satellite_imagery_binder_notebook

November 10, 2020

1 Exploring long-term urban changes through satellite imagery

1.0.1 Meixu Chen [1], Dominik Fahrner [2,3], Daniel Arribas-Bel[1], Francisco Rowe [1]

[1] Geographic Data Science Lab, Department of Geography and Planning, University of Liverpool, Roxby Building, 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom

[2] Department of Geography annd Planning, School of Environmental Sciences, University of Liverpool, Liverpool, L69 7ZT, United Kingdom

[3] Institute for Risk and Uncertainty, University of Liverpool, Chadwick Building, Peach Street, Liverpool, L7 7BD, United Kingdom

1.1 Abstract

Satellite imagery is often used to study and monitor Earth surface changes. The open availability and extensive temporal coverage of Landsat imagery has enabled changes in temperature, wind, vegetation and ice melting speed for a period of up to 46 years. Yet, the use of satellite imagery to study cities has remained underutilised, partly due to the lack of a methodological approach to capture features and changes in the urban environment. This notebook offers a framework based on Python tools to demonstrate how to batch-download high-resolution satellite imagery; and enable the extraction, analysis and visualisation of features of the built environment to capture long-term urban changes.

Keywords: satellite imagery, image segmentation, urbanisation, cities, urban change, computational notebooks

1.2 Table of Contents

1.2.1 Introduction

Sustainable urban habitats are a key component of many global challenges. Efficient management and planning of cities are pivotal to all 17 UN Sustainable Development Goals (SDGs). Over 90% of the projected urban population growth by 2050 will occur in less developed countries (UN, 2019). Concentrated in cities, this growth offers an opportunity for social progress and economic development but it also imposes major challenges for urban planning. Prior work on urbanisation has identified the benefits of agglomeration and improvements in health and education, which tend to outweigh the costs of congestion, pollution and poverty (Glaeser and Henderson, 2017). Yet research has remained largely focused on Western cities, developing a good understanding of urban areas in high-income, developed countries (Glaeser and Henderson, 2017). Much less is known about urban habitats in less developed countries. This gap is partly due to the lack of comprehensive data sources capturing the dynamics of urban structures in less developed countries.

Cities in Asia provide a unique setting to explore the challenges triggered by rapid urbanisation. The share of urban population in Asia is currently at turning point transitioning to exceed the share of rural population. Currently Asia is home to over 53% of the urban population globally and the share of urban population is projected to increase to 66% by 2050 (UN, 2019). Developing monitoring tools to understand the past and current urbanisation process is key to guide appropriate urban planning and policy strategies.

Recent technological developments can help overcome the paucity in spatially-detailed urban data in less developed countries. The combination of geospatial technology, cheap computing and new machine learning algorithms has ushered in an age of new forms of data, producing brand new data sets and repurposing existing sources. Satellite imagery represents a key source of information. Photographs from the sky have existed for decades, but their use in the context of socioeconomic urban research has been limited. Image data has been hard to process and understand for social scientists. Yet recent developments in machine learning and artificial intelligence have made images computable and turned these data into brand new information to be explored by quantitative urban researchers. Satellite data can be openly accessible, provide high temporal and a global coverage at a reasonably high spatial resolution.

This notebook illustrates an analytical framework based on Python tools which enables batch download, image feature extraction, analysis and visualisation of high-resolution satellite imagery to capture long-term urban changes. The source of satellite data and administritive boundaries data are from NASA's Landsat satellite programme and ArcGIS Online. The Python libraries used in this notebook are the following:

- Landsat images in Google Cloud Storage: The Google Cloud Storage is accessed using an API to download Landsat imagery
- Matplotolib:A Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Numpy: Adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions
- Pandas: Provides high-performance, easy-to-use data structures and data analysis tools
- GeoPandas: Python library that simplifies working with geospatil data
- Folium: Python library that enables plotting interactive maps using leaflet

- Glob: Unix style pathname pattern expansion
- GDAL: Library for geospatial data processing
- Landsat578: Simple Landsat imagery download tool
- L8qa: Landsat processing toolbox
- Rasterio: Library for raster data processing
- Scikit-image: Collection of algorithms for image processing
- Wget: Pure python download utility
- OpenCV: Library for image processing
- scikit-learn: Machine learning in Python. Simple and efficient tools for data mining and data analysis.

We can import them al as follows:

[1]:
```
%matplotlib inline

#load external libraries
import matplotlib.pyplot as plt
from matplotlib import colors
import pandas as pd
import numpy as np
import geopandas as gpd
import folium
import os, shutil
import glob
import gdal
import wget
from landsat import google_download
from google_download import GoogleDownload
from l8qa.qa import write_cloud_mask
import rasterio
import rasterio as rio
from rasterio import merge
from rasterio.plot import show
from rasterio.mask import mask
from skimage import io,exposure, transform,data
from skimage.color import rgb2hsv, rgb2gray
from skimage.feature import local_binary_pattern
from sklearn.cluster import KMeans
import matplotlib.cm as cm
from sklearn import preprocessing
from rasterio.enums import Resampling
import seaborn as sns
import itertools

wdir= os.getcwd()
```

The remainder of this paper is structured as follows. The next section introduces the Landsat satellite imagery, study area Shanghai, and process on how to batch download and pre-process satellite data. Section 3 proposes our methods to extract different features including colour, texture, vegetation and built-up from imagery. Section 4 performs clustering method on the extracted features, and section 5 interprets the results and gain insights from them. Finally, section 5 concludes by providing a summary of our work and avenues for further reserach using our proposed framework.

1.2.2 Data and Study Area

Landsat Imagery We draw data from the NASA's Landsat satellite programme. It is the longest standing programme for Earth observation (EO) imagery (NASA, 2019). Landsat satellites have been orbiting the Earth for 46 years providing increasingly higher resolution imagery. Landsat Missions 1-3 offer coarse imagery of 80m covering the period from 1972 to 1983. Landsat Missions 4-5 provides images of 30m resolution covering the the period from 1983 to 2013 and Landsat Missions 7-8 are currently collecting enhanced images at 15m capturing Cirrus and Panchromatic bands, in addition to the traditional RGB, Near-, Shortwave-Infrared, and Thermal bands. The Landsat 6 mission was unsuccessful due to the transporting rocket not reaching orbit. Landsat imagery is openly available and offers extensive temporal coverage streching for 46 years. The table below provides a summary overview of the operation, revisit time and image resolution for the Landsat programme.

Mission	Operational time	Revisit time	Resolution
Landsat 1	1972-1978	18 d	80 m
Landsat 2	1975-1982	18 d	80 m
Landsat 3	1978-1983	18 d	80 m
Landsat 4	1983-1993	16 d	30 m
Landsat 5	1984-2013	16 d	30 m
Landsat 7	1999-present	16 d	15 m
Landsat 8	2013-present	16 d	15 m

Additional Earth observation programmes exist. These programmes also offer freely accessible imagery at a higher resolution.

Provider	Programme	Operational time	Revisit time	Resolution
European Space Agency	Sentinel	2015-present	5 d	10m
Planet Labs	Rapideye PlanetscopeSkysat	2009-present	4/5 d to daily	up to 0.8 m
NASA	Orbview 3	2003-2007	< 3 d	1-4 m
NASA	EO-1	2003 -2017	–	10-30 m

Study Area In this analysis, we examine urban changes in Shanghai, China. Shanghai has experienced rapid population growth. Between 2000 and 2010, Shanghai's population rose by 7.4 million from 16.4 million to 23.8 million. It is annual growth rate of 3.8 percent over 10 years. While the pace of population expansion has been less acute, Shanghai's population has continued to grow. In 2018, an estimated 24.24 million people were living in Shanghai experiencing a population expansion of approximately 8 million since 2010.

To extract satellite imagery, a first step is to identify the shape of the geographical area of interest. To this end, we use a polygon shapefile (Shapefile source). These polygons represent the Shanghai metropolitan area, so they include the city centre and surrounding areas. These polygons will be used as a bounding box to identify and extract relevant satellite images. We need to ensure the shapefile is in the same coordinate reference system (CRS) as the satellite imagery (WGS84 or EPSG:4326).

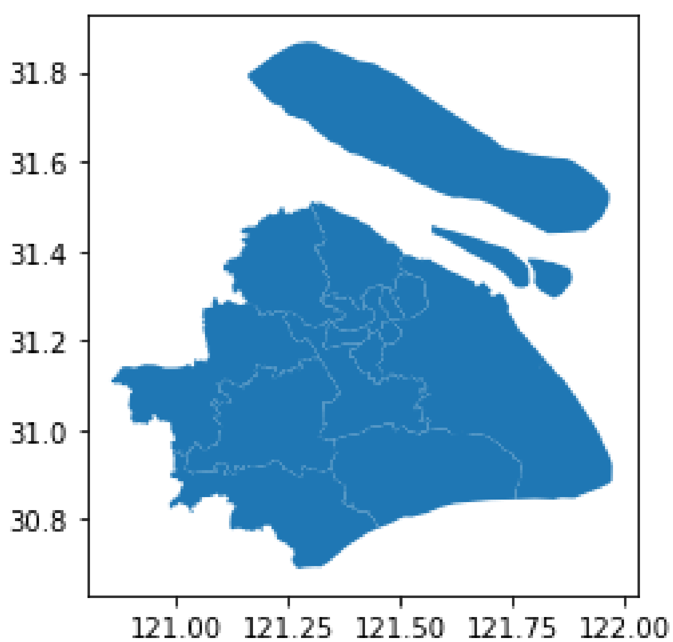
[2]:
```
# Specify the path to your shapefile
directory = os.path.dirname(wdir)
shp = 'shang_dis_merged/shang_dis_merged.shp'
```

[3]:
```
# Certify that the shapefile is in the right coordinate system, otherwise␣
↪reproject it into the right CRS
def shapefile_crs_check(file):
    global bbox
    bbox = gpd.read_file(file)
    crs = bbox.crs
    data = crs.get("init", "")
    if 'epsg:4326' in data:
        print('Shapefile in right CRS')
    else:
        bbox = bbox.to_crs({'init':'epsg:4326'})
    return bbox.plot()
```

[4]:
```
shapefile_crs_check(shp)
```

```
Shapefile in right CRS
```

[4]: <AxesSubplot:>



The world reference system (WRS) from NASA is a system to identify individual satellite imagery scenes using path-row tuples instead of absolute latitude/longitude coordinates. The latitudinal center of the image correpsonds to the row, the longitudinal center to the path. This system allows to uniformly catalogue satellite data across multiple missions and provides an easy to use reference system for the end user. It is necessary to note that the WRS was changed between Landsat missions, due to a difference in swath patterns of the more recent Landsat satellites (NASA, 2019). The WRS1 is used for Landsat missions 1-3 and the WRS2 for Landsat missons 4,5,7,8. In order to obtain path-row tuples of relevant satellite images for an area of interest (AOI), it is necessary to intersect the WRS shapefile (either WRS1 or WRS2, depending on the Landsat satellite you would like to obtain data from) with the AOI shapefile. The resulting path-row tuples will later be used to locate and download the corresponding satellite images from the Google Cloud Storage. The output of the intersection between WRS and AOI files can be visualised using an interactive widget. The map below shows our area of interest in purple and the footprints of the relevant Landsat images on top of an OpenStreetMap basemap.

[ ]:
```
# Donwload the WRS 2 file to later intersect the shapefile with the WRS path/
↪row tuples to identify
# relevant Landsat scenes
def sat_path():

    url = 'https://landsat.usgs.gov/sites/default/files/documents/
↪WRS2_descending.zip'

    # Create folder for WRS2 file
    if os.path.exists(os.path.join('Landsat_images','wrs2')):
        print('folder exists')
    else:
        os.makedirs(os.path.join('Landsat_images','wrs2'))

    WRS_PATH = os.path.join('Landsat_images','WRS2_descending.zip')
    LANDSAT_PATH = os.path.dirname(WRS_PATH)

    # The WRS file is only needed once thus we add this loop
    if os.path.exists(WRS_PATH):
        print('File already exists')
    # Downloads the WRS file from the URL given and unzips it
    else:
        wget.download(url, out = LANDSAT_PATH)
        shutil.unpack_archive(WRS_PATH, os.path.join(LANDSAT_PATH, 'wrs2'))
```

[ ]:
```
%%time
# WARNING: this will take time the first time it's executed
```

```
# depending on your connection
sat_path()
```

[ ]:
```
# Intersect the shapefile with the WRS2 shapefile to determine relevant path/
↪row tuples
def get_pathrow():
    global paths,rows,path,row, wrs_intersection

    wrs=gpd.GeoDataFrame.from_file(os.path.
↪join('Landsat_images','wrs2','WRS2_descending.shp'))
    wrs_intersection=wrs[wrs.intersects(bbox.geometry[0])]
    paths,rows=wrs_intersection['PATH'].values, wrs_intersection['ROW'].values

    for i, (path,row) in enumerate(zip(paths,rows)):
        print('Image', i+1, ' -path:', path, 'row:', row)
```

[ ]:
```
get_pathrow()
```

[ ]:
```
# Visualise the output of the intersection with the shapefile using Folium

# Get the center of the map
xy = np.asarray(bbox.centroid[0].xy).squeeze()
center = list(xy[::-1])

# Select a zoom
zoom = 8

# Create the most basic OSM folium map
m = folium.Map(location = center, zoom_start = zoom, control_scale=True)

# Add the bounding box (bbox) GeoDataFrame in red using a lambda function
m.add_child(folium.GeoJson(bbox.__geo_interface__, name = 'Area of Interest',
                           style_function = lambda x: {'color': 'purple',␣
↪'alpha': 0}))


# Iterate through each polygon of paths and rows intersecting the area
for i, row in wrs_intersection.iterrows():
    # Create a string for the name containing the path and row of this Polygon
    name = 'path: %03d, row: %03d' % (row.PATH, row.ROW)
    # Create the folium geometry of this Polygon
    g = folium.GeoJson(row.geometry.__geo_interface__, name=name)
    # Add a folium Popup object with the name string
    g.add_child(folium.Popup(name))
    # Add the object to the map
    g.add_to(m)
m
```

[10]:
```python
# Display number of images and Path/Row of the image
for i, (path,row) in enumerate(zip(paths,rows)):
    print('Image', i+1, ' -path:', path, 'row:', row)
```

```
Image 1  -path: 118 row: 38
Image 2  -path: 119 row: 38
```

Note that here you have two options: 1) continuing and executing the code reported in the next two sections on data donwload and image cropping, or 2) skipping these sections and proceeding to the image mosaicing sections. We recommend 2) as the processing of unzipping every folder may take long causing the JupyterLab instance to crash.

1.2.3 Data download and pre-processing

We now have relevant path and row tuples for our area of analysis. So we can proceed to download satellite images, wich are stored on the Google Cloud. To download images, we specify certain parameters: time frame, cloudcover in percentage (0-100 %) and satellite mission (1-5,7,8). The here used Landsat578 API automatically searches the Google Cloud for scenes with the specified parameters and downloads matching images. In order to search the Google Cloud for relevant images, a list of available needs to be downloaded when the code is run for the first time. The list provides basic information of the satellite images and since Landsat data acquistion is ongoing, is updated continuously. Thus, if data from the latest aquistion data is required, it is recommended to re-download the file list before running the code.

We use satellite imagery from Landsat 5 scene taken in 1984 and Landsat 8 taken in 2019 to determine neighbourhood changes over time. Landsat 5 scenes can be obtained from two different sensors, the Multispectral Scanner System and the Thematic Mapper, which provide 4 and 7 bands, respectively. The Multispectral Scanner System (MSS) is used in Landsat 1-3 and was superseded by the Thematic Mapper (TM). The MSS provides a green and red band (Band numbers: 1,2) and two infrared bands (Band numbers: 3,4), while the TM provides bands covering red,blue and green (Band numbers: 1,2,3), near-infrared (Band numbers: 4), short-wave infrared (Band numbers:5,7) and thermal infrared (6). Each downloaded scene contains all bands with one image per band. The different bands can then be stacked in order to highlight various Earth surface processes. In this exercise, scenes from the MSS and TM are downloaded, but only data from the TM is used for analysis.

The Operational Land Imager (OLI) aboard Landsat 8 provides multispectral bands (bands 1-7 and 9) with a resolution of 30 meters and a panchromatic band (band 8) with a resolution of 15 meters (Barsi et al., 2014a). The Thermal Infrared Sensor (TIRS) provides thermal infrared images (bands 10 and 11) with a resolution of 100 meters (Barsi et al., 2014b). The Landsat 8 satellite has a swath width of 185 km for the OLI and TIRS instruments, so one scene usually captures the extent of a city. In other cases, the geographical area of interest may extend beyond one image so that multiple images may be needed (Barsi et al., 2014b, Knight & Kvaran, 2014). Given the revisit time of 16 days, usually cloud free images can be retrieved for most cities on bi-weekly or monthly basis (Roy et al., 2014). The folder and filename of each scene provides information about the satellite, instrument, path/row tuple and date.

The tables below shows which general information of the downloaded scenes can be inferred from the folder and file names of each individual scene:

FOLDER:

LXPPPRRRYYYYDDDGSIVV

Parameter	Meaning
L	Landsat
X	Sensor ("C"=OLI/TIRS combined, "O"=OLI-only, "T"=TIRS-only, "E"=ETM+, "T"="TM, "M"=MSS)
PPP	WRS path
RRR	WRS row
YYYY	Year
DDD	Julian day of year
GSI	Ground station identifier
VV	Archive version number

IMAGE:

LXSS_LLLL_PPPRRR_YYYYMMDD_yyyymmdd_CC_TX

Parameter	Meaning
L	Landsat
X	Sensor ("C"=OLI/TIRS combined, "O"=OLI-only, "T"=TIRS-only, "E"=ETM+, "T"="TM, "M"=MSS)
SS	Satellite ("07"=Landsat 7, "08"=Landsat 8)
LLL	Processing correction level (L1TP/L1GT/L1GS)
PPP	WRS path
RRR	WRS row
YYYYMMDD	Acquisition year, month, day
yyyymmdd	Processing year, month, day
CC	Collection number (01, 02, …)
TX	Collection category ("RT"=Real-Time, "T1"=Tier 1, "T2"=Tier 2)

[ ]:
```
# Download Tile list from Google - only needs to be done when first running the␣
↪code
# NOTE this cell is using the ! magic, which runs command line processes from a␣
↪Jupyter
# notebook. Make sure the `landsat` tool, from the `landsat578` package is␣
↪installed
# and available

#Path to index file
Index_PATH = os.path.join(directory + '/index.csv.gz')
```

```
if os.path.exists(Index_PATH):
    print('File already exists')
else:
    !landsat --update-scenes yes
```

[ ]:
```
# Define Download function to acquire scenes from the Google API
def landsat_download(start_date, end_date, sat,path,row,cloud,output):
    g=GoogleDownload(start=start_date, end=end_date, satellite=sat, path=path,␣
↪row=row, max_cloud_percent=cloud, output_path=output)
    g.download()
```

[ ]:
```
# Specify start/end date (in YYYY-MM-DD format), the cloud coverage of the␣
↪image (in %) and the satellite
# you would like to acquire images from (1-5,7,8). In this case we acquire a␣
↪recent scene from Landsat 8
# with a cloud coverage of 5 %.

start_date = '2019-01-01'
end_date = '2019-02-20'
cloud = 5
satellites = [8]
output = os.path.join(directory + '/Lansat_images/')
```

[ ]:
```
# Loop through the specified satellites for each path and row tuple
for sat in satellites:
    for i, (path,row) in enumerate(zip(paths,rows)):
        print('Image', i+1, ' -path:', path, 'row:', row)
        landsat_download(start_date, end_date,sat,path,row,cloud,output)
```

[ ]:
```
# The above step is repeated to acquire a Landsat 5 scene from 1984 with 5 %␣
↪cloud coverage.
start_date = '1984-04-22'
end_date = '1984-04-24'
cloud = 5
satellites = [5]
output = os.path.join(directory + '/Lansat_images/')
```

[ ]:
```
# Loop through the specified satellites for each path and row tuple
for sat in satellites:
    for i, (path,row) in enumerate(zip(paths,rows)):
        print('Image', i+1, ' -path:', path, 'row:', row)
        landsat_download(start_date, end_date,sat,path,row,cloud,output)
```

[ ]:
```
# Delete Scenes that were acquired using the MSS:
outdir = os.listdir(output)
for i in outdir:
```

```
    if 'LM' in os.path.basename(i):
        try:
            shutil.rmtree(os.path.abspath(os.path.join(output,os.path.
↪basename(i))))
        except OSError as e:
            print ("Error: %s - %s." % (e.filename, e.strerror))
```

Image Cropping Satellite imagery is large. The size per image can easily equate to 1 GB. It often makes the data processing and analysis computationally expensive. Cropping the obtained scenes to the relevant region of the image enables faster processing and analysing by significantly reducing the size of the input.

[ ]:
```
# Define cropping function using command line gdalwarp.
## Note: The BQA band is the quality assessment band, which has a different no␣
↪data value (1) than the other
## bands (0), which makes it necessary to us a different croping function.

def crop(inraster,outraster,shape):
        !gdalwarp -cutline {shape} -srcnodata 0 -crop_to_cutline {inraster}␣
↪{outraster}

def crop_bqa(inraster,outraster,shape):
        !gdalwarp -cutline {shape} -srcnodata 1 -crop_to_cutline {inraster}␣
↪{outraster}
```

[ ]:
```
# Loop through every folder and a create an image cropped to the extent of the␣
↪shapefile
# save it with the original name and the extension _Cropped
for t in range(0,12):
    for filename in glob.glob((output+'/**/*_B{}.tif').format(t),␣
↪recursive=True):
        inraster = filename
        outraster = filename[:-4] + '_Cropped.tif'
        crop(inraster, outraster, shp)
for filename in glob.glob(output+'/**/*.tif'):
    if 'BQA.TIF' in i:
        inraster = i
        outraster = i[:-4] + '_Cropped.tif'
        crop_bqa(inraster,outraster,shp)
```

Image mosaic As indicated above, a single Landsat scene may not cover the full extent of a city due to the satellite's flight path as can be observed from the interactive map. Creating a mosaic of two or more images is thus often needed to produce a single image that covers the enterity of the area under analysis.

```
[ ]: # Read in the relevant Landsat 8 files
output = 'Landsat_images/'
images = sorted(os.listdir(output))
dirpath1 = os.path.join(output, images[0])
dirpath2 = os.path.join(output, images[1])
mosaic_n = os.path.join(output,'Mosaic/')
search = 'L*_Cropped.tif'
query1 = os.path.join(dirpath1,search)
query2 = os.path.join(dirpath2,search)
files1 =  glob.glob(query1)
files2 =  glob.glob(query2)
files1.sort()
files2.sort()
if os.path.exists(mosaic_n):
    print('Output Folder exists')
else:
    os.makedirs(mosaic_n)
```

```
[ ]: # Match bands together and create a mosaic. Since the BQA band and the␣
 ↪cloudmask have different denominations
# than the other bands, these images have to be merged together separately.
def mosaic_new(scene1,scene2):
    src_mosaic =[]
    string_list=[]
    for i,j in zip(scene1,scene2):
        for k in range(1,12):
            string_list.append('B{}_Cropped'.format(k))
        for l in range(0,11):
            if string_list[l] in os.path.basename(i) and os.path.basename(j):
                src1 = rasterio.open(i)
                src2 = rasterio.open(j)
                src_mosaic = [src1,src2]
                mosaic,out_trans = rasterio.merge.merge(src_mosaic)
                out_meta = src1.meta.copy()
                out_meta.update({"driver": "GTiff", 'height':mosaic.
 ↪shape[1],'width':mosaic.shape[2],
                                'transform':out_trans})
                outdata = os.path.join(mosaic_n,'B{}_mosaic.tif'.format(l))
                with rasterio.open(outdata,'w',**out_meta) as dest:
                    dest.write(mosaic)
        # Mosaic Quality Assessment Band
        if 'BQA_Cropped' in os.path.basename(i) and os.path.basename(j):
            bqa1 = rasterio.open(i)
            bqa2 = rasterio.open(j)
            bqa_mosaic = [bqa1,bqa2]
            mosaic_,out_trans = rasterio.merge.merge(bqa_mosaic,nodata=1)
            out_meta = bqa1.meta.copy()
```

```
            out_meta.update({"driver": "GTiff", 'height':mosaic_.
↪shape[1],'width':mosaic_.shape[2],
                             'transform':out_trans})
            outdata = os.path.join(mosaic_n,'BQA_mosaic.tif')
            with rasterio.open(outdata,'w',**out_meta) as dest:
                dest.write(mosaic_)

            # Mosaic of Cloudmask
            search = 'cloudmask.tif'
            query3 = os.path.join(dirpath1,search)
            query4 = os.path.join(dirpath2,search)
            files3 =  glob.glob(query3)
            files4 =  glob.glob(query4)
            for i,j in zip(files3,files4):
                if 'cloudmask' in os.path.basename(i)and os.path.basename(j):
                    cloudmask1 = rasterio.open(i)
                    cloudmask2 = rasterio.open(j)
                    cloud_mosaic = [cloudmask1,cloudmask2]
                    mosaic_c,out_trans = rasterio.merge.
↪merge(cloud_mosaic,nodata=1)
                    out_meta = cloudmask1.meta.copy()
                    out_meta.update({"driver": "GTiff", 'height':mosaic_c.
↪shape[1],'width':mosaic_c.shape[2],
                                     'transform':out_trans})
                    outdata = os.path.join(mosaic_n,'Cloudmask_mosaic.tif')
                    with rasterio.open(outdata,'w',**out_meta) as dest:
                        dest.write(mosaic_c)
```

[ ]:
```
mosaic_new(files1,files2)
```

[ ]:
```
# Read in the relevant files for the Landsat 5 scenes
images = sorted(os.listdir(output))
dirpath_o1 = os.path.join(output, images[2])
dirpath_o2 = os.path.join(output, images[3])
mosaic_o = os.path.join(output,'Mosaic_old/')
query_o1 = os.path.join(dirpath_o1,search)
query_o2 = os.path.join(dirpath_o2,search)
files_o1 =  glob.glob(query_o1)
files_o2 =  glob.glob(query_o2)
files_o1.sort()
files_o2.sort()
if os.path.exists(mosaic_o):
    print('Output Folder exists')
else:
    os.makedirs(mosaic_o)
```

```
[ ]: # Match bands together and create a mosaic. Since the BQA band and the␣
     ↪cloudmask have different denominations
     # than the other bands, these images have to be merged together separately.
     def mosaic_old(scene_o1,scene_o2):
         src_mosaic =[]
         string_list=[]
         for i,j in zip (scene_o1,scene_o2):

             for k in range(1,8):
                 string_list.append('B{}_Cropped'.format(k))
             for l in range(0,7):
                 if string_list[l] in os.path.basename(i) and os.path.basename(j):
                     src1 = rasterio.open(i)
                     src2 = rasterio.open(j)
                     src_mosaic = [src1,src2]
                     mosaic,out_trans= rasterio.merge.merge(src_mosaic)
                     out_meta = src1.meta.copy()
                     out_meta.update({"driver": "GTiff", 'height':mosaic.
     ↪shape[1],'width':mosaic.shape[2],
                                     'transform':out_trans})
                     outdata = os.path.join(mosaic_o,'B{}_mosaic.tif'.format(l))
                     with rasterio.open(outdata,'w',**out_meta) as dest:
                         dest.write(mosaic)
             # Mosaic Quality Assessment Band
             if 'BQA_Cropped' in os.path.basename(i) and os.path.basename(j):
                 bqa1 = rasterio.open(i)
                 bqa2 = rasterio.open(j)
                 bqa_mosaic = [bqa1,bqa2]
                 mosaic_,out_trans= rasterio.merge.merge(bqa_mosaic,nodata=1)
                 out_meta = bqa1.meta.copy()
                 out_meta.update({"driver": "GTiff", 'height':mosaic_.
     ↪shape[1],'width':mosaic_.shape[2],
                                 'transform':out_trans})
                 outdata = os.path.join(mosaic_o,'BQA_mosaic.tif')
                 with rasterio.open(outdata,'w',**out_meta) as dest:
                     dest.write(mosaic_)

                 # Mosaic of Cloudmask
                 search = 'cloudmask.tif'
                 query_o3= os.path.join(dirpath_o1,search)
                 query_o4 = os.path.join(dirpath_o2,search)
                 files_o3 =  glob.glob(query_o3)
                 files_o4 =  glob.glob(query_o4)
                 for i,j in zip(files_o3,files_o4):
                     if 'cloudmask' in os.path.basename(i)and os.path.basename(j):
                         cloudmask1 = rasterio.open(i)
                         cloudmask2 = rasterio.open(j)
```

```
                    cloud_mosaic = [cloudmask1,cloudmask2]
                    mosaic_c,out_trans= rasterio.merge.
↪merge(cloud_mosaic,nodata=1)
                    out_meta = cloudmask1.meta.copy()
                    out_meta.update({"driver": "GTiff", 'height':mosaic_c.
↪shape[1],'width':mosaic_c.shape[2],
                                    'transform':out_trans})
                    outdata = os.path.join(mosaic_o,'Cloudmask_mosaic.tif')
                    with rasterio.open(outdata,'w',**out_meta) as dest:
                        dest.write(mosaic_c)
```

[ ]:
```
mosaic_old(files_o1,files_o2)
```

1.2.4 Natural-colour (True-colour) composition

Our downloaded data from Landsat 8 and Landsat 5 have different band designations. Combining different satellite bands are useful to identify features of the urban environment: vegetation, built up areas, ice and water. We create a standard natural-colour composition image using Red, Green and Blue satellite bands. This colour composition best reflects the natural environment. For instance, trees are green; snow and clouds are white; and, water is blue. Landsat 8 has 11 bands with bands 4, 3 and 2 corresponding to Red, Green and Blue respectively. Landsat 5 has 7 bands with bands 3, 2 and 1, corresponding to Red, Green and Blue. We perform layer stacking to produce a true colour image composition to gain understanding of the local area before extracting and analying features of the urban environment.

[5]:
```
# Normalise the bands to so that they can be combined to a single image
def normalize(array):
    """Normalizes numpy arrays into scale 0.0 - 1.0"""
    array_min, array_max = array.min(), array.max()
    return ((array - array_min)/(array_max - array_min))
```

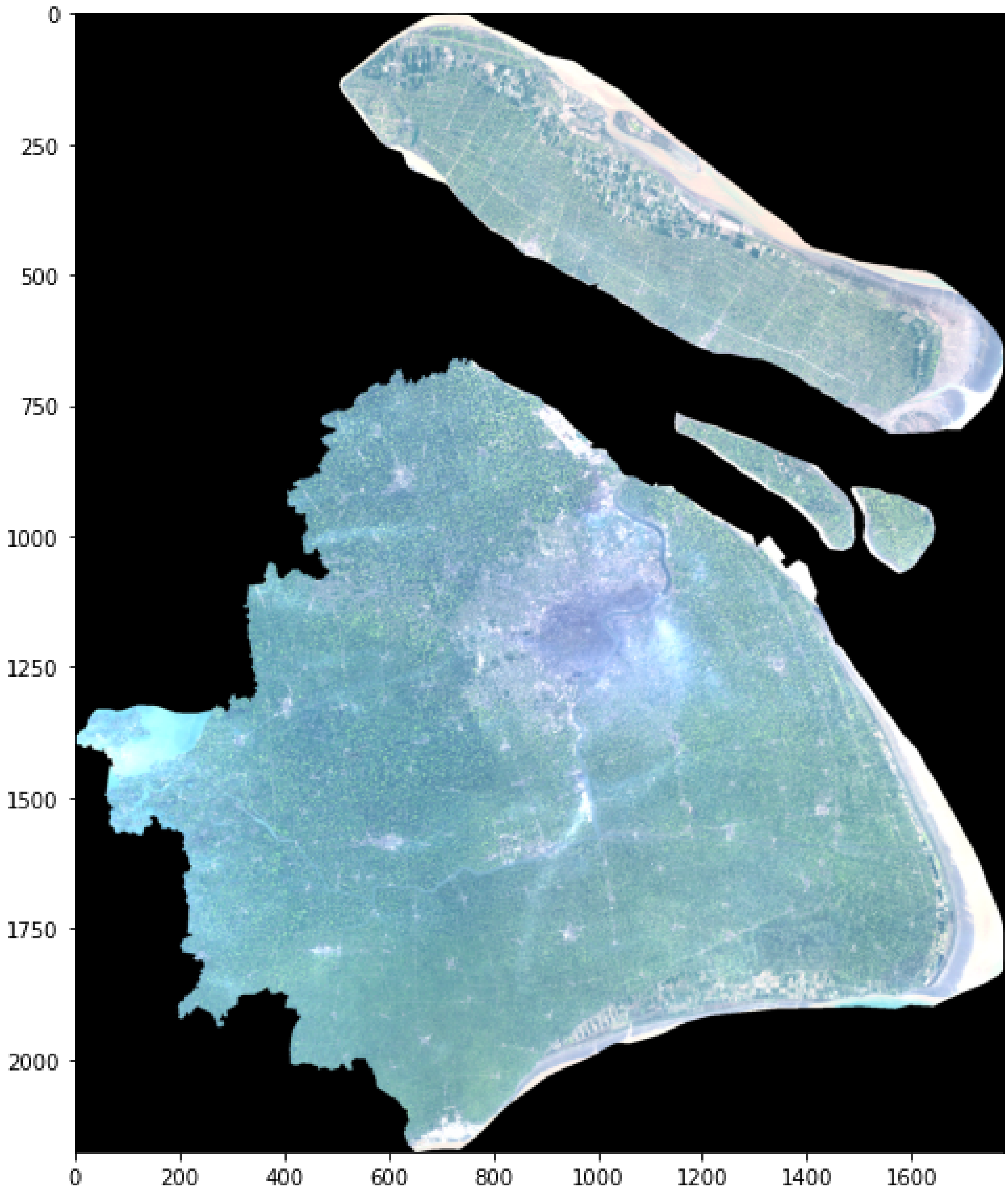
[6]:
```
# Adjust the intensity of each band for visualisation.
# This is a way of rescaling each band by clipping the pixels that are outside␣
↪the specified range to
# the range we defined. By adjusting the gamma, we change the brightness of the␣
↪image with gamma >1
# reuslting in a brighter image. However there are more complex methods such as␣
↪top of the atmosphere
# corrections, which subtracts any atmospheric interference from the image.
# For the purpose of this notebook, this way is sufficient.
def rescale_intensity(image):
    p2, p98 = np.percentile(image, (0.2, 98))
    img_exp = exposure.rescale_intensity(image, in_range=(p2, p98))
    img_gamma = exposure.adjust_gamma(img_exp, gamma=2.5,gain=1)
    return(img_gamma)
```

[7]:
```python
# Downsample image resolution with factor 0.5 for displaying purposes.
def downsample(file):
    downscale_factor=0.5
    data = file.read(1,
        out_shape=(
            file.count,
            int(file.height * downscale_factor),
            int(file.width * downscale_factor)
        ),
        resampling=Resampling.bilinear
    )
    # scale image transform
    transform = file.transform * file.transform.scale(
        (file.width / data.shape[-1]),
        (file.height / data.shape[-2])
    )
    return data
```

[8]:
```python
# Use rasterio to open the Red, Blue and Green bands of the mosaic image from␣
 ↪1984 to create an RGB image
# **NOTE**: The Mosaic names do not correspond to the actual band designations␣
 ↪as python starts
# counting at 0!
with rasterio.open('Landsat_images/Mosaic_old/B0_mosaic.tif') as band1_old:
    b1_old=downsample(band1_old)
with rasterio.open('Landsat_images/Mosaic_old/B1_mosaic.tif') as band2_old:
    b2_old=downsample(band2_old)
with rasterio.open('Landsat_images/Mosaic_old/B2_mosaic.tif') as band3_old:
    b3_old=downsample(band3_old)
```

[9]:
```python
# Normalise the bands so that they can be combined to a single image
red_old_n = normalize(b3_old)
green_old_n = normalize(b2_old)
blue_old_n = normalize(b1_old)

# Apply the function defined before to make more natural-looking image
red_adj = rescale_intensity(red_old_n)
green_adj = rescale_intensity(green_old_n)
blue_adj = rescale_intensity(blue_old_n)

# Stack the three different bands together
rgb_2 = np.dstack((red_adj,green_adj,blue_adj))

# Visualise the true color image
fig,ax = plt.subplots(figsize=(10,10))
ax.imshow(rgb_2)
plt.show()
```

```
plt.close()
del rgb_2,b1_old,b2_old,b3_old,red_adj,green_adj,blue_adj
```

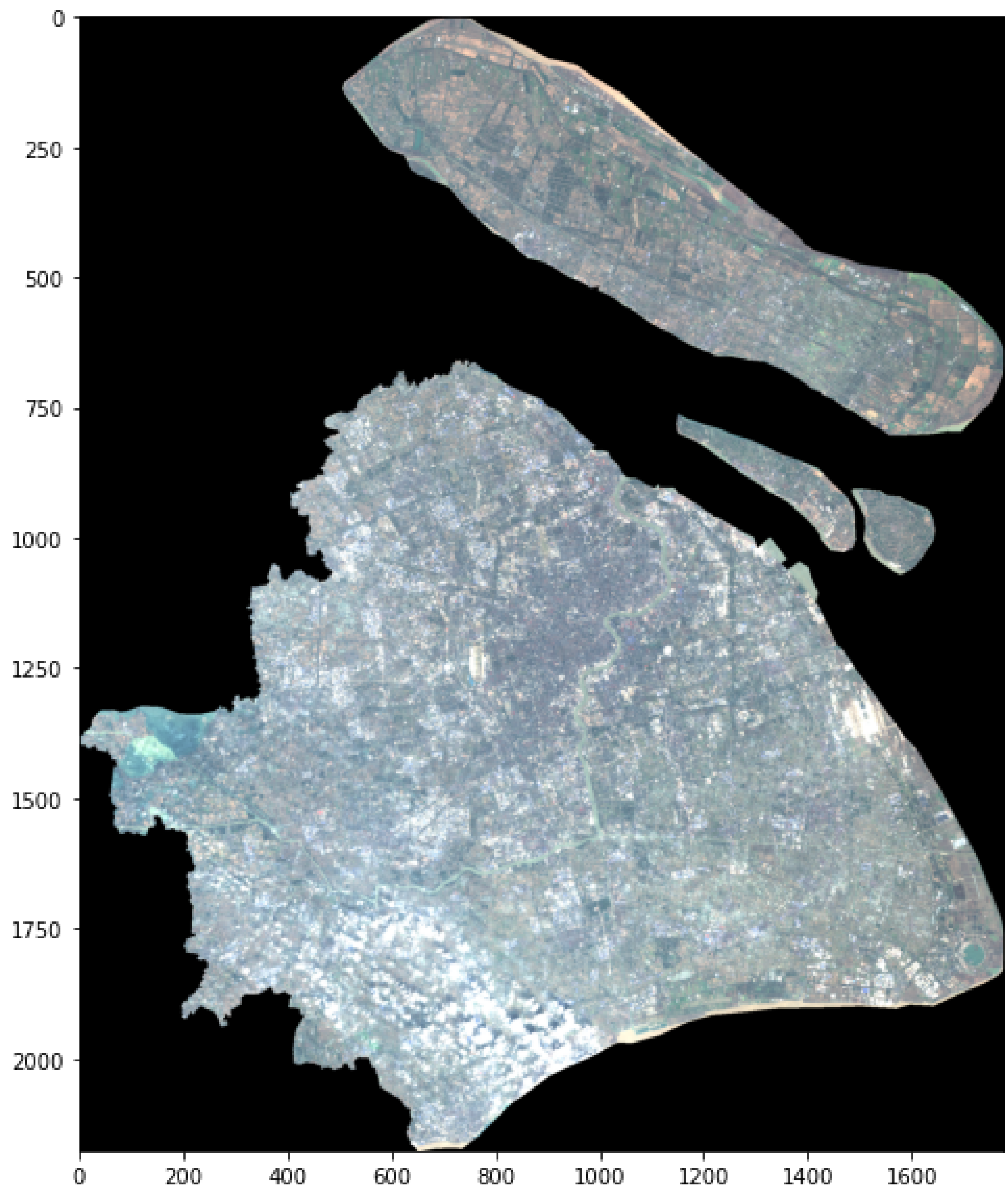


[10]:
```
# Use rasterio to open the Red, Blue and Green bands of the mosaic image from
↪2019 to create an RGB image
# **NOTE**: The Mosaic names do not correspond to the actual band designations
↪as python starts
```

```
# counting at 0!!
with rasterio.open('Landsat_images/Mosaic/B1_mosaic.tif') as band2_new:
    b2_new = downsample(band2_new)
with rasterio.open('Landsat_images/Mosaic/B2_mosaic.tif') as band3_new:
    b3_new = downsample(band3_new)
with rasterio.open('Landsat_images/Mosaic/B3_mosaic.tif') as band4_new:
    b4_new = downsample(band4_new)
```

[11]:

```
# Normalise the bands so that they can be combined to a single image
red_new_n = normalize(b4_new)
green_new_n = normalize(b3_new)
blue_new_n = normalize(b2_new)

# Apply the function defined before to make more natural-looking image
red_rescale = rescale_intensity(red_new_n)
green_rescale = rescale_intensity(green_new_n)
blue_rescale = rescale_intensity(blue_new_n)

# Stack the three different bands together
rgb = np.dstack((red_rescale, green_rescale, blue_rescale))

# Here we adjust the gamma (brightness) for the stacked image to achieve a more␣
 ↪natural looking image.
rgb_adjust = exposure.adjust_gamma(rgb, gamma = 1.5, gain=1)

# Visualise the true color image
fig,ax = plt.subplots(figsize=(10,10))
ax.imshow(rgb_adjust)
plt.show()
plt.close()
del␣
 ↪rgb,red_new_n,green_new_n,blue_new_n,red_rescale,green_rescale,blue_rescale,rgb_adjust
```



1.2.5 Feature extraction

Since the above two maps show that urban neighbourhoods of Shanghai have undergone dramatic changes over time in colour, texture, greenary, buildings, etc., the next stage is to gain valuable information out of satellite imgaes and interpret these changes. Since the images we have downloaded are in city scale, which cover more than a thousand kilometer and less detailed. Therefore, feature extraction is performed to get a reduced representation of the intial image but informative features for subsequent analysis and better interpretation.

We examine four sets of features based the above two maps: colour and texture features from true colour imagery (i.e. RGB bands composition represented by bands 1-3 and bands 2-4 in 1984 and 2019), and vegetation features and built-up features from Red, near infrared (NIR) and shortwave infrared (SWIR) bands, represented by bands 3-5 and bands 4-6 in 1984 and 2019. A more detailed information about the meaning of each band can be found from here. In this analysis, colour features measure the colour moments of ture colour imagery to interpret colour distribution; texture features apply LBP (Local binary patterns) texture spectrum model to show spatial distribution of intensity values in an image; vegetation features calculate the NDVI (Normalised difference vegetation index) to capture the amount of vegetation, and built-up features caculate NDBI (Normalised difference built-up index) to highlight artifically constructed areas.

The administritive divisions of Shanghai have experienced tremendous changes in the last tens of years (MCAPRC, 2018), thus, we will conduct feature extration of imagery on the current administritive boundaries to explore if satellite imagery can be used to reflect and interpret urban changes. The figure below shows the spatial distribution of each administritative area with relative labels in Shanghai.

[12]:
```
#read administritive boundary shapefile of Shanghai
poly = gpd.read_file(shp)


f, ax = plt.subplots(1, figsize = (9,9))
poly.plot(ax = ax)
#create a new colomn, in order to plot polygon labels (i.e. name) in the map
poly['coords']=poly['geometry'].apply(lambda x:x.representative_point().coords[:
 ↪])
poly['coords']=[coords[0] for coords in poly['coords']]
for idx, row in poly.iterrows():
    ax.annotate(text=row['Name'],xy=row['coords'],va='center',ha='center',alpha␣
 ↪= 0.8, fontsize = 8)
plt.axis('equal')
plt.axis('off')
f.suptitle('Administritive divisions of Shanghai', fontsize = 15)
```

[12]:
```
Text(0.5, 0.98, 'Administritive divisions of Shanghai')
```

The above figure shows that administritive divisions of 'Chongming' in the north appear three geometries. Therefore, it is necessary to check if they belong to a single administrive unit

[13]:
```python
poly.loc[poly['Name']== 'Chongming']
```

[13]:
```
   OID_       Name  SymbolID  AltMode  Base  Clamped  Extruded Snippet  \
0     0  Chongming         0        0   0.0       -1         0    None
3     0  Chongming         0        0   0.0       -1         0    None
5     0  Chongming         0        0   0.0       -1         0    None
```

```
                                           PopupInfo  Shape_Leng  Shape_Area  \
0  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    1.921339    0.133777
3  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    0.552764    0.009068
5  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    0.298602    0.005598

                                            geometry  \
0  POLYGON Z ((121.24055 31.71320 0.00000, 121.24…
3  POLYGON Z ((121.77366 31.31691 0.00000, 121.75…
5  POLYGON Z ((121.84247 31.29451 0.00000, 121.83…

                                      coords
0  (121.51489159858086, 31.654863000000034)
3    (121.7157753452659, 31.38690150000005)
5   (121.84187767409576, 31.33721550000007)
```

Since Chongming administritive division consist of three seperate geometries, which may confuse our further analysis. As a result, we dissolved these geometries into a single geometric feature and take a look at the new dataset. The below table shows that Chongming administritive division is now consist of multipolygons which includes all polygons as a whole.

[14]:
```python
#Dissolve geometries with the identical names together
poly = poly.dissolve(by = 'Name').reset_index()
#Have a look at the new table and we can see that chongming districts in the third row
#have been dissolved into a single administritive unit
poly.head()
```

[14]:
```
        Name                                          geometry  OID_  \
0    Baoshan  POLYGON Z ((121.29753 31.50113 0.00000, 121.29…     0
1  Changning  POLYGON Z ((121.34851 31.23906 0.00000, 121.34…     0
2  Chongming  MULTIPOLYGON Z (((121.24055 31.71320 0.00000, …     0
3   Fengxian  POLYGON Z ((121.35276 30.97814 0.00000, 121.36…     0
4    Hongkou  POLYGON Z ((121.46171 31.31811 0.00000, 121.46…     0

   SymbolID  AltMode  Base  Clamped  Extruded Snippet  \
0         1        0   0.0       -1         0    None
1         0        0   0.0       -1         0    None
2         0        0   0.0       -1         0    None
3         4        0   0.0       -1         0    None
4         4        0   0.0       -1         0    None

                                           PopupInfo  Shape_Leng  Shape_Area  \
0  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    1.141426    0.027744
1  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    0.371964    0.003515
2  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    1.921339    0.133777
3  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    1.318697    0.067998
4  <html xmlns:fo="http://www.w3.org/1999/XSL/For…    0.241918    0.002211
```

```
                              coords
0   (121.4139260994209, 31.388573500000064)
1  (121.38161741583315, 31.212733000000043)
2  (121.51489159858086, 31.654863000000034)
3   (121.56065980897392, 30.89682200000007)
4   (121.47207625707068, 31.28110650000005)
```

Image processing Further pre-processing of satellite imagery is needed before feature extraction. This pre-processing involves three steps: * Masking(cropping) of raster files (i.e., Blue, Green, Red, Nir and SWIR bands) into each administrative district polygon; * Image enhancement to improve the quality and content of the original image; and, * Band stacking based on each neighbourhood unit.

[15]:
```
#open raster files
file_list_old = sorted(glob.glob('Landsat_images/Mosaic_old' + "/*.
 ↪tif",recursive = True))
files_old = [rio.open(filename) for filename in file_list_old]
```

[16]:
```
file_list = sorted(glob.glob('Landsat_images/Mosaic' + "/*.tif"))
files = [rio.open(filename) for filename in file_list]
```

Before cropping all raster files into each polygon in the vector file (i.e. Shanghai administrative area shapefile), we have to ensure they have the same coordinate reference system (CRS). Once matched, the cropping process is prepared to go.

[17]:
```
poly.crs
```

[17]:
```
{'init': 'epsg:4326'}
```

[18]:
```
#check the crs of one band of satellite imagery
files[0].crs
```

[18]:
```
CRS.from_epsg(32651)
```

[20]:
```
#reproject the vector file to make it consistent with raster files
poly = poly.to_crs('EPSG:32651')
```

[21]:
```
#get each neighbourhood geographic boundary based on administritive area data
geo = [poly.__geo_interface__['features'][i]['geometry']
            for i in range(len(poly))]
```

[22]:
```
#clip R,G,B bands separately by each poly, so get pixel values in each poly and␣
 ↪save them into a list
out_image = [[] for i in range(5)]
img_old = [[] for i in range(5)]
```

```python
#x: Blue,Green,Red,NIR and SWIR bands, y: 16 polygons from vertor file
for x,y in itertools.product(range(5),range(len(geo))):
    #out_image[0] means masked Blue band polygon
    out_image[x].append(mask(files_old[0:5][x], [geo[y]], crop=True))
    #image enhancement: normalisation and Histogram Equalization
    img_old[x].append(exposure.equalize_hist(normalize(out_image[x][y][0][0])))
del out_image,files_old
```

[23]:
```python
#clip R,G,B bands separately by each poly, so get pixel values in each poly and␣
↪save them into a list
out_image = [[] for i in range(5)]
img_new = [[] for i in range(5)]

#x: Blue,Green,Red,NIR and SWIR bands, y: 16 polygons from vertor file
for x,y in itertools.product(range(5),range(len(geo))):
    #out_image[0] means masked Blue band polygon
    out_image[x].append(mask(files[0:5][x], [geo[y]], crop=True))
    #image enhancement: normalisation and Histogram Equalization
    img_new[x].append(exposure.equalize_hist(normalize(out_image[x][y][0][0])))
del out_image,files
```

[24]:
```python
#have a look at the pixel values of one geographic area in blue band
img_new[0][0]
```

[24]:
```
array([[0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378],
        [0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378],
        [0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378],
        …,
        [0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378],
        [0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378],
        [0.48515378, 0.48515378, 0.48515378, …, 0.48515378, 0.48515378,
         0.48515378]])
```

[25]:
```python
#stack R,G,B bands together for later feature extraction
bb = [img_old[0][x].astype(np.float) for x in range(len(geo))]
bg = [img_old[1][x].astype(np.float) for x in range(len(geo))]
br = [img_old[2][x].astype(np.float) for x in range(len(geo))]
```

[26]:
```python
rgb_old = [np.dstack((br[x],bg[x],bb[x])) for x in range(len(geo))]
```

[27]:
```python
bb = [img_new[0][x].astype(np.float) for x in range(len(geo))]
bg = [img_new[1][x].astype(np.float) for x in range(len(geo))]
```

```
br = [img_new[2][x].astype(np.float) for x in range(len(geo))]
```

[28]:
```
rgb_new = [np.dstack((br[x],bg[x],bb[x])) for x in range(len(geo))]
```

Colour features Colour features are used to extract the characteristics of colours from satellite imagery. A commonly used method to extract colour feauters is to compute colour moments of an image. Colour moments provide a measurement of colour similiarity between images (Keen, 2005). Basically, colour probability distribution of an image are characterised by a range of unique moments. The mean, standard deviation and skewness these three central moments are generally used to identify colour distribution. Here we extract colour features on HSV (Hue, Saturation and Value) colour space because it corresponds to human vision and has been widely used in computer vision. HSV colour space can be converted from RGB colour channels, Hue represents the colour portion, saturation represents the amount of gray in a particular colour (0 is gray), and Value represents the brightness of the colour (0 is black). Therefore, the true-colour imagery is characterised by a total of nine moments - three moments for each HSV channel in the same units.

[29]:
```
#interpret the color probability distribution by computing low order color␣
↪moments(1,2,3)
def color_moments(img):
    if img is None:
        return
    # Convert RGB to HSV colour space
    img_hsv = rgb2hsv(img)
    # Split the channels - h,s,v
    h, s, v = [img_hsv[:,:,i] for i in [0,1,2]]
    # Initialize the colour feature
    color_feature = []
    # N = h.shape[0] * h.shape[1]
    # The first central moment - average
    h_mean = np.mean(h)  # np.sum(h)/float(N)
    s_mean = np.mean(s)  # np.sum(s)/float(N)
    v_mean = np.mean(v)  # np.sum(v)/float(N)
    color_feature.extend([h_mean, s_mean, v_mean])
    # The second central moment - standard deviation
    h_std = np.std(h)  # np.sqrt(np.mean(abs(h - h.mean())**2))
    s_std = np.std(s)  # np.sqrt(np.mean(abs(s - s.mean())**2))
    v_std = np.std(v)  # np.sqrt(np.mean(abs(v - v.mean())**2))
    color_feature.extend([h_std, s_std, v_std])
    # The third central moment - the third root of the skewness
    h_skewness = np.mean(abs(h - h.mean())**3)
    s_skewness = np.mean(abs(s - s.mean())**3)
    v_skewness = np.mean(abs(v - v.mean())**3)
    h_thirdMoment = h_skewness**(1./3)
    s_thirdMoment = s_skewness**(1./3)
    v_thirdMoment = v_skewness**(1./3)
    color_feature.extend([h_thirdMoment, s_thirdMoment, v_thirdMoment])
```

```python
    return color_feature
```

[30]:
```python
#create and initialize a data table to store colour feastures
color_mom_old = pd.DataFrame(color_moments(rgb_old[0]))
#add the rest columns by assigning 9 color moments in each poly
for i in range(1,len(rgb_old)):
    color_mom_old[i]= color_moments(rgb_old[i])
    i = i+1
```

[31]:
```python
#create and initialize a data table
color_mom_new = pd.DataFrame(color_moments(rgb_new[0]))
#add the rest columns by assigning 9 color moments in each poly
for i in range(1,len(rgb_new)):
    color_mom_new[i]= color_moments(rgb_new[i])
    i = i+1
```

[32]:
```python
#Data manipulation
color_old_var = color_mom_old.T
#assign column names
color_old_var.columns =␣
 ↪['h_mean','s_mean','v_mean','h_std','s_std','v_std','h_skew','s_skew','v_skew']
#set geographic name as index
color_old_var= color_old_var.set_index(poly.Name)
```

[33]:
```python
color_new_var = color_mom_new.T
color_new_var.columns =␣
 ↪['h_mean','s_mean','v_mean','h_std','s_std','v_std','h_skew','s_skew','v_skew']
color_new_var= color_new_var.set_index(poly.Name)
```

As we have created two new tables for colour features in the year of 1984 and 2019, it would be helpful to have a view of the tables and see how they look like. The below two tables show nine variables (column) representing colour features within five adminstritive division of Shanghai (row).

[34]:
```python
#check the information of colour feature
color_old_var.head()
```

[34]:
```
             h_mean    s_mean    v_mean     h_std     s_std     v_std  \
Name
Baoshan    0.272161  0.052081  0.644148  0.327094  0.072457  0.185309
Changning  0.221412  0.051564  0.659894  0.288368  0.075177  0.174455
Chongming  0.153807  0.017394  0.742309  0.272162  0.035627  0.102347
Fengxian   0.339613  0.112915  0.605758  0.321941  0.122805  0.243621
Hongkou    0.249526  0.063704  0.650725  0.309825  0.087439  0.187805

             h_skew    s_skew    v_skew
Name
Baoshan    0.356713  0.090057  0.199446
```

```
Changning  0.330803  0.092504  0.187627
Chongming  0.332916  0.051184  0.120603
Fengxian   0.347226  0.144392  0.257670
Hongkou    0.347968  0.106194  0.200131
```

[35]:
```
color_new_var.head()
```

[35]:
```
             h_mean    s_mean    v_mean     h_std     s_std     v_std  \
Name
Baoshan    0.231070  0.035865  0.638941  0.297847  0.052048  0.180189
Changning  0.231849  0.031294  0.649237  0.304689  0.048471  0.167002
Chongming  0.157306  0.016473  0.742402  0.282495  0.031843  0.101771
Fengxian   0.295539  0.086197  0.605431  0.302731  0.097695  0.243596
Hongkou    0.239944  0.037582  0.638613  0.303830  0.055047  0.182938

             h_skew    s_skew    v_skew
Name
Baoshan    0.336779  0.067866  0.195591
Changning  0.344394  0.062974  0.182483
Chongming  0.345289  0.043360  0.119800
Fengxian   0.329388  0.115702  0.257234
Hongkou    0.339615  0.070552  0.197624
```

Texture features To extract texture features, we use a Local Binary Pattern (LBP) approach. LBP searches for pixels adjacent to a central point and tests whether these surrounding pixels are greater or less than the central pixel and generate a binary classification (Pedregosa et al., 2011). In theory, eight adjacent neighbour pixels in grayscale are set to compare with one central pixel value by 3 * 3 neighbourhood threshold, and consider the result as 1 or 0 (Ojala et al., 1996). Thus, these eight surrounding binary numbers correspond to LBP code for the central pixel value, determining the texture pattern of that threshold. Texture features are then the distribution of a collection of LBPs over an image.

[36]:
```
#convert a RGB image into Grayscale,which takes less space for analysis
gray_images_old = [rgb2gray(rgb_old[i]) for i in range(len(rgb_old))]
gray_images_new = [rgb2gray(rgb_new[i]) for i in range(len(rgb_new))]
```

[37]:
```
# settings for LBP
radius = 1 #radius = 1 refers to a 3*3 patch/window scale
n_points = 8 * radius # the number of circularly symmetric neighbour set points
method = 'uniform'#finer quantization of the angular space which is gray scale␣
↪and rotation invariant

lbps_old = [local_binary_pattern(gray_images_old[i],n_points,radius,method) for␣
↪i in range(len(rgb_old))]
lbps_new = [local_binary_pattern(gray_images_new[i],n_points,radius,method) for␣
↪i in range(len(rgb_new))]
```

[38]:
```
#n_bins are the same in each neighbourhood
n_bins = int(lbps_old[0].max()+1)
#define a function to count the number of points in a given bin of LBP␣
↪distribution histogram
def count_hist(x):
    return np.histogram(lbps_old[x].ravel(),density=True, bins=n_bins,range=(0,␣
↪n_bins))
#Assign counts to a new list, return the higtogram vector features in this␣
↪cell(polygon)
hist_features_old = [count_hist(i)[0] for i in range(len(rgb_old))]
```

[39]:
```
#Extract texture features of another year based on same method
n_bins = int(lbps_new[0].max()+1)

def count_hist(x):
    return np.histogram(lbps_new[x].ravel(),density=True, bins=n_bins,range=(0,␣
↪n_bins))

#Assign counts to a new list, return the higtogram vector features in this␣
↪cell(polygon)
hist_features_new = [count_hist(i)[0] for i in range(len(rgb_new))]
```

Same with opertations on colour features, this time we build two new tables for texture features, with each row present administritive division and each column represent texture feature.

[40]:
```
#The histogram features are the texture features
texture_old_var = pd.DataFrame([hist_features_old[a] for a in␣
↪range(len(rgb_old))])
texture_old_var.columns = ['LBP'+ str(i) for i in range(n_bins)]
texture_old_var = texture_old_var.set_index(poly.Name)
#Have a look at the table with texture features of administrive division of␣
↪Shanghai in 1984
texture_old_var.head()
```

[40]:
```
                LBP0      LBP1      LBP2      LBP3      LBP4      LBP5  \
Name
Baoshan     0.035093  0.041960  0.040705  0.068394  0.078389  0.067483
Changning   0.036086  0.046078  0.041956  0.059792  0.060040  0.064422
Chongming   0.025822  0.029946  0.021757  0.034058  0.039580  0.036787
Fengxian    0.055508  0.066468  0.051230  0.072002  0.073767  0.073928
Hongkou     0.042018  0.050562  0.043542  0.059326  0.056759  0.070718

                LBP6      LBP7      LBP8      LBP9
Name
Baoshan     0.040339  0.041101  0.520053  0.066483
Changning   0.037538  0.043385  0.539416  0.071285
Chongming   0.024035  0.029158  0.709444  0.049413
```

```
Fengxian   0.052907  0.065099  0.382110  0.106981
Hongkou    0.039691  0.046490  0.510429  0.080465
```

[41]:
```
#The histogram features are the texture features
texture_new_var = pd.DataFrame([hist_features_new[a] for a in␣
 ↪range(len(rgb_new))])
texture_new_var.columns = ['LBP'+ str(i) for i in range(n_bins)]
texture_new_var = texture_new_var.set_index(poly.Name)
#Have a look at the table with texture features of administrive division of␣
 ↪Shanghai in 2019
texture_new_var.head()
```

[41]:
```
               LBP0      LBP1      LBP2      LBP3      LBP4      LBP5  \
Name
Baoshan    0.043059  0.047740  0.040768  0.058617  0.068075  0.056809
Changning  0.042641  0.050118  0.037668  0.051370  0.058422  0.061528
Chongming  0.025468  0.029637  0.023332  0.035217  0.047621  0.036412
Fengxian   0.051206  0.061050  0.052882  0.081410  0.097157  0.079233
Hongkou    0.047032  0.054172  0.042940  0.054392  0.055014  0.068051

               LBP6      LBP7      LBP8      LBP9
Name
Baoshan    0.037410  0.045565  0.524189  0.077767
Changning  0.035235  0.047082  0.539452  0.076482
Chongming  0.023590  0.028650  0.704424  0.045649
Fengxian   0.051521  0.060437  0.369931  0.095174
Hongkou    0.038789  0.048937  0.507320  0.083353
```

1.2.6 Vegetation and built-up features

Vegetation features and built-up features can be measured by calculating fundamental NDVI and NDBI indices in each administritive area repectively. The Normalized Difference Vegetation Index (NDVI) is a normalized index, using Red and NIR bands to display the amount of vegetation (NASA, 2000). The use of NDVI maximizes the reflectance properties of vegetation by minimizing NIR and maximizing the reflectance in the red wavelength. The measure is used to distinguish vegatation in regions, as more vegataopm will affect the ratio of visible light absorbed and near-infrared light reflected. The formula is as follows:

```
NDVI = (NIR - Red)/(NIR + Red)
```

The output value of this index is between -1.0 and 1.0. Close to 0 represents no vegetation, close to 1 indicates the highest possible density of green leaves, and close to -1 indicates water bodies.

The Normalized Difference Built-up Index (NDBI) uses the NIR and SWIR bands to highlight artificially constructed areas (built-up areas) where there is a typically a higher reflectance in the shortwave infrared region than the near infrared region (Zha et al., 2003). The index is a ratio type that reduces the effects of differences in terrain illumination and atmospheric effects. The formula is as follows:

```
NDBI = (SWIR - NIR) / (SWIR + NIR)
```

Also, the output value of this index is between -1 to 1. Higher values represent built-up areas whereas negative value represent water bodies.

After calculating these two indices, vegetation features and built-up features can be measured by calculating average values of index values within each administritive area.

- Vegetation features

[42]:
```
#identify red and NIR band to each neighbourhood unit in 1984
red_old, nir_old = img_old[2],img_old[3]
# Calculate ndvi, assign 0 to nodata pixels
ndvi_old = [np.where((nir_old[i] + red_old[i])==0, 0,
                    (nir_old[i]-red_old[i])/(nir_old[i] + red_old[i]))
            for i in range(len(poly))]
```

[43]:
```
#identify red and NIR band to each neighbourhood unit in 1984
red_new, nir_new = img_new[2],img_new[3]
# Calculate ndvi, assign 0 to nodata pixels
ndvi_new = list(map(lambda i: np.where((nir_new[i] + red_new[i])==0, 0,
                    (nir_new[i]-red_new[i])/(nir_new[i] + red_new[i])),
            list(range(len(poly)))
    ))
```

[44]:
```
veg_old_var = pd.DataFrame([np.mean(ndvi_old[i]) for i in range(len(poly))],
                       index = poly.Name, columns = ['veg_mean'])
```

[45]:
```
veg_new_var = pd.DataFrame([np.mean(ndvi_new[i]) for i in range(len(poly))],
                       index = poly.Name, columns = ['veg_mean'])
```

- Built-up features

[46]:
```
#identify red and NIR band to each neighbourhood unit in 1984
nir_old, swir_old = img_old[3],img_old[4]
# Calculate ndvi, assign 0 to nodata pixels
ndbi_old = [np.where((nir_old[i] + swir_old[i])==0., 0,
                    (swir_old[i] - nir_old[i])/(nir_old[i] + swir_old[i]))
            for i in range(len(poly))]
```

[47]:
```
#identify red and NIR band to each neighbourhood unit in 1984
nir_new, swir_new = img_new[3],img_new[4]
# Calculate ndvi, assign 0 to nodata pixels
ndbi_new = list(map(lambda i: np.where((nir_new[i] + swir_new[i])==0., 0,
                    (swir_new[i] - nir_new[i])/(nir_new[i] + swir_new[i])),
            list(range(len(poly)))
                   ))
```

[48]:
```
builtup_old_var = pd.DataFrame([np.mean(ndbi_old[i]) for i in range(len(poly))],
                               index = poly.Name, columns = ['builtup_mean'])
```

[49]:
```
builtup_new_var = pd.DataFrame([np.mean(ndbi_new[i]) for i in range(len(poly))],
                               index = poly.Name, columns = ['builtup_mean'])
```

The two new tables we created as shown below contain both vegetation features (NDVI) and builtup features (NDBI), with the mean value of vegetation features and builtup features (two columns) calculated at each administritive division (row).

[50]:
```
veg_built_old = pd.concat([veg_old_var,builtup_old_var], axis = 1)
veg_built_old.head()
```

[50]:
```
           veg_mean  builtup_mean
Name
Baoshan   -0.002218      0.000611
Changning -0.002147      0.000582
Chongming -0.000805      0.000190
Fengxian  -0.007201      0.001499
Hongkou   -0.004648     -0.000313
```

[51]:
```
veg_built_new = pd.concat([veg_new_var,builtup_new_var], axis = 1)
veg_built_new.head()
```

[51]:
```
           veg_mean  builtup_mean
Name
Baoshan   -0.001801      0.001938
Changning -0.001515      0.000774
Chongming -0.000705      0.000318
Fengxian  -0.008185     -0.000408
Hongkou   -0.002057     -0.000277
```

1.2.7 Feature clustering

Now we have four types of features: colour, texture, vegetation and built-up area for Shanghai in 1984 and 2019. These features are the embodiment of urban changes and vary greatly due to rapid urbanisation and development. Therefore, the subsequent task is to identify systematic patterns from these integrated features for analysis of urban changes, such as whether several administritive areas share similar patterns. A clustering method is required within this context to group these geographical division that are similar within each other but diffent between them. Considering the ease of computation and fast implementation, we use generalised and the most popular k-means clustering to identify representative types of neighbourhoods based on multiple features. K-means clustering partitions the data by creating k groups of equal variance, minimising the within-cluster sum of squares (Pedregosa et al., 2011). We can perform K-means using the package scikit-learn, which is a powerful machine learning package for Python.

[52]:
```
#merge all features together
features_old_var = pd.concat([color_old_var,texture_old_var,veg_old_var,
 ↪builtup_old_var], axis = 1)
features_old_var.head()
```

[52]:
```
              h_mean    s_mean    v_mean     h_std     s_std     v_std  \
Name
Baoshan     0.272161  0.052081  0.644148  0.327094  0.072457  0.185309
Changning   0.221412  0.051564  0.659894  0.288368  0.075177  0.174455
Chongming   0.153807  0.017394  0.742309  0.272162  0.035627  0.102347
Fengxian    0.339613  0.112915  0.605758  0.321941  0.122805  0.243621
Hongkou     0.249526  0.063704  0.650725  0.309825  0.087439  0.187805

              h_skew    s_skew    v_skew      LBP0  …      LBP2      LBP3  \
Name                                                …
Baoshan     0.356713  0.090057  0.199446  0.035093  …  0.040705  0.068394
Changning   0.330803  0.092504  0.187627  0.036086  …  0.041956  0.059792
Chongming   0.332916  0.051184  0.120603  0.025822  …  0.021757  0.034058
Fengxian    0.347226  0.144392  0.257670  0.055508  …  0.051230  0.072002
Hongkou     0.347968  0.106194  0.200131  0.042018  …  0.043542  0.059326

                LBP4      LBP5      LBP6      LBP7      LBP8      LBP9  \
Name
Baoshan     0.078389  0.067483  0.040339  0.041101  0.520053  0.066483
Changning   0.060040  0.064422  0.037538  0.043385  0.539416  0.071285
Chongming   0.039580  0.036787  0.024035  0.029158  0.709444  0.049413
Fengxian    0.073767  0.073928  0.052907  0.065099  0.382110  0.106981
Hongkou     0.056759  0.070718  0.039691  0.046490  0.510429  0.080465

            veg_mean  builtup_mean
Name
Baoshan    -0.002218      0.000611
Changning  -0.002147      0.000582
Chongming  -0.000805      0.000190
Fengxian   -0.007201      0.001499
Hongkou    -0.004648     -0.000313

[5 rows x 21 columns]
```

[53]:
```
#merge all features together
features_new_var = pd.concat([color_new_var,texture_new_var,veg_new_var,
 ↪builtup_new_var], axis=1)
features_new_var.head()
```

[53]:
```
              h_mean    s_mean    v_mean     h_std     s_std     v_std  \
Name
Baoshan     0.231070  0.035865  0.638941  0.297847  0.052048  0.180189
```

```
Changning  0.231849  0.031294  0.649237  0.304689  0.048471  0.167002
Chongming  0.157306  0.016473  0.742402  0.282495  0.031843  0.101771
Fengxian   0.295539  0.086197  0.605431  0.302731  0.097695  0.243596
Hongkou    0.239944  0.037582  0.638613  0.303830  0.055047  0.182938

             h_skew    s_skew    v_skew      LBP0  …      LBP2      LBP3  \
Name                                               …
Baoshan    0.336779  0.067866  0.195591  0.043059  …  0.040768  0.058617
Changning  0.344394  0.062974  0.182483  0.042641  …  0.037668  0.051370
Chongming  0.345289  0.043360  0.119800  0.025468  …  0.023332  0.035217
Fengxian   0.329388  0.115702  0.257234  0.051206  …  0.052882  0.081410
Hongkou    0.339615  0.070552  0.197624  0.047032  …  0.042940  0.054392

               LBP4      LBP5      LBP6      LBP7      LBP8      LBP9  \
Name
Baoshan    0.068075  0.056809  0.037410  0.045565  0.524189  0.077767
Changning  0.058422  0.061528  0.035235  0.047082  0.539452  0.076482
Chongming  0.047621  0.036412  0.023590  0.028650  0.704424  0.045649
Fengxian   0.097157  0.079233  0.051521  0.060437  0.369931  0.095174
Hongkou    0.055014  0.068051  0.038789  0.048937  0.507320  0.083353

           veg_mean  builtup_mean
Name
Baoshan   -0.001801      0.001938
Changning -0.001515      0.000774
Chongming -0.000705      0.000318
Fengxian  -0.008185     -0.000408
Hongkou   -0.002057     -0.000277

[5 rows x 21 columns]
```

The above two tables reveal the integrated 21 features across our four sets of image features and their differences at geographical division in magnitude between 1984 and 2019. Since k-means clustering is one of the machine learning algorithms, which generally expect data transformation for preprocessing before fitting the algorithm. We therefore use one of the most popular rescale methods to standadise these features to lie between 0 and 1 based on MinMaxScaler() function in scikit-learn package. The motivation of this method relies on the robustness to very small standard deviation. This preprocess ensures individual features of dataset have the same scale that standard normally distributied.

[54]:
```
#Last preprocessing step before machine learning: data rescaling
min_max_scaler = preprocessing.MinMaxScaler()
np_scaled = min_max_scaler.fit_transform(features_old_var)
oldvar_scale = pd.DataFrame(np_scaled)
oldvar_scale.columns = features_old_var.columns

oldvar_scale.head()
```

[54]:
```
     h_mean    s_mean    v_mean     h_std     s_std     v_std    h_skew  \
0  0.636975  0.359694  0.281144  1.000000  0.422465  0.580121  1.000000
1  0.363843  0.354334  0.396450  0.295018  0.453664  0.504220  0.000000
2  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  0.081547
3  1.000000  0.990530  0.000000  0.906183  1.000000  0.987872  0.633860
4  0.515153  0.480226  0.329302  0.685631  0.594329  0.597572  0.662480

     s_skew    v_skew      LBP0  …     LBP2      LBP3      LBP4      LBP5  \
0  0.417053  0.568198  0.286043  …  0.588494  0.717523  0.837853  0.613055
1  0.443305  0.483028  0.316677  …  0.627346  0.537772  0.441710  0.551933
2  0.000000  0.000000  0.000000  …  0.000000  0.000000  0.000000  0.000000
3  1.000000  0.987806  0.915923  …  0.915397  0.792921  0.738052  0.741784
4  0.590183  0.573137  0.499704  …  0.676601  0.528034  0.370871  0.677669

       LBP6      LBP7      LBP8      LBP9  veg_mean  builtup_mean
0  0.564716  0.332298  0.714937  0.032829  0.647652      0.766995
1  0.467707  0.395850  0.744082  0.042063  0.654899      0.764480
2  0.000000  0.000000  1.000000  0.000000  0.790941      0.730085
3  1.000000  1.000000  0.507310  0.110712  0.142243      0.844884
4  0.542270  0.482237  0.700451  0.059718  0.401200      0.686044

[5 rows x 21 columns]
```

[55]:
```
min_max_scaler = preprocessing.MinMaxScaler()
np_scaled = min_max_scaler.fit_transform(features_new_var)
newvar_scale = pd.DataFrame(np_scaled)
newvar_scale.columns = features_new_var.columns

newvar_scale.head()
```

[55]:
```
     h_mean    s_mean    v_mean     h_std     s_std     v_std    h_skew  \
0  0.442195  0.278123  0.301061  0.463123  0.308221  0.547531  0.504942
1  0.446860  0.212560  0.370616  0.659603  0.254002  0.455458  0.773771
2  0.000000  0.000000  1.000000  0.022307  0.002006  0.000000  0.805338
3  0.828667  1.000000  0.074677  0.603370  1.000000  0.990252  0.244056
4  0.495392  0.302755  0.298842  0.634913  0.353666  0.566728  0.605083

     s_skew    v_skew      LBP0  …     LBP2      LBP3      LBP4      LBP5  \
0  0.338747  0.541837  0.526221  …  0.548142  0.506561  0.410501  0.406220
1  0.271126  0.448129  0.513894  …  0.450686  0.349686  0.216767  0.500209
2  0.000000  0.000000  0.007423  …  0.000000  0.000000  0.000000  0.000000
3  1.000000  0.982533  0.766492  …  0.928936  1.000000  0.994141  0.852811
4  0.375880  0.556374  0.643383  …  0.616412  0.415111  0.148374  0.630101

       LBP6      LBP7      LBP8      LBP9  veg_mean  builtup_mean
0  0.494801  0.499487  0.726866  0.060848  0.856451      1.000000
1  0.416926  0.544291  0.749995  0.058415  0.893850      0.699181
```

```
2  0.000000  0.000000  1.000000  0.000000  1.000000       0.581435
3  1.000000  0.938629  0.493097  0.093827  0.020274       0.393903
4  0.544153  0.599054  0.701302  0.071432  0.822844       0.427538

[5 rows x 21 columns]
```

Above two tables are the results of data transformation in 1984 and 2019. To identify robust and consistent clustering results, we merge them into a single one based their common geographical units (see table below). The column names ended with '_x' and '_y' represent features extracted in 1984 and 2019, resepctively. This table is the one prepared for the final k-mean clustering analysis. The dominant parameter in k-means clustering is the number of clusters (i.e., k), determining the optimal numbers of clusters is therefore becomes a fundanmental issue. We select a direct and popular elbow method as an example to assess the resulting patitions, testing nine different solutions varying k from 2 to 10. Basically, the idea of elbow method is to define clusters to minimise the total intra-cluster variation or total within-cluster sum of square (WSS). The optimal number can be determined by ploting the curve of WSS according to different k clusters and the location of a bend is considered as an indicator of the appropriate number for k.

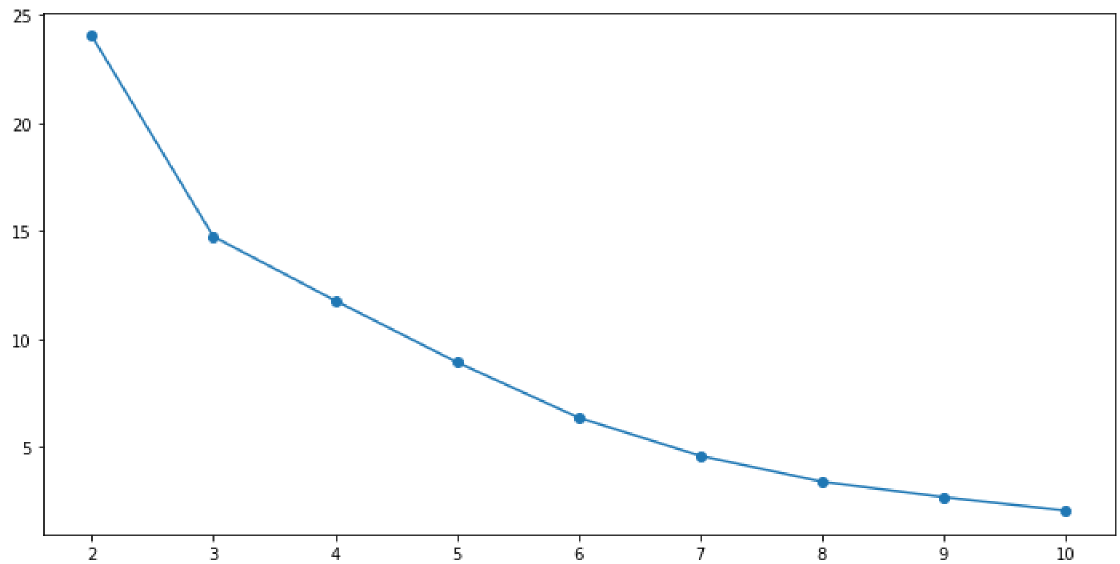
[56]:
```
merged_var = pd.merge(oldvar_scale, newvar_scale, left_index = True,␣
 ↪right_index = True)
merged_var.head()
```

[56]:
```
   h_mean_x  s_mean_x  v_mean_x   h_std_x   s_std_x   v_std_x  h_skew_x  \
0  0.636975  0.359694  0.281144  1.000000  0.422465  0.580121  1.000000
1  0.363843  0.354334  0.396450  0.295018  0.453664  0.504220  0.000000
2  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  0.081547
3  1.000000  0.990530  0.000000  0.906183  1.000000  0.987872  0.633860
4  0.515153  0.480226  0.329302  0.685631  0.594329  0.597572  0.662480

   s_skew_x  v_skew_x    LBP0_x  …    LBP2_y    LBP3_y    LBP4_y    LBP5_y  \
0  0.417053  0.568198  0.286043  …  0.548142  0.506561  0.410501  0.406220
1  0.443305  0.483028  0.316677  …  0.450686  0.349686  0.216767  0.500209
2  0.000000  0.000000  0.000000  …  0.000000  0.000000  0.000000  0.000000
3  1.000000  0.987806  0.915923  …  0.928936  1.000000  0.994141  0.852811
4  0.590183  0.573137  0.499704  …  0.616412  0.415111  0.148374  0.630101

     LBP6_y    LBP7_y    LBP8_y    LBP9_y  veg_mean_y  builtup_mean_y
0  0.494801  0.499487  0.726866  0.060848    0.856451        1.000000
1  0.416926  0.544291  0.749995  0.058415    0.893850        0.699181
2  0.000000  0.000000  1.000000  0.000000    1.000000        0.581435
3  1.000000  0.938629  0.493097  0.093827    0.020274        0.393903
4  0.544153  0.599054  0.701302  0.071432    0.822844        0.427538

[5 rows x 42 columns]
```

[57]:
```
#elbow analysis
cluster_range = range( 2, 11 )
```

```
cluster_errors = []

for num_clusters in cluster_range:
  clusters = KMeans( num_clusters )
  clusters.fit( merged_var)
  cluster_errors.append( clusters.inertia_ )
clusters_df = pd.DataFrame( { "num_clusters":cluster_range, "cluster_errors":␣
↪cluster_errors } )
plt.figure(figsize=(12,6))
plt.plot( clusters_df.num_clusters, clusters_df.cluster_errors, marker = "o" )
```

[57]: [<matplotlib.lines.Line2D at 0x2e51f000320>]



This figure indicates that 6 (i.e. knee in the plot) is the optimal number of k clusters for the features extracted from both years of satellite imagery. The number of 6 is therefore assigned to k to fit the kmeans clustering model, varying labels are subsequently matched to features dataset.

[58]:
```
np.random.seed(0)
k = 6

cls = pd.Series(KMeans(n_clusters=k, max_iter = 1000, n_init = 1000,␣
↪random_state = 24).fit_predict(merged_var))
```

After implementing k-means clustering on our constructed dataset, the label of each cluster is assigned to the last columns of data for further interpretation (as shown below).

[59]:
```
#Assign the each cluster number to the merged data
merged_var = merged_var.assign(lbls=cls)
merged_var.index = features_old_var.index
```

```
merged_var.head()#last columns represent class labels
```

[59]:
```
           h_mean_x  s_mean_x  v_mean_x   h_std_x   s_std_x   v_std_x  \
Name
Baoshan    0.636975  0.359694  0.281144  1.000000  0.422465  0.580121
Changning  0.363843  0.354334  0.396450  0.295018  0.453664  0.504220
Chongming  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000
Fengxian   1.000000  0.990530  0.000000  0.906183  1.000000  0.987872
Hongkou    0.515153  0.480226  0.329302  0.685631  0.594329  0.597572

           h_skew_x  s_skew_x  v_skew_x    LBP0_x  …    LBP3_y    LBP4_y  \
Name                                               …
Baoshan    1.000000  0.417053  0.568198  0.286043  …  0.506561  0.410501
Changning  0.000000  0.443305  0.483028  0.316677  …  0.349686  0.216767
Chongming  0.081547  0.000000  0.000000  0.000000  …  0.000000  0.000000
Fengxian   0.633860  1.000000  0.987806  0.915923  …  1.000000  0.994141
Hongkou    0.662480  0.590183  0.573137  0.499704  …  0.415111  0.148374

             LBP5_y    LBP6_y    LBP7_y    LBP8_y    LBP9_y  veg_mean_y  \
Name
Baoshan    0.406220  0.494801  0.499487  0.726866  0.060848    0.856451
Changning  0.500209  0.416926  0.544291  0.749995  0.058415    0.893850
Chongming  0.000000  0.000000  0.000000  1.000000  0.000000    1.000000
Fengxian   0.852811  1.000000  0.938629  0.493097  0.093827    0.020274
Hongkou    0.630101  0.544153  0.599054  0.701302  0.071432    0.822844

           builtup_mean_y  lbls
Name
Baoshan          1.000000     1
Changning        0.699181     1
Chongming        0.581435     2
Fengxian         0.393903     3
Hongkou          0.427538     1

[5 rows x 43 columns]
```

1.2.8 Interpretation

To understand the analysis result, the mean of each feature across each cluster can be calculated to uncover the feature differences among clusters. A categorical barplot shown below presents how the average of all features changed between 1984 and 2019. Besides, a choropleth map is created to visualise the spatial distribution of catogories/clusters by varing colours.

[60]:
```
#calculate the mean of features for each class
k6_mean = merged_var.groupby('lbls').mean()
k6_mean
```

[60]:
```
      h_mean_x  s_mean_x  v_mean_x   h_std_x   s_std_x   v_std_x  h_skew_x  \
lbls
0     0.803863  0.749195  0.146109  0.895068  0.749907  0.834330  0.633322
1     0.426195  0.381664  0.384955  0.543494  0.481686  0.523486  0.450601
2     0.113097  0.088470  0.837305  0.210939  0.118478  0.127151  0.316506
3     0.949856  0.995265  0.004559  0.876415  0.991048  0.993936  0.676387
4     0.721478  0.979475  0.114017  0.567528  0.998762  0.933619  0.244557
5     0.471719  0.427622  0.373358  0.634773  0.554207  0.559062  0.554391

      s_skew_x  v_skew_x   LBP0_x  …   LBP2_y    LBP3_y    LBP4_y  \
lbls                               …
0     0.720249  0.823005  0.454389  …  0.760955  0.841492  0.838157
1     0.478586  0.504149  0.382849  …  0.510348  0.446913  0.309060
2     0.107534  0.124469  0.070301  …  0.140384  0.172060  0.158486
3     0.975688  0.993903  0.882717  …  0.964468  0.940470  0.808656
4     0.971796  0.925791  1.000000  …  0.907557  0.660840  0.487015
5     0.560426  0.534555  0.426212  …  0.570189  0.416572  0.116072

        LBP5_y    LBP6_y    LBP7_y    LBP8_y    LBP9_y  veg_mean_y  \
lbls
0     0.722830  0.794093  0.652172  0.600350  0.070401    0.714016
1     0.470514  0.466278  0.466282  0.748782  0.053443    0.852035
2     0.147885  0.115339  0.054215  0.939642  0.006619    0.959421
3     0.863404  0.987573  0.969315  0.488089  0.104292    0.010137
4     1.000000  0.950033  0.913027  0.521429  0.107485    0.713309
5     0.221324  0.445842  0.718360  0.000000  1.000000    0.895112

      builtup_mean_y
lbls
0           0.835832
1           0.622854
2           0.572280
3           0.196952
4           0.986139
5           0.568988

[6 rows x 42 columns]
```

The above table displays the mean values of all features in two years at varying groups. For more interpretability, a few data munging steps are required to generate visual representations.

[61]:
```python
#Rearrange our data in a way that every row is one feature in a class
k6_mean = k6_mean.stack()
k6_mean.head()
```

[61]:
```
lbls
0     h_mean_x    0.803863
```

```
      s_mean_x    0.749195
      v_mean_x    0.146109
      h_std_x     0.895068
      s_std_x     0.749907
dtype: float64
```

[62]:
```
#convert multi-indices into single index
k6_mean = k6_mean.reset_index()
#renmae the columns
k6_mean = k6_mean.rename(columns = {'lbls': 'Class','level_1': 'Features', 0:␣
↪'Values'})
k6_mean.head()
```

[62]:

	Class	Features	Values
0	0	h_mean_x	0.803863
1	0	s_mean_x	0.749195
2	0	v_mean_x	0.146109
3	0	h_std_x	0.895068
4	0	s_std_x	0.749907

[63]:
```
#rename feature names in Feature column
old  = k6_mean.loc[k6_mean['Features'].str.contains('x') ==  True, :]
new  = k6_mean.loc[k6_mean['Features'].str.contains('y') ==  True, :]
#add a new column to represent time
old = old.assign(Time = 1984)
new = new.assign(Time = 2019)
#remove '_x' and '_y' in the table to make feature names for both years are the␣
↪same
old['Features'] = old['Features'].str.replace('_x', '')
new['Features'] = new['Features'].str.replace('_y', '')
```

[64]:
```
#create a new dataframe to store the mean of each feature each cluster with time
data = pd.concat([old,new])
data.head()
```

[64]:

	Class	Features	Values	Time
0	0	h_mean	0.803863	1984
1	0	s_mean	0.749195	1984
2	0	v_mean	0.146109	1984
3	0	h_std	0.895068	1984
4	0	s_std	0.749907	1984

The above table reveals different categorical information, with each row represents the number of class, the feature name, the mean value of the feature and the year when the feature is extracted. We can then visualise this table in the following barplot to understand the pattern from image features.

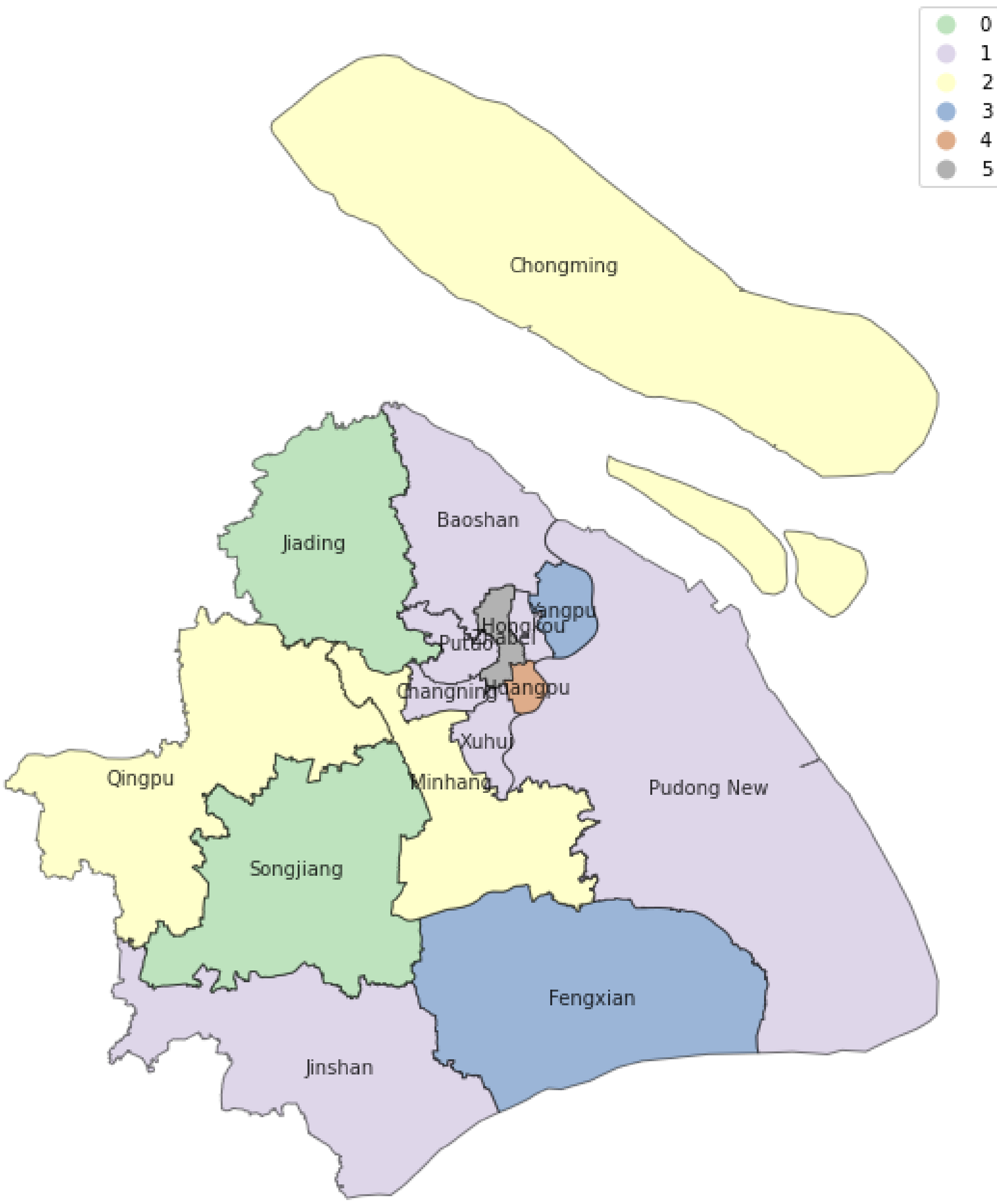
```
[65]: #visualise the distribution of mean values by features, class and time
sns.catplot( data = data, x = 'Features', y = 'Values',row = 'Class', hue =␣
 ↪'Time',kind = 'bar',\
             aspect = 5, height = 3, palette = 'Accent')
```

[65]: <seaborn.axisgrid.FacetGrid at 0x2e5288d5cf8>


```
[66]: #plot clustering results for two different years
f, ax = plt.subplots(1, figsize=(10, 12))
```

```
#plot cluster results
poly = poly.drop('coords', axis = 1)
poly.assign(lbls=cls)\
     .plot(column='lbls', categorical=True, linewidth=1, alpha=0.5,␣
 ↪ax=ax,legend = True,cmap = 'Accent', edgecolor = 'black')
#add labels for geographical units
poly['coords']=poly['geometry'].apply(lambda x:x.representative_point().coords[:
 ↪])
poly['coords']=[coords[0] for coords in poly['coords']]
for idx, row in poly.iterrows():
    ax.annotate(text=row['Name'],xy=row['coords'],va='center',ha='center',alpha␣
 ↪= 0.8, fontsize = 10)

#remove axes and set aspect ratio so that the data units are the same in every␣
 ↪direction
ax.axis('off')
ax.axis('equal')
```

[66]: (290053.0696196473, 407301.6741094636, 3389866.639388826, 3533566.430983904)



From the above two figures(barplot and choropleth map) we can see a few striking differences across clusters, or classes. For class 4, only one administritive area (i.e. Huangpu area/)is grouped, displayed in the middle of noth-east areas. The mean values for this class are mostly high in both years except a couple of features such as v_mean, LBP4 and LBP9 features. The brightness(v_mean) for this area is highly low and it became completely black over the time. H_mean value is high in both years, demonstrating that the dominating colour is blue, which represent water. This corresponds to famous area of The Bund, with its river skyline, which is part of this polygon. The vegetation built-up features indicate that this area has experienced a remarkable change, from more vegetation and few buildings to less vegetation and completely constructed/urbanisation.

Class 0 and class 1 are relatively consistent compared to other classes, implying that the urban areas in purple and yellow colours almost remained unchanged during the past 35 years. Besides, these two classes have similar transformation such as more vegetation coverage and less buildings for the current year of 2019. However, class 0 has more brightness and more green colour based on v_mean, h_mean and veg_mean features, and class 1 has higher h_mean, h_std, h_skew and built-up_mean, implying these two areas have water covered and were highly constructed.

Class 2 distributed at north and west areas in the map, which is extremly diverse and unique among all categories. It has the highest brightness features and LBP8 texture features, while the rest mean values of colour and texture features are highly low, especially for LBP9 where almost zero values in both years. The values for h_mean, s_mean and v_mean display that the primary colour for these areas is red with little gray and much brightness, representing that these areas include more bare ground or soil and thus probably rural areas. Adversely, class 5 has zero values for LBP8 but highest values for LBP9 in both years. It contains only one administritive area (i.e. Zhabei area/), surrounded by class 4 and class 0. Similarly, the area in class 5 has more vegetation but slightly less built-up areas over the past years. Class 3 contains two areas distributed at the south and surrounded by class 1 from the map. The feature values in class 5 are mostly extremely high, while the veg_mean and built-up_mean for current year are the least, thus indicating that these areas have more water over the time.

1.2.9 Conclusion

Urbanisation has significantly changed the interaction between humans and the surrounding environment, which poses new challenges in a multitude of fields including construction and city planning, hazard mitigation or disease control. It is essential to quantify and assess urbanisation over time to enable policy makers and planners to make informed decisions about future urban changes.This notebook shows the potential of open source satellite imagery to exploring urban changes and proposes a simple method framework for automatic data collection and features extraction to determine urbanisation over time using Python as a tool.

1.2.10 References

Barsi, J. A., Lee, K., Kvaran, G., Markham, B. L., & Pedelty, J. A. (2014a). The spectral response of the Landsat-8 operational land imager. Remote Sensing, 6(10), 10232-10251.

Barsi, J. A., Schott, J. R., Hook, S. J., Raqueno, N. G., Markham, B. L., & Radocinski, R. G. (2014b). Landsat-8 thermal infrared sensor (TIRS) vicarious radiometric calibration. Remote Sensing, 6(11), 11607-11626.

Clark, D. (2004). Urban world/global city. Routledge.

Glaeser, E. and Henderson, J.V., 2017. Urban economics for the developing world: An introduction. Journal of Urban Economics, 98, pp.1-5.

Knight, E. J., & Kvaran, G. (2014). Landsat-8 operational land imager design, characterization and performance. Remote Sensing, 6(11), 10286-10305.

Ministry of Civial Affairs of the People's Republic of China (2018). Change of administrative divisions at or above the county level. [online] Available at: http://202.108.98.30/description?dcpid=1 [Accessed 10 Oct. 2019]

Roy, D. P., Wulder, M. A., Loveland, T. R., Woodcock, C. E., Allen, R. G., Anderson, M. C., ... & Scambos, T. A. (2014). Landsat-8: Science and product vision for terrestrial global change research. Remote sensing of Environment, 145, 154-172.

NASA, 2019. Landsat Science. [online] Available at: https://landsat.gsfc.nasa.gov/ [Accessed 10 Sep. 2019]

NASA, 2000. Normalized Difference Vegetation Index (NDVI). [online] Available at: https://earthobservatory.nasa.gov/features/MeasuringVegetation/measuring_vegetation_2.php [Accessed 30 Oct. 2019]

Noah Keen, "Color Moments", February 2005.

Ojala, T., Pietikäinen, M. and Harwood, D. (1996), A Comparative Study of Texture Measures with Classification Based on Feature Distributions. Pattern Recognition 19(3):51-59.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.

United Nations, Department of Economic and Social Affairs, Population Division (2019). World Urbanization Prospects 2018: Highlights (ST/ESA/SER.A/421).

Zha, Y., J. Gao, and S. Ni (2003). "Use of Normalized Difference Built-Up Index in Automatically Mapping Urban Areas from TM Imagery." International Journal of Remote Sensing 24, 3, 583-594.

[ ]: