

Enhancing Collaborative Data Collection Using Mapillary

Jay Dahlstrom, Christian Matthews, Tommy Nguyen
August, 2017

Recommended Course of Action

Our team has devised four core Python-based script tools in order to aid Mapillary in their development. These script tools are aimed at providing efficient data filtering, automating representation of photos, and a method of verifying data quality. The four tools are comprised of: geojson to shapefile converter, a spatial analysis tool to create heat maps and priority ranking amongst photos, a false positive analyzer, and a tool to automate publishing to a web map. Mapillary will be able to use these tools as a foundation for processing their photos while also integrating and developing them further for use in their APIs. From the results, users and Mapillary will better understand and represent relationships among various aspects of a complex world.

Table of Contents

1. Introduction.....	5
1.1 Background.....	5
1.2 Goal.....	5
1.3 Objectives.....	5
1.4 Scope.....	6
1.5 Social-Ecological System.....	7
2. Design & Methods.....	7
2.1 Data Acquisition.....	7
2.2 Software Requirements.....	8
2.3 Python Scripts.....	9
2.3.1 Mapillary to ESRI (GeoJSON to Shapefile).....	10
2.3.2 Priority Ranking and Heat Map Generation (Analyze Mapillary Photos).....	13
2.3.3 Image to Object Distance Calculation (Calculate Distance).....	15
2.3.4 Data to ArcGIS Online (Publish Service).....	18
2.4 ArcGIS Online.....	21
3. Results.....	21
3.1 GeoJSON to Shapefile.....	21
3.2 Results of Analyze Mapillary Photos.....	23
3.3 Results of Shapefile to ArcGIS Online.....	24
3.4 Results of False Positive Analysis.....	25
3.5 Results of ArcGIS Online Web Map.....	26
4. Discussion.....	26
4.1 Successes.....	26
4.2 Lessons Learned.....	27
4.3 Recommend Course of Action.....	28
4.4 Future Implementations.....	28
5. Business Case and Implementation Plan.....	30
6. Literature Cited.....	31
7. Appendix.....	31

Table of Figures

Figure 1 - Toolbox	9
Figure 2 - GeoJSON to Shapefile Dialogue.....	10
Figure 3 - GeoJSON Example	11
Figure 4 - GeoJSON to Shapefile Code	12
Figure 5 - Analyze Mapillary Photos Dialogue.....	13
Figure 6 - Analyze Mapillary Photos Code	14
Figure 7 - Calculate Distance Dialogue	15
Figure 8 - Calculate Distance Code	16
Figure 9 - Publish Service Dialogue	18
Figure 10 - Publish Service Code.....	20
Figure 11 - Map of Converted Shapefile	22
Figure 12 - Heat Map.....	23
Figure 13 - Ranked Grid.....	24
Figure 14 - ArcGIS Online.....	24
Figure 15 - Output Attribute Table.....	25
Figure 16 - Scatter Plot of Distance Results.....	25
Figure 17 - Ranked Grid Web Map	26

1. Introduction

1.1 Background

Since the launch of Mapillary in 2013, the company has obtained “over 155 million photos contributed over seven continents” (Mapillary 2017). The photos are used to compile large vista datasets, from which Mapillary has constructed object class categories through their image processing. Insightful information can then be extracted from this data. Current uses of this data include editing OpenStreetMap (OSM), extracting recognized traffic signs, and integrating the data with ArcGIS Pro and ArcGIS Online. As photos continue to be uploaded from many sources, Mapillary is continuously seeking ways to manage and understand the dynamic relationships among these datasets.

To meet this objective, Mapillary needs their data to not only be used, but also for their collection and internal processes to be efficient and accurate. Currently, the semantic segmentation process will classify an object with a minimum of 64 pixels. But, even when combined with a specific confidence interval, this process can lead to false-positive features being tagged. Introducing false-positives into datasets can prove extremely problematic when users begin extracting data and using it for other purposes. For instance, a transit network model can be seriously altered by a bike lane dataset that includes bike paths that do not exist. In addition, while collecting as much data as possible is in any company’s best interest, it is our view that for a flagship coverage city, more of the same data doesn’t necessarily improve coverage or quality. Not having enough data for certain object classes hurts the overall data extraction efforts of users.

1.2 Goal

Mapillary’s specific request for this partnership with the University of Washington was either:

1. A clear and well researched case study which leveraged their computer vision data to identify a social issue specific to the city in focus.
2. A solution that enabled us to verify and improve the accuracy of our segmentation data.

After discussing internally and in meetings with Mapillary, we decided that this project would need to meet both of these requests. We determined that a multi-step analytical process would best meet the sponsor’s needs. The motivation for this process was the unique data involved, the flexibility of the case study, and the yearning to incorporate all aspects of the UW MGIS program. By performing a case study of the developed tools, we seek to provide the necessary example for Mapillary to see the benefits of this repeatable process.

1.3 Objectives

Any process must meet the two objectives: enhancing the quality control of Mapillary’s image processing while also increasing the efficiency of data capture efforts.

The process that we have developed meets these objectives through four major components. The first component is the ability for Mapillary to easily and automatically convert their own data into a traditional geospatial format. By converting the data to a geospatial format, the data is opened up to a greater level of spatial analytics using desktop GIS software. The second component performs a proximity analysis in order to enhance the quality control process. However, this component does require complete comparison datasets and a substantial amount of manual effort. The third component takes this data and determines not only the current coverage level, but where users should target for future data capture. Because flagship cities already have ample coverage, we believe that targeted capturing of specific and critical datasets increases the effectiveness of Mapillary's users. The last component automatically publishes these results to an ArcGIS Online account for access when creating web maps or web applications.

1.4 Scope

Based on Mapillary's requests and our given timeline, this project has a defined and limited scope. Tools and data will be created and managed using Esri's ArcGIS Desktop software. This will include creating Python tools that integrate with ArcGIS Desktop and using the shapefile and file geodatabase format for geographic data. Web maps will be created using ArcGIS Online.

Because of its designation by Mapillary as a flagship of data coverage, San Francisco was chosen for the case study. While we had an original request for several object classes, due to a lack of data in some, we finalized on using benches, bridges, and bike routes. These datasets provided ample coverage across the city that could be used in our process. Additionally, the Metropolitan Transportation Committee had datasets with near complete coverage of these same classes.

1.5 Social-Ecological System

	Environmental	Economic	Social
National	Assess and project current and future impacts of populations and resource relationships for country	Budgeting and allocation of funds to states.	Development of relationships of data that spans across all systems at varying scales.
San Francisco	Assessing street-level impacts (construction on green spaces, pollution, high rise development)	Updates of infrastructure integrity and well-being of population	Method of communication, promotion of sharing improvements.
Citizen	Represent pollution sources from street-level images.	Promote upkeep and maintenance of underserved areas	Contribution and connecting with other focal scales. Involve users across sociodemographic spectrum.

2. Design & Methods

2.1 Data Acquisition

Before designing the Python tools, work was performed with Mapillary and local agencies to acquire the data necessary to test the tools. Mapillary data is accessible through the Mapillary API, which, at the time of this paper, was version 3. All of the data received from Mapillary and used in this project can be accessed with this API. The public API does put certain restrictions on requests including restricting the number of records returned by each call to the API at a single page of data (Mapillary 2017). In addition, formulating these requests requires significant coding experience with Representational State Transfer (REST). There are code examples on the Mapillary API homepage that can be altered for this purpose.

Given the time limitations placed on this project, we did not develop our own REST calls to interact with the data directly. Instead, Chris Beddow with Mapillary fulfilled all of our data needs. As a Mapillary employee he has administrative access to the data that bypasses the limitations described above.

Data from Mapillary, regardless if it is derived from the API or supplied by an employee with the company comes in GeoJSON format. GeoJSON is an “[Open source] format for encoding a variety of geographic data structures” (GeoJSON 2017). The format is commonly used in open source web applications, such as those running on Leaflet. It is, however, not a format that Esri applications (ArcGIS Desktop, etc.) can read natively. It needs to be either transformed into an Esri format (shapefile, etc.) or converted using the Data Interoperability extension (which requires its own paid license). As will be described in more detail later in this section, our first script was developed in response to this limitation in Esri software. However, QGIS can read data in the GeoJSON format natively.

The Mapillary data only represents half the data needs. There was also a need to acquire spatially verified datasets for the Mapillary object classes that were under review. The Metropolitan Transportation Commission (MTC) provided the bike lane and bridge data. MTC is a “transportation planning, financing and coordinating agency for the nine-county San Francisco Bay Area” (MTC 2017). MTC hosts both datasets, along with many other social, environmental, and economic datasets on their open data website. Both of these object classes were made available as shapefiles.

The bench dataset was found on the DataSF data portal and was provided by the San Francisco Department of Recreation and Parks. This object class was not available as a shapefile. Some of the download options included CSV, XML, JSON and RDF. For this project, the data was downloaded as a CSV and loaded into ArcGIS as X,Y data. From there, the data was exported as a shapefile for further analysis.

Depending on the location of analysis, the provider of these comparison datasets will vary. Typically, city departments and agencies are the best place to start as they often track the same granular data that Mapillary collects. If the city lacks the data then the county should be the next stop followed by the state. The state level typically does not collect data that would be used in this sort of analysis, but results could vary depending on the location. Lastly, one could use OSM to find these datasets. However, the completeness of OSM datasets should be confirmed before analysis, as OSM is a volunteer driven effort.

2.2 Software Requirements

The four Python tools for this project were built using version 2.7.10 of the programming language. 2.7.10 is not the current release of Python, which at the time of this report is 3.6.1. The reason for using an older version of Python is that the ArcGIS plugin, arcpy, is built based on the 2.7.X releases of Python and has not been ported over to the latest version. There are significant changes to the core functionality of Python in 3.0 and above.

Any installation of ArcGIS Desktop comes with an installation of Python 2.7.10 and the arcpy library. The arcpy library will not work without a valid ArcGIS license. In addition, the Spatial Analyst extension is required to run one of the scripts. No additional extensions or functionality are required beyond the core components of ArcGIS Desktop and the Spatial Analyst extension.

Finally, an ArcGIS Online Organization account is required to run one of the scripts. The organizational account is required because data is published and hosted to ArcGIS Online with this script. Hosting data costs credits on ArcGIS Online which are not available with the free ArcGIS Online accounts. A developer account might also work since those accounts come with a handful of credits but this project did not test to confirm this. It is also possible to use ArcGIS Server instead of ArcGIS Online with a few alterations to the script, but the script as is will not work with ArcGIS Server.

2.3 Python Scripts

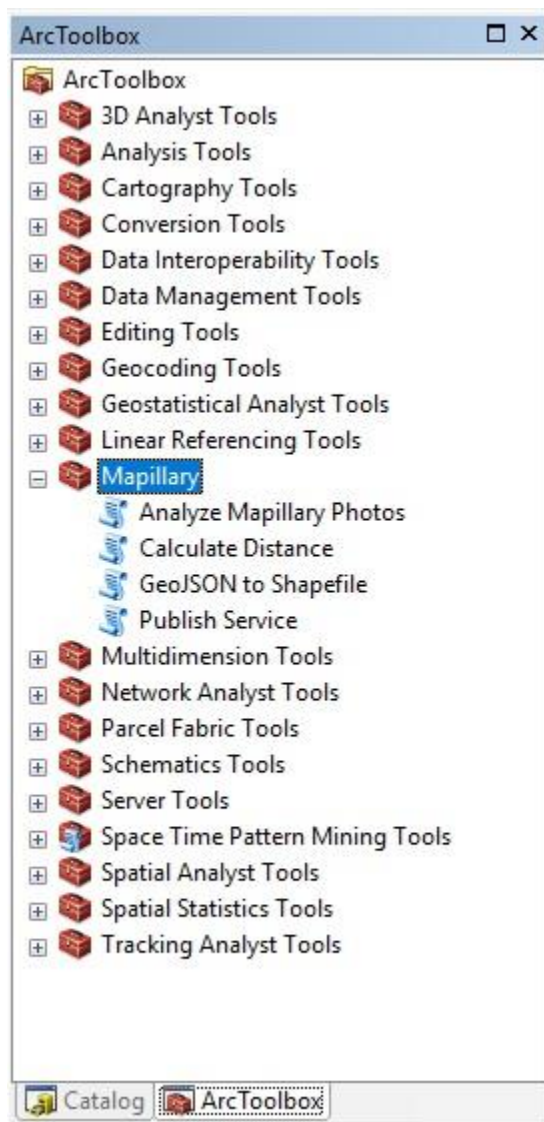


Figure 1 - Toolbox

2.3.1 Mapillary to ESRI (GeoJSON to Shapefile)

Library	Purpose
arcpy	This library allows Python scripts to leverage all of the available tools within ArcGIS Desktop. In this script the library is used to create and then insert data from a GeoJSON field into a new feature class
json	A library that allows Python to access the data stored within JSON and GeoJSON files. Data is made available to Python in the form of a dictionary.

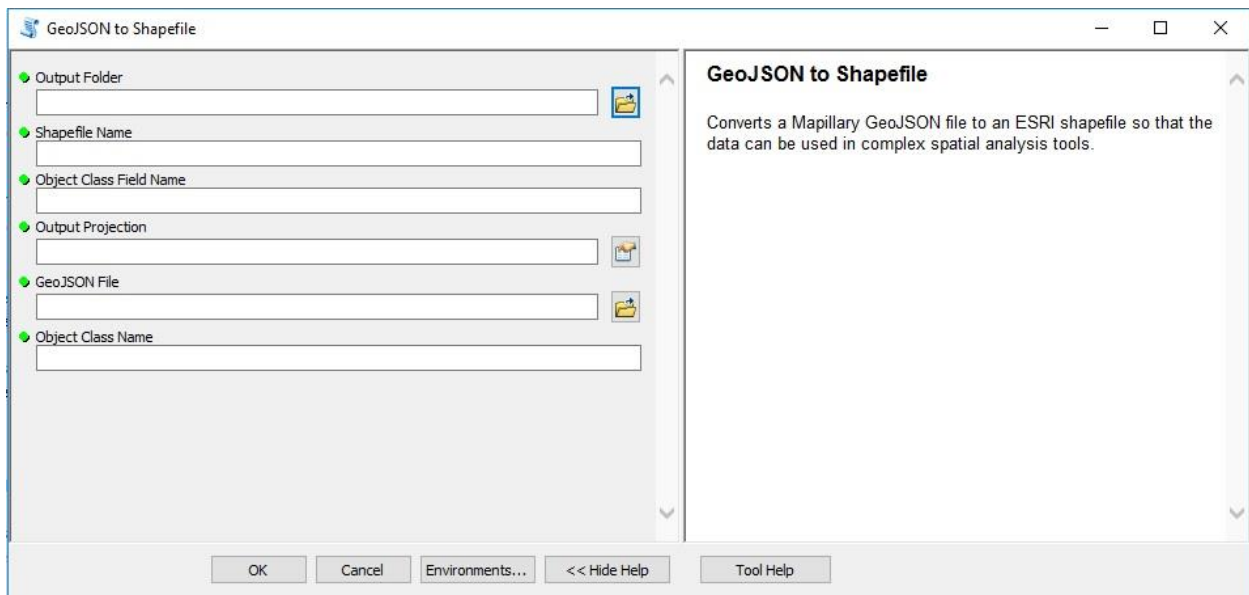


Figure 2 - GeoJSON to Shapefile Dialogue

The Mapillary to Esri script converts GeoJSON data extracted from Mapillary’s API or from a data request that is fulfilled by a Mapillary employee into an Esri shapefile. With a shapefile, it is possible to run spatial analysis in a desktop GIS application like ArcGIS and QGIS. The first step of each script described in this report is to collect input from the user of the script. All four tools are designed to run as custom ArcToolbox scripts and should be run within ArcGIS Desktop or ArcCatalog.

```

{"type": "FeatureCollection",
 "features": [{"type": "Feature",
  "properties": {
    "key": "rqq3SNRM42zESKpQJ__g4Q",
    "nature--sky": 0.20261001571327597,
    "construction--structure--bridge": 0.3211133767458705,
    "construction--structure--building": 0.018203736311988905,
    "object--vehicle--truck": 0.0010320877627236769,
    "object--vehicle--car": 0.058742735629493836,
    "construction--flat--road": 0.3137599090114236,
    "marking--continuous--solid": 0.023902044180431403,
    "marking--discrete--other-marking": 0.0003108978853560984,
    "nature--vegetation": 0.002989027081639506,
    "construction--flat--sidewalk": 0.011858835292514414,
    "construction--flat--curb-cut": 0.00021595369616989046,
    "object--support--pole": 0.008086310362045879,
    "object--traffic-sign--front": 0.0025950537019525655,
    "construction--barrier--wall": 0.0018956927669933066,
    "construction--barrier--curb": 0.006587132476852275,
    "marking--continuous--dashed": 0.000005722176865674555,
    "void--ground": 0.00008689012611284852,
    "object--billboard": 0.00034650172165129334,
    "geometry": {
      "coordinates": [-122.394914, 37.766294],
      "type": "Point"
    }
  }
}

```

Figure 3 - GeoJSON Example

After launching the tool, a dialogue box appears that describes the intent of the script and what outputs the user can expect. The first input from the user is the location of a folder where an output shapefile will be saved. Next the user provides the name of the new shapefile, the name of a new field to store the Mapillary image segmentation data for the desired object class, followed by the desired coordinate system. Finally, the user will input the path to the GeoJSON file and Mapillary object class name for the object they want to extract. An example object class name is 'construction-flat-bike-lane'. The object class names can be found on the Mapillary Vistas website or in the GeoJSON file, the GeoJSON data can be viewed in a text editor like notepad.

```

def CreateFeatureClass(OutLocation, Name, ObjectFieldName):
    """
    Function that takes an output folder, feature class name and output
    field name and creates a new shapefile to load the Mapillary data into
    from the GeoJSON data. The function adds a field for the unique key
    associated with each photo and sets the spatial reference to Web
    Mercator (the Mapillary projection).
    """
    arcpy.CreateFeatureclass_management(OutLocation, Name, "POINT")
    # Need to set if/else statement for .shp extension
    FeatureClassPath = OutLocation + r'\\' + Name + r'.shp'
    arcpy.AddField_management(FeatureClassPath, 'Key', "TEXT")
    arcpy.AddField_management(FeatureClassPath, ObjectFieldName, "DOUBLE")
    SR = arcpy.SpatialReference(4326)
    arcpy.DefineProjection_management(FeatureClassPath, SR)
    return FeatureClassPath

def ExtractData(FilePath, ObjectKey):
    """
    Function that extracts the data from the GeoJSON file and returns a nested
    list that will be used to populate the feature class from the new shapefile.
    The path to the GeoJSON file and the key for the object being imported are
    required as inputs.
    """
    DataTransferList = []
    with open(FilePath) as RawData:
        DataDictionary = json.load(RawData)

    for Row in DataDictionary['features']:
        DataTransferList.append([Row['properties']['key'], Row['properties'][ObjectKey], Row['geometry']['coordinates']])
    return DataTransferList

def InsertData(FeatureClassPath, Fields, DataList):
    """
    Function that inserts the output of a nested list into the new shapefile.
    Requires the path of the shapefile and the nested list to function properly.
    """
    Cursor = arcpy.da.InsertCursor(FeatureClassPath, Fields)
    for Row in DataList:
        Cursor.insertRow(Row)
    del Cursor

```

Figure 4 - GeoJSON to Shapefile Code

After collecting the required user inputs from the tool dialogue box, the script creates a new shapefile with the name and path provided by the user. The coordinate system of the shapefile is initially set to the Mapillary spatial reference, Web Mercator. This is necessary to ensure that the initial data is imported and projected properly. Following later in the script the user defined coordinate system is applied. Two fields are added, one with the user provided name for the segmentation data and a second for the unique photo key that Mapillary adds to each photo.

At this point the script shifts to extracting the data from the GeoJSON file. The script iterates through each feature in the file and since the dictionary keys are either known (key and geometry) or provided by the user for the object class field, the values associated with each key can be extracted. The data from each row of the GeoJSON file is stored in a Python list in the form of [Key, Object Class, Geometry]. That list is in turn stored within another list, making a nested list. The format of the nested list is as follows [[Key, Object Class, Geometry], [Key, Object Class, Geometry], ...]. The reason for the nested list is that the data needs to be formatted this way to load it into the shapefile.

Finally, with the help of an arcpy insert cursor, the data from the GeoJSON file (now in a nested list) is loaded into the previously built shapefile. The cursor iterates through the outer list and inserts the contents of each inner list into their own row of the shapefile. The Mapillary key is added to the

key field, the object class data is added to the user named field, and XY data is added to the Shape@XY field. The final step in the script is to project the shapefile into the user requested coordinate system. It is imperative that the user defined coordinate system be applied after the data is loaded into the shapefile. Unless the user requests Web Mercator, the data will not appear in the correct location if it is applied before loading the data.

At the conclusion of the script the user is provided with a new shapefile in the specified directory, is in their desired coordinate system, and includes all of the object class data from Mapillary.

2.3.2 Priority Ranking and Heat Map Generation (Analyze Mapillary Photos)

Library	Purpose
arcpy	This library allows Python scripts to leverage all of the available tools within ArcGIS Desktop. Spatial analyst tools are utilized in this script to analyze the distribution of photos and known objects and complete a multi-criteria analysis.

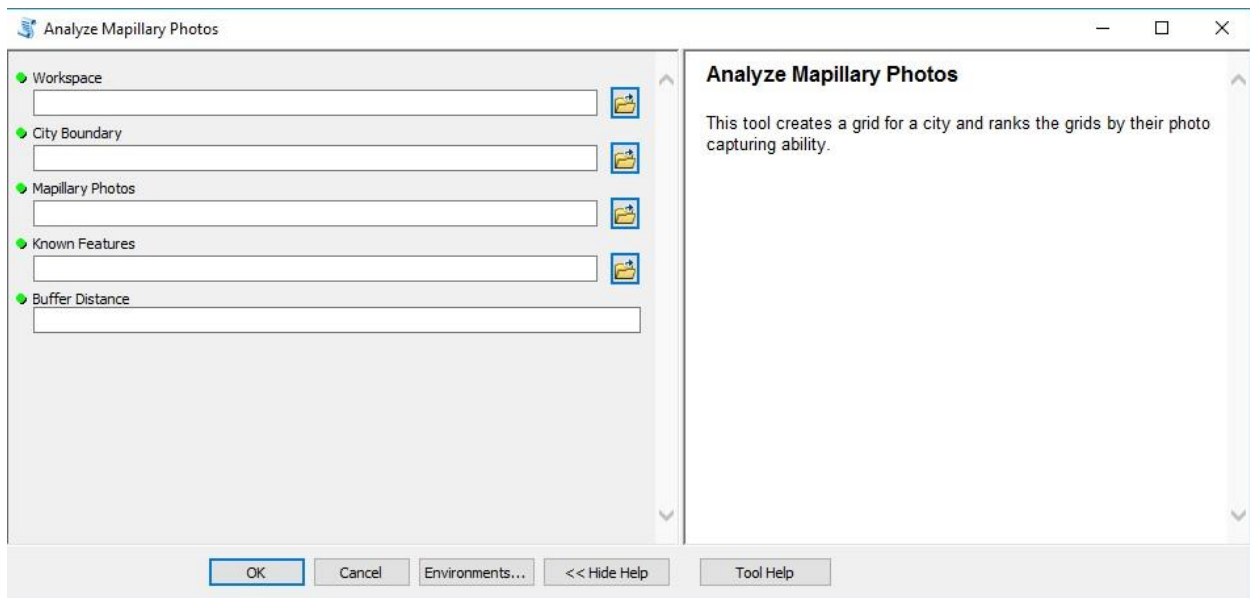


Figure 5 - Analyze Mapillary Photos Dialogue

The purpose of this tool is to analyze Mapillary’s photos in two ways. The first part generates a heat map based on the density of the input point data. This aspect requires the Spatial Analyst extension. The second part analyzes where those Mapillary points are, compares them to known features, and then determines which areas of the chosen city should be targeted to gather more data on the specific input features.

Like the previous tool, after launch, a dialogue box appears that initially describes the intent of the script and what outputs the user can expect. The first input from the user is the location of a

workspace for the data. This location should be a GDB or MDB containing the rest of the necessary data. Next, the user provides file paths to the feature class or shapefile of the city boundary, the input photo data, the known data, and a buffer distance. The buffer distance is the distance in meters that the input data class can be determined through semantic segmentation. For example, if bike lanes can be determined from 150 meters, then the entry would be "150".

```
# Add fields to grid feature class
arcpy.AddField_management(GridRanking, "MCE_Photo_Count", "DOUBLE")
arcpy.AddField_management(GridRanking, "MCE_Shape_Area", "DOUBLE")
arcpy.AddField_management(GridRanking, "Priority", "DOUBLE")

# Calculate max values for area and photos
MaxArea = []
rows = arcpy.SearchCursor(GridRanking)
for row in rows:
    MaxArea.append(row.getValue("SUM_Shape_Area"))
del rows
MaxArea.sort()
MaxArea = MaxArea[-1]

MaxPhotos = []
rows = arcpy.SearchCursor(GridRanking)
for row in rows:
    MaxPhotos.append(row.getValue("Join_Count"))
del rows
MaxPhotos.sort()
MaxPhotos = MaxPhotos[-1]

# MCE calculation/priority ranking
arcpy.CalculateField_management(GridRanking, "MCE_Photo_Count", "1-([Join_Count]/ " + str(MaxPhotos)+")", "VB", "")
arcpy.CalculateField_management(GridRanking, "MCE_Shape_Area", "[SUM_Shape_Area]/ " + str(MaxArea), "VB", "")
arcpy.CalculateField_management(GridRanking, "Priority", "(0.5* [MCE_Photo_Count])+(0.5* [MCE_Shape_Area])", "VB", "")
```

Figure 6 - Analyze Mapillary Photos Code

At this point the script shifts to creating the heat map. The script uses default values to run the Kernel Density tool. This tool creates a raster of the density values of the input data.

Next, the tool creates an equal 0.5 mile grid across the whole city boundary. This grid is the foundation for the eventual priority ranking. The input known data is then buffered based on the previously entered buffer distance. This buffer distance represents the possible area from where a photo could be taken of the object and be classified as a specific feature. The total count of the input photos within these buffers is then appended to the grid squares. This is done to determine how many classified photos are within each grid that are also within the possible photo area. Next, the total area of the buffers within each grid is appended to each grid. This is done to know which grids have the greatest possible area from where a photo could be taken and classified. Then, these two total values are converted to a scale value of 0 to 1. Lastly, a multi-criteria evaluation field is added and calculated as the average value of these two scaled values. To interpret the result, higher values represent areas with a lower number of already classified photos and larger areas from where to take photos.

2.3.3 Image to Object Distance Calculation (Calculate Distance)

Library	Purpose
arcpy	This library allows Python scripts to leverage all of the available tools within ArcGIS Desktop. In the calculate distance script, arcpy is used to prepare the feature classes for analysis, perform the distance calculation and populate the Mapillary feature class with the new data.
os	The operating system library allows Python to interact with the underlying computer operating system. In this case, os is used to dynamically create a working folder to store draft data used in the script.
shutil	Shutil, like os, allows Python to interact with the underlying operate system. Shutil is beneficial because it simplifies the coding required to perform more complex tasks. Deleting directories that contain files requires extra coding, coding that is eliminated with shutil. Shutil is used to delete old versions of the working folder.
sys	The system library, like os and shutil, provides tools to interact with the operating system. For this script, sys is used to find the directory containing the script. The path is used by os to create the working folder.

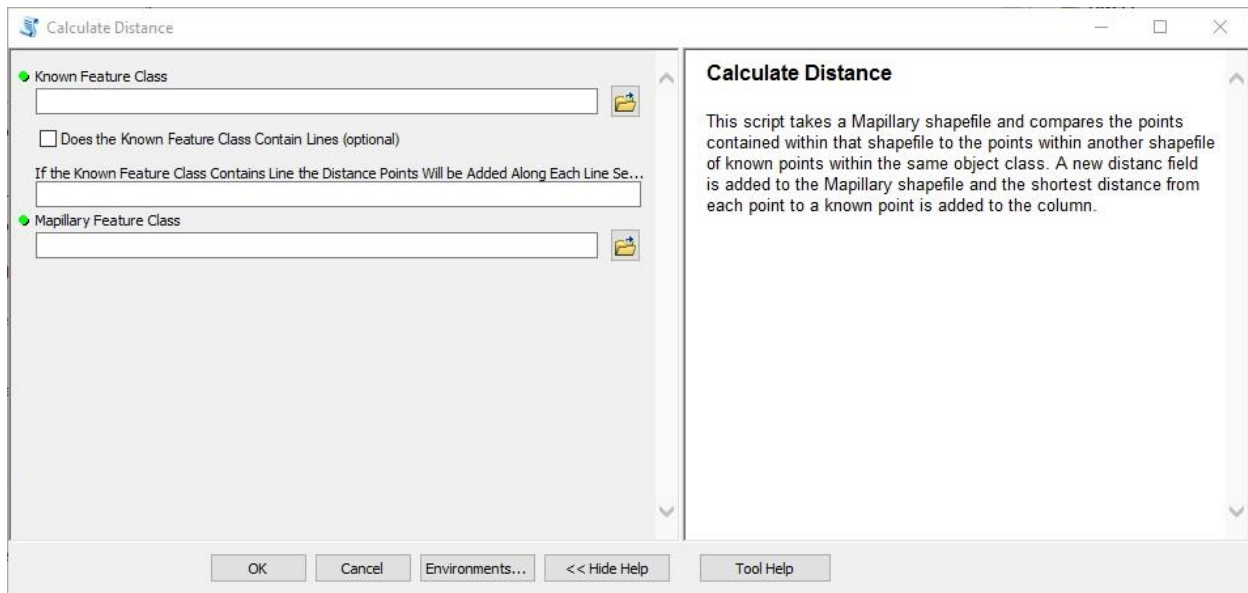


Figure 7 - Calculate Distance Dialogue

The purpose of the Image to Object Distance Calculation script is to take a set of Mapillary photos containing a particular object class and compare those locations to the locations of points representing known objects. The known object data is typically derived from a local government

that manages the particular object. In this analysis, known object locations were identified for bridges, bike lanes, and benches.

As with the other tools, after launching the tool in ArcGIS Desktop or ArcCatalog, a dialogue box appears requesting information from the user. The first input is the file path to the feature class containing known object locations. The tool can accept either point or line data for the known object class. If the input data contains points, then the user can skip optional inputs two and three and proceed to the fourth input. If on the other hand, the known feature class contains lines, then the user needs to check the checkbox for the second input. The third input is the distance at which points will be added along each line segment to allow the script to perform the analysis. The script will create a new shapefile containing points along each line segment at the user specified distance. The units of the provided distance are in the unit of measurement used by the coordinate system of the Mapillary feature class (feet, meters, degrees, etc.). Finally, the user, regardless of the type of input data, will provide the file path to the Mapillary shapefile. This is likely the shapefile created in the Mapillary to Esri script.

```
def CalculateShortestDistance(MapillaryList, KnownList):
    """
    Function that takes two input lists containing X,Y values and calculates
    the distance between each Mapillary point and the nearest known point.
    The shortest distance is added to each row of the Mapillary list.
    """
    for MapillaryPoint in MapillaryList:
        # set an arbitrarily high number for the minimum distance at the start of each row
        MinDistance = 99999999
        for KnownPoint in KnownList:
            # find the distance between the points
            Xdistance = math.pow((MapillaryPoint[0]-KnownPoint[0]),2)
            Ydistance = math.pow((MapillaryPoint[1]-KnownPoint[1]),2)
            Sum = Ydistance + Xdistance
            Distance = math.sqrt(Sum)
            # compare this distance to the previous minimum and if less than update the minimum distance value
            # if the new value is higher than pass
            if Distance < MinDistance:
                MinDistance = Distance
            else:
                pass
        MapillaryPoint.append(MinDistance)

def InsertDistanceValues(MapillaryList):
    """
    Function that takes the updated Mapillary list from the calculate shortest distance
    function and inserts the new distance value into a new distance field. The insert
    cursor compares the key values between the feature class and list to find a match
    then inserts the list distance value into the distance field
    """
    arcpy.AddField_management(FcMapillary, 'Distance', "DOUBLE")
    with arcpy.da.UpdateCursor(FcMapillary, ["Key", "Distance"]) as Cursor:
        for Row in Cursor:
            for Item in MapillaryList:
                if Row[0] == Item[2]:
                    Row[1] = Item[3]
                    Cursor.updateRow(Row)
            else:
                pass
    del Cursor
```

Figure 8 - Calculate Distance Code

Once the inputs are submitted by the user, the tool creates a temporary working directory to store scratch data over the course of the script. A new folder is created in the same directory as the script

file. If the script was run previously, the old working directory is deleted and new empty directory is added in its place. With the directory created, the script prepares the known object feature class for analysis. The projection of the known feature class is set to the same projection as the Mapillary feature class to allow for distance calculations. A new feature class with the projection is added to the working directory and that feature class is used in the analysis, unless it contains lines. If the known object class data consists of lines instead of points, the script creates a new point feature class. Points are added along each line segment at the user specified distance. From here, the point feature class is used in the analysis to follow. Finally, X and Y fields are added to both the Mapillary and known object feature classes.

At this point, the feature classes are ready for analysis. A search cursor extracts the X and Y values from both feature classes and places the coordinates into a list for each row. The attributes for each row are added to one of two lists, one for each feature class, to create a nested list. For the Mapillary feature class the unique Mapillary key is included with each row. The key will be used later on in the script to import the distance calculations back into the Mapillary feature class.

With the XY data now in two lists the script uses a “for” loop to iterate through each row in the Mapillary feature class. Within that for loop is another for loop that iterates through each row in the known feature class for each Mapillary photo. The purpose of the nested for loop is to find the shortest distance between the photo and a known point and this is only possible by checking each known point and comparing its location to the photo’s location. An arbitrarily high value of 99,999,999 is assigned to a minimum distance value. If the calculated distance is less than that value then the new distance value is assigned to the minimum distance variable replacing the old value. The process continues for the rest of the rows in the known feature class. At the end of the inner for loop, the minimum value is appended to the end of the nested Mapillary list. The calculations repeat for each row in the Mapillary feature class.

Once all of the minimum distance values are identified and added to each of the nested lists, the script calls an insert cursor. The insert cursor adds the newly identified distance data to the distance column for each row. The distance column is created just before calling the insert cursor and is the type ‘Double’. For each row in the feature class, the cursor iterates through each row of the list until it finds a match in the key values. When a match is identified, the distance value from the list is inserted. Again, this continues for each row in the feature class.

At this point, the script concludes and the user is provided with an updated feature class of Mapillary photo locations that contains the distance to the nearest known object. With this data and the semantic segmentation data, it is possible to create scatter plots that will identify outliers in the data. These outliers could be false positives identified by the Mapillary algorithm.

2.3.4 Data to ArcGIS Online (Publish Service)

Library	Purpose
arcpy	This library allows Python scripts to leverage all of the available tools within ArcGIS Desktop. In the fourth script, arcpy leverages the tools within ArcGIS Desktop that allow information to be published to either ArcGIS Server or ArcGIS Online.
os	The operating system library allows Python to interact with the underlying computer operating system. In this case, os is used to dynamically create a working folder to store draft data used in the script.
shutil	Shutil, like os, allows Python to interact with the underlying operate system. Shutil is beneficial because it simplifies the coding required to perform more complex tasks. Deleting directories that contain files requires extra coding, coding that is eliminated with shutil. Shutil is used to delete old versions of the working folder.
sys	The system library, like os and shutil, provides tools to interact with the operating system. For this script, sys is used to find the directory containing the script. The path is used by os to create the working folder.
xml	XML is a library that allows Python to interact with XML files. The service data that is published to ArcGIS Online is transmitted as XML data. This library allows the script to dynamically update the contents of the XML file to change the service settings.

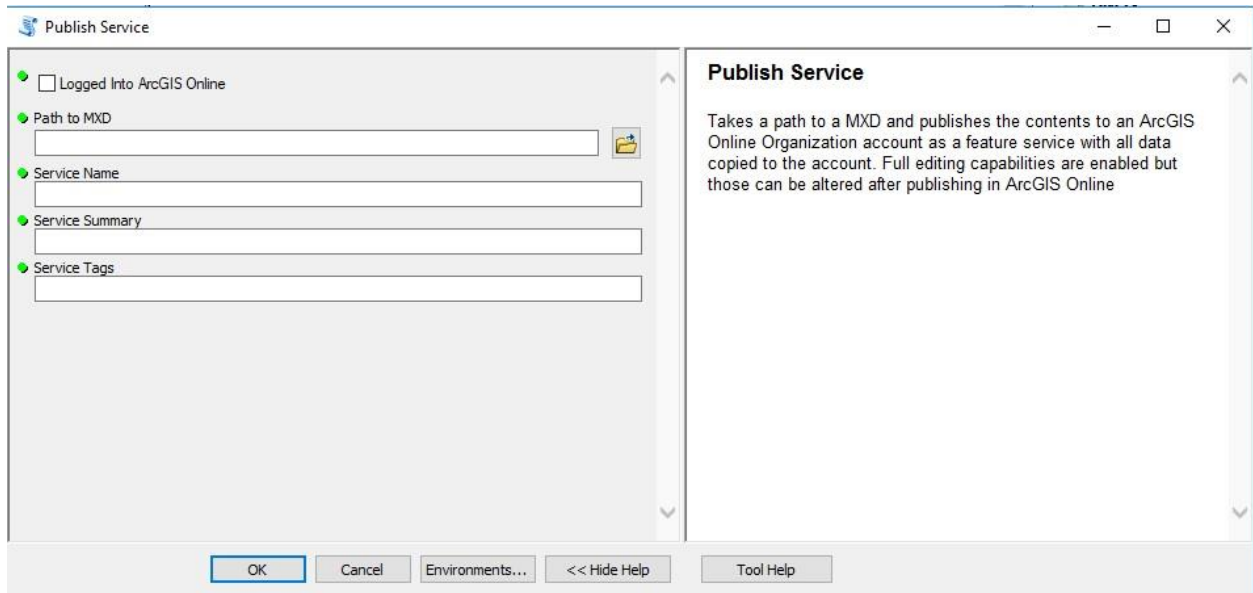


Figure 9 - Publish Service Dialogue

The Data to ArcGIS Online script is designed to upload the contents of an ArcGIS Map Document (MXD) to an ArcGIS Online organization account. With the data in ArcGIS Online, users are able to create web maps from the results of the analysis. When the data is transferred to ArcGIS Online, the symbols set in ArcMap are maintained. It is therefore recommended that symbolizing be performed prior to running this script and all layers be added to the map. Each layer in the map will be published as its own layer in the service.

To run this script the user is required to run the tool through ArcMap or ArcCatalog as a custom script tool. This tool comes in the same custom toolbox as the other three tools. Prior to executing the tool, the user needs to sign into their ArcGIS Online account within ArcMap. This can be performed by opening the File menu and selecting Sign in. Note that this needs to be an account with ArcGIS Online credits, a free account will not work. With the tool dialogue box open check the box for the first input that you are signed into ArcGIS Online. Next use file explorer to provide the path to the MXD file that has data you want to publish to your account. Lastly, provide a name for the service in the third box and a summary and tags (separated by commas) in the fourth and fifth boxes respectively. Note that if the name of the service is the same as an existing service in your account, the new service will overwrite the old one.

```

tagsType = doc.getElementsByTagName('Type')
for tagType in tagsType:
    if tagType.parentNode.tagName == 'SVCManifest':
        if tagType.hasChildNodes():
            tagType.firstChild.data = "esriServiceDefinitionType_Replacement"

tagsState = doc.getElementsByTagName('State')
for tagState in tagsState:
    if tagState.parentNode.tagName == 'SVCManifest':
        if tagState.hasChildNodes():
            tagState.firstChild.data = "esriSDState_Published"

# Change service type from map service to feature service
typeNameNames = doc.getElementsByTagName('TypeName')
for typeName in typeNameNames:
    if typeName.firstChild.data == "MapServer":
        typeName.firstChild.data = "FeatureServer"

# Turn off caching
configProps = doc.getElementsByTagName('ConfigurationProperties')[0]
propArray = configProps.firstChild
propSets = propArray.childNodes
for propSet in propSets:
    keyValues = propSet.childNodes
    for keyValue in keyValues:
        if keyValue.tagName == 'Key':
            if keyValue.firstChild.data == "isCached":
                keyValue.nextSibling.firstChild.data = "false"

# Turn on feature access capabilities
configProps = doc.getElementsByTagName('Info')[0]
propArray = configProps.firstChild
propSets = propArray.childNodes
for propSet in propSets:
    keyValues = propSet.childNodes
    for keyValue in keyValues:
        if keyValue.tagName == 'Key':
            if keyValue.firstChild.data == "WebCapabilities":
                keyValue.nextSibling.firstChild.data = "Query,Create,Update,Delete,Uploads,Editing"

# Write the new draft to disk
f = open(newSDdraft, 'w')
doc.writexml( f )
f.close()

```

Figure 10 - Publish Service Code

With the user provided parameters stored in their own variables, the script proceeds to create a temporary working directory to store intermittent files. The temporary directory is created in the same way as the temporary directory in the Image to Object Distance Calculation script. The only difference is that this directory has a different name. The first step in actually publishing the service is to create a draft service definition that includes the MXD, service name, summary, and tags. In addition, the location where the data will be published is identified. It is in this part of the script where the destination could be altered to an instance of ArcGIS Server.

With a draft service definition saved to the temporary directory, the XML library is used to open the contents of the file in Python. The script iterates through the contents of the service to set the desired parameters. In this case the feature service functionalities are enabled (editing, creating and querying) and any caching settings removed. Once the edits have been made to the service definition the document is saved. At this point, the script analyzes the service draft for any errors. If

an error is found, the script ends and the cause of the error is printed to the console in ArcMap. If no issues are found then the contents of the service are published to ArcGIS Online.

At the successful conclusion of the script the user is provided with a hosted feature service in their ArcGIS Online account. Note that all of the data associated with the layers in the MXD are also copied to the account. One should consider the size of these datasets and the cost in terms of credits to their account before publishing any data with this script. The new service can be used to create web applications with the Web AppBuilder or Story Maps.

2.4 ArcGIS Online

With the data now available within ArcGIS Online it is possible to create web applications with the results of the analysis. For example, the results of the grid analysis could be overlaid on top of the study area. This is done by simply creating a new web map and adding the published service. From there it is possible to change the symbols if needed, set popup windows, and filter the data to create the ideal web map. The map can then be shared with your organization or can be made public and available to all. ArcGIS Online allows groups to quickly share GIS data with non-technical audiences.

Since the focus of this project was on developing scripts to help Mapillary analyze their segmentation algorithms, web maps were a byproduct of this effort. The exact process of creating these maps will not be covered in further detail. The best way to learn how to use ArcGIS Online is to review the numerous Esri help pages available online.

3. Results

3.1 GeoJSON to Shapefile

The GeoJSON to Shapefile script outputs a Shapefile based on the input GeoJSON features. Figure 11 shows the export of the bike lane GeoJSON file.

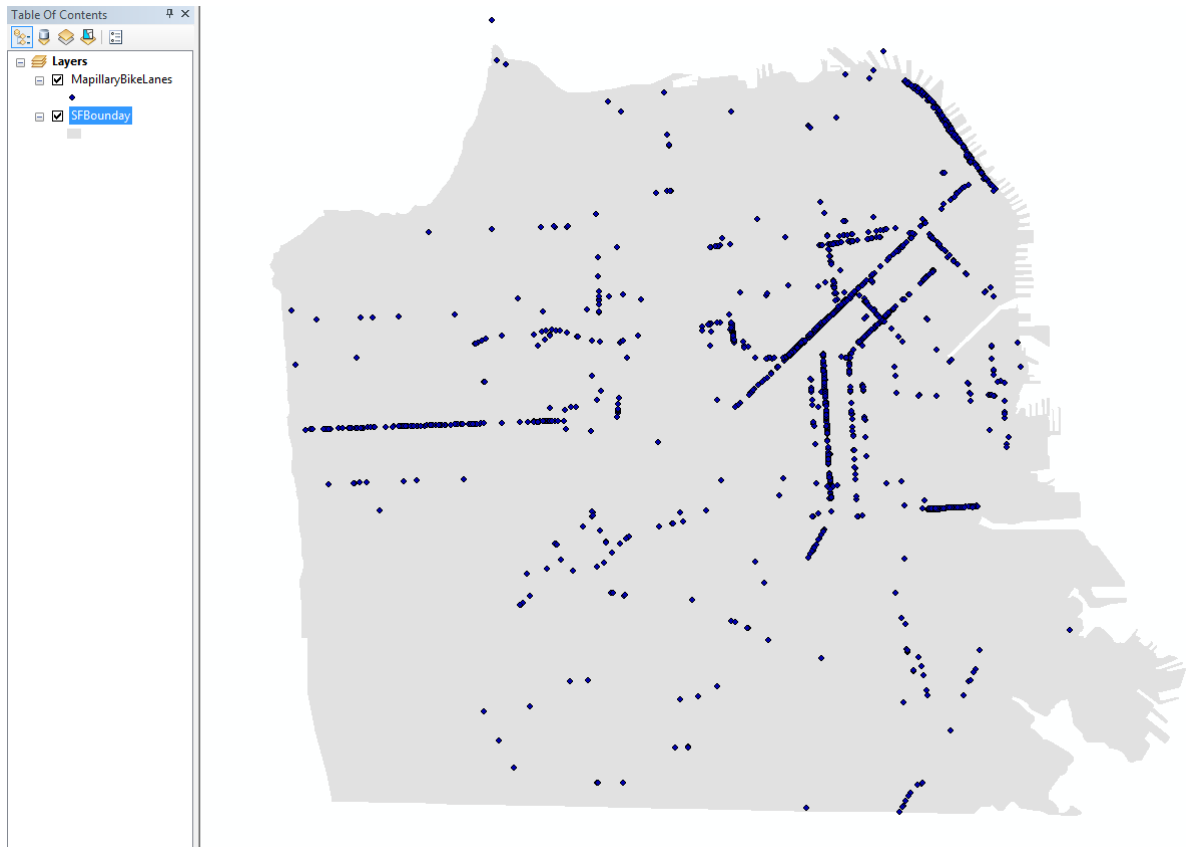


Figure 11 - Map of Converted Shapefile

3.2 Results of Analyze Mapillary Photos

The Analyze Mapillary Photos tool results in two datasets: a heat map of the input Mapillary photos and a ranked grid of the input city.

Seen in Figure 12, the highest density of Mapillary's bike lane photographs are in the northeast portion of the city along the waterfront and along major roadways going into the center of the city.

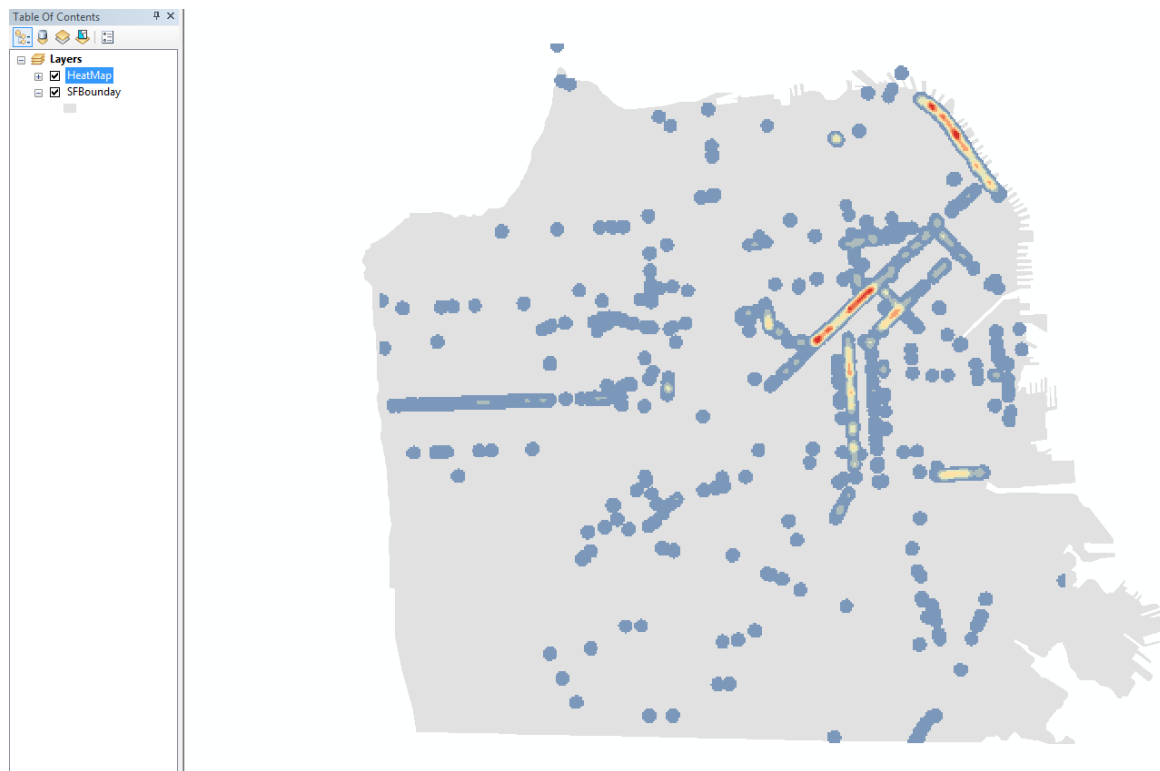


Figure 12 - Heat Map

Figure 13 displays the ranked grid as a graduated color scheme where low priorities for targeting bike lanes are blue and high values are red. The ranked grid is very dispersed regarding areas of priority. The northeast part of the city has a small section of grids with a low priority.

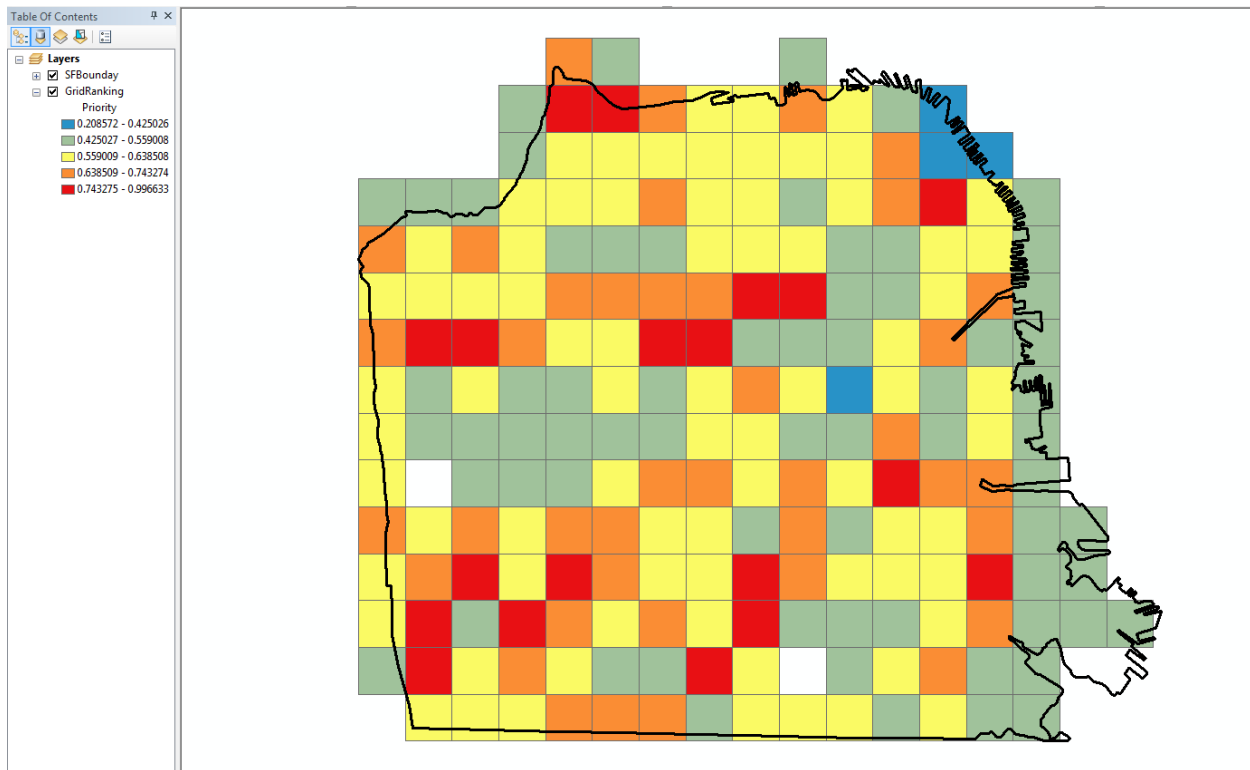


Figure 13 - Ranked Grid

3.3 Results of Shapefile to ArcGIS Online

The Shapefile to ArcGIS Online tool results in the input data published as a feature service. This feature service is the ranked grid seen in Figure 13.

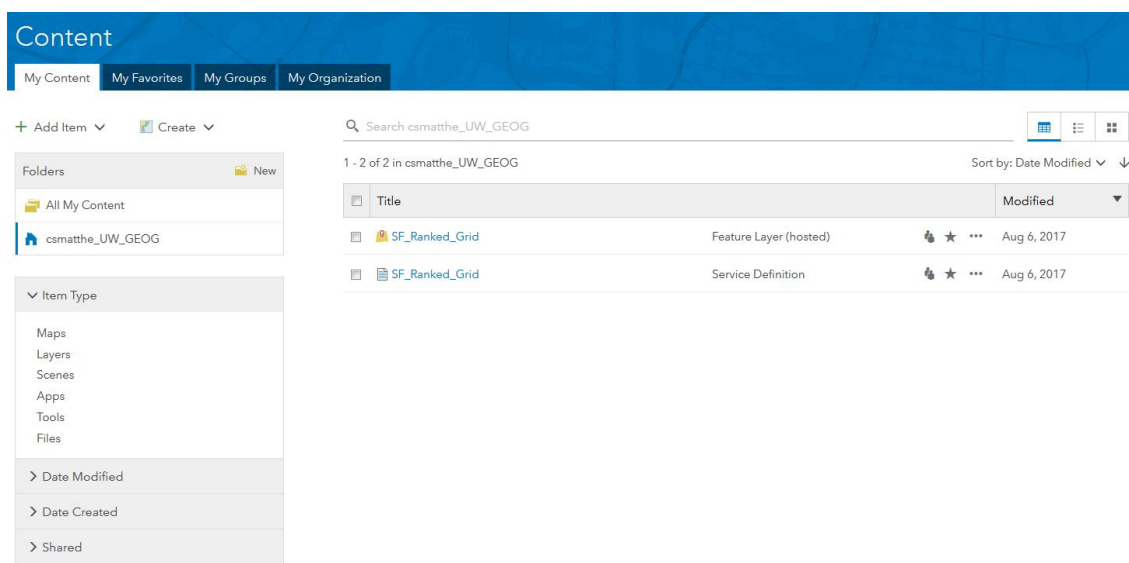


Figure 14 - ArcGIS Online

3.4 Results of False Positive Analysis

The false positive tool adds the distance column to each record in the shapefile/feature class.

OBJECTID *	Shape *	Key	BikeData	POINT_X	POINT_Y	Distance
1	Point	wGQA4RbnbrVkQprHjUQZIQ	0.044336	6006710.486525	2110107.76106	186.619763
2	Point	9kCicrT5x323WxXiYTxEw	0.049437	6006781.2839	2110164.29666	277.183516
3	Point	_PhsfPDCGZnrgrHmRtDhQ	0.032129	6006708.573396	2110064.622646	162.840449
4	Point	Xbk258lmXOf_GgwHkUHgow	0.081559	6007325.992313	2110725.72499	361.598779
5	Point	U73kHl6LB4YAGIYAdA0PyQ	0.078314	6006903.676975	2104981.38236	648.227183
6	Point	YqEywq-6_nLc2rfgyakbw	0.070056	6000234.989744	2096448.037781	320.854489
7	Point	6zCVyyWD-wrSqTlW6O9kjQ	0.044645	5997104.772426	2099575.170125	81.806859

Figure 15 - Output Attribute Table

Utilizing the data, a scatter plot was made in Tableau which shows the distance from a known object against the pixel percentage. Outliers seem apparent at 800 or more feet, however each point should be verified manually.

Scatter Plot of Distance Between Photos and Known Objects Against Pixel Percentage

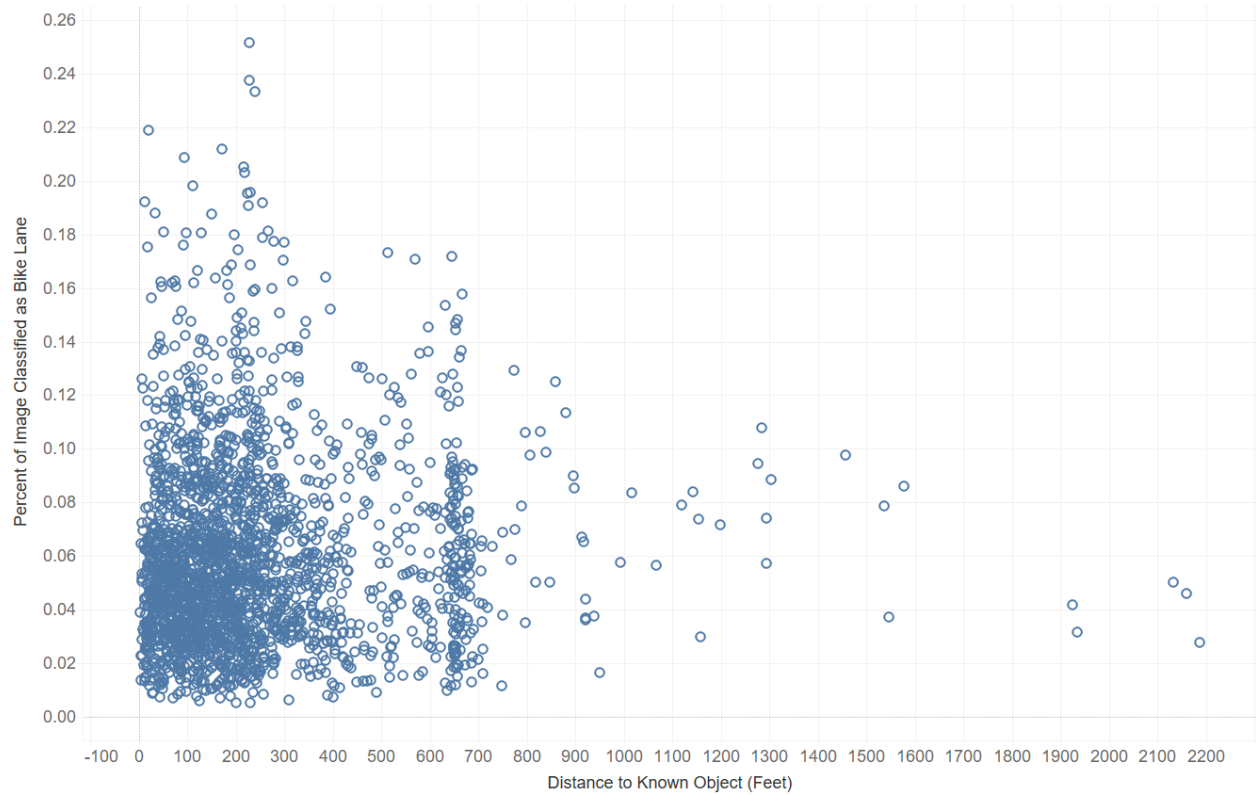


Figure 16 - Scatter Plot of Distance Results

3.5 Results of ArcGIS Online Web Map

ArcGIS Online allows for simple web map creation that can be disseminated to a wider audience than a static map. These maps also provide dynamic interaction so map users can manipulate, pan, and zoom with the data.

The below figure is a web map of the feature service seen in Figure 14.

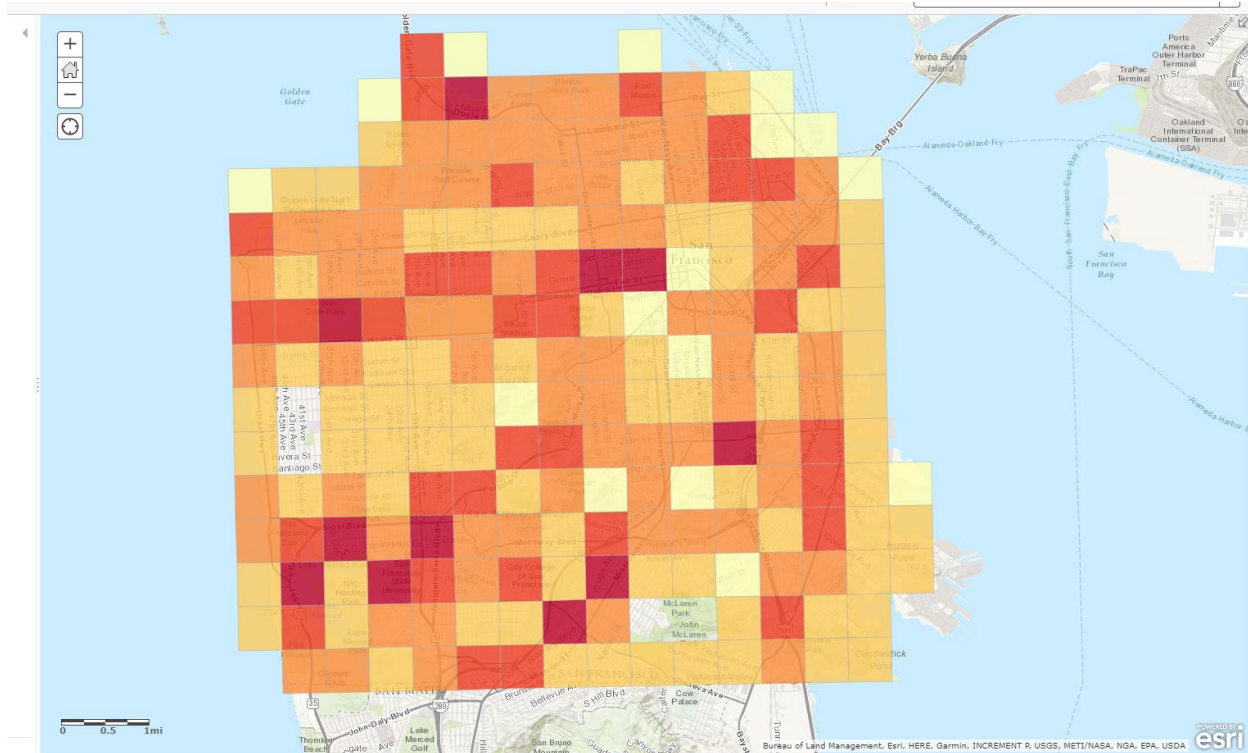


Figure 17 - Ranked Grid Web Map

4. Discussion

4.1 Successes

During the time in our capstone course, we were able to build script tools that were focused on enhancing existing applications used at Mapillary. Despite focus shifts that set us back valuable time, we were able to create a GeoJSON to Shapefile tool that is efficient, portable, and significantly faster than what we've found on the web. Given more time, we would be able to transfer the functionality into free and open source software such as QGIS. Automation of processing GeoJSON data into a shapefile is not available in either software package as a standard feature. The Data Interoperability extension allows you to convert GeoJSON features in ArcGIS, but this extension requires an additional cost expenditure. With QGIS it is possible to display the data and manually

export the data out as a shapefile. However, this process is rather tedious and slow. An efficient GeoJSON data converter is not something that we could find on the web so this tool appears to be a new addition to spatial industry.

From our tools, we were also able to create results that represented false positives for Mapillary's vision team. They are able to use this information to improve and test their semantic segmentation algorithm. Without tools like these, the vision team would need to look at all of their images to check for errors. Even with random sampling of a small number of photos this would be a time consuming and inefficient task. With a tool that identifies potential false positives, Mapillary is able to focus their efforts on potential problem areas. These tools are pilot versions and when integrated into existing APIs, it will have the ability to be replicated with other assets in different locations.

Multi-criteria evaluation allows for the evaluation of multiple and sometimes conflicting criteria. Previous research has proven the effectiveness of integrating multi-criteria evaluation with GIS (Store and Kangas, 2001). Multi-criteria evaluation was selected as one of our tool's outputs because when targeting locations for photos, we wanted to know not only where there weren't photos already, but where we could take photos from. While we know that additional variables would be more useful in a final evaluation, we argue that with the available data, these two variables provide the best evaluation. By weighting these values equally, the resulting layer is determined as the most optimal location for users to target for capturing specific features.

The heat map generation and multi-criteria evaluation provide useful insights currently not being tracked. They are able to use the heat maps to not only know where their data is, but also the magnitude of this data. This dataset can also be easily implemented in a web map and shared widely. For future use, the multi-criteria evaluation results can be used for competitions and specific targeting.

4.2 Lessons Learned

Given our successes we have room for improvement in many areas of our project. First, we wanted to start with what we assumed was an advantage by selecting the city of San Francisco. According to Mapillary, the city had a great percentage of data coverage for their object classes. We soon discovered great coverage of Mapillary data does not translate into others having the same robust publically available data. The lack of robust comparison data would challenge our design, testing, and implementation. The datasets we used to compare our results were often unavailable online or were incomplete.

In addition, the current process of comparing our results proved tedious and manual. This does leave room for improvement and for future development of a more efficient method of comparing datasets. Secondly, we spent some time learning other methods with which we were unfamiliar, to produce results. We quickly found learning a new programming language and the complexities along with it would require more time than the course allowed, hence the utilization of Esri-based

software. We would have preferred to develop tools with QGIS or to create web applications with Leaflet, but it can take months to learn either of those. This was time we simply did not have if we wanted to meet our deadlines.

When we entered into the project we knew the potential challenges that may arise. We did not anticipate changing our project focus several times to meet the needs of both Mapillary and the curriculum. Despite reevaluating and redeveloping our scope, we were able to meet the program and capstone partner deadlines.

4.3 Recommend Course of Action

In the immediate future, it is our recommendation that Mapillary identify other cities and areas to repeat the analysis outlined in this paper. Ideally this would be in locations with easily accessible and complete comparison data. In terms of identifying false positives or hotspots to promote image collection, it is more important to have robust control data than it is to have significant photo coverage. At the start of the project we looked for areas that had lots of photos. If we had to do it again, we would have instead identified locations with object class data that we could use in our analysis. From there we would pull the Mapillary data for that location instead of the other way around.

After performing this analysis in several other locations, it is recommended to combine all of the potential false positives into one group and take a random sampling of the points for human review. If the population of photos is small enough then they can be reviewed manually, but likely there will be too many photos to be practical. The results of the human analysis should help inform the segmentation algorithms. If there are false positives then the algorithm can be improved to address the issues. On the other hand if there are not false positives then Mapillary has a selling point to use with the local agencies that manage the area. Mapillary can sell the fact that their platform is a more effective means of inventorying their assets.

Another change that can be implemented is the use of the hot spot statistical analysis tool to determine where to run Complete the Map challenges. These are challenges that Mapillary puts on in different cities to promote their users to add photos to their platform. The hot spot analysis tool can be used to make data driven decisions on where to statistically focus their efforts. If there are certain assets that Mapillary wants to add they can target areas where photos are most likely to capture them.

4.4 Future Implementations

The first step to improve upon these scripts should be the porting over of the functionality to open source Python libraries. Given the time limitations that have been discussed in detail, it was not possible for the team to develop scripts using libraries they were unfamiliar with before the course.

Given more time, the functionality would have been developed using the QGIS equivalent libraries. The group only performed a cursory review of the functionality offered by the QGIS add-in but it should be possible to replicate the first three scripts. The ArcGIS Online script is the only script that cannot be ported over to open source. In an open source environment ArcGIS Online would not be necessary. Instead, a new script should be developed to convert the results back to GeoJSON. The GeoJSON data could then be used in open source web mapping applications like Leaflet. In this way the data would come full circle.

A couple quick improvements to the scripts that would improve analysis are automated scatter plot generation and the selection of a random set of points for humans to review in more detail. A scatter plot could be generated to compare distance from a known object class to the segmentation data (e.g. the percent of an image classified as the object). Our current assumptions based on the data show that photos further away from known features have a lower pixel count. The further the photo is from an object the smaller it will appear. The smaller an object in an image, the fewer pixels that will be classified than if the photo was right in front of the same object. By implementing a scatter plot generation, users would be able to more easily visualize how distance and the pixel levels are correlated. However, this is very dependent on the known data being accurate. This should be relatively straightforward with the matplotlib library. The team planned to implement this functionality but ran out of time before the conclusion of the course.

For random sampling, implementing a random photo selection would be more statistically accurate for point comparison. This would ensure that the spatial location would not be the main contributor to any inaccuracies in the segmentation process. It would also limit the amount of work required to check image segmentation. A new field could be added to the feature class with a value from a random number generator. For example, the twenty points nearest zero could be selected for more thorough analysis.

Currently, the Analyze Mapillary Photos tool exports the heat map as a raster. For integration with custom web maps (Leaflet), this data needs to be converted to a vector data format (points, lines or polygon) instead of raster (grid). While we did attempt this using Python, we ran into roadblocks with the Raster Calculator tool. This function can be performed manually, but, automation would ease and speed up the process. This does not alter the analysis in any way but improves the functionality of the data in terms of communicating the results.

A couple improvements in the more distant future are the ability for the scripts to take location viewshed into consideration and an improved MCE analysis. The multi-criteria evaluation and distance tools assume that features can be photographed from anywhere within the city or area. However, we know that this is not accurate for most features. This is because in real life, other objects such as buildings can block the view of users to the object. Implementing a viewshed analysis using LIDAR would ensure that areas out of view would not be included in the multi-criteria evaluation and that distances would be calculated to the nearest object in view, if there is one. This can be done in GIS manually and can most likely be implemented in Python but it will

require a significant amount of coding. However, the results would be far more robust than those in the current scripts.

The multi-criteria evaluation can be improved in several ways, one of which is the incorporation of user locations. Without knowing where active users live or primarily spend their time, the multi-criteria evaluation does not take into account how close users are to where the grids are located. By adding in the locations of users, the multi-criteria evaluation would then factor in the “ease” of getting to specific grids.

There are likely many more changes that can be made to these scripts and those changes will be defined by Mapillary in order to align the scripts with their business goals. As those goals change, the functionality of the scripts will also need to adapt. Hopefully with the foundation provided by these pilot scripts, Mapillary can easily adapt them into their existing and future business processes.

5. Business Case and Implementation Plan

The final project should include a business case, including a financial and strategic analysis and workflow plan, for implementation of your recommended course of action. The implementation plan including "who" will be doing what, when, with what expected outcome and at what estimated cost versus what intended benefits for the organization.

The team at Mapillary has formed a solid foundation on which our additions are built upon. As the company continues to grow, so will the number of users and the data collected by them every day. The ability for users to contribute data to Mapillary’s database at virtually zero cost provides great benefit to both the user and the host. As the database continues to grow exponentially, there must be ways to filter, verify, and manage each photo passing through. With the addition of our analysis kit we hope Mapillary will be able to allocate manual efforts to other tasks while allowing the scripts to perform routine quality assurance checks and data maintenance.

Automating analysis of photos as they are received will allow Mapillary to determine data cold spots. These areas will give Mapillary an idea of where users are inactive and strategize on ways to develop promotional efforts to those areas. This process is aided by the use of automated web map publishing to view and communicate the need to fill in gaps. Furthermore, these analysis results provide a reference to compare other factors such as social, ecological, and environmental aspects. The addition of various scripts has the ability to connect the environment to the people and the local governments that help operate them. Lastly, Mapillary has the tools to build upon and integrate further, providing a public viewing interface to draw in participation from both cities and their citizens.

Local governments are given a unique opportunity to collaborate with its citizens. Mapillary allows beneficial relationships to blossom which benefit livability and access across the social spectrum.

The benefit of crowdsourcing data collection is that anyone with a camera can contribute. Those that want to be more involved with their communities and cities are given a chance to share their photos and help develop a database of useful information. This information can then be used to develop functional and effective improvements to areas that may once have been overlooked. These improvements may include infrastructure, assessing green spaces, determining availability to integrate better road sharing opportunities, and more. Mapillary can use our kit as a springboard to sharing, collaborating, and connecting people to bring change to areas that are needed.

6. Literature Cited

GeoJSON. "GeoJSON." Accessed August 5, 2017. <https://geojson.org/>

Mapillary. "About Mapillary." Accessed August 4, 2017. <https://www.mapillary.com/about>

Mapillary. "Mapillary API Documentation." Mapillary API. Accessed August 13, 2017.
<https://www.mapillary.com/developer/api-documentation/>.

Metropolitan Transportation Commission (MTC). "About MTC." Accessed August 5, 2017.
<http://mtc.ca.gov/about>

Store, Ron, and Jyrki Kangas. 2001. "Integrating spatial multi-criteria evaluation and expert knowledge for GIS-based habitat suitability modelling". *Landscape and Urban Planning*, Volume 55, Issue 2, Pages 79-93, ISSN 0169-2046, [http://dx.doi.org/10.1016/S0169-2046\(01\)00120-7](http://dx.doi.org/10.1016/S0169-2046(01)00120-7).

7. Appendix

Script Code

https://github.com/james1992/UW_Capstone_Mapillary/