Tagamajig: Image Recognition via Crowdsourcing

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Gregory Martin Simpson

December 2016

© by Gregory Martin Simpson, 2016

All Rights Reserved.

To my sister, Rebecca. If I can do it, so can you.

Acknowledgements

I would like to thank my advisor and committee chair, Dr. Mark Dean, for his invaluable support and counsel throughout my graduate career. I would like to thank my committee members, Dr. Bradley Vander Zanden and Dr. Chad Steed, for their significant guidance and feedback throughout the lifespan of this project, as well as their considerable contributions to my graduate and undergraduate education. I would like to thank my colleague and friend, Kelley Deuso, for presenting me with the opportunity to pursue my graduate degree and for her continuous encouragement to persevere during the most difficult portions of my graduate education. I would also like to thank my other current and former fellow Graduate Teaching Assistants, Mwamba Bowa and Derek Lusby, for their hard work and patience with me on those occasions when I was overwhelmed with my studies.

I would like to thank my parents for their immeasurable support throughout my pursuit of this degree...and the last one, and the one before that. I could not have done this without their constant words of comfort and advice, and I truly appreciate their encouragement and their confidence in my abilities. Finally, I would like to thank my friends, Janine and Richard, for their unceasing encouragement and patience with me, even on my worst of days. Their constant reality checks, reminders to keep things in perspective, and occasional insistence at taking a break and living life allowed me to survive the last year and a half, and I am forever grateful.

Abstract

The University of Tennessee, Knoxville (UTK) Library possesses thousands of unlabeled gray-scale photographs from the Smoky Mountains circa the 1920s - 1940s. Their current method of identifying and labeling attributes of the photographs is to do so manually. This is problematic both because of the scale of the collection as well as the reliance on an individual's limited knowledge of the area's numerous landmarks.

In the past few years, similar dilemmas have been tackled via an approach known as crowd computing. Some examples include Floating Forests, in which users are asked to identify and mark kelp forests in satellite images, and Ancient Lives, which enlists users help in transcribing 2000-year-old manuscripts that Oxford University researchers had struggled to efficiently translate for over a century.

For this particular problem, we propose releasing the image collections to the public through a web application. The application would target outdoor enthusiasts, conservationists, or professionals such as geologists, rangers, or historians who are familiar with the region and would find interest in helping to label the more recognizable photos. Users would "tag" landmarks using a hierarchically sorted data set of landmark names accessible via an incremental search.

With sufficient participation, the image collection could be efficiently categorized and labeled beyond what is currently feasible using the librarys limited number of personnel. Furthermore, this application could easily be adapted to categorize other unlabeled image collections if provided with the proper data set for tagging.

Table of Contents

1 Introduction **1**

1.1 Project Overview 1

1.2 Inspiration 2

1.3 Objectives 3

2 Design **5**

2.1 Application Overview 5

2.2 Utility Views 6

2.2.1 User Registration and Login 6

2.2.2 Image Collection Navigation 7

2.3 Image Tagging Interface 9

2.4 Aggregate Results View 14

2.5 Administrative Panel 18

3 Implementation **22**

3.1 Back-end 24

3.1.1 Tagging Data Set and MySQL Database 24

3.1.2 Laravel Framework 29

3.1.3 Helper Scripts 35

3.2 Front-end 35

3.2.1 Laravel Blade Templates 35

3.2.2 Bootstrap Framework 36

3.2.3 jQuery Plugins 37

4 Evaluation **40**

4.1 Alternative Design Considerations 40

4.2 Limitations 41

4.2.1 Limitations of Image Tagging Interface 41

4.2.2 Limitations of Aggregate Results View 42

4.3 Additional Applications 44

5 Future Work **46**

5.1 Sorting Collection by Landmark Type 46

5.2 Aggregate View by Landmark Type 47

5.3 Google Maps Integration 47

5.4 Tablet/Touchscreen Support 48

5.5 Digital Assets Management Integration 48

5.6 User Contributions to Data Set 49

6 Conclusion **50**

Bibliography **52**

Vita **56**

List of Figures

2.1 User Registration Form 7
2.2 Application Login Form 7
2.3 Paginated Image Collection View 8
2.4 Image Tagging User Interface 9
2.5 Image Tagging Incremental Search 11
2.6 Tag Marker Mouse Hover 12
2.7 Tagging a Feature by Type 13
2.8 Aggregate Results Heatmap 15
2.9 Heatmap Adjusted After Removal of Most Prominent Landmark . . . 16
2.10 Heatmap Opacity Adjusted to 100% 17
2.11 Colorblind Friendly Heatmap with 100% Opacity 18
2.12 Admin Photo Management Table 19
2.13 Admin Panel - Two Landmark Write-ins To Be Reviewed 20
2.14 Manage Write-ins Table 20

3.1 Application Software Stack 24
3.2 Diagram of Project Database Tables 25
3.3 Diagram of Related Tables in Project Database 27
3.4 Example Migration - Users Table 30
3.5 Example Eloquent Query 31

4.1 Aggregate Results Heatmap - Limitations 43

4.2 Example Use - Sports Teams and Players 45

5.1 Potential Google Maps Integration 48

Chapter 1

Introduction

1.1 Project Overview

This project was motivated by the needs of the University of Tennessee Library Digital Collections Department. The library possesses thousands of scanned digital images across dozens of collections, many of which are unlabeled. The library's current method for labeling these images is for a single employee to do so manually. This is problematic both because of the sheer size of the image collections as well as the dependence on an individual's limited knowledge of a particular image set. For example, the image collection which was targeted as a proof of concept for this project contains gray-scale images of a particular region of the Smoky Mountains, most of which were taken between the 1920s and 1940s. While many of these images contain landmarks which may be recognizable to individuals with extensive knowledge of the area, relying on a single person with limited familiarity of the region to accurately classify such a large and diverse collection of images proved to be time-consuming and impractical.

The first solution proposed for this particular challenge was the use of landmark identification using existing image processing techniques. However, this approach was soon abandoned due to several limiting factors. Firstly, many of the landmarks

in question are located in remote areas and therefore lack a significant number of existing labeled photographs needed to act as a training set for an image recognition neural network. Secondly, the appearance of the majority of the landmarks change over the course of months, days, or even minutes. A particular waterfall, for example, may appear drastically different from one photograph to the next, as environmental changes may affect the flow of water or even cause some or all of the waterfall to freeze. With the proper training set, it could be possible to classify the image as containing a waterfall, but labeling the specific landmark in question would be impossible, even with a substantial collection of images of that particular landmark to act as a training set. Finally, the collection itself would be a limiting factor in the use of automated object recognition, as the digital collection consists of scanned gray-scale photographs, many of which are almost 100 years old. While the Digital Collections Department's efforts to catalog these photographs have resulted in a collection which is easily recognizable to the average viewer, the quality is far from perfect due to the age of the photographs and would be unsuitable for use in a study of object identification via a trained neural network.

After abandoning the computer vision approach to the problem at hand, focus shifted to a solution utilizing crowdsourcing. With the Smoky Mountains National Park drawing a large number of outdoor enthusiasts and history buffs each year, a crowdsourcing application targeting such individuals with more extensive familiarity of the region seemed the most practical and realistic means of acquiring accurate labels for the image collection.

1.2 Inspiration

This project's primary influence was Ancient Lives, developed by Oxford University [24]. The university has been in possession of hundreds of thousands of untranslated Greek texts since the 1880s. Prior to the introduction of Ancient Lives to the public, the university had only managed to translate a small portion of the text collection due

to a limited number of available personnel and the time-consuming nature of the work. After its release, Ancient Lives proved to be a major success, with volunteers providing over 7 million transcriptions in only a few years. Ancient Lives allows users to select the exact point on each scanned papyrus image that they wish to identify and mark this location with a letter from a predefined set of Greek alphabetic symbols. This process served as the primary inspiration for the landmark tagging process developed for use in this project.

Another crowdsourcing web application, Floating Forests [25], was also examined as a source of possible inspiration. Floating Forests presents users with satellite images in an "endless slideshow" and asks users to identify kelp forests in the images by outlining them with their mouse. This idea of a continuous one-way slideshow was initially considered as a means of presenting users of this application with new images from the Smoky Mountains collection. However, this approach was eventually abandoned for reasons that will be described in detail in the Design section below.

1.3 Objectives

The primary requirement of the library's Digital Collections Department is a means of sorting and classifying vast quantities of unlabeled photographs. Given the department's needs and the decision to tackle the problem at hand through the use of crowdsourcing, the application's primary focus was determined to be the delivery a user-friendly interface for tagging specific landmarks in the given set of photographs. Emphasis on a simple, intuitive interface is stressed due to the varied levels of computer literacy of the target users. However, the interface must find balance between ease of use and depth to allow for users to provide enough feedback via the available data set in order to provide the library with meaningful results. To allow users with less familiarity of the region with the opportunity to assist with cataloging the image collection, the application must provide users with the ability to mark landmark types in addition to specific landmarks. While this data might not

be as directly valuable to the library, it could be used to categorize the collection in a more meaningful and organized way rather than presenting users with all possible photographs that need to be identified.

To promote participation and a sense of group loyalty among users (and discourage purposeful manipulation or skewing of results), users and application administrators will be able to view aggregate results for each photograph with a visual representation of each individual tag submitted for a photograph. Each tag visualization would be weighted in such a way as to represent the degree to which other users agree or disagree by submitting the same landmark for the same photograph.

Finally, the application was developed with the goal of being able to accept new image collections and data sets with relative ease, allowing the application to easily be adapted for use in the identification of numerous other types of unlabeled image sets. Other potential image types, as well as many potential future additions to the application, will be discussed in Chapter 5.

Chapter 2

Design

This chapter will provide an overall description of the user-facing layout of the application, a description of some key terms used throughout this report, and a detailed description of each front-end element of the application.

2.1 Application Overview

The application can be described as being divided into three primary sections.

Image-Tagging Interface The image-tagging interface is the primary view through which users contribute to the labeling of the image collection. Using this view, users are able to examine a specific image and label or **tag** specific notable features within the image. A **feature** refers to any attribute within the image that users may deem significant enough to warrant labeling. Users are given two options for tagging a feature within the image: the user can tag the feature as being a **landmark**, which is a feature unique to the region (e.g. "Abrams Falls", "Bright Hill Cemetery"), or the user can tag the feature by **type**, which is a categorical description of a feature to which each landmark must be assigned (e.g. "Abrams Falls" is of type "falls", "Bright Hill Cemetery" is of type "cemetery").

Aggregate Results View The aggregate results view utilizes a heatmap overlaid on an image to illustrate all tags which users have submitted for the image. The view allows users to filter the heatmap to only display results for certain submitted landmarks and modify the heatmap's settings to suit the user's individual preferences.

Administrative Panel The administrative panel provides users with admin privileges the ability to manage all assets of the application, including user accounts, available images, and the contents of the data set used for tagging image features.

Each view is explored in detail in the following sections.

2.2 Utility Views

2.2.1 User Registration and Login

Before exploring or contributing to the image collection, users are required to create a unique user account. A user account is necessary to prevent a user from submitting multiple identical tags for a single image. It also makes it possible for the user to review and edit previously submitted tags. New users must provide a unique username and email address as well as a password. An email address is necessary for resetting a user's password in the event that the user's password is forgotten. The user registration form is displayed in Figure 2.1.

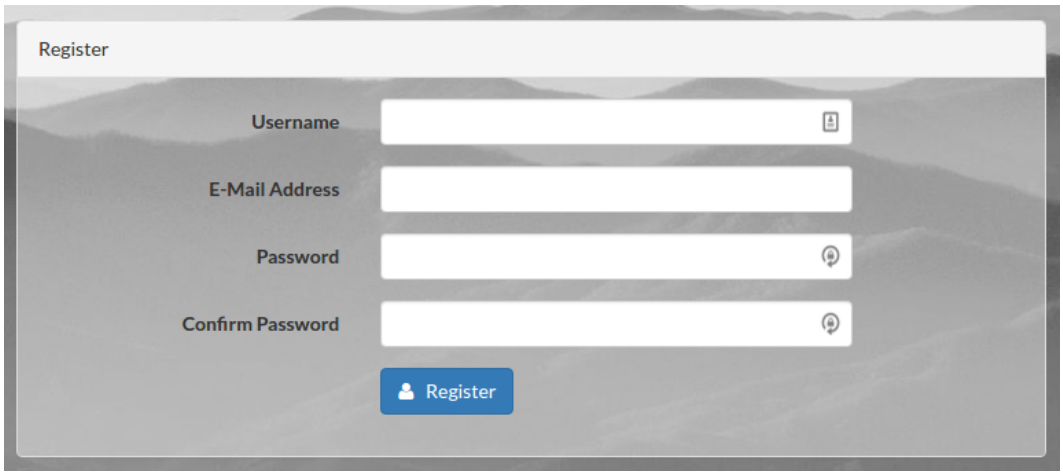

Figure 2.1: User Registration Form

After registering, users can sign in via the login form on the application's homepage, which can be seen in Figure 2.2.

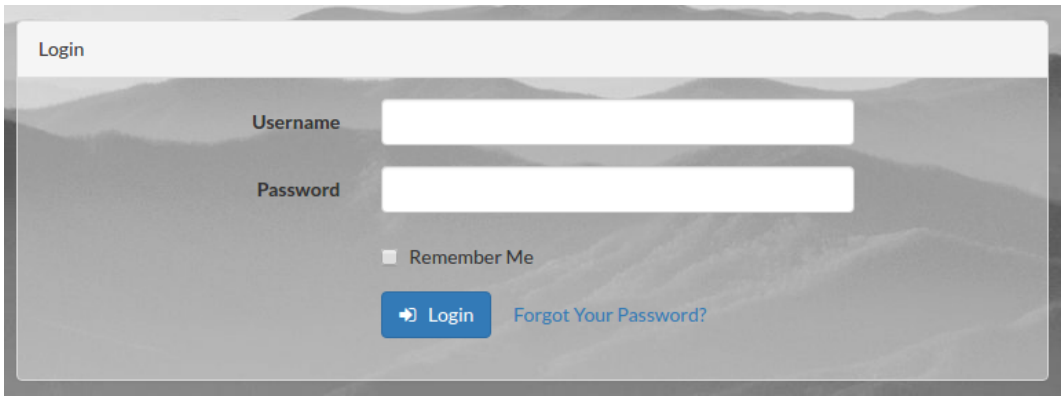

Figure 2.2: Application Login Form

2.2.2 Image Collection Navigation

Navigation of images both for the purposes of tagging image features or viewing aggregate results is managed through the pagination of the image collection, allowing users to view a subset of the collection on each page. This paginated thumbnail view is shown in Figure 2.3.

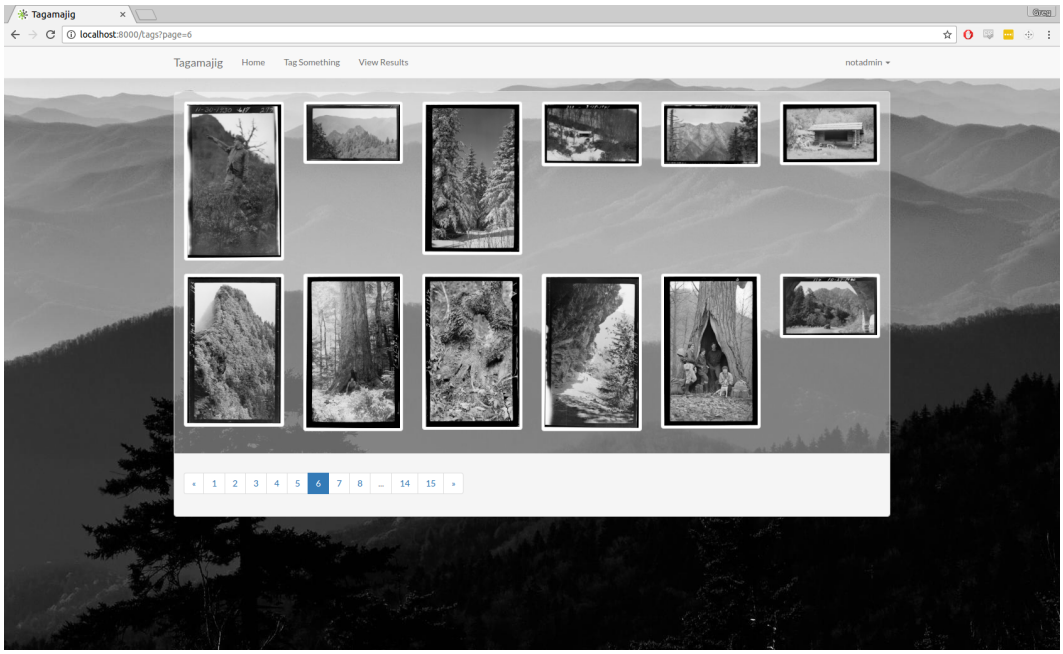

Figure 2.3: Paginated Image Collection View

The use of image thumbnails allows users to decide at a glance whether the image may contain recognizable features, or at least be clear enough to make out a particular feature type which the user can tag for further categorization of a particular image. Initially, the concept of a unidirectional "endless" image slideshow was considered for the purpose of displaying photos to be tagged, with new photos being dynamically appended to the end of the slideshow using AJAX queries. However, after developing and testing this interface, concerns arose with regards to user frustration, as many of the photos in the test collection are simply unrecognizable (a close-up shot of a flower, for example). With no control over what photo is appended to the slideshow with each AJAX call, it is possible that users could be continuously presented with unrecognizable photos, frustrating them to the point of quitting. The one-way slideshow concept also did not allow for users to return to previously viewed photos and update their contributed tags. By instead using simple thumbnail pagination, users are able to quickly scan and skip unrecognizable photos as well as return to previously viewed images and alter their past tags for each photograph.

2.3 Image Tagging Interface

The core component of the application is the user interface for tagging specific features within images. A high emphasis was placed upon providing the users with an interface which is uncluttered and intuitive. The interface can be seen in Figure 2.4. To assist users with identifying specific features within the images, the interface supports image zoom via either a mouse wheel or the zoom buttons located in the lower left corner of the image viewing container. Panning the zoomed image is supported via clicking and dragging the left mouse button.

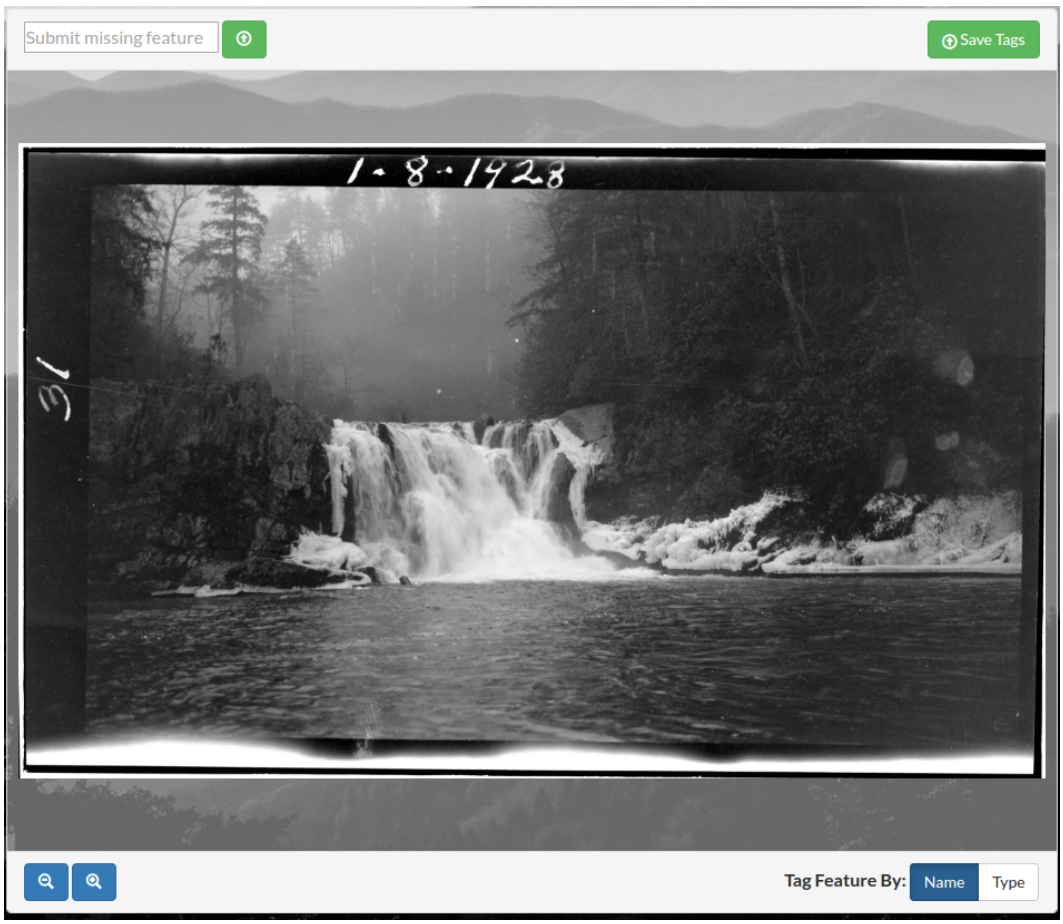

Figure 2.4: Image Tagging User Interface

To tag a specific point within the image as containing a particular landmark, a user simply needs to click on that point within the image to begin the tagging process.

Upon clicking, an opaque tag marker appears at the point of the click, and an attached container with an empty select element is attached to the tag marker. When the user enters one or more letters into the dynamic select element, it executes an AJAX request to the application's back-end which returns all possible matching landmarks and populates the select element with the returned landmark names. This real-time incremental search is useful for two reasons. Firstly, it can assist the user in their attempt to identify a particular landmark. For example, if a user is attempting to label a waterfall but cannot remember the name of the fall from memory, the user can input "fall" into the incremental search box, and a narrowed selection of landmarks containing the substring "fall" will be returned, allowing the user to browse a much more restricted and manageable list of possible matches in an attempt to jog his or her memory. Secondly, constraining users to only submit landmarks that exist within the database's Landmarks table via a select element, as opposed to a free-form input field where any input would be accepted, reduces the probability of unintentional user error as well as intentional submissions of incorrect (or even inappropriate) names. The dynamic search element of the tagging process is visible in Figure 2.5.

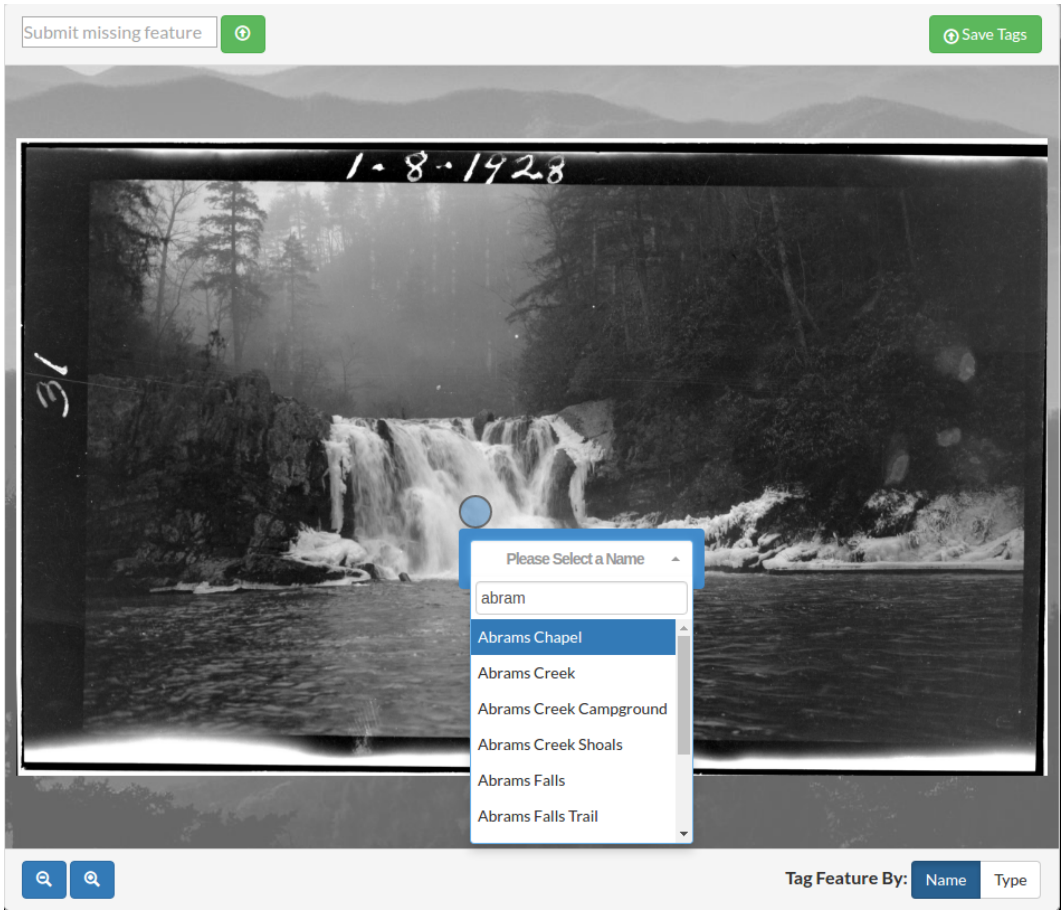

Figure 2.5: Image Tagging Incremental Search

The user may cancel the tagging action by clicking on another point in the image or by clicking the tag marker icon, which displays an 'X' when hovered. If the user has found a landmark name with which they wish to tag the feature, they may complete the tagging process of the feature by either clicking the appropriate option within the dynamic select element or scrolling to the proper select element with the mouse wheel or keyboard keys and pressing the Enter key.

After successfully tagging a feature with the desired landmark name, the tag marker icon remains in place. If the user hovers the mouse over the tag marker, a pop-up container displayed above or below the marker reveals the selected name of the landmark as can be seen in Figure 2.6. If the user wishes to remove the tag,

he or she simply needs to click the tag marker icon itself, marked with an 'X' upon hovering.

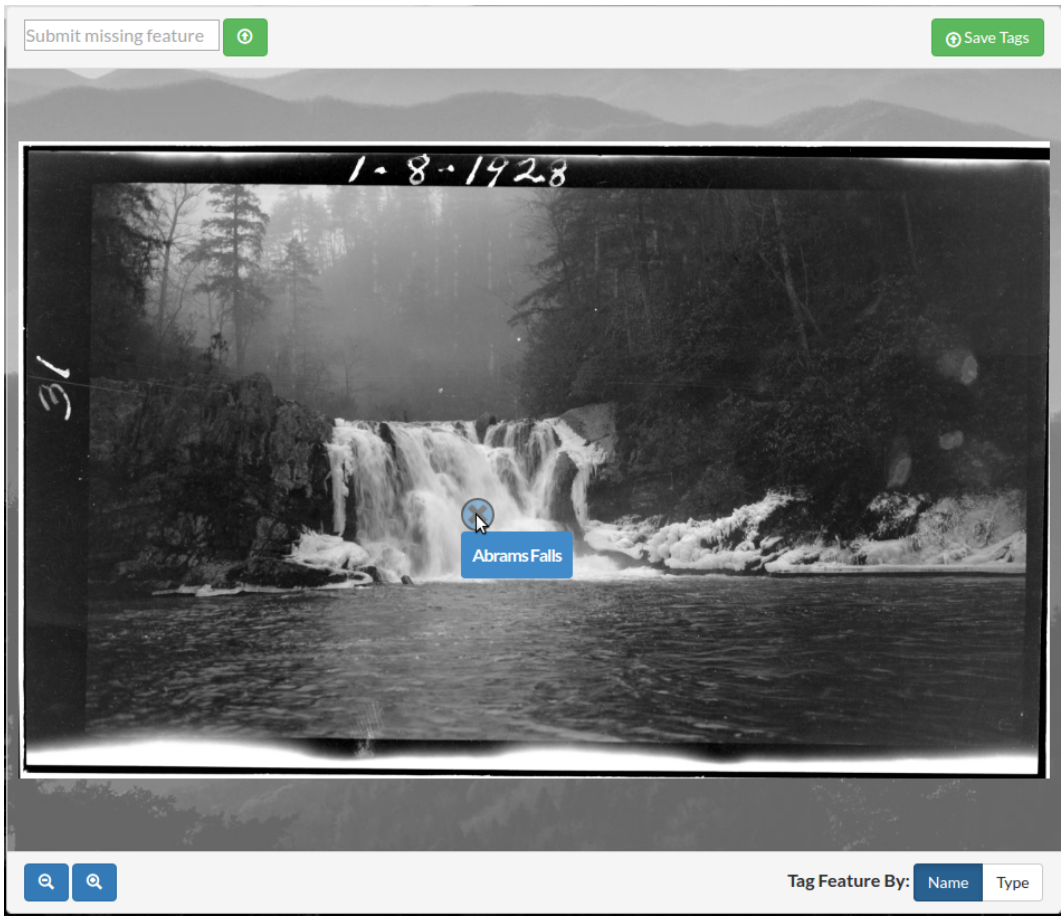

Figure 2.6: Tag Marker Mouse Hover

For instances in which a user may be unable to identify a feature as being a specific, unique landmark, he or she can still contribute to the identification of the image by tagging the feature's type. Like the landmark tagging function, tagging a feature by type provides the user with a dynamic incremental search element with which to tag the feature. Because the number of types is significantly less than the number of unique landmark names present in the database, the select element is populated with all type options before the user begins typing. However, the user can still use the keyboard to restrict the number of choices via an incremental search of available types. To differentiate a tag by type versus a tag by specific landmark

name, landmark name tags are color-coded in blue while type tags are colored green. An example of both types of tags can be seen in Figure 2.7. In this example, the user has already tagged the waterfall with a specific landmark name and is currently in the process of tagging the stream in the foreground by type.

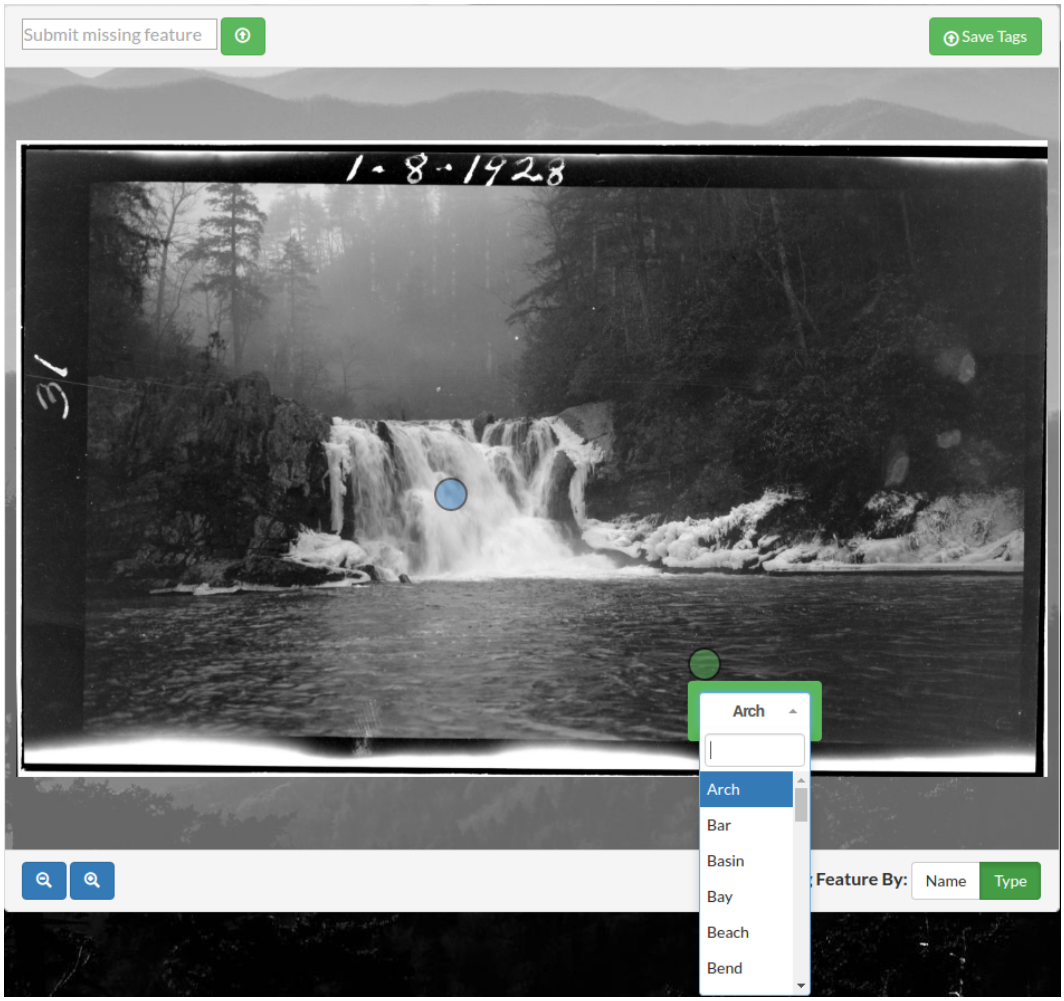

Figure 2.7: Tagging a Feature by Type

When a user finishes tagging a particular photo, the user may save and submit their tags by clicking the "Save Tags" button located in the upper right corner of the image tagging container. The "Save Tags" button saves all tags made by the user (and deletes all previously submitted user tags for the image) via an AJAX request

and presents the user with the option of remaining on the current page or returning to the previous collection of paginated thumbnails to continue tagging other images.

Finally, if a user believes that a particular landmark is missing from the list of available landmarks, the user can submit a request that the landmark be added by using the text input field in the upper left corner of the image container. Upon submission of the write-in request, administrators are alerted of the request via the admin panel where they are given the option of adding the write-in to the available choice of landmarks.

2.4 Aggregate Results View

To compliment the contribution of feature identification by individual users, an interface was developed to display tags provided by all users for each image. The interface uses a weighted rainbow heatmap to display each individual tag provided for the image, with tags representing more "popular" landmarks for the image being represented by "warmer" colors such as oranges and reds, and tags for landmarks which were chosen less often in comparison to the total number of contributed tags for the photo being represented with "cooler" colors such as blues and greens. In the example in Figure 2.8, we can see that the majority of users have provided a tag of "Abrams Falls" on or around the waterfall in the center of the image. Because this landmark is the most prominent of all tags provided for the image, all tags representing this landmark are colored red, while less popular tags such as "Abrams Creek" and "Bald River" which appear in the stream located in the bottom of the photo are painted with cooler colors.

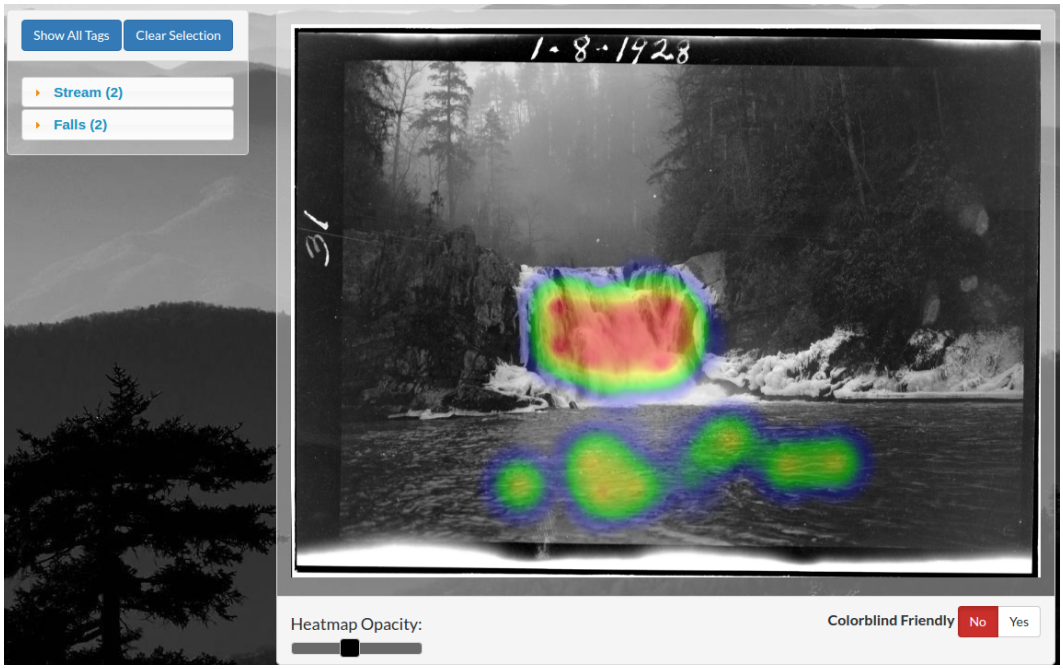

Figure 2.8: Aggregate Results Heatmap

Users may utilize the panel on the left side of the aggregate view panel to control which tags representing particular landmarks are displayed. The "Clear Selection" button will deselect all landmarks, while the "Show All Tags" button will select all landmarks, returning the view to its initial state upon loading the view. Individual landmarks can be toggled using buttons which are located in a drop-down menu under their respective type. Each type drop-down label also provides the number of landmarks in the image that are of that particular type. In the example provided in Figure 2.9, the user has deselected the most popular tag, "Abrams Falls", resulting in the weighting of the remaining tags in the heatmap to be recalculated. Because "Abrams Creek" is the most popular tag following the removal of "Abrams Falls", each "Abrams Creek" tag is now displayed as red instead of yellow as before.

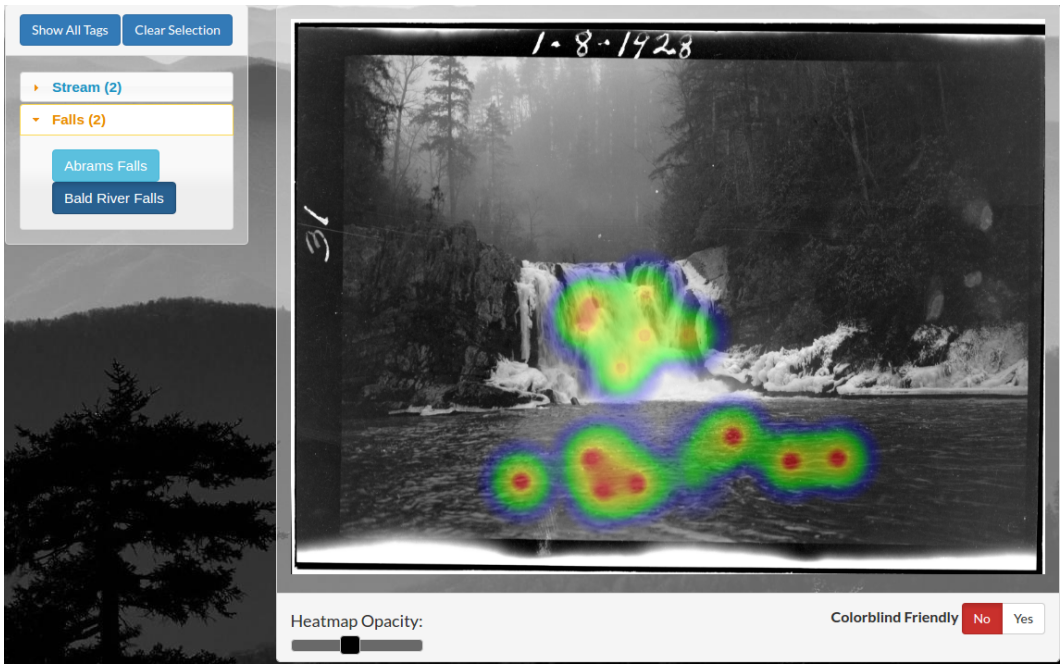

Figure 2.9: Heatmap Adjusted After Removal of Most Prominent Landmark

Upon initial load of the aggregate view, the opacity of the heatmap defaults to 45%. However, the user may adjust the opacity to their liking using the opacity slider located in the bottom left corner of the aggregate view container. The slider supports the adjustment of the heatmap's opacity from 0 to 100%, allowing the user to completely remove the heatmap temporarily in order to more easily explore features behind the tags or make the heatmap completely nontransparent to make each tag, especially for features which were tagged less often, more easy to discern. An example of the aggregate view with the heatmap adjusted to 100% opacity is shown in Figure 2.10.

Figure 2.10: Heatmap Opacity Adjusted to 100%

Finally, for individuals who are affected by red-green color blindness, a colorblind-friendly option is available by selecting the option in the bottom right corner of the aggregate view container. This changes the heatmap to utilize a sequential single hue color scheme deemed "colorblind safe". Like the rainbow color scheme which is utilized by the heatmap by default, the opacity of the colorblind safe heatmap is fully adjustable. The colorblind-friendly color scheme is shown in Figure 2.11, adjusted to use an opacity value of 100%. The color scheme for the colorblind-friendly mode was selected from the online resource "ColorBrewer" [4].

Figure 2.11: Colorblind Friendly Heatmap with 100% Opacity

2.5 Administrative Panel

An administrative panel is provided to allow user with admin privileges the ability to manage the application's content. The main panel is divided into three accordion sections: Photos, Landmarks, and Users. Under the photos section of the accordion element, administrators are given two options: "Add Photos", which directs the user to a simple page for uploading one or more new images, and "Manage Photos", which takes the user to a table view of all images currently in the database. The table provides pagination of the image collection as well as the ability to order the table by image name, number of tags for each image, and whether the image is currently active and able to be viewed and tagged by users. The table also provides support for instant search to narrow results. The image management table is shown in Figure 2.12.

Manage Photos

Show 10 entries Search:

Filename	Number of Tags	Active
roth0011_result.png	25	Yes
roth0105_result.png	2	Yes
roth0121_result.png	1	Yes
roth0001_result.png	0	Yes
roth0002_result.png	0	Yes
roth0003_result.png	0	Yes
roth0004_result.png	0	Yes
roth0005_result.png	0	Yes
roth0006_result.png	0	Yes
roth0007_result.png	0	Yes

Showing 1 to 10 of 170 entries Previous 1 2 3 4 5 ... 17 Next

Figure 2.12: Admin Photo Management Table

Selecting a particular photo from the table will provide the administrator with a view identical to that of the aggregate results heatmap view available to normal users. In addition to the heatmap container, the view includes the count of the total number of tags provided by users for the image, controls for setting the image to active or inactive, and controls for permanently removing the image from the collection.

Under the Landmarks section of the main admin panel, administrators are presented with the following options: "Manage Landmarks", "Manage Write-Ins", "Add Landmark", and "Add Type". An icon next to "Manage Write-Ins" presents the number of landmark write-ins submitted by users that remain to be reviewed by administrators. An example of the main administrative panel with two write-ins waiting to be evaluated can be seen in Figure 2.13.

Figure 2.13: Admin Panel - Two Landmark Write-ins To Be Reviewed

"Manage Landmarks" takes administrators to a table view similar to the view provided by the "Manage Photos" link. Like the "Manage Photos" table, the landmarks table supports instant search and allows administrators to sort landmarks by name or by type. Selecting a landmark from the landmarks table brings the administrator to an "Update Landmark" form which is pre-populated with the landmark's existing attributes. The form allows administrators to update the landmark's name, type, county, state, latitude, and longitude, as well as delete the landmark. The "Manage Write-Ins" option displays the list of current write-in landmarks submitted by users and provides administrators with the option to make a particular write-in a new landmark, delete a particular write-in, or delete all existing write-in suggestions. An example of this table is displayed in Figure 2.14.

Figure 2.14: Manage Write-ins Table

Selecting "Add to Landmarks" for a particular write-in will load a form identical to the "Update Landmark" form but with only the "name" field being pre-populated

with the name suggested by the write-in. The administrator is then required to assign a type to the landmark before submitting it. County, state, latitude, and longitude are all optional fields when creating a new landmark. When the landmark is submitted, the write-in is automatically deleted.

The "Add Landmark" option brings the administrator to an identical form but lacking any pre-populated fields. Finally, the "Add Type" provides a form consisting of a single text input field, allowing the administrator to submit a new type which can be assigned to new or existing landmarks for future tagging purposes.

The final section of the main admin panel is the "Users" section. This section offers two options: "Manage Users" and "Add User". Like the "Manage Photos" and "Manage Landmarks" options, the "Manage Users" option loads an interactive table listing all users. Users are able to be sorted by username, email address, or admin status, and like the other tables used for managing of photos and landmarks, the users table supports instant search and dynamic pagination. Selecting a particular user from the list gives administrators the option to delete that user and all tags associated with the user account. The "Add User" option allows an administrator to manually create a new user and give the new user admin privileges if needed.

Chapter 3

Implementation

Implementation details of the application will be discussed in detail in the following sections. However, for the purposes of clarity, we will first provide a brief description of each software component that was used in the development of the application.

Database

- **MySQL** - MySQL is an open-source relational database management system provided by Oracle Corporation [12].

Application Frameworks

- **Laravel** - Laravel is an open-source web application framework. It utilizes the model-view-controller (MVC) software architectural pattern and provides features to assist with the management of database relations, front-end view templating, and application security [17].
  - **Eloquent ORM** - Eloquent ORM (Object-Relational Mapping) is Laravel's approach to managing an application's database records. An object-oriented model is defined for each database table and is used to interact with database records, as opposed to record manipulation via raw SQL queries [16].

- **Blade Templates** - Blade is Laravel's templating engine for providing dynamic views to end users. Blade supports template inheritance to reduce the likelihood of copy-paste errors across views with shared layouts [15].
- **Artisan** - Artisan is Laravel's command-line interface, providing features to support application development such as generating application assets and performing database operations [14].

- **Bootstrap** - Bootstrap is an open-source front-end web framework. It provides templates for responsive front-end design and ships by default with Laravel [13].

Scripting

- **Python** - Python is an open-source high-level programming language provided by the Python Software Foundation [1]. For this project, Python scripts were used for parsing the landmarks data set and seeding tables within the project database.

- **jQuery** - jQuery is a JavaScript library developed by John Resig and the jQuery Foundation [18]. jQuery's primary purpose is to simplify client-side scripting by providing a simple but feature-rich syntax for easier manipulation of DOM elements. jQuery was used to implement the application's image tagging interface as well as in several open-source third-party plugins described in the following sections.

Software Stack A diagram summarizing the application's overall software stack is displayed in Figure 3.1.

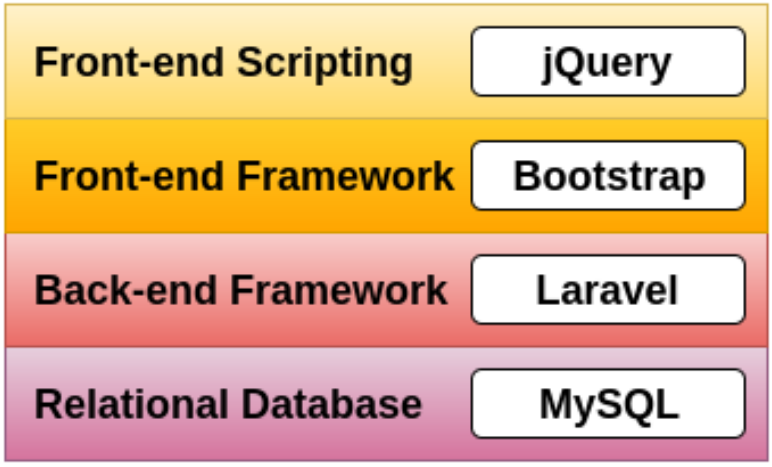

Figure 3.1: Application Software Stack

3.1 Back-end

3.1.1 Tagging Data Set and MySQL Database

One of the first priorities in the development of the application was obtaining a list of landmarks which could be used to provide users with options for tagging notable features within the Smoky Mountains photo collection. An appropriate data set was found via the United States Board on Geographic Names, which is responsible for maintaining uniform naming of geographic landmarks throughout the U.S. government [21]. The Board provides lists of geographic landmarks for the entire country and divided by state. Each element in these landmarks lists contains an excess of information about each landmark, including the landmark name, landmark type (mountain range, lake, etc.), state alphabetic abbreviation, county name, landmark latitude and longitude, and the landmark's elevation. The Tennessee geographic data set was utilized for this application, and the specific landmarks used was narrowed down by only using landmarks whose location was within a certain distance of the center of the Smoky Mountains National Park, where the photos in the image collection were taken.

MySQL was used as the relational database management system for this project. A diagram of the complete database is displayed in Figure 3.2 (ER model created with MySQL Workbench [11]). Before discussing table relationships within the project's database, independent tables with no inherent relations will be reviewed.

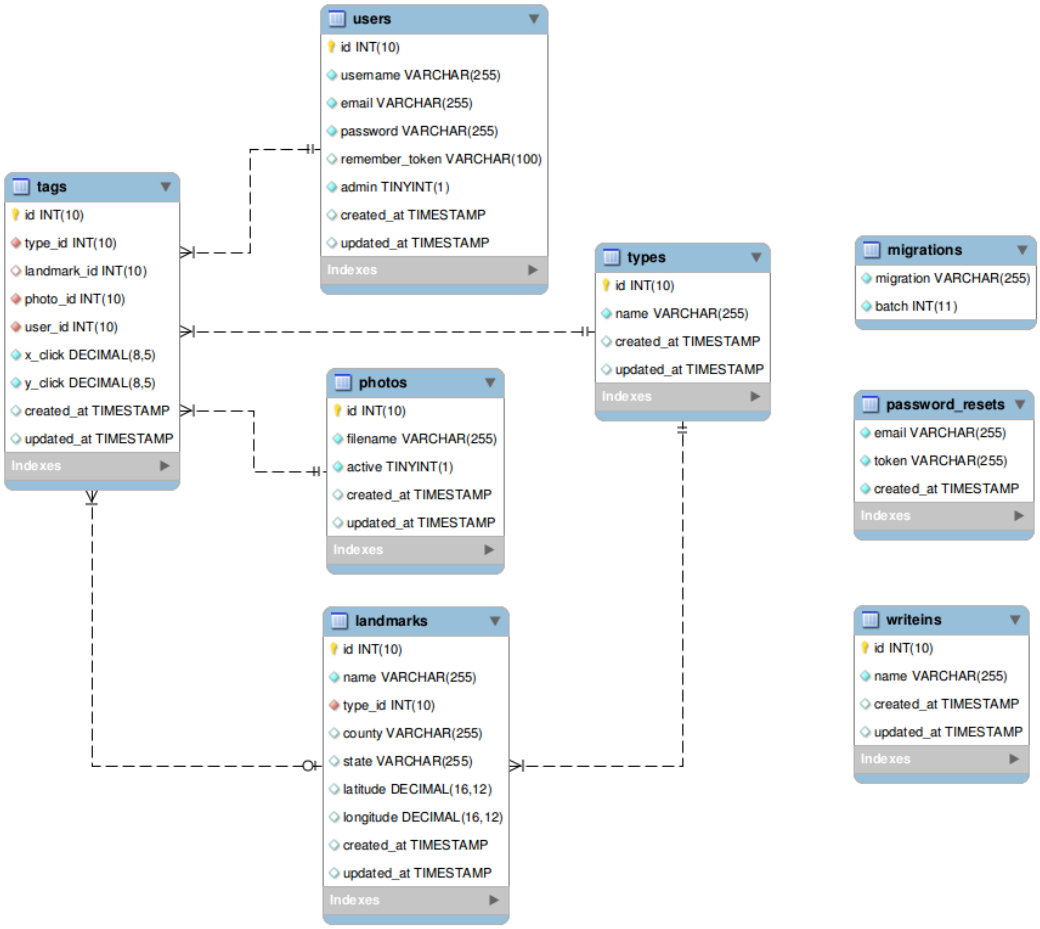

Figure 3.2: Diagram of Project Database Tables

Independent Tables

Migrations Table The Migrations Table is generated by default by the Laravel PHP framework utilized for this project. Its purpose is to track any changes to the application's database schema, such as the creation of a new table, updating of a column within a table, addition of foreign key constraints, or any other modifications

to the target database, which are referred to as "migrations" by Laravel. Tracking these migrations via the Migrations Table allows Laravel to track changes to the database in a manner similar to version control systems used for managing source code, allowing users to easily rescind previous modifications to the database as well as easily share database schemas with coworkers. Laravel's database migrations will be discussed in more detail in the "Laravel Framework" subsection below.

Password Resets Table The Password Resets Table is generated by Laravel to assist with user authentication. If a user forgets his or her password and requests that it be reset, an email is sent to the user which contains a password reset link which is appended with a unique token. The token, user email, and timestamp of the reset request are stored in the Resets Table. When the user follows the link provided in the email to reset their password, the unique token contained in their password reset URL is compared against the token stored for that user's email address in the Password Resets Table to prevent any malicious attempts by unauthorized individuals to reset a user's password. By default, the unique token is set to expire one hour after the password reset link is sent and the table entry is created.

Landmark Write-in Table The Landmark Write-in Table contains names of landmarks submitted by users who believe that the current set of landmark names offered for tagging of image features is incomplete. After being reviewed by an administrator, the specific entry in the table is deleted regardless of whether the administrator decides to accept or reject the user's suggestion.

Table Relationships

Relationships between database tables are implemented through the use of foreign keys. Specific table relationships can be examined in the entity relationship model in Figure 3.3. Primary keys are labeled with a yellow key icon. Foreign keys are labeled

with red icons. Columns which are defined as **NOT NULL** are labeled with filled icons, while the icons for nullable columns are unfilled.

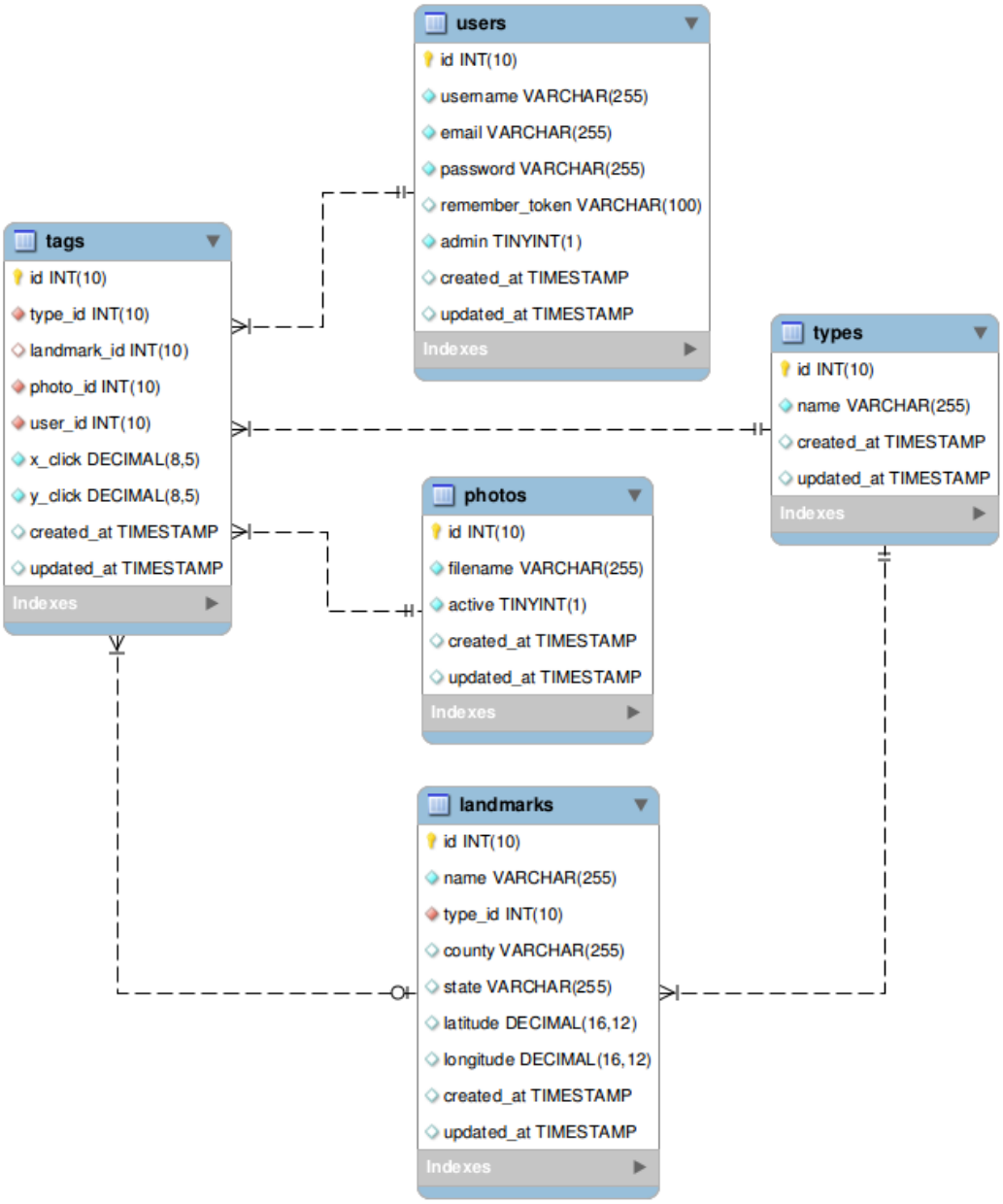

Figure 3.3: Diagram of Related Tables in Project Database

All tables use the Laravel naming convention of *id* as their primary auto-incrementing key. While some may argue that this naming convention is insufficient and requires increased aliasing when making complex queries, Laravel's Object

Relational Model (ORM) provides a layer of abstraction on top of MySQL and mitigates the need to use raw MySQL queries in most circumstances. Laravel's Eloquent ORM will be described in more detail in a later subsection. All tables created using Laravel migrations also contain timestamps for the time and date that the record was created and most recently updated by default.

Users Table Other than the standard auto-incrementing primary key, *created_at* timestamp, and *updated_at* timestamp, the Users Table contains the following fields: *username*, *email*, *password*, *remember_token*, and *admin*. The *password* field is encrypted using Laravel's implementation of the bcrypt hashing function. Bcrypt protects against rainbow table attacks through the use of a salt while also preventing brute-force attacks by increasing hash time with each iterative call for comparison to hashed values stored in a user's record. The *remember_token* is used to compare against a stored cookie on the user's machine when a user utilizes the "Remember Me" option when logging in. Finally, the *admin* field is a simple Boolean value to indicate whether the user has administrative privileges.

Photos Table The Photos Table contains two unique fields: *filename* and *active*. As the name implies, *filename* contains the file name of the photograph. The *active* field is a Boolean value indicating whether the image is currently available to be viewed and tagged by users and can be toggled by administrators via the admin panel.

Types Table The Types Table contains only one unique field: *name*, the name of a landmark type to which a specific landmark can belong ("stream", "cave", or "cemetery", for example).

Landmarks Table The Landmarks Table has a many-to-one relationship with the Types Table through the use of the foreign key *type_id*. Through the use of Laravel Migration foreign key constraints, all landmark records whose *type_id* corresponds to

a Type record that is deleted are deleted as well through the use of the **cascade** command when defining the foreign key reference in the migration. The *name* field is required and must be unique to the table. All remaining fields in the Landmarks Table - *county*, *state*, *latitude*, and *longitude* - are optional.

Tags Table Each entry in the Tags Table is used to record a single tag event on an image by a particular user. Therefore, the following foreign keys are needed to accurately record all necessary information: *type_id*, *landmark_id*, *photo_id*, and *user_id*. Like the foreign key relation defined in the Landmarks Table, all foreign keys in the Tags Table are constrained in that deleting a record in another table that corresponds to a foreign key entry in the Tags Table will result in any Tag records with that foreign key to be deleted as well. Because the Tags table has a many-to-one relationship with the Landmarks Table, which itself has a many-to-one relationship with the Types Table, the inclusion of the *type_id* foreign key in the Tags Table may appear to be redundant. However, because a user can tag a photo feature by specific landmark name *or* type, the *type_id* foreign key is necessary, as a tag by type does not contain a *landmark_id*. This is also why *type_id*, *photo_id*, and *user_id* are all defined as **NOT NULL** while *landmark_id* is nullable. The other unique entries in the Tags table, *x_click* and *y_click*, are also required. They contain the normalized coordinates of the tag within the image for easy translation regardless of how the image may be scaled on the user's machine or later displayed on the aggregate results view.

3.1.2 Laravel Framework

The open-source PHP web framework known as Laravel was used as the backbone of this project. Laravel provides support for web applications utilizing a model-view-controller design pattern and provides tools to support a multitude of features including dependency management, database query abstraction, traffic routing, user authentication, and unit testing. Laravel includes a commmand-line interface known as Artisan to assist with application development.

Database Migrations

Laravel's database migrations act as a sort of version control for an application's database. Using Artisan, developers can create migrations which are responsible for creating, updating, and deleting tables in the project database. Each migration consists of two methods: *up* and *down*. The *up* method is used to perform the desired operation upon the database, and the *down* method should be written to undo whatever action was taken in the *up* method. For example, if a migration's purpose is to create a new Users table, the *up* method should use the appropriate Laravel syntax to create a new table with the desired columns and constraints, and the *down* method should simply drop the Users table. An example Laravel migration - this one used to create the Users table for this application - is shown in Figure 3.4.

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    public function up(){
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('username')->unique();
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->boolean('admin')->default(0);
            $table->timestamps();
        });
    }

    public function down(){
        Schema::drop('users');
    }
}
```

Figure 3.4: Example Migration - Users Table

After the desired migrations are created, they can be executed in the order in which they were created via the command line using the artisan command `php`

`artisan migrate`. To undo the previous migration (calling the *down* method on all migrations which were most recently executed), the artisan command `php artisan migrate:rollback` can be used. Finally, invoking the command `php artisan migrate:reset` will call the *down* method on all project migrations. This migrations system makes it incredibly easy to track database changes during the lifespan of the project's development and easily deploy the project on a new server when ready for production. Laravel's database migrations were utilized for the entirety of this project.

Eloquent ORM

Instead of directly interacting with the application's MySQL database via raw queries, Laravel supports an object-relational mapping known as Eloquent. Using the Eloquent ORM, each table in the database has an analogous model which is used to query the database as well as insert, update, or delete records. A skeleton model (example model User in this case) can be generated from the command line using the Artisan command `php artisan make:model User`. Using the `--migration` flag will generate a database migration that corresponds to the model. Models can then be used to query the corresponding database table using the Laravel syntax and return one or more instances of the class. Figure 3.5 illustrates the Laravel ORM equivalent of the raw MySQL query `select * from users where admin = 1 order by name desc;` before iterating over the returned array of User objects.

```
$users = User::where('admin',1)->orderBy('name','desc')->get();

foreach ($users as $user) {
    echo $user->email;
}
```

Figure 3.5: Example Eloquent Query

Updating, creating, and deleting records are similarly straightforward thanks to Eloquent's syntactic sugar. However, one of Eloquent's greatest strengths is its ability

to define relationships among models in the form of model methods. For example, in this application there exists a one-to-many relationship between a User record and Tags, i.e. a user may have many tags. By declaring the method

```
public function users(){
    return $this->belongsTo('App\User');
}
```

within the Tag class and declaring the method

```
public function tags(){
    return $this->hasMany('App\Tag');
}
```

within the User class, we may then query the database for a count of all tags belonging to the user with the following command:

```
$tag_count = $user->tags()->count();
```

Using Laravel's relationships in combination with its simple query building syntax makes the handling of large collections of database records much more in line with the principles of object-oriented paradigm and greatly reduces development time and error rate in comparison to making raw database queries.

Controllers

Controllers are used by Laravel to handle the bulk of request-related logic, including parsing of the request, querying the database via the Eloquent syntax, working with raw data, and returning data and views to the client-side user. Controllers provide a simple way to group related requests into a single class. The following custom controllers were utilized for this application:

UserController The User Controller is responsible for returning the user home page view and handling all logic related to the user changing his or her password.

PhotoController The Photo Controller handles creating the paginated thumbnail view for browsing photos as well as the Aggregate Results View for an individual image.

TagController The Tag Controller is responsible for returning the Image Tagging Interface view for a particular image. It also saves all submitted user tags and write-in landmarks for an image and makes the appropriate query and returns the JSON results for the interface's incremental search.

AdminController The Admin Controller handles all administrative tasks such as uploading new photos, loading the main admin panel view, or granting admin privileges to new users.

Security

Laravel provides a number of tools to assist with the development of a secure web application.

Authentication Laravel ships with several authentication controllers out of the box, including controllers for user login, registration, and password reset. The Artisan command `php artisan make:auth` will create the necessary routes and views to support these built-in authentication controllers. The `Auth` facade (similar to a static class interface) can be used to check whether a user is authenticated and retrieve an instance of the current authenticated user.

Middleware Laravel Middleware provides a convenient mechanism for filtering requests within the application. Within a middleware, the incoming request can be checked to meet certain criteria before allowing the user's request to proceed. If the middleware rule is not met, the user can be automatically redirected to the appropriate page. For example, the `Admin.php` middleware confirms that the requesting user has admin privileges. If the user lacks the proper permissions, they

are redirected to their homepage. Middleware can be attached to individual routes or groups of routes within `routes.php`. However, a more convenient approach is often to assign the middleware to the appropriate controller by calling the `middleware` method in the controller's constructor. For example, this application's `AdminController.php` constructor appears as so:

```
public function __construct(){
    $this->middleware(['auth', 'admin']);
}
```

This attaches both the `admin` middleware as well as the standard `auth` middleware to the controller. If a user's request calls any method within this controller, the user must be both authenticated and have admin privileges. Because of the way that middleware "stacks", if the user is authenticated but does not have admin privileges, they will be redirected to the user's personal homepage. However, if they lack authentication, they will be redirected to the application's login page.

Form Validation Laravel also provides support for form validation which is simplified through the use of syntactic sugar. The example below illustrates how Laravel's `validate` method handles form input from a post request.

```
$this->validate($request, [
    'email' => 'required|email|unique:users|max:255',
    'password' => 'required|confirmed'
]);
```

The above code snippet is used to validate a form for creating a new user. The `validate` method does the following: for the 'email' line, it confirms that the user has provided an email, that it is a valid email format, that the email is unique to the Users table, and that it is a maximum of 255 characters. For the 'password' line, the `validate` method confirms that the user provides an password and that the

password matches the password provided in the form's *password confirmation* field. If any of these conditions are not met, the user is automatically returned to the form and alerted with the appropriate errors.

3.1.3 Helper Scripts

Several Python scripts were created to assist with the seeding of the application database. Two scripts were responsible for parsing of the Board of Geographic Names data set and creating the appropriate records within the Types and Landmarks tables. Another script accepts the directory which contains the image collection as a command line argument and seeds the Photos table with an entry for each image's filename and path. A final script was used for seeding the Users table with randomly generated users, with usernames and emails made by concatenating two randomly chosen words from the standard Unix `words` file located in `/usr/share/dict/`. The generation of random users was necessary for testing purposes.

3.2 Front-end

3.2.1 Laravel Blade Templates

Laravel creates user-side views through the use of their Blade templating engine. Blade templates work primarily through the use of two key features: *template inheritance* and *sections*. The idea behind these features is simple - rather than rewriting portions of markup which are shared across views (for example, a navigational header, or the inclusion of certain CSS stylesheets or JavaScript files), a template can be created which contains a markup "skeleton" which can be shared across a series of views. HTML markup specific to a certain view can then be inserted in specific section of the template using the `@yield` directive in the appropriate area within the template. For example, when writing the master template of an

application, the `<body>` element can be left blank except for a `@yield` directive, as follows:

```
<body>
    @yield('body_content')
</body>
```

Each child view that needs to inherit the parent template may then do so by *extending* the parent template and inserting that view's specific HTML markup into the target section of the inherited template using the `section` directive. For example, extending the above template, assuming the template is named `main_layout.blade.php`:

```
@extends('main_layout')


@section('body_content')
    <p>This will appear within the body of the main layout template.</p>
@endsection
```

Blade templating greatly reduces copy-paste errors across views and simplifies the changing of stylesheets and scripts which are utilized by multiple views within the application. Blade also includes syntactic sugar for simplifying the displaying of data and execution of PHP control structures such as branches and loops.

3.2.2 Bootstrap Framework

Laravel ships with the Bootstrap CSS and JavaScript front-end framework by default. Bootstrap is a responsive front-end framework that easily scales from mobile devices to desktop machines with little or no modification necessary. The default Bootstrap theme was utilized for this project. However, custom Bootstrap themes can easily be applied to the application to override the default theme at a later time if desired.

3.2.3 jQuery Plugins

The responsive elements of this application's user interfaces were developed primarily using the jQuery JavaScript library. jQuery provides a simpler syntax for manipulating DOM elements and handling events and provides support for the development of plugins and widgets. A custom plugin was developed to support the image tagging interface, while existing plugins were utilized for the aggregate results view and other miscellaneous tasks.

Tagger Plugin

The semi-opaque tag marker icons and the pop-up containers displaying the names of tagged landmarks were inspired by the design of the open-source **Taggd** jQuery Plugin developed by Tim Severien [19]. However, the underlying method of implementing the plugin is quite different and borrows much more heavily from the **imgNotes** jQuery Plugin by Wayne Mogg [9]. The Tagger plugin is attached to the currently viewed image and is responsible for maintaining state information for all tags that are currently attached to the image, including the *landmark_id* and/or *type_id* of each tag as well as each tag's normalized coordinates within the image. The Tagger plugin is responsible for all event handling related to the tag markers, including creation of new tag markers when clicking on a new point in the image, displaying of tag names when hovering over a tag marker, and deleting a tag marker on click. The plugin also handles all AJAX requests to the server for the incremental search implemented for the purpose of narrowing tag options. Finally, the plugin provides `export` and `import` methods which are used to send all existing tags to the server to be saved, and repopulate the image with tags previously submitted by the user, respectively.

Plugin Dependencies The Tagger plugin relies on several dependencies in order to function. The primary dependency is the **ImgViewer** plugin, also developed by

Wayne Mogg [10]. ImgViewer supports basic user interaction with the image itself, including zooming and panning via click and drag of the left mouse button. The plugin also translates zoomed coordinates to properly scaled coordinates for use by the Tagger plugin. The ImgViewer plugin itself relies on two other plugins: **Zoetrope** [2] and **jquery-mousewheel** [7], which handle basic animations and mouse wheel behavior. Finally, the **Select2** plugin provides support for dynamic select elements with incremental search via AJAX request [5].

Heatmap Plugin

The heatmap plugin utilized for the aggregate results view of the application, **heatmap.js**, was developed by Patrick Wied [23]. The heatmap plugin was wrapped in a custom jQuery plugin (**photoresults.js**) which was responsible for configuring and instantiating the heatmap plugin. Upon first loading the results view, the photoresults.js plugin iterates over all tags that exist in the aggregate view and determines which tag occurs most frequently in order to properly weight the value of each tag entry in the heatmap. The photoresults.js plugin also handles all events related to landmark filtering, the opacity slider, and the colorblind-friendly switch and reconfigures the heatmap settings accordingly when a change in one of the input variables is detected.

Miscellaneous Plugins

Several other open-source jQuery plugins were used throughout the project. **Bootstrap-slider** [8], developed by Kyle Kemp and Rohit Kalkur, was used for the opacity slider in the aggregate results view. **Bootstrap-checkbox** [22] was used for the toggle switches which control the tag-by-landmark-name versus tag-by-type option in the image tagging interface as well as the colorblind-friendly toggle switch in the aggregate results view. The **match-height** plugin [6] was used to match the height

of adjacent Bootstrap elements. Finally, the **DataTables** plugin [20] was utilized for the pagination and management of all assets within the admin panel.

Chapter 4

Evaluation

4.1 Alternative Design Considerations

Before the current approach of allowing users to select a specific image to tag via paginated thumbnails was implemented, an alternative design was considered. This alternative approach would have acted as an "endless" one-way slideshow, allowing the user to stay on a single page, provide tags for an image, and then select the scroll button on the right side of the image to automatically submit their tags and provide the user with a new image. This approach was strongly inspired by Zooniverse's "Floating Forests" web application discussed in this composition's introduction. However, this design has several shortcomings which more negatively affected the user experience than anticipated.

The first shortcoming to this approach is the risk of overwhelming the user with unidentifiable images. The photo collection for which this application was developed contains many images which users can simply not be expected to recognize, either due to the ambiguous content of the image or the quality of the image itself. Because this application relies on promoting participation and feedback by providing users with content that they find interesting, delivering a series of images that are unrecognizable increases the risk that users abandon the application due to boredom or frustration.

The second shortcoming of this design is that users are unable to revisit previously viewed images and view or update their original tags. By abandoning the slideshow approach and providing full access to the image collection via paginated thumbnail view, both of these issues are resolved. The thumbnails allow users to quickly scan a portion of the collection for noteworthy images while ignoring images which appear difficult or impossible to identify. This layout also allows users to revisit previously tagged images, view their past input, and update their contributed tags if necessary.

4.2 Limitations

While initial design and development has produced a working application, examination of the finished product reveals some shortcomings or room for improvement with regards to some design choices. Some of these flaws could be rectified relatively quickly by implementing relatively simple additions to an existing interface, while other shortcomings would require more extensive examination and reevaluation of the root purpose of a particular interface before implementing a redesigned solution.

4.2.1 Limitations of Image Tagging Interface

While emphasis was placed on the delivery of a simple and user-friendly interface for tagging features, there is still room for the improvement of at least two minor components within the tagging view.

The first element that may be improved is the tag markers. While these markers are easily created and destroyed by the user, they cannot be relocated after their creation at this time. This means that if a user wishes to modify a tag's location within the image, the user must destroy the original tag before creating a new tag at the desired position. A more intuitive way of doing this would be to implement drag-and-drop support for existing tags, allowing the user to drag an existing tag to a

new location by clicking and holding the primary mouse button over the tag marker while moving it with the mouse.

A second potential improvement to the tagging interface concerns the current method of saving tags. At this time, the user is required to update their contribution to a particular image by clicking the "Save Tags" button located in the upper right corner of the image tagging container. Clicking this button sends all user tag data for the image to the server for storage via an AJAX request. This is potentially problematic as a user could easily forget to use the "Save Tags" button before leaving the page. A potential solution is to remove the button and simply update the user's tag records for the image by sending an AJAX request each time the user creates or deletes any tag. This would increase traffic overhead between the client and server but would alleviate the burden of the user being required to remember to manually save their input before leaving the page.

4.2.2 Limitations of Aggregate Results View

The Aggregate Results View was meant as more of a proof-of-concept illustrating one potential way that the combined input for all users could be visualized for an individual image. The interface succeeds in revealing the locations within an image that users tend to consider most likely to contain a notable feature, and coloring more popular tag choices in darker, more "intense" colors does somewhat succeed in highlighting which user opinions are more popular at a glance. However, this approach does have a limitation, which can be more easily understood by examining Figure 4.1.

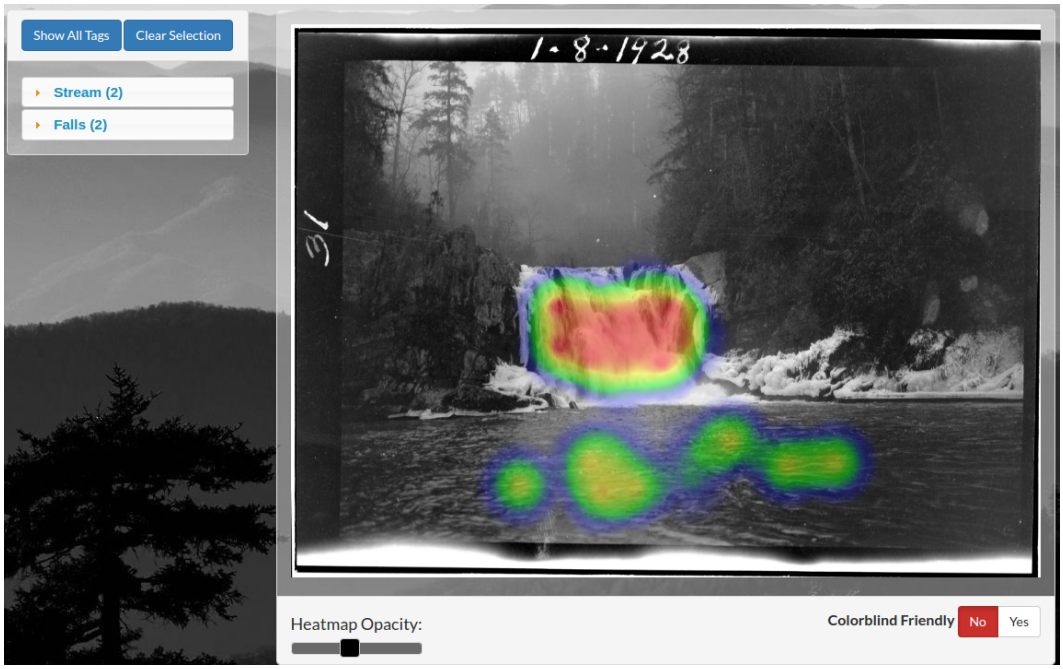

Figure 4.1: Aggregate Results Heatmap - Limitations

In this example, the overwhelming majority of users have labeled the waterfall in the center of the photo as "Abrams Falls", with a few users labeling the same waterfall as "Bald River Falls". However, even though the individual tags submitted as "Bald River Falls" are more lightly colored to indicate that it was a less popular choice among user submissions, these tags for "Bald River Falls" are completely concealed by the more popular and darkly colored heatmap markers for "Abrams Falls". In hindsight, this should be expected, as users will naturally place "competing" tags for landmarks of the same type in the same general area of the image where the landmark appears. However, this leads to a situation where the majority opinion for the name of a particular landmark tends to obscure less popular user opinions. This can be observed in Figure 4.1 above, where it is apparent that users provided many more tags for the waterfall in the image than the stream beneath it. While it is likely that the popular opinion is correct if a sufficient number of users provide input, there are rare instances in which the less popular opinion may be more accurate, in which case it would be useful to have a method of viewing this user input as well.

One possible solution to reveal less popular tags which may be concealed by more popular submissions would be to somehow highlight the heatmap marks of a particular landmark when the user hovers over the landmark name on the selection panel located on the left side of the screen. When hovering over a button for a particular landmark name, all other landmark tags which are currently active within the heatmap could be redrawn using a "diluted" or more transparent color scheme, allowing the desired landmark tags to be visible while retaining their properly weighted color values based on their frequency versus the entire visible tag set.

4.3 Additional Applications

One of the major strengths of the application is the ease with which it can be applied to a wide variety of image collections for the purposes of labeling and categorization. All that is required is a target image collection and an appropriate data set for tagging the desired "landmarks", and the application can be deployed with little or no modification needed to its underlying functionality. In fact, even if specific feature names are not available within a new data set, providing only a data set for the Types table still allows users to categorize images for easier handling and labeling of the target image collection at a later time.

The example provided in Figure 4.2 illustrates a theoretical image collection which could be targeted for labeling and organization using the application.

Figure 4.2: Example Use - Sports Teams and Players

The example image is a screen capture of a soccer match, and the example data set uses soccer teams for the Types table and individual players for the Landmarks table. Much like the Smokey Mountains collection and data set used for developing the application, users can label specific "landmarks" (in this case, specific players), or simply provide a tag specifying that a landmark type (the football club) is present within the image, allowing users who are more familiar with the tagged football club to explore images containing those tags in hopes of identifying specific players.

Chapter 5

Future Work

While the application produced for this study is a complete and fully functioning piece of software, it can be considered a backbone for a much more ambitious project when considering the possible additions to its existing functionality.

5.1 Sorting Collection by Landmark Type

While tags by landmark type are presently able to be collected, users are unable to browse images based on which images contain certain Type tags at this time. Allowing users to view thumbnails of images which have been tagged as containing a particular landmark type would allow users who are more familiar with a particular landmark type to focus their efforts on providing specific landmark names for these images. For example, a cemetery is a landmark type which is easily distinguishable for the average user, but most users are unlikely to recognize the particular cemetery present within an image. However, a local historian with extensive knowledge of the area's historic cemeteries could likely recognize and tag many of these landmarks with relative ease if he or she was presented with a collection of images containing only such landmark types. Updating the paginated Image Collection Navigation view to support narrowing by type would be a relatively simple addition that could

increase the rate at which images containing type tags are later correctly identified with regards to a specific landmark name.

5.2 Aggregate View by Landmark Type

Potential shortcomings of the current Aggregate Results View are discussed in Section 4.2.2. Beyond its present flaws, the interface does not have a way of displaying tags by type at this time. While efforts should first be focused on addressing the present issues of the view when displaying tags by landmark name, future work could also incorporate a complimentary view of tags by type.

5.3 Google Maps Integration

An untapped resource within the existing tagging data set is the existence of latitude and longitude data for each specific landmark within the data set. One possible use for this data would involve integrating the most likely location of each image into a Google Maps plugin. The latitude and longitude for the most popular landmark from each image could be extracted and used to place a marker within a Maps plugin, similar to the example seen in Figure 5.1.

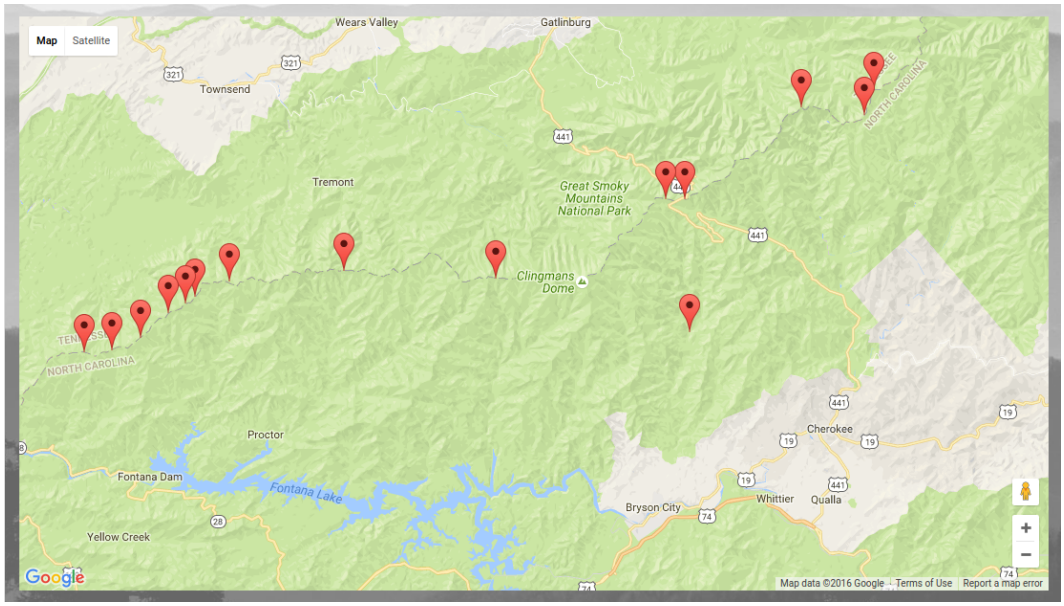

Figure 5.1: Potential Google Maps Integration

Each marker could display a thumbnail of the "best guess" image when hovering and activate a pop-up modal containing the corresponding image when clicked.

5.4 Tablet/Touchscreen Support

Support for tablets as well as touchscreen laptops and desktops should be a relatively simple addition thanks to the use of the flexible Bootstrap framework and the simplicity of the image tagging interface. Expanding support to smartphones is not recommended due to their limited screen size.

5.5 Digital Assets Management Integration

The application's image collection is currently stored locally within the public images directory. To increase flexibility regarding the control of available images and decrease the storage demand on the application's host server, the application could be modified to use the library's digital assets management system. The management system is built around the Islandora framework, an open-source framework created to manage

and distribute digital collections [3]. Because the library's management system utilizes a RESTful API to manipulate the Islandora framework, integrating the system with the application to serve images from the remote digital collection should be a relatively straightforward undertaking.

5.6 User Contributions to Data Set

At this time, users can only contribute to the tagging data set by suggesting "missing" landmarks via the write-in text box in the image tagging view. If the write-in suggestion is accepted by an administrator, that administrator is responsible for providing all other metadata for the new landmark entry, including latitude and longitude, type, and county. One possible future addition to the application could allow users to provide suggestions for any or all of these fields when submitting a write-in. Users could also possibly be allowed to browse the existing landmark data set and provide feedback similar to the write-in submission feature if the user believes that existing records within the data set are inaccurate.

Chapter 6

Conclusion

This project has focused on the creation of an application for the purposes of labeling an unsorted image collection via the contributions of volunteer users. Users are able to label notable features within an image either as a specific, unique landmark or as a more broad "type" to which each landmark belongs. This allows users with a broad range of knowledge about the area to contribute to the labeling and categorizing of the image collection. Users who are more familiar with the region may be able to identify a feature as being a particular landmark with relative ease, while those who are less knowledgeable of the area may still contribute to the collection's organization by labeling a feature's type present within each image which would be incredibly difficult using existing image processing or pattern recognition techniques. re While the application's development focused on a specific image set, it was designed such that a completely unrelated image collection and complementary data set could be imported with relative ease, allowing it to be used for a variety of image types. Furthermore, this application is in many ways a foundation upon which many more robust and practical features may be added to increase its use both to its administrators and its volunteer users. Nevertheless, the application is fully functional in its present state and fulfills the primary goal of this project - to provide a tool for assisting the

University of Tennessee Library's Digital Collections Department in the classification of unlabeled images.

Bibliography

[1] Python, accessed April 04, 2016. https://www.python.org/. 23

[2] Zoetrope, accessed June 05, 2016. https://github.com/FormstoneClassic/Zoetrope. 38

[3] Islandora, accessed September 28, 2016. http://http://islandora.ca/. 49

[4] Brewer, C., and Harrower, M. ColorBrewer, accessed July 08, 2016. http://colorbrewer2.org/. 17

[5] Brown, K., and Vaynberg, I. Select2, accessed June 08, 2016. https://github.com/select2/select2. 38

[6] Brummitt, L. matchHeight, accessed June 05, 2016. https://github.com/liabru/jquery-match-height. 38

[7] jQuery Foundation. jquery-mousewheel, accessed June 05, 2016. https://github.com/jquery/jquery-mousewheel. 38

[8] Kemp, K., and Kalkur, R. bootstrap-slider, accessed June 22, 2016. https://github.com/seiyria/bootstrap-slider. 38

[9] Mogg, W. ImgNotes, accessed June 05, 2016. https://github.com/waynegm/imgNotes. 37

[10] Mogg, W. ImgViewer, accessed June 05, 2016. https://github.com/waynegm/imgViewer. 38

[11] MySQL. MySQL Workbench, accessed August 29, 2016. http://www.mysql.com/products/workbench/. 25

[12] Oracle Corporation. MySQL, accessed April 04, 2016. http://www.mysql.com. 22

[13] Otto, M., and Thornton, J. Bootstrap, accessed May 09, 2016. http://getbootstrap.com/. 23

[14] Otwell, T. Artisan, accessed May 09, 2016. https://laravel.com/docs/5.3/artisan. 23

[15] Otwell, T. Blade Templates, accessed May 09, 2016. https://laravel.com/docs/5.3/blade. 23

[16] Otwell, T. Eloquent ORM, accessed May 09, 2016. https://laravel.com/docs/5.3/eloquent. 22

[17] Otwell, T. Laravel, accessed May 09, 2016. https://laravel.com/. 22

[18] Resig, J. jQuery, accessed June 02, 2016. https://jquery.com/. 23

[19] Severien, T. Taggd, accessed June 04, 2016. https://timseverien.com/taggd/v3/. 37

[20] SpryMedia Ltd. DataTables, accessed August 06, 2016. https://datatables.net/. 39

[21] United States Board on Geographic Names. Domestic and Antarctic Names, accessed April 04, 2016. http://geonames.usgs.gov/domestic/download_data.htm. 24

[22] Vasily, A. Bootstrap-checkbox, accessed June 23, 2016. https://github.com/vsn4ik/bootstrap-checkbox. 38

[23] Wied, P. heatmap.js, accessed June 22, 2016. https://github.com/pa7/heatmap.js. 38

[24] Williams, A., et al. A Computational Pipeline for Crowdsourced Transcriptions of Ancient Greek Papyrus Fragments. *2014 IEEE International Conference on Big Data* (2014). 2

[25] Zooniverse. *Floating Forests*, accessed March 14, 2016. http://www.floatingforests.org. 3

Vita

Gregory Simpson was born in Bristol, TN to the parents of Bryan and Kathryne Simpson. He is the older of two children, having one sister, Rebecca. He attended Van Pelt Elementary, Virginia Middle, and Virginia High Schools in Bristol, VA. After completing high school, he enrolled at the University of Tennessee, Knoxville, and pursued an undergraduate degree in Psychology. He obtained a Bachelors of Arts degree in Psychology from the University of Tennessee in May 2007 before continuing on to work in the field of mental health care. Gregory returned to the University of Tennessee in 2012 to pursue a second undergraduate degree in Computer Science. He obtained a Bachelors of Science degree in Computer Science in May 2015 before accepting a graduate teaching assistantship at the university. Gregory completed his Masters of Science degree in Computer Science in December 2016.