Geospatial analysis in Scala

Roxana Tesileanu

roxana.te@web.de
INCDS, Romania

October 2017

Contents

1 Introduction

The aim of the present paper is to investigate the use of some of the existing libraries for geospatial analysis available in Scala, the Geospatial Data Abstraction Library (GDAL) and Geotrellis, for performing the main geospatial analysis tasks: manipulating vector and raster data (geoprocessing) and geostatistics. The former task will be approched using GDAL and the later using Geotrellis.

2 Geoprocessing using GDAL

Using a programming language for geospatial analysis allows you to customize your analyses instead of being limited to what the software user interface allows. This is one of the most important advantages of open source software [3]. The GDAL library is one of the open source libraries used in this work. It was written in C and C++ and has bindings for several languages (Java, Perl and Python).
In order to use GDAL, you need to install it on your machine, and for its import in Scala you need to install its Java bindings along with it. For installation details you can look at the GDAL homepage http://www.gdal.org/, download GDAL and follow the instructions for building from source, which might not be an easy task, depending on your operating system. Thanks to the efforts of the UbuntuGIS team (https://wiki.ubuntu.com/UbuntuGIS), on Ubuntu, the installation procedure of GDAL and its bindings is done rapidly. Firstly, you need to add the ubuntugis PPA, which offers the official stable UbuntuGIS packages, to your system (https://launchpad.net/ ubuntugis/+archive/ubuntu/ppa). This is done with the commands:

sudo add-apt-repository ppa:ubuntugis/ppa
sudo apt-get update.

Next, you install GDAL on your machine with the commands [4] [2] (http://www.sarasafavi.com/installing-gdalogr-on-ubuntu.html, https://packages.ubuntu.com/source/trusty/gdal):

sudo apt-get install libproj-dev, gdal-bin, libgdal-dev, libgdal-doc
sudo apt-get update.

Finally, you add the Java bindings to your GDAL package (https://launchpad.net/ubuntu/+source/proj):

sudo apt-get install libgdal-java, libproj-java.

In order to import GDAL in Scala, you have to add its jar to the project's classpath. An easy way of managing dependencies of a Scala project is to use SBT (for further details see [5]). In this way you can take advantage of the most convenient way to place the gdal jar to the project's classpath, namely, to place a copy of it into the lib directory of the Scala project, now that the actual installation has already taken place.

The following subsections will offer a background in geoprocessing, starting with manipulating vector data (reading and writing files of different vector data formats and performing overlay and proximity analyses), and continuing with manipulating raster data (reading and writing files of different raster data formats, resizing pixels, performing moving window analyses and map algebra).

2.1 Types of spatial data

Spatial data are divided in two categories: vector data and raster data. Vector data provide information about distinct features in space, i.e. different distinct items of interest, and are made up of points, lines and polygons [3]. The features of interest could be for example:

- roads, rivers, road networks, hidrological networks, country boundaries, city boundaries as examples of features represented by lines,
- mountain peaks, volcano peaks, weather stations, restaurants, as examples of features represented by points, and
- lakes, oceans, ownership status as examples of features represented by polygons.

Features have attributes attached to them such as the name of the individual observations (for example the wheather stations's name) and other recorded variables (like for example different concentrations of air pollutants, temperature or wind regime for each individual weather station). As it can be noticed, the multiple attributes which can be attached to features, can be of different types, and they actually represent different types of recorded variables (they might be dicrete or continuous numerical variables or categorical variables).

On the other hand, raster data provide information about characteristics of interest which take the form of a continuum like gradients, with no distinct boundaries. They are represented as two- or three-dimensional arrays of data values which form grids of values [3]. Because they can cope well with gradients, they capture local variation more easily than vector geometries, and are used in digital elevation models (DEMs). Also because the data source is pixel-based (e.g. aerial photos, satellite imagery) they can be used in vegetation mapping.

2.2 Reading vector data

The main objective of vector data analysis is to investigate relationships between features, by overlapping them on another or measuring distances between them [3]. A typical example for vector analyses is the investigation of GPS-collared wildlife to see the direction of travel, distances covered and how they interact with man-made features like roads [3].
In order to perform such vector-based analyses, we need to be able to read, edit and write vector data. This kind of functionality is offered by the OGR Simple Features Library for geoprocessing vector data, which is included in GDAL.

At this point it is noted that the Scala code relating to using the GDAL functionality introduced in this document has its origins in the Python code written by Chris Garrard in her book "Geoprocessing with Python" (2016). The main reason for the transition towards using Scala for geospatial analysis is the use of

Scala's functional nature for the further processing of geodata, by using higher-order functions.

There are many different types of vector data formats. Among the most widely used ones are: the ESRI shapefile, the GeoJSON file, or the SpatiaLite or PostGIS databases. The ESRI shapefile format requires a minimum of three binary files, each of which serves a different purpose: geometry information is stored in .shp and .shx files, and attribute values are stored in a .dbf file. You need to make sure they are all grouped in the same folder, because they work together [3]. The GeoJSON format is used mainly for web-mapping applications and is a plain text file which can be easily examined. The GeoJSON format consists of a single file. Vector data can also be stored in relational databases with spatial extensions. The most widely used spatial extensions are SpatiaLite (for SQLite databases) and PostGIS (for PostgreSQL). You can check other vector data formats supported by GDAL at http://www.gdal.org/ogr_formats.html.

The OGR package of GDAL contains the classes used for geoprocessing vector data. The OGR Java Application Programming Interface (API) [1] (http://gdal.org/java/) lists them all. Among them there are the classes: Driver, DataSource, Layer, Geometry and Feature. In order to handle vector geodata with OGR, we need to understand how the geospatial information is organized in OGR. The spatial vector data is stored in a data source (for example a shapefile, a GeoJSON file, or a SpatiaLite or PostGIS database). This data source object can have one or more layers, one for each dataset contained in the data source. Many vector formats, such as shapefiles can only contain one dataset, thus have one layer, but others like SpatiaLite can contain multiple datasets, thus have multiple layers. Each layer contains a collection of features, which holds the geometries (like for example points, lines, polygons) and their attributes [3].

The first step in accessing any vector data is to open the data source. For this you need to use a driver specific to the data format. Each vector data format has its own driver, which is used to read and write a particular format [3]. In order to make the configured OGR drivers available we call the RegisterAll() function at the beginning of the analysis. The RegisterAll() function is placed in the package OGR of the GDAL library, in the class ogr, so we need to call it using org.gdal.ogr.ogr.RegisterAll() (or, we can also import the class to save typing, but it remains unclear where the function resides inside GDAL).

Code snippet 1: accessing a shapefile using OGR in Scala You can find a shapefile on the internet at one of the sources indicated under http://gisgeography.com/best-free-gis-data-sources-raster-vector/ and try the following snippet with your shapefile using the Scala REPL in SBT while you are in the Scala project's directory:

```
org.gdal.ogr.ogr.RegisterAll()
```

```
val dataSource = org.gdal.ogr.ogr.Open("example.shp")
dataSource: org.gdal.org.DataSource = org.gdal.ogr.DataSource@305c6b70

dataSource.GetLayerCount
res1: Int = 1

val lyr = dataSource.GetLayer(0)
lyr: org.gdal.ogr.Layer = org.gdal.ogr.Layer@305ca330

lyr.GetName
res2: String = example

lyr.GetFeatureCount
res3: Long = 927

val feat0 = lyr.GetFeature(0)
feat0 : org.gdal.ogr.Feature = org.gdal.ogr.Feature@3164b9a0

dataSource.delete()
```

Code snippet 1: explanation In order to access the vector geodata, we make the OGR drivers available with RegisterAll(). Then, we create a variable called dataSource in which we store the DataSource object. We obtain it by opening the shapefile (or any other OGR file format) with Open(). We check how many layers (i.e. datasets) are available in the DataSource object by calling GetLayerCount on it. Further, we retrieve the first layer (which has the index 0), with GetLayer(0), obtaining a variable called lyr in which we store it. You can check its name or the number of features it contains, with GetName or GetFeatureCount. You can see the available functions on the lyr with the help of autocompletion. Autocompletion works for example by typing lyr. followed by a <TAB>. This lists all the methods available for that object. We continue, by retrieving the first feature (which has the index 0) of the layer called lyr with GetFeature(0). You can check with autocompletion the methods available on it. At the end of the code snippet we close the data source.

Geodata would actually be "normal" data without georeferencing. In the next subsection, we learn how to deal with spatial reference systems of already georeferenced features, and how to construct our own geometries and features and how to georeference them.

2.3 Georeferencing data

- investigate georeferenced data
- construct new geometries and features

- georeference new features

In order to be able to locate some coordinates on a map, you need to know what spatial reference system is used for the coordinates and what spatial reference system uses the map. If they are not the same, you need to perform transformations from one spatial reference system to another. Georeferencing the data means adding the information regarding the spatial reference system used when defining the features.

A spatial reference system is made of three components [3]:

- a coordinate system,
- a datum and,
- a projection.

The set of coordinates is provided by a coordinate system, the datum specifies the model used to represent the curvature of the earth and a projection is used to transform the three-dimensional globe to a two-dimensional map. A set of coordinates is represented as set of two pieces of information: the latitude and the longitude. Positive latitude values are north of the equator, and positive longitudes are east of the prime meridian. Multiple methods exist for specifying latitude and longitude coordinates: decimal degrees (DD), degrees decimal minutes (DM) and degrees minutes seconds (DMS) [3]. But, if you don't specify the datum used, the same set of latitude and longitude coordinates can refer to slightly different locations, because different datums represent different ellipsoids of different shapes. One of the most widely used datums is the World Geodetic System, last revised in 1984. This datum, also called WGS84, is also used by the Global Positioning System (GPS). WGS84 has a global coverage. But most datums model the curvature of the earth in a more localized area (a continent or a country). A datum designed for an area will not work well elsewhere [3]. Sometimes the difference between two datums can be of hundreds of meters for the same set of coordinates.

Projections convert the coordinates of a point on the globe to the coordinates of a point on a two-dimensional plane. In doing so, it creates distortions. The type of distortion depends on how the conversion is done. If you would like to preserve local shapes, you should use conformal projections, like for example Universal Transverse Mercator (UTM) [3]. To keep the amount of area the same, you should use equal-area projections, like for example the Lambert equal-area or Gall-Peters projections. Thus, sometimes using geographic coordinates (lat/lon), is not appropriate, depending on the aims of your analysis, and you need to choose a projection instead. Also, note that projections are not tied to specific datums, so knowing the projection of your data is not enough. You also have to know the datum used.

You can search for spatial reference systems on the spatialreference.org website under http://spatialreference.org/.

2.3.1 Spatial reference systems in OGR

OGR provides ways of storing and converting the information regarding the spatial reference system (SRS) used for vector data in its package called OGR Spatial Reference (OSR). You can find out the SRS used by a georeferenced layer, by calling the GetSpatialRef function on it. If the layer is not georeferenced it returns null.

```
lyr.GetSpatialRef
res10: org.gdal.osr.SpatialReference =
GEOGCS["GCS_WGS_1984",
  DATUM["WGS_1984",
    SPHEROID["WGS_84",6378137,298.257223563]],
  PRIMEM["Greenwich",0],
  UNIT["Degree",0.017453292519943295],
  AUTHORITY["EPSG","4326"]]


lyr.GetSpatialRef
res14: org.gdal.osr.SpatialReference =
PROJCS["ETRS_1989_LAEA",
  GEOGCS["GCS_ETRS_1989",
    DATUM["European_Terrestrial_Reference_System_1989",
      SPHEROID["GRS_1980",6378137.0,298.257222101]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433]],
  PROJECTION["Lambert_Azimuthal_Equal_Area"],
  PARAMETER["False_Easting",4321000.0],
  PARAMETER["False_Northing",3210000.0],
  PARAMETER["longitude_of_center",10.0],
  PARAMETER["latitude_of_center",52.0],
  UNIT["Meter",1.0]]
```

The first spatial reference from above is not a projected SRS, because it has a GEOCS entry only, without a PROJCS one. But, we can still see that the datum used is WGS1984, the spheroid used is WGS84, the unit used for the set of coordinates is degree and the authority giving the ID code is EPSG (short for European Petroleum Survey Group). The second spatial reference from above is a projected one.

Note

This document is "under construction". The current version is available on my GitHub profile under the multivariate_analyses project repository: https://github.com/RoxanaTesileanu/multivariate_analyses/blob/master/literature_analysis/geospatial_scala/geospatial_scala.pdf.

References

[1] GDAL Java API.

[2] UbuntuGIS.

[3] Chris Garrard. *Geoprocessing with Python*. Manning Publications Co., Shelter Island, 2016.

[4] Sara Safavi. Installing GDAL/OGR on Ubuntu, 2015.

[5] Roxana Tesileanu. Using Linux as a development platform for Scala projects, 2017.