user: a cell size and a raster value field.

The cell size for the new raster to be created is selected as the second parameter in the tool's UI (Figure 5.5). The raster's cell size plays an important role in both the process of creating weighted overlay services and in the quality of the weighted raster overlay analysis. All of the raster layers included in a weighted overlay service must have cells of the same size. If the cell sizes are not consistent with each other, then the data values will not line up and the resulting analysis model would be subject to flaws and error. The Prepare the Input Dataset tool is designed to help ensure that all the raster layers can be processed to have the same cell size.

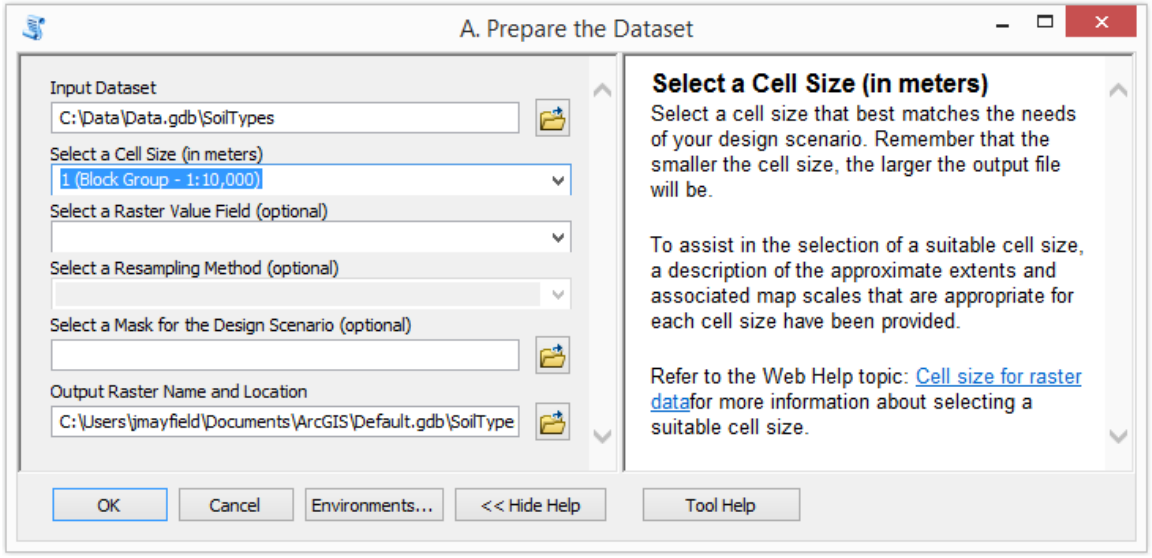

Figure 5.5 Parameter 2: Select a Cell Size

The raster value field is selected as the third parameter in the tool's UI (Figure 5.6). This parameter is only needed for vector inputs and is disabled for raster inputs. For this parameter, the user is instructed to select the attribute field that should be used to assign values to the raster cells during the vector to raster conversion step. Because various input datasets represent a wide range of phenomena and there is no standard for field

names, this parameter cannot be automatically selected by the tool; the user must select which field should be used to assign values in the new raster grid.

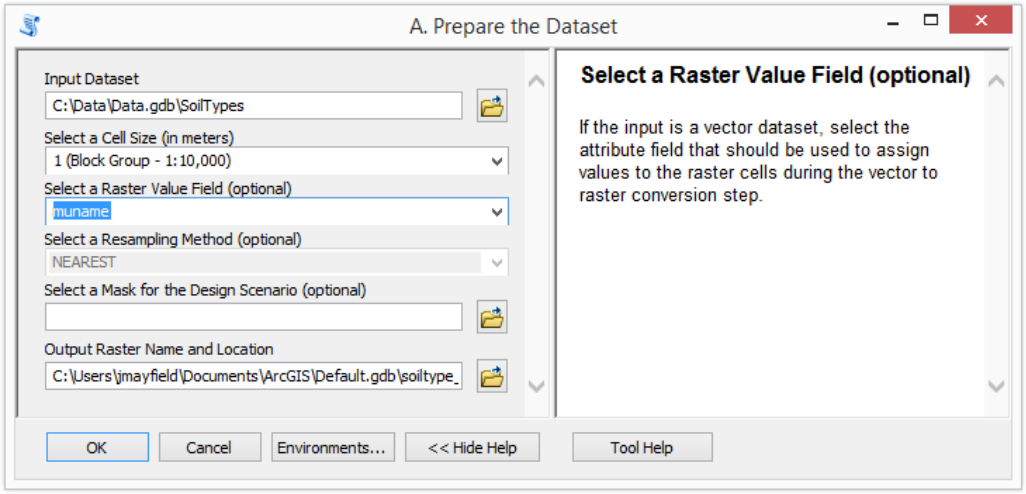

Figure 5.6 Parameter 3: Select a Raster Value Field

5.1.4 Step 4 – Clip the Area of Interest

The Prepare the Input Dataset tool is designed to process datasets from a variety of sources. When datasets come from different sources, it is uncommon for them to have the same boundaries or extent – a requirement for each of the layers in a weighted overlay service. The tool accommodates these discrepancies by providing a way for the user to clip or mask the raster to a defined area of interest. If the input dataset needs to be clipped to a boundary or extent, a feature or raster layer that defines the area of interest for the design scenario can be added as the fifth parameter (Figure 5.7). This layer is used in the arcpy.sa.ExtractByMask() function to remove any cells outside the defined boundary. This step also includes building raster pyramids to improve the processing performance throughout the weighted overlay service creation workflow.

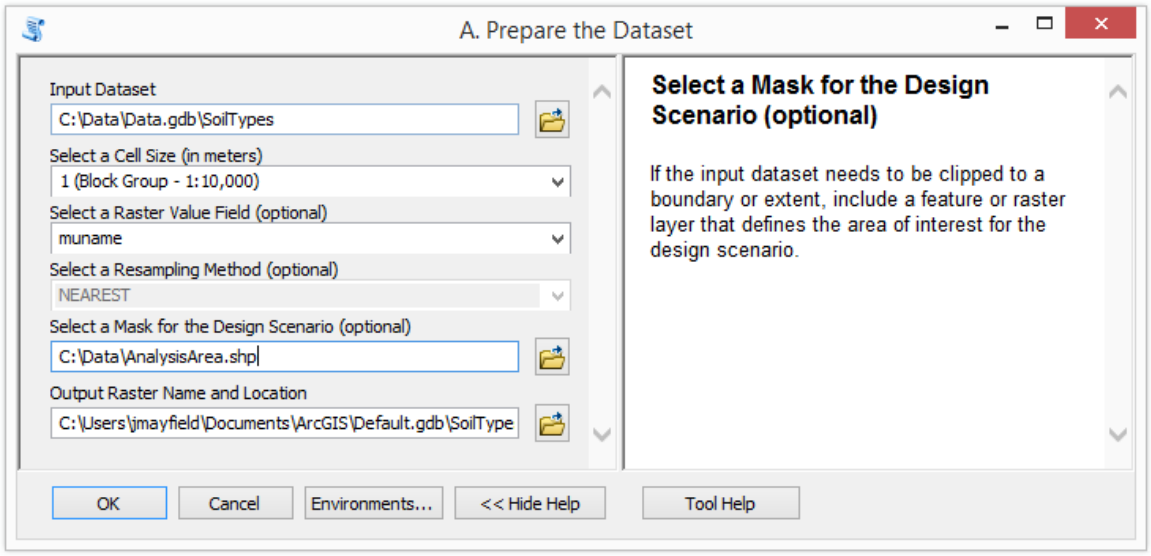

Figure 5.7 Parameter 4: Select a Mask for the Design Scenario

5.1.5 Step 5 – Output the Prepared Raster

After the input dataset has been described, projected, rasterized or resampled, cells re-sized, and clipped, it is ready for the next process in the weighted overlay creation workflow. The final step in this tool is to save the prepared raster as a geoTIFF. In the last parameter, the user is instructed to select the destination file location for the output raster, and give the output a unique name (Figure 5.8). The output must be saved as a geoTIFF, so the name must end with the ".tif" suffix. A validation rule checks against this requirement and prevents the tool from running until the output meets the criteria.

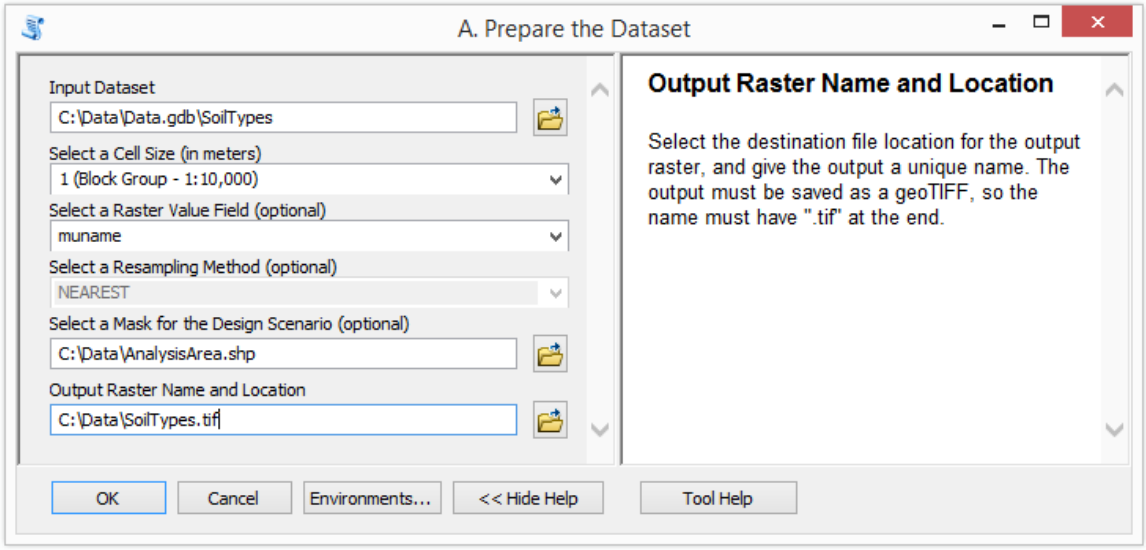

Figure 5.8 Parameter 6: Output Raster Name and Location

The Prepare the Input Dataset tool is designed to walk the user through a series of data processing steps that are needed to prepare the input dataset for the next process in the weighted overlay service creation workflow – classifying the raster dataset.

5.2 Classify the Raster Dataset

The Classify the Raster Dataset tool classifies the input raster dataset and adds the weighted overlay data required by the GeoPlanner for ArcGIS application. The output of this tool is written to an XML file that is associated with the input raster for further processing in the Create a Mosaic Dataset process of the weighted overlay service creation workflow. The Classify the Raster Dataset tool was designed to simplify and automate the process of selecting a classification method, calculating the new classes, labeling and weighting the classes, and then recording that information in a properly formatted XML file.

There are seven parameters for the tool: Input Raster, Classification Method, Number of Classes, Field for Class Labels, Description, Metadata, and URL (Figure 5.9). The

input, classification method, number of classes, and the class label field are required parameters, and the description, metadata, and URL parameters are optional. Depending on the input data type, nominal or continuous, the second, third, and fourth parameters have backend logic in place to provide recommended defaults to the user.

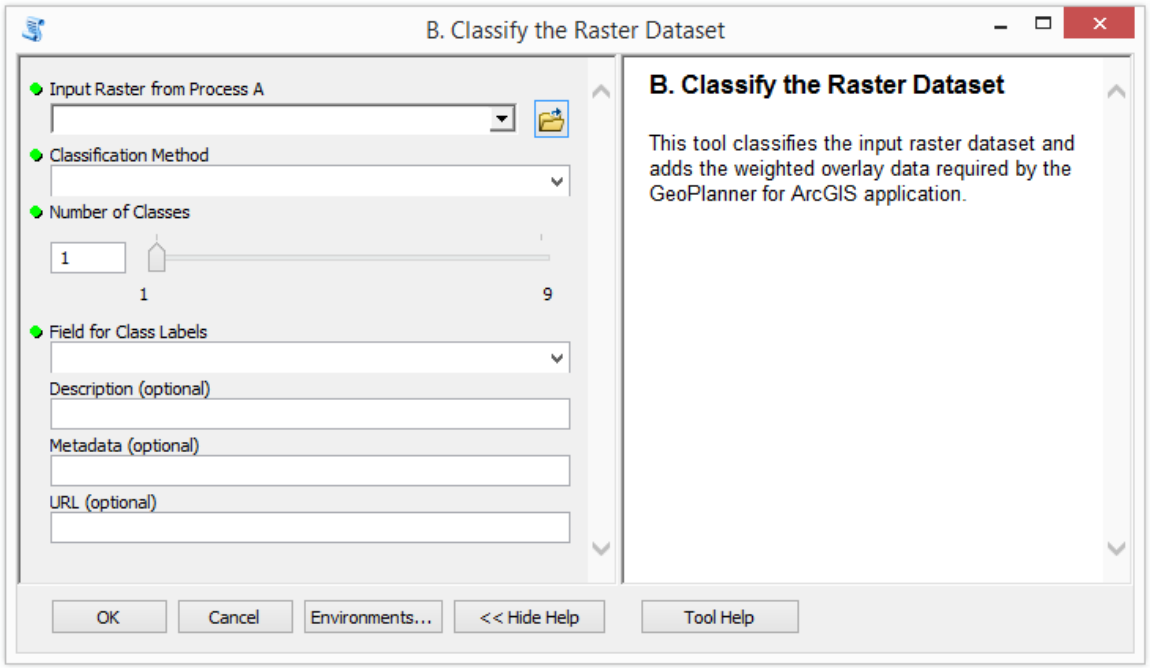

Figure 5.9 Classify the Raster Dataset Tool

5.2.1 Step 1 – Input Prepared Raster Dataset and Determine Raster Type

The first step of the Classify the Raster Dataset tool is to input the prepared geoTIFF raster that was produced previously by the Prepare the Input Dataset tool. Because the data type for this parameter is set to Raster Layer, this first field only accepts raster files (Figure 5.10). Additionally, when a raster is added to this parameter, validation rules confirm that the input is a geoTIFF and that it is in the Web Mercator (Auxiliary Sphere) projection. Input rasters that do not meet this criteria return this error message: “The input raster must be a TIFF. Run this dataset through the 'Prepare the Dataset' tool to

prepare it for this tool." This ensures that the outputs of this tool will be compatible with the other layers in the mosaic dataset and with the weighted overlay service requirements.

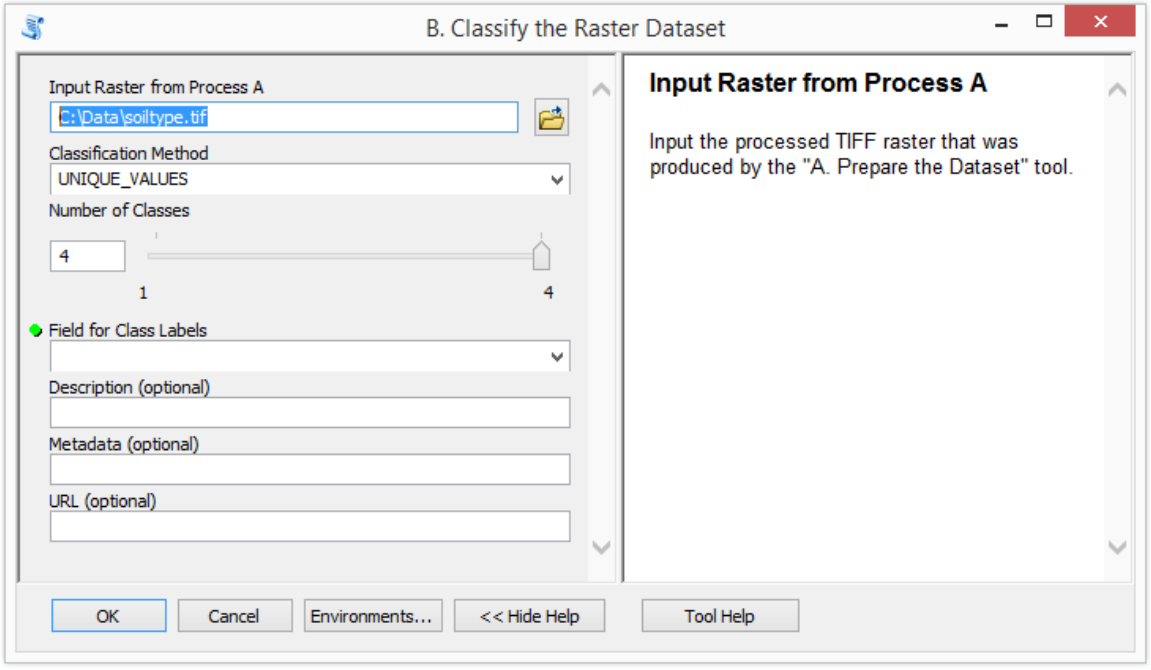

Figure 5.10 Step 1: Input the Prepared Raster

After the raster has been added by the user and validated by the tool, a raster object is created in Python to both enable raster statistics calculations and to improve processing performance. The tool then determines the pixel type and depth of the raster by running the arcpy.GetRasterProperties_management() function, which returns the "VALUETYPE" property. The raster value type is a combination of the pixel type (e.g., unsigned, signed, or complex) and the pixel depth (e.g., 8-bit, 16-bit, or 32-bit). From this value type information, the tool determines whether the input raster represents nominal or continuous data based on the most common application of each specific pixel type and depth combination. For example, an 8-bit unsigned raster would be considered a nominal raster type, and a 32-bit floating point double precision raster would be considered a

continuous raster type. This raster type is used in the following steps to recommend classification methods and the number of classes to be created.

5.2.2 Step 2 – Classify the Raster Dataset

The classification method used to group the input raster's values has a significant impact on the final weighted overlay analysis. The Classify the Raster Dataset tool has been designed to improve the weighted overlay service creation workflow in a number of ways. First, the tool clearly defines and describes each available classification method by outlining the benefits, drawbacks, and their typical applications. Second, the tool automatically recommends a classification method based on the raster type. Third, the tool streamlines the classification process to facilitate user testing and experimentation by iterating between classification methods and the number of classes used. This allows the user to determine which classification method best suits the needs of the design project. Based on the scope of this project, there are three classification methods available in the tool: equal intervals, quantiles, and unique values. Other classification methods, such as standard deviation or Jenk's natural breaks, can also provide useful analytical perspectives but were not included in the scope of this project.

Classifying the raster by equal intervals groups the cell values into classes that contain an equal range of values. This method is good for showing different classes when there are not great differences between most of the values. This type of classification method is straightforward and relatively easy to interpret. When the input is a continuous raster, the tool automatically sets the classification method parameter to "Equal Interval" and the number of classes to "5" by default. (Figure 5.11).

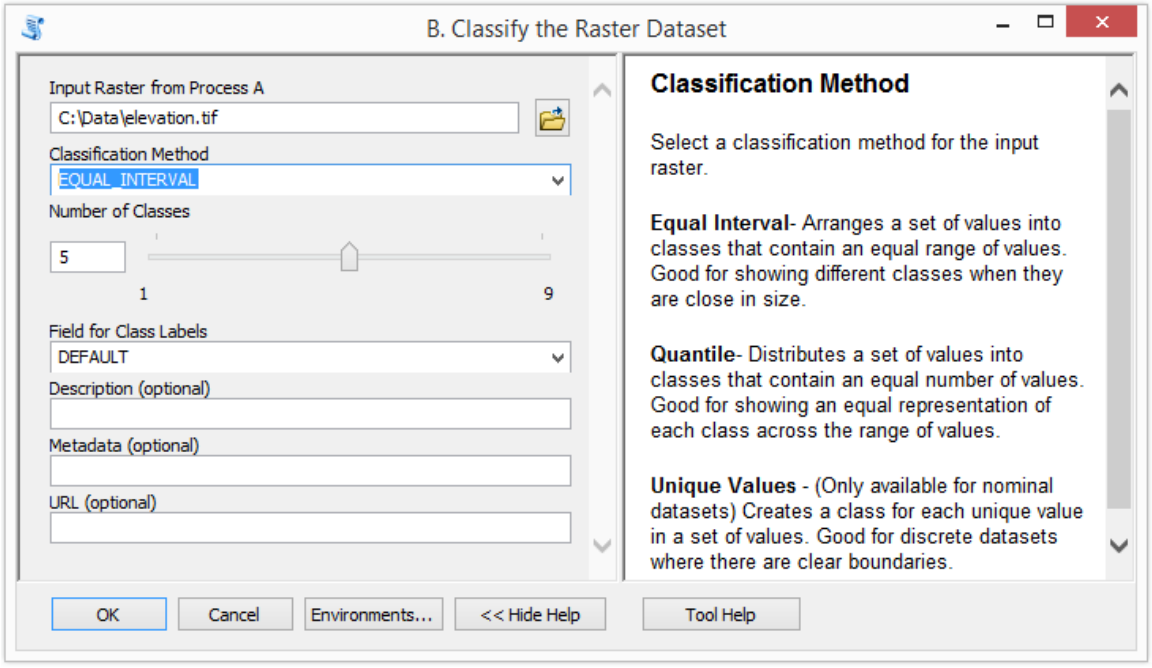

Figure 5.11 Select the Classification Method and Number of Classes

The tool calculates equal intervals by first calculating the class range value, then calculating the break point values, and finishes by calculating the class ranges. The class range value is calculated by dividing the raster's value range by the number of classes to be created. The first break point is calculated by adding the class range value to the minimum cell value. Each subsequent break point is determined by adding the class range value to the previous break point value. These break point values are appended to a list that is fed into the next step (Figure 5.12).

```
# Calculate the Break Points
while counter < numClasses:
    if prevItem == -9999:
        # Calculate with integers
        prevItem = (int(min) + int(classRangeValue))
        breakpoints.append(int(prevItem))
        counter+=1
        continue
    classVal = prevItem + float(classRangeValue)
    breakpoints.append(int(classVal))
    prevItem = classVal
    counter+=1
breakpoints.append(int(max))
```

Figure 5.12 Python Code for Calculating the Break Points

For the last step of the process, the class ranges are defined by adding the minimum and maximum values for each class to a new class ranges list based on the raster's minimum and maximum values and the break point values list (Figure 5.13). When this list is interpreted by the weighted overlay service, the class ranges are defined by including the minimum value and excluding the maximum value for each class (a minimum inclusive and maximum exclusive approach). This means that the maximum value of the first class will be the same value as the minimum value of the next class and so on.

```
# Calculate the Class Ranges (Input Ranges)
while cntr < numClasses:
    if prvItem == -9999:
        prvItem = (breakpoints[indxctr])
        classRanges.append(int(min))
        classRanges.append(prvItem)
        continue
    clssValMin = (breakpoints[indxctr])
    indxctr+=1
    clssValMax = (breakpoints[indxctr])
    classRanges.append(clssValMin)
    classRanges.append(clssValMax)
    cntr+=1
```

Figure 5.13 Python Code for Defining the Class Ranges

The quantile classification method is similar to equal intervals, but instead of equal range values, the classes themselves have an equal number of values. In this method, the raster's cell values are distributed into classes that each have an equal number of values, so every class is approximately the same size. Quantiles are useful in mapping and visualization for showing an equal representation of each class across the range of values. This classification method is also a suitable option for continuous raster datasets.

The process for calculating quantiles in the tool is similar to the equal intervals method with the biggest difference being the way the break points list is created. First, a raster array is created to capture all of the dataset's values in a new list. This list of values

is then input into a series of np.percentile() functions (from the NumPy site package) that have been configured to return quantile values for each potential class range from one to nine classes. These quantile values are then used as the break point values that are then used to define the class range values in the same manner as the equal intervals method.

The third classification method in the Classify the Raster Dataset tool is the unique values approach. This method creates a class for each unique value in the input raster dataset. This option is only available to integer type rasters because creating a class for each unique value in a float type raster would, in the majority of cases, produce more classes than would be useful or meaningful in a weighted raster overlay analysis. The unique values approach is good for discrete datasets where there are clear boundaries or nominal features. When the tool's input is a nominal dataset, the unique values classification method is automatically selected and the Number of Classes parameter defaults to the number of unique values in the dataset.

The raster classification step is the core of this tool. The tool significantly improves the process by simplifying and automating the steps to classify, define, and record the values for the input raster dataset. The final steps in this process are related the recording task and associating related fields and metadata to the input raster dataset.

5.2.3 Step 3 – Add Related Fields

In addition to the class range values that are calculated in the second step, there are a number of other fields that are added and configured for the raster dataset. These fields provide required information, such as the weights for each class (output values) and the labels for each class (range labels). There are also optional fields that can be included to provide more information about the raster layer, such as the raster description, metadata,

and source URL. The Classify the Raster Dataset tool has been designed to accommodate the input of all of these fields.

The required output value field is used to assign weights to each class in the raster layer. These weights can range from a scale of 0 to 9 and can be adjusted in the weighted overlay modeling UI inside the GeoPlanner application. These weights are used in the weighted overlay analysis to emphasize or de-emphasize classes, depending on their significance to the overlay model. There is no UI parameter for this field, instead the tool automatically assigns each class a default weight of "5." When the weighted overlay service is opened in GeoPlanner, all of the classes have the same weight and the user can adjust each class weight as needed.

The Field for Class Labels parameter is used to define the required range labels field. In the weighted overlay service, each class in a layer has a descriptive label. For nominal datasets, the class labels are selected from the raster's attribute table (Figure 5.14).

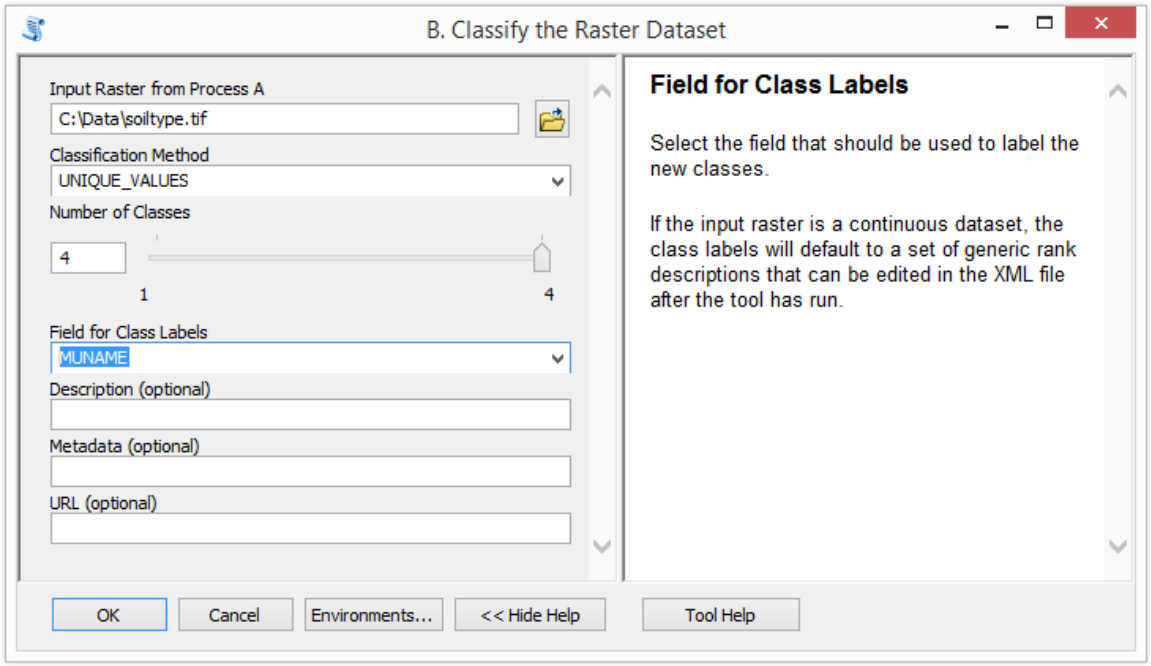

Figure 5.14 Select the Field for Class Labels

If Unique Values has been selected as the classification method, the label values in the selected field are associated with the corresponding cell values during the Classify the Raster step. For continuous datasets, the tool automatically assigns a generic set of raster value descriptions (e.g., Low, Middle, High) for the class labels. These labels are used for each dataset when the classes are weighted in the GeoPlanner for ArcGIS application (Figure 5.15). If the user determines that any of these labels need to be modified, these labels can be edited in the output XML file.

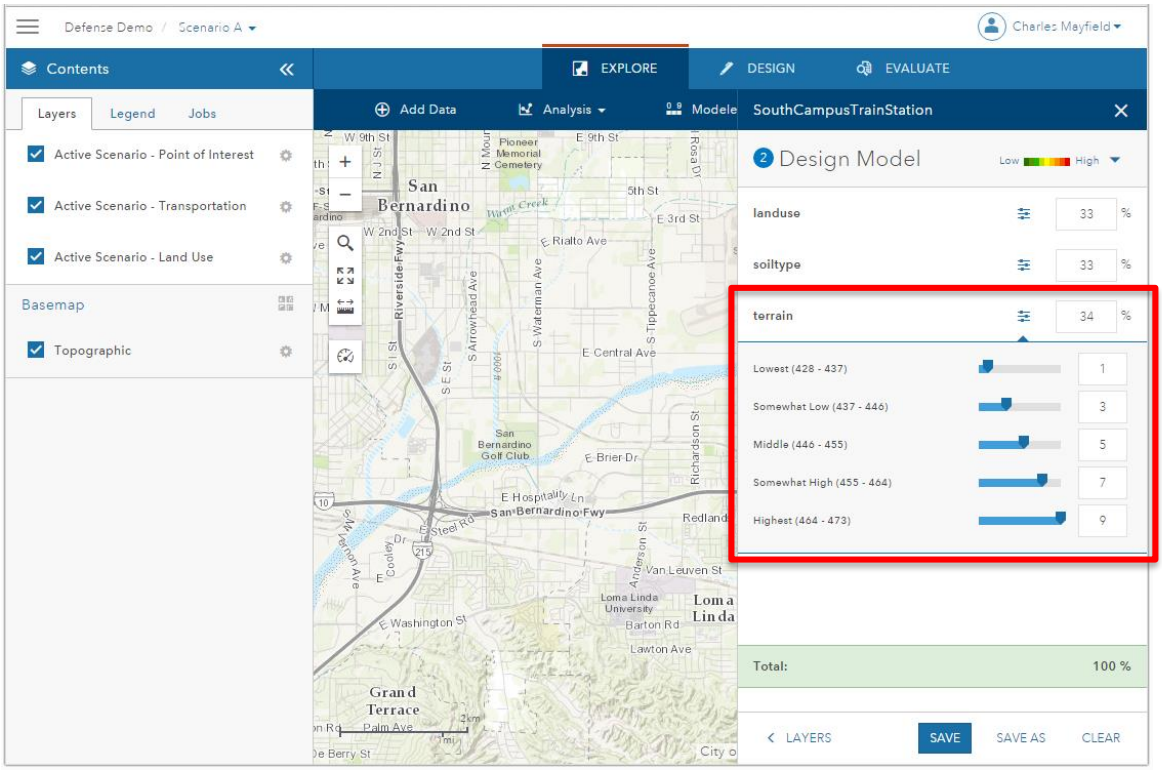

Figure 5.15 Default class labels for continuous datasets in GeoPlanner

In addition to the required raster fields, the Classify the Raster Dataset tool, includes parameters for the optional description, metadata, and URL raster fields. The Description field can be used to describe the raster layer. This text is displayed as part of the raster layer information in the weighted overlay service in GeoPlanner. The Metadata field can be used to record any metadata associated with the raster layer, such as its source,

creation date, limitations, and so forth. If the raster layer was created from a hosted image layer, the URL field can be used to link back to the original source.

5.2.4 Step 4 – Write Data to an XML File

The final step in the Classify the Raster Dataset process is to write all of the data to an output XML file. The data written to this XML file includes the class ranges, weights, and labels, as well as any information included in the optional field parameters. This XML file is associated with the input raster by assigning a title, or file name, that matches the raster. The name of the raster is cleaned up by removing the ".tiff" suffix and adding ".aux.wo.xml" to the end. This XML file is automatically saved to the same location as the input raster. When the Create Mosaic Dataset process is run, the tool loads both raster and XML file together into the mosaic. As a raster layer in the GeoPlanner weighted overlay model, the application reads the XML to determine its classes, weights, and labels.

The two tools, Prepare the Input Dataset and Classify the Raster Dataset, simplify the weighted overlay service creation workflow for the end user by reducing the number of required parameters and automating a number of steps in each process. Additionally, the tools reduce the chance of user error by programmatically writing the raster classification values and other data directly to the output XML file.

Chapter 6 – Results and Analysis

At the conclusion of this project, two custom Python tools have been developed to facilitate the creation of weighted overlay services for the GeoPlanner for ArcGIS application. These tools successfully satisfy the core requirements of the project to improve the weighted overlay service creation workflow by reducing the number of steps and parameters in the UI, by simplifying the front end of the dataset preparation and classification processes, and by automating the creation of the output XML file.

These two tools developed for this project, Prepare the Input Dataset and Classify the Raster Dataset, were delivered to the client in February of 2016. The tools met the client's acceptance criteria for functionality and usability and are expected to improve the end-user experience for GeoPlanner customers. Because these tools were designed to complement the existing WROS toolset, the GeoPlanner for ArcGIS team is preparing them for a general release to the public at a date in the near future. The efforts undertaken for the project have provided material benefit to the project client and the end-users of the GeoPlanner application. The results of the end product, a new weighted overlay service and a composite raster as they appear in the GeoPlanner for ArcGIS application, are displayed in in Figure 6.1.

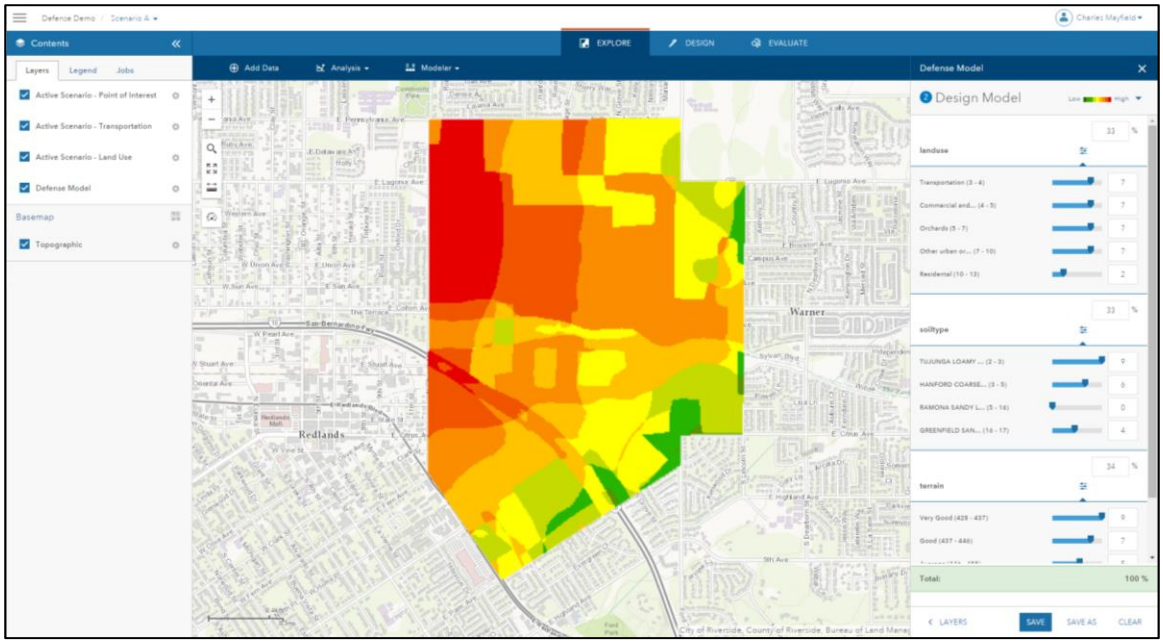

Figure 6.1 Weighted overlay service and composite raster in GeoPlanner

While ultimately successful, the project presented a host of difficulties and challenges. The original expectation for the project was to create a single tool that would completely replace the WROS toolset and the weighted overlay service creation workflow. However, during the requirements gathering phase and initial tool development phase, a number of technical limitations were discovered that made that original goal unrealistic. At the same time, a deepening understanding of the workflow and its individual processes led to a realization that the most significant challenges for end users were in the first two steps. There was little need to revise the two latter steps, which are straightforward and seldom present any difficulty to GeoPlanner users. After discussing this new understanding with the project's client and faculty advisor, the scope of the project was revised to focus on the most important elements of the workflow, the input dataset preparation and classification processes.

The actual development of the tools themselves was another significant challenge of this project. The project manager started this project with no software development experience, so in order to build these tools, Python coding skills had to be learned and the entire software development process had to be understood and applied. This required learning how to gather, define, and prioritize requirements. To this end, many hours were spent searching ArcPy documentation, reading through support forums online, and consulting with other Python developers. This experience demonstrated that novice software developers can successful in their efforts if they know the right questions to ask and where to go find the answers to those questions.

In the tool development phase, as the tools' code base started to take shape, debugging and troubleshooting skills had to be developed to understand and resolve issues that were preventing the tools from executing as designed and expected. Learning how to identify errors, and becoming more efficient at quickly fixing them, is key to the successfully completing this phase of the project. During the final phase of the software development process, testing played an important role in ensuring that the tools could handle a variety of different data type and variables. It is easy to feel successful when only a narrow subset of data types are ever used; it takes imagination and some optimism to test the tools with real-world datasets that can sometimes break the tool in unexpected ways. Testing was difficult but it resulted in a more robust set of tools that can handle more than just carefully curated synthetic data.

The process of developing two new custom Python tools was educational. It introduced new concepts of software development and presented the opportunity to put many of these concepts into practice as the new tools were built from the bottom up.

However, this was only an introduction to the field, and there are still many things to be learned and improved upon.

Chapter 7 – Conclusions and Future Work

The high-level goal of this project was to automate the classification of thematic rasters for weighted overlay analysis in the GeoPlanner for ArcGIS application. To reach this goal a number of specific objectives were defined. The specific requirements associated with these objectives evolved over the course of the project, but the core purpose remained unchanged. The final deliverable included two custom Python tools that replace and complement existing tools in the Weighted Raster Overlay Services toolset. These new tools significantly improve the end-user experience by reducing the number of required steps in the weighted overlay service creation workflow, by providing guidance and recommendations for best practices, and by automating much of the intermediate processes of the workflow. These two new tools satisfied the client's requirements and expectations.

This project has addressed the most difficult and challenging aspects of the weighted overlay service creation workflow, but there are still additional improvements that could be made in future projects. The first version of the Classify the Raster Dataset tool only includes three classification methods, equal interval, quantiles, and unique values. Future development could incorporate other popular classification methods such as standard deviation and Jenk's natural breaks. This project addressed the first two processes of the workflow with two new tools, and these tools process one dataset at a time. Tools that could handle multiple datasets at the same time could potentially make the user experience even easier. Additionally, if a tool was built to classify multiple prepared rasters at the same time, the input rasters and resulting XML files could automatically be

mosaicked together. Automating the Create a Mosaic Dataset process would further reduce end-user effort in the weighted overlay service creation workflow.

Outside of the context of the GeoPlanner for ArcGIS application, there are elements of this project that could be incorporated in other software products and applications. The idea of using Maximum Location Accuracy and map scale to determine an appropriate cell size in raster resampling would be a significant improvement to some of the default cell size selection methodologies in practice today.

The Prepare the Input Dataset and Classify the Raster Dataset tools are evidence that existing processes and workflows can be enhanced by approaching the solution from a customer-centric perspective. The effort to design, build, test, and deploy these tools will translate into a more users creating their own weighted overlay services. When these services are consumed in weighted raster overlay modeler in GeoPlanner, the user has access to a powerful analytical tool that can provide a wealth of information related to their design project – a generic tool becomes specialized, and generalized information becomes more specific and relevant. This project enables users to gain and share new insights and perspectives from their own datasets.

Works Cited

Carver, S. J. (1991). Integrating multi-criteria evaluation with geographical information systems. *International Journal of Geographic Information Systems*, 321-339.

Chrisman, N. (2004). *Charting the Unknown: How Computer Mapping at Harvard Became GIS.* Redlands: Esri Press.

Collins, M. G., Steiner, F. R., & Rushman, M. J. (2001). Land-Use Suitability Analysis in the United States: Historical Development and Promising Technological Achievements. *Environmental Management*, 611-621.

Dragut, L., & Blaschke, T. (2006). Automated classification of landform elements using object-based image analysis. *Geomorphology*, 330-344.

Esri. (2016, January 28). *GeoPlanner for ArcGIS*. Retrieved from ArcGIS Marketplace: https://marketplace.arcgis.com/listing.html?id=5e99f4fa519949209cd3da2966fd543b

Hall, G. B., Wang, F., & Subaryono. (1992). Comparision of Boolean and fuzzy classification methods in land suitability analysis by using geographical information systesms. *Environment and Planning A*, 497-516.

Hengl, T. (2006). Finding the right pixel size. *Computers and Geosciences*, 1283-1298.

Hopkins, L. (1977). Methods for Generating Land Suitability Maps: A Comparative Evaluation. *Journal of the American Institute of Planners*, 380-400.

Iwahashi, J., & Pike, R. J. (2007). Automated classifications of topography from DEMs by an unsupervised nested-means algorithm and a three-part geometric signature. *Geomorphology*, 409-440.

Jankowski, P. (1995). Integrating geographical information systems and multiple criteria decision-making methods. *International Journal of Geographic Information Systems*, 251-273.

Malczewski, J. (2000). On the Use of Weighted Linear Combination Method in GIS: Common and Best Practice Approaches. *Transactions in GIS*, 5-22.

Malczewski, J. (2004). GIS-based land-use suitability analysis: a critical overview. *Progress in Planning*, 3-65.

McHarg, I. L. (1995). *Design with Nature.* New York: J. Wiley. (Reprinted from *Design with Nature,* 1969, Garden City, New York: Natural History Press)

Rossiter, D., & Hengl, T. (2002). Technical note: creating geometrically-correct photo-interpretations, photomosaics, and base maps for a project GIS. *Technical Report*. Enschede, The Netherlands: ITC, Department of Earth Systems Analysis.

Steiner, F., McSherry, L., & Cohen, J. (2000). Land suitability analysis for the upper Gila River watershed. *Landscape and Urban Planning*, 199-214.

Tobler, W. (1988). "Resolution, Resampling, and All That". In H. Mounsey, & R. Tomlinson, *Building Data Bases for Global Science* (pp. 129-137). London: Taylor and Francis.

Tomlinson, R. F. (1999). *Geographic Information Systems.* New York: Wiley.

Van der Merwe, J. H. (1997). GIS-aided land evaluation and decision-making for regulating urban expansion: A South African case study. *GeoJournal*, 135-151.

Vink, A. (1975). *Land Use in Advancing Agriculture, vol. 10.* New York: Springer.

Wilson, K. A. (2007). *Building Complex and Site Categorization Using Similarity to a Prototypical Site.* Redlands: Unpublished master's thesis, University of Redlands.

Appendix A. Python Toolbox Code

The Python toolbox code for the Prepare the Input Dataset tool and the Classify the

Raster Dataset tool.

```
import arcpy
from arcpy import env
from arcpy.sa import *
arcpy.CheckOutExtension("Spatial")
import os
import numpy as np
import types
import string, random, os
import xml
import xml.etree.cElementTree as ET
arcpy.env.overwriteOutput = True

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of the
        .pyt file)."""
        self.label = "Mayfield - WROS Tools"
        self.alias = "MayfieldWROSTools"

        # List of tool classes associated with this toolbox
        self.tools = [A_PrepareDataset, B_ClassifyRaster]

class A_PrepareDataset(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "A. Prepare the Input Dataset"
        self.description = "This tool prepares the input dataset for classification."
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        param0 = arcpy.Parameter(
            displayName="Input Dataset",
            name="InputDataset",
            datatype="DEDatasetType",
            parameterType="Required",
            direction="Input")

        param01 = arcpy.Parameter(
```

```
        displayName="Select a Cell Size (in meters)",
        name="cellSizeInput",
        datatype="GPString",
        parameterType="Required",
        direction="Input")
    param01.filter.type = 'ValueList'
    param01.filter.list = ["600 (Large State - 1:6,000,000)", "300 (Medium State -
1:3,000,000)", "150 (Counties - 1:1,500,000)", "75 (County - 1:750,000)", "32 (Metro
Area - 1:320,000)", "16 (Cities - 1:160,000)", "8 (City - 1:80,000)", "4 (Town -
1:40,000)", "2 (Neighborhood - 1:20,000)", "1 (Block Group - 1:10,000)", "0.5 (Street -
1:5,000)"]

    param02 = arcpy.Parameter(
        displayName="Select a Raster Value Field",
        name="rasterValueField",
        datatype="Field",
        parameterType="Required",
        direction="Input")
    param02.filter.list = ['Text', 'Short', 'Long', 'Single', 'Double', 'Date', 'OID', 'Raster',
'GlobalID']
    param02.parameterDependencies = [param0.name]

    param03 = arcpy.Parameter(
        displayName="Select a Resampling Method",
        name="resamplingType",
        datatype="GPString",
        parameterType="Optional",
        direction="Input")
    param03.filter.type = 'ValueList'
    param03.filter.list = ["NEAREST","BILINEAR"]

    param04 = arcpy.Parameter(
        displayName="Select a Mask for the Design Scenario",
        name="mask",
        datatype="DEFeatureClass",
        parameterType="Optional",
        direction="Input")

    param05 = arcpy.Parameter(
        displayName="Output Raster Name and Location",
        name="outputRaster",
        datatype="DERasterDataset",
        parameterType="Required",
        direction="Output")

    params = [param0,param01,param02,param03,param04,param05]
```

```
        return params

    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."""
        if (parameters[0].value):
            # Disable raster value field for raster inputs
            userInput = parameters[0].valueAsText
            desc = arcpy.Describe(userInput)

            if desc.datasetType == "FeatureClass":
                prepped = "vector"
            elif desc.datasetType == "RasterDataset":
                prepped = "raster"
            else:
                parameters[0].setErrorMessage("Input feature classes must be a raster or vector
dataset.")

            if prepped == "vector":
                parameters[2].enabled = 1 #Turn Raster Value Field parameter ON
                parameters[3].enabled = 0 #Turn Resampling Method parameter OFF
            else:
                parameters[2].value = " "
                parameters[2].enabled = 0 #Turn Raster Value Field parameter OFF
                parameters[3].enabled = 1 #Turn Resampling Method parameter ON

                # Default the resampling method to NEAREST
                if not parameters[3].altered:
                    parameters[3].filter.type = "ValueList"
                    parameters[3].filter.list = ["NEAREST", "BILINEAR"]
                    parameters[3].value = "NEAREST" #Because this is the safe assumption
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter.  This method is called after internal validation."""
        # Verify that the input file is a vector or raster dataset and that the feature classes are
only line and polygon geometry
        if parameters[0].value:
            userInput = parameters[0].valueAsText
            desc = arcpy.Describe(userInput)
```

```
            if desc.datasetType == "FeatureClass":
                prepped = "vector"
                if desc.shapeType == "Point":
                    parameters[0].setErrorMessage("Input feature classes must have line or
polygon geometry.")
                elif desc.shapeType == "MultiPoint":
                    parameters[0].setErrorMessage("Input feature classes must have line or
polygon geometry.")
                elif desc.shapeType == "MultiPatch":
                    parameters[0].setErrorMessage("Input feature classes must have line or
polygon geometry.")
            elif desc.datasetType == "RasterDataset":
                prepped = "raster"
            else:
                parameters[0].setErrorMessage("The input must be a vector feature class or a
raster dataset.")

        # Verify that the output file is a TIFF
        if parameters[5].value:
            output = parameters[5].valueAsText
            if not output.endswith("tif"):
                parameters[5].setErrorMessage("The output raster must be a geoTIFF. Add '.tif'
to the end of the file name.")

        if parameters[5].altered:
            output = parameters[5].valueAsText
            if not output.endswith("tif"):
                parameters[5].setErrorMessage("The output raster must be a geoTIFF. Add '.tif'
to the end of the file name.")
        return

    def execute(self, parameters, messages):
        """The source code of the tool."""
        userInput = parameters[0].valueAsText
        cellSizeInput = parameters[1].valueAsText
        rasterValueField = parameters[2].valueAsText
        resamplingType = parameters[3].value
        mask = parameters[4].value
        outputRaster = parameters[5].valueAsText
        arcpy.env.nodata = "PROMOTION"

        # Set cell size based on user input
        if cellSizeInput == "600 (Large State - 1:6,000,000)":
            cellSize = 600
        elif cellSizeInput == "300 (Medium State - 1:3,000,000)":
```

```
            cellSize = 300
        elif cellSizeInput == "150 (Counties - 1:1,500,000)":
            cellSize = 150
        elif cellSizeInput == "75 (County - 1:750,000)":
            cellSize = 75
        elif cellSizeInput == "32 (Metro Area - 1:320,000)":
            cellSize = 32
        elif cellSizeInput == "16 (Cities - 1:160,000)":
            cellSize = 16
        elif cellSizeInput == "8 (City - 1:80,000)":
            cellSize = 8
        elif cellSizeInput == "4 (Town - 1:40,000)":
            cellSize = 4
        elif cellSizeInput == "2 (Neighborhood - 1:20,000)":
            cellSize = 2
        elif cellSizeInput == "1 (Block Group - 1:10,000)":
            cellSize = 1
        elif cellSizeInput == "0.5 (Street - 1:5,000)":
            cellSize = 0.5
        else:
            cellSize = 0.5

        # Set global variables
        arcpy.env.outputCoordinateSystem = arcpy.SpatialReference(3857) #3857 is the
WKID for WGS_1984_Web_Mercator_Auxiliary_Sphere
        arcpy.env.cellSize = cellSize

        def main():
            # Determine the Input Dataset Type
            desc = arcpy.Describe(userInput)
            if desc.datasetType == "FeatureClass":
                prepped = prepVector(userInput)
            else:
                prepped = prepRaster(userInput)

        def prepVector(inVector):
            projected = "projected"
            outRaster = "outRaster"

            # Project the Vector Dataset
            arcpy.Project_management(inVector, projected,
arcpy.env.outputCoordinateSystem)

            # Repair the Vector Dataset
            arcpy.RepairGeometry_management(projected)
```

```
        # Convert Vector to Raster
        arcpy.FeatureToRaster_conversion (projected, rasterValueField, outRaster,
cellSize)
        outRaster2 = Raster(outRaster)

        # Clip the Raster Dataset and Build Pyramids
        if mask and mask != "#":
           clipRaster = arcpy.sa.ExtractByMask(outRaster2, mask)
           preppedRaster = clipRaster
           arcpy.BuildPyramids_management(preppedRaster)
           preppedRaster.save(outputRaster)
        else:
           preppedRaster = outRaster
           arcpy.BuildPyramids_management(preppedRaster)
           preppedRaster2 = Raster(preppedRaster)
           preppedRaster2.save(outputRaster)
        return

     def prepRaster(inRaster):
        # Define the intermediate file
        int = parameters[5].valueAsText
        if int.endswith(".tif"):
           intRaster = int.replace(".tif","_int")

        # Clip the Raster Dataset and Build Pyramids
        if mask and mask != "#":
           clipRaster = arcpy.sa.ExtractByMask(inRaster, mask)
           preppedRaster = clipRaster
           arcpy.BuildPyramids_management(preppedRaster)
        else:
           preppedRaster = inRaster
           arcpy.BuildPyramids_management(preppedRaster)

        # Project the Raster Dataset
        arcpy.ProjectRaster_management(preppedRaster, intRaster,
arcpy.env.outputCoordinateSystem, resamplingType, cellSize)
        projectedRaster = arcpy.Raster(intRaster)
        projectedRaster.save(outputRaster)
        arcpy.Delete_management(intRaster)
        return
     main()
     return


class B_ClassifyRaster(object):
   def __init__(self):
```

```
        """Define the tool (tool name is the name of the class)."""
        self.label = "B. Classify the Raster Dataset"
        self.description = "This tool classifies the input raster dataset and adds the weighted
overlay data required by the GeoPlanner for ArcGIS application."
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        param0 = arcpy.Parameter(
            displayName="Input Raster from Process A",
            name="raster",
            datatype="GPRasterLayer",
            parameterType="Required",
            direction="Input")

        param01 = arcpy.Parameter(
            displayName="Classification Method",
            name="classMethod",
            datatype="GPString",
            parameterType="Required",
            direction="Input")
        param01.filter.type = "ValueList"
        param01.filter.list = ["EQUAL_INTERVAL", "QUANTILE",
"UNIQUE_VALUES"]

        param02 = arcpy.Parameter(
            displayName="Number of Classes",
            name="numClasses",
            datatype="GPLong",
            parameterType="Required",
            direction="Input")
        param02.filter.type = "Range"
        param02.filter.list = [1,9]

        param03 = arcpy.Parameter(
            displayName="Field for Class Labels",
            name="labelField",
            datatype="Field",
            parameterType="Required",
            direction="Input")
        param03.filter.list = ['Text', 'Short', 'Long', 'Single', 'Double', 'Date', 'OID', 'Raster',
'GUID', 'GlobalID']
        param03.parameterDependencies = [param0.name]

        param04 = arcpy.Parameter(
            displayName="Description",
```

```
        name="Description",
        datatype="GPString",
        parameterType="Optional",
        direction="Input")

    param05 = arcpy.Parameter(
        displayName="Metadata",
        name="Metadata",
        datatype="GPString",
        parameterType="Optional",
        direction="Input")

    param06 = arcpy.Parameter(
        displayName="URL",
        name="url",
        datatype="GPString",
        parameterType="Optional",
        direction="Input")

    param07 = arcpy.Parameter(
        displayName="Output XML File",
        name="outputXML",
        datatype="GPRasterLayer",
        parameterType="Derived",
        direction="Output")
    param07.parameterDependencies=[param0.name]

    params = [param0,param01,param02,param03,param04,param05,param06,param07]
    return params

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."""

    if (parameters[0].value):
    # Assign variables
        raster = parameters[0].valueAsText
        classMethod = parameters[1].value
        numClasses = parameters[2].value
        labelField = parameters[3].value
        rasterTitle = "" #For the XML file
```

```
            rasterType = "" #Continuous or Nominal
            classLabels = []
            classRanges = []
            outputWoXmlFiles=[]

        # Describe the input raster and clean up the Raster Title
            desc = arcpy.Describe(raster)
            rasterTitle = desc.name
            rasterExtension = desc.extension
            rasterTitle = rasterTitle.replace(rasterExtension,"")
            if rasterTitle.endswith("."):
                rasterTitle=rasterTitle.replace(".","")

        # Create a raster object to improve performance
            rasterObject = Raster(raster)

        # Determine and assign the raster type (NOMINAL or CONTINUOUS)
            rasterValTypResult = arcpy.GetRasterProperties_management (raster,
"VALUETYPE") #Returns the pixel type and depth
            rasterValTyp = rasterValTypResult.getOutput(0) #Value Types between 9 and 14
are floating point, doubles, or complex

            if rasterValTyp == "1": #2-bit unsigned (0 to 3)
                rasterType = "NOMINAL"
            elif rasterValTyp == "2": #4-bit unsigned (0 to 15)
                rasterType = "NOMINAL"
            elif rasterValTyp == "3": #8-bit unsigned (0 to 255)
                rasterType = "NOMINAL"
            elif rasterValTyp == "4": #8-bit signed (-128 to 127)
                rasterType = "NOMINAL"
            elif rasterValTyp == "5": #16-bit unsigned (0 to 65535)
                rasterType = "NOMINAL"
            elif rasterValTyp == "6": #16-bit signed (-32768 to 32767)
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "7": #32-bit unsigned integer (0 to 4,294,967,295)
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "8": #32-bit signed integer (approx. -2 billion to +2 billion)
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "9": #32-bit floating point single precision
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "10": #64-bit floating point double precision
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "11": #8-bit complex
                rasterType = "CONTINUOUS"
            elif rasterValTyp == "12": #16-bit complex
                rasterType = "CONTINUOUS"
```

```
        elif rasterValTyp == "13": #32-bit complex
            rasterType = "CONTINUOUS"
        elif rasterValTyp == "14": #64-bit complex
            rasterType = "CONTINUOUS"
        else:
            arcpy.AddWarning("The raster type could not be determined.")

    # Assign defaults for Nominal datasets
        if rasterType == "NOMINAL":
            unqValCntResult = arcpy.GetRasterProperties_management (raster,
"UNIQUEVALUECOUNT") #Number of unique values in the raster
            unqValCnt = int(unqValCntResult.getOutput(0))
            if not parameters[1].altered:
                parameters[1].value = "UNIQUE_VALUES" #Default classification method
to Unique Values
                parameters[2].filter.type = "Range"
                parameters[2].filter.list = [1,unqValCnt]
                parameters[2].value = unqValCnt #Default number of class to the number of
unique values

    # Assign defaults for Continuous datasets
        elif rasterType == "CONTINUOUS":
            if not parameters[1].altered:
                parameters[1].filter.type = "ValueList"
                parameters[1].filter.list = ["EQUAL_INTERVAL", "QUANTILE"]
#Removes Unique Values from the pick list
                parameters[1].value = "EQUAL_INTERVAL" #Default to Equal Interval
            if not parameters[2].altered:
                parameters[2].value = "5" #Default to 5 classes for continuous rasters
            if not parameters[3].altered:
                parameters[3].value = "DEFAULT" #Assigns default class labels

    # Define default labels for continous datasets
        if numClasses == 1:
            classLabels = ["Present"]
        elif numClasses == 2:
            classLabels = ["Low", "High"]
        elif numClasses == 3:
            classLabels = ["Low", "Middle", "High"]
        elif numClasses == 4:
            classLabels = ["Lowest", "Somewhat Low", "Somewhat High", "Highest"]
        elif numClasses == 5:
            classLabels = ["Lowest", "Somewhat Low", "Middle", "Somewhat High",
"Highest"]
        elif numClasses == 6:
```

```
            classLabels = ["Lowest", "Very Low", "Somewhat Low", "Somewhat High",
"Very High", "Highest"]
        elif numClasses == 7:
            classLabels = ["Lowest", "Very Low", "Somewhat Low", "Middle",
"Somewhat High", "Very High", "Highest"]
        elif numClasses == 8:
            classLabels = ["Lowest", "Very Low", "Low", "Somewhat Low", "Somewhat
High", "High", "Very High", "Highest"]
        elif numClasses == 9:
            classLabels = ["Lowest", "Very Low", "Low", "Somewhat Low", "Middle",
"Somewhat High", "High", "Very High", "Highest"]

    # Define the raster statistics
        min = rasterObject.minimum
        minResult = arcpy.GetRasterProperties_management (raster, "MINIMUM")
        minNumber = (minResult.getOutput(0))
        max = rasterObject.maximum
        maxResult = arcpy.GetRasterProperties_management (raster, "MAXIMUM")
        maxNumber = (maxResult.getOutput(0))
        range = (float(maxNumber) - float(minNumber))
        noDataValue = rasterObject.noDataValue

# Classify the Raster
    # UNIQUE VALUES
        if classMethod == "UNIQUE_VALUES":
            # Define variables for Unique Values
            breakpoints = []
            classLabels = []
            classRanges = []
            prevItem = -9999
            counter = 1
            indxctr = 0
            field = "Value" #Hard coded value to simplify the process

            # Create lists for unique values and class labels
            cursor = arcpy.SearchCursor(raster)
            for row in cursor:
                classVal = row.getValue(field)
                classLbl = row.getValue(labelField)
                breakpoints.append(classVal)
                classLabels.append(classLbl)
            breakpoints.append(9999)

            # Calculate the class ranges (Input Ranges)
            while counter < numClasses:
                if prevItem == -9999:
```

```
                min = breakpoints[indxctr]
                prevItem = (breakpoints[indxctr] + 1)
                classRanges.append(min)
                classRanges.append(prevItem)
                continue
            classValMin = (breakpoints[indxctr] + 1)
            indxctr+=1
            classValMax = (breakpoints[indxctr] + 1)
            classRanges.append(classValMin)
            classRanges.append(classValMax)
            counter+=1

    # EQUAL INTERVAL
        if classMethod == "EQUAL_INTERVAL":
            # Define variables
            breakpoints = []
            classRanges = []
##              classRangeValue = (float(range) / float(numClasses)) #For precise
calculations
            classRangeValue = (int(range) / int(numClasses)) #For integer requirement
            counter = 1
            prevItem = -9999
            cntr = 1
            prvItem = -9999
            indxctr = 0

            # Calculate the Break Points
            while counter < numClasses:
                if prevItem == -9999:
                    # Calculate with integers
                    prevItem = (int(min) + int(classRangeValue)) #For integer requirement
                    breakpoints.append(int(prevItem)) #For integer requirement
##                      # Calculate with floating point for greater precision
##                      prevItem = (float(min) + float(classRangeValue))
##                      breakpoints.append(float(prevItem))
                    counter+=1
                    continue
                classVal = prevItem + float(classRangeValue)
##                  breakpoints.append(float(classVal)) #For precise calculations
                breakpoints.append(int(classVal)) #For integer requirement
                prevItem = classVal
                counter+=1
            breakpoints.append(int(max)) #For integer requirement

            # Calculate the class ranges (Input Ranges)
            while cntr < numClasses:
```

```
            if prvItem == -9999:
                prvItem = (breakpoints[indxctr])
                classRanges.append(int(min)) #For integer requirement
                classRanges.append(prvItem)
                continue
            clssValMin = (breakpoints[indxctr])
            indxctr+=1
            clssValMax = (breakpoints[indxctr])
            classRanges.append(clssValMin)
            classRanges.append(clssValMax)
            cntr+=1

    # QUANTILE
        elif classMethod == "QUANTILE":
            # Define variables
            breakpoints = []
            classRanges = []
            counter = 1
            prevItem = -9999
            indxctr = 0
            rasterValues = []

            # Calculate the raster values
            rasterArray = arcpy.RasterToNumPyArray(raster)
            rows, cols = rasterArray.shape
            for rowNum in xrange(rows):
                for colNum in xrange(cols):
                    value = rasterArray.item(rowNum, colNum)
                    if value != noDataValue:
                        rasterValues.append(value)
            data = rasterValues

            # Quantile calculations for each class range
            q11 = int(np.percentile(data, 11.0))
            q13 = int(np.percentile(data, 13.0))
            q14 = int(np.percentile(data, 14.0))
            q17 = int(np.percentile(data, 17.0))
            q20 = int(np.percentile(data, 20.0))
            q22 = int(np.percentile(data, 22.0))
            q25 = int(np.percentile(data, 25.0))
            q29 = int(np.percentile(data, 29.0))
            q33 = int(np.percentile(data, 33.0))
            q38 = int(np.percentile(data, 38.0))
            q40 = int(np.percentile(data, 40.0))
            q43 = int(np.percentile(data, 43.0))
            q44 = int(np.percentile(data, 44.0))
```

```
q50 = int(np.percentile(data, 50.0))
q56 = int(np.percentile(data, 56.0))
q57 = int(np.percentile(data, 57.0))
q60 = int(np.percentile(data, 60.0))
q63 = int(np.percentile(data, 63.0))
q66 = int(np.percentile(data, 66.0))
q67 = int(np.percentile(data, 67.0))
q71 = int(np.percentile(data, 71.0))
q75 = int(np.percentile(data, 75.0))
q78 = int(np.percentile(data, 78.0))
q80 = int(np.percentile(data, 80.0))
q83 = int(np.percentile(data, 83.0))
q86 = int(np.percentile(data, 86.0))
q88 = int(np.percentile(data, 88.0))
q89 = int(np.percentile(data, 89.0))
q100 = int(max)

# Calculate the break points
if numClasses == 1:
   breakpoints.append(q100)
elif numClasses == 2:
   breakpoints.append(q50)
   breakpoints.append(q100)
elif numClasses == 3:
   breakpoints.append(q33)
   breakpoints.append(q66)
   breakpoints.append(q100)
elif numClasses == 4:
   breakpoints.append(q25)
   breakpoints.append(q50)
   breakpoints.append(q75)
   breakpoints.append(q100)
elif numClasses == 5:
   breakpoints.append(q20)
   breakpoints.append(q40)
   breakpoints.append(q60)
   breakpoints.append(q80)
   breakpoints.append(q100)
elif numClasses == 6:
   breakpoints.append(q17)
   breakpoints.append(q33)
   breakpoints.append(q50)
   breakpoints.append(q66)
   breakpoints.append(q83)
   breakpoints.append(q100)
elif numClasses == 7:
```

```
                breakpoints.append(q14)
                breakpoints.append(q29)
                breakpoints.append(q43)
                breakpoints.append(q57)
                breakpoints.append(q71)
                breakpoints.append(q86)
                breakpoints.append(q100)
            elif numClasses == 8:
                breakpoints.append(q13)
                breakpoints.append(q25)
                breakpoints.append(q38)
                breakpoints.append(q50)
                breakpoints.append(q63)
                breakpoints.append(q75)
                breakpoints.append(q88)
                breakpoints.append(q100)
            elif numClasses == 9:
                breakpoints.append(q11)
                breakpoints.append(q22)
                breakpoints.append(q33)
                breakpoints.append(q44)
                breakpoints.append(q56)
                breakpoints.append(q67)
                breakpoints.append(q78)
                breakpoints.append(q89)
                breakpoints.append(q100)
            else:
                parameters[1].setErrorMessage("The QUANTILE classification method
cannon be used with dataset. Please select another classifcation method.")

            # Calculate the class ranges (Input Ranges)
            while counter < numClasses:
                if prevItem == -9999:
                    prevItem = (breakpoints[indxctr])
                    classRanges.append(int(min)) #For integer requirement
                    classRanges.append(prevItem)
                    continue
                classValMin = (breakpoints[indxctr])
                indxctr+=1
                classValMax = (breakpoints[indxctr])
                classRanges.append(classValMin)
                classRanges.append(classValMax)
                counter+=1

    # Add XML Info
        catalogPath = parameters[0].value
```

```
            inaux="{}.aux.wo.xml".format(catalogPath)
            labelField = parameters[3].value
            rasterDescription = parameters[4].valueAsText
            rasterMetadata = parameters[5].valueAsText
            rasterUrl = parameters[6].valueAsText

            # Define Output Values
            outputValues = []
            counterOV = 1
            while counterOV <= numClasses:
               outputValues.append(5)
               counterOV+=1

            # Create the wo.xml file and write data to it
            pmdataset=ET.Element("PAMDataset")
            metadata=ET.SubElement(pmdataset,"Metadata")
            ET.SubElement(metadata,"MDI", key="Title").text=rasterTitle
            ET.SubElement(metadata,"MDI", key="Description").text=rasterDescription
            ET.SubElement(metadata,"MDI", key="Metadata").text=rasterMetadata
            ET.SubElement(metadata,"MDI", key="url").text=rasterUrl
            ET.SubElement(metadata,"MDI",
key="InputRanges").text=','.join(map(str,classRanges))
            ET.SubElement(metadata,"MDI",
key="OutputValues").text=','.join(map(str,outputValues))
            #ET.SubElement(metadata,"MDI", key="NoDataRanges").text=str(noDataValue)
            ET.SubElement(metadata,"MDI", key="NoDataRanges").text=""
            ET.SubElement(metadata,"MDI",
key="RangeLabels").text=','.join(map(str,classLabels))
            #ET.SubElement(metadata,"MDI", key="NoDataRangeLabels").text="NoData
Range Lables"
            ET.SubElement(metadata,"MDI", key="NoDataRangeLabels").text=""
            tree=ET.ElementTree(pmdataset)
            tree.write(inaux)
            outputWoXmlFiles.append(inaux)

            arcpy.SetParameter(1,outputWoXmlFiles)
         return

      def updateMessages(self, parameters):
         """Modify the messages created by internal validation for each tool
         parameter.  This method is called after internal validation."""
         # Verify that the input file is a TIFF
         if parameters[0].value:
            input = parameters[0].valueAsText
            if not input.endswith("tif"):
```

```
            parameters[0].setErrorMessage("The input raster must be a TIFF. Run this
dataset through the 'Prepare the Input Dataset' tool to prepare it for this tool.")

        desc = arcpy.Describe(input).spatialReference
        code = desc.factoryCode
        if not code == 3857:
            parameters[0].setErrorMessage("The input raster must be projected in Web
Mercator (Auxiliary Sphere). Run this dataset throught 'Prepare the Input Dataset' tool to
prepare it for this tool.")

    if parameters[0].altered:
        input = parameters[0].valueAsText
        if not input.endswith("tif"):
            parameters[0].setErrorMessage("The input raster must be a TIFF. Run this
dataset through the 'Prepare the Input Dataset' tool to prepare it for this tool.")
    return

  def execute(self, parameters, messages):
    """The source code of the tool."""
    if arcpy.CheckExtension("Spatial")!="Available":
        arcpy.AddError("Spatial Analyst extension is not available")
        return

    return
```