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 how the Geospatial Data Abstraction Library (GDAL) [1] can be used in Scala, for performing the main geospatial analysis tasks: manipulating vector and raster data (geoprocessing) and geospatial data analysis.

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 [2].

This work uses the GDAL open source library [1] developed at the Open Source Geospatial Foundation (OSGeo) www.osgeo.org. GDAL 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 [3] [4] (http://www .sarasafavi.com/installing-gdalogr-on-ubuntu.html, https://packages.ubuntu.co m/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.ne t/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 [2]. 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 [2]. 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 [2]. 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 [2].
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 [2]. 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 [2].

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 [2]. 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 [2]:

- 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) [2]. 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 [2]. 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) [2]. 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 epsg.io website under http://epsg.io/. The SRSs for Romania for example can be found under the link: http://epsg.io/?q=Romania.

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. It has a PROJCS entry, the projection used is the LAEA (Lambert Azimuthal Equal-Area projection). The datum is ETRS 1989 (European Terrestrial Reference System 1989). The SRS is thus the ETRS1989/LAEA. The unit of measure is 1.0 m.

The function GetSpatialRef called on a layer returns a SpatialReference object. Georeferenced data already have such an object defined, if not the GetSpatialRef function returns null. In order to create a SpatialReference object for your layers and geometries you need to firstly create an empty SpatialReference object, then you have to import into the empty SpatialReference object the information on the SRS you want to use, turning it into a valid SpatialReference object.

Code snippet 2: creating a SpatialReference object using OGR in Scala In this code I will use the ETRS1989/LAEA, the Pulkovo1942(58)/ Stereo70 and the WGS84/Pseudo-Mercator SRS. The first one is the SRS for all Europe and it is used for statistical mapping at all scales and other purposes where true area representation is required (http://spatialreference.org/ref/epsg/ 3035/), the second SRS is for Romania and is used in large and medium scale topographic mapping and engineering surveys (http://spatialreference.org/ref/eps g/3844/), and the third is used for rendering maps in GoogleMaps, OpenStreetMap, Bing a.o. (http://epsg.io/3857). Google Earth uses WGS84 with geographic coordinates (lat/lon), unprojected, with EPSG code 4326 (https://gis.s tackexchange.com) (see the first SRS example at page 7, section 2.3.1).

```
val newSR = new org.gdal.osr.SpatialReference()
newSR: org.gdal.osr.SpatialReference =

newSR.ImportFromEPSG(3035)
res1: Int = 0

newSR
res2: org.gdal.osr.SpatialReference =
PROJCS["ETRS89 / LAEA Europe",
  GEOGCS["ETRS89",
    DATUM["European_Terrestrial_Reference_System_1989",
      SPHEROID["GRS 1980",6378137,298.257222101,
        AUTHORITY["EPSG","7019"]],
      TOWGS84[0,0,0,0,0,0,0],
        AUTHORITY["EPSG","6258"]],
    PRIMEM["Greenwich",0],
        AUTHORITY["EPSG","8901"]],
```

```
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4258"]],
  PROJECTION["Lambert_Azimuthal_Equal_Area"],
  PARAMETER["latitude_of_center",52],
  PARAMETER["longitude_of_center",10],
  PARAMETER["false_easting",4321000],
  PARAMETER["false_northing",3210000],
  UNIT["metre",1,
    AUTHORITY["EPSG","9001"]],
  AUTHORITY["EPSG","30...

val newSR2 = new ogr.gdal.osr.SpatialReference()
newSR2: org.gdal.osr.SpatialReference =

newSR2.ImportFromProj4("+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000
+y_0=3210000 +ellps=GRS80 +units=m +no_defs ")
res3: Int = 0

newSR2
res4: org.gdal.osr.SpatialReference=
PROJCS["unnamed",
  GEOGCS["GRS 1980(IUGG, 1980)",
    DATUM["unknown",
      SPHEROID["GRS80",6378137,298.257222101],
      TOWGS84[0,0,0,0,0,0,0]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433]],
  PROJECTION["Lambert_Azimuthal_Equal_Area"],
  PARAMETER["latitude_of_center",52],
  PARAMETER["longitude_of_center",10],
  PARAMETER["false_easting",4321000],
  PARAMETER["false_northing",3210000],
  UNIT["Meter",1]]

val newSR3 = new org.gdal.osr.SpatialReference()
newSR3: org.gdal.osr.SpatialReference =

newSR3.ImportFromWkt("""PROJCS["ETRS89 / ETRS-LAEA",
|     GEOGCS["ETRS89",
|       DATUM["European_Terrestrial_Reference_System_1989",
|         SPHEROID["GRS 1980",6378137,298.257222101,
|           AUTHORITY["EPSG","7019"]],
|         AUTHORITY["EPSG","6258"]],
|       PRIMEM["Greenwich",0,
|         AUTHORITY["EPSG","8901"]],
```

```
|       UNIT["degree",0.01745329251994328,
|         AUTHORITY["EPSG","9122"]],
|       AUTHORITY["EPSG","4258"]],
|     UNIT["metre",1,
|       AUTHORITY["EPSG","9001"]],
|     PROJECTION["Lambert_Azimuthal_Equal_Area"],
|     PARAMETER["latitude_of_center",52],
|     PARAMETER["longitude_of_center",10],
|     PARAMETER["false_easting",4321000],
|     PARAMETER["false_northing",3210000],
|     AUTHORITY["EPSG","3035"],
|     AXIS["X",EAST],
|     AXIS["Y",NORTH]]""")
res9: Int = 0


newSR3
res10: org.gdal.osr.SpatialReference =
PROJCS["ETRS89 / ETRS-LAEA",
  GEOGCS["ETRS89",
    DATUM["European_Terrestrial_Reference_System_1989",
      SPHEROID["GRS 1980",6378137,298.257222101,
        AUTHORITY["EPSG","7019"]],
      AUTHORITY["EPSG","6258"]],
    PRIMEM["Greenwich",0,
      AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,
      AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4258"]],
  UNIT["metre",1,
    AUTHORITY["EPSG","9001"]],
  PROJECTION["Lambert_Azimuthal_Equal_Area"],
  PARAMETER["latitude_of_center",52],
  PARAMETER["longitude_of_center",10],
  PARAMETER["false_easting",4321000],
  PARAMETER["false_northing",3210000],
  AUTHORITY["EPSG","3035"],
  AXIS["X",EAST],
  AXIS["...

val roSR1 = new org.gdal.osr.SpatialReference()
roSR1: org.gdal.osr.SpatialReference =

roSR1.ImportFromEPSG(3844)
res11: Int = 0
```

```
roSR1
res12: org.gdal.osr.SpatialReference =
PROJCS["Pulkovo 1942(58) / Stereo70",
  GEOGCS["Pulkovo 1942(58)",
    DATUM["Pulkovo_1942_58",
      SPHEROID["Krassowsky 1940",6378245,298.3,
        AUTHORITY["EPSG","7024"]],
      TOWGS84[2.329,-147.042,-92.08,0.309,-0.325,-0.497,5.69],
      AUTHORITY["EPSG","6179"]],
    PRIMEM["Greenwich",0,
      AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
      AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4179"]],
  PROJECTION["Oblique_Stereographic"],
  PARAMETER["latitude_of_origin",46],
  PARAMETER["central_meridian",25],
  PARAMETER["scale_factor",0.99975],
  PARAMETER["false_easting",500000],
  PARAMETER["false_northing",500000],
  UNIT["metre",1,
  A...

val roSR2 = new org.gdal.osr.SpatialReference()
roSR2: org.gdal.osr.SpatialReference =

roSR2.ImportFromWkt("""PROJCS["Pulkovo 1942(58) / Stereo70",
|  GEOGCS["Pulkovo 1942(58)",
|    DATUM["Pulkovo 1942(58)",
|      SPHEROID["Krassowsky 1940",6378245.0,298.3,
|        AUTHORITY["EPSG","7024"]],
|      TOWGS84[33.4,-146.6,-76.3,-0.359,-0.053,0.844,-0.17326243724756094],
|      AUTHORITY["EPSG","6179"]],
|    PRIMEM["Greenwich",0.0,
|      AUTHORITY["EPSG","8901"]],
|    UNIT["degree",0.017453292519943295],
|    AXIS["Geodetic latitude",NORTH],
|    AXIS["Geodetic longitude",EAST],
|    AUTHORITY["EPSG","4179"]],
|  PROJECTION["Oblique Stereographic",
|    AUTHORITY["EPSG","9809"]],
|  PARAMETER["central_meridian",25.0],
|  PARAMETER["latitude_of_origin",46.0],
|  PARAMETER["scale_factor",0.99975],
|  PARAMETER["false_easting",500000.0],
|  PARAMETER["false_northing",500000.0],
```

```
|   UNIT["m",1.0],
|   AXIS["Northing",NORTH],
|   AXIS["Easting",EAST],
|   AUTHORITY["EPSG","3844"]]""")
res16: Int = 0

roSR2
res17: org.gdal.osr.SpatialReference =
PROJCS["Pulkovo 1942(58) / Stereo70",
  GEOGCS["Pulkovo 1942(58)",
    DATUM["Pulkovo 1942(58)",
      SPHEROID["Krassowsky 1940",6378245.0,298.3,
        AUTHORITY["EPSG","7024"]],
      TOWGS84[33.4,-146.6,-76.3,-0.359,-0.053,0.844,-0.17326243724756094],
      AUTHORITY["EPSG","6179"]],
    PRIMEM["Greenwich",0.0,
      AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.017453292519943295],
    AXIS["Geodetic latitude",NORTH],
    AXIS["Geodetic longitude",EAST],
    AUTHORITY["EPSG","4179"]],
  PROJECTION["Oblique Stereographic",
    AUTHORITY["EPSG","9809"]],
  PARAMETER["central_meridian",25.0],
  PARAMETER["latitude_of_origin",46.0],
  PARAMETER["scale_factor",0.99975],
    PAR...

val googleMapsSR = new org.gdal.osr.SpatialReference()
googleSR: org.gdal.osr.SpatialReference =

googleMapsSR.ImportFromEPSG(3857)
res18: Int = 0

googleMapsSR
res19: org.gdal.osr.SpatialReference =
PROJCS["WGS 84 / Pseudo-Mercator",
  GEOGCS["WGS 84",
    DATUM["WGS_1984",
      SPHEROID["WGS 84",6378137,298.257223563,
        AUTHORITY["EPSG","7030"]],
      AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
      AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
      AUTHORITY["EPSG","9122"]],
```

```
    AUTHORITY["EPSG","4326"]],
  PROJECTION["Mercator_1SP"],
  PARAMETER["central_meridian",0],
  PARAMETER["scale_factor",1],
  PARAMETER["false_easting",0],
  PARAMETER["false_northing",0],
  UNIT["metre",1,
    AUTHORITY["EPSG","9001"]],
  AXIS["X",EAST],
  AXIS["Y",NORTH],
  EXTENSION["PROJ4","+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0
+lon_0=0.0 +x_0=0.0 +y...
```

Code snippet 2: Explanation In code snippet 2 we've created new empty SpatialReference objects, in which we've imported information on the used SRS using more sources: the EPSG code, the PROJ4 string, and the WKT string. There are also more ways of importing. You can inspect them with autocompletion on an empty SpatialReference object.

In this section we've seen it is important to know the SRS used for your geodata. You can create a SRS using an empty SpatialReference object and importing the information needed using its EPSG code, WKT or PROJ4 string. The important fact to know, if your data comes from the GPS, is that this system uses WGS84 unprojected (EPSG code 4326). If you use Google Earth (EPSG code 4326) and try to map them on Google Maps (EPSG code 3857) you don't have the same SRS. You need to be able to make transformations between different SRSs. We will do this in the following sections, after we learn how to create individual geometries and entire layers.

2.3.2 Creating OGR geometries and vector layers

In this section I will use points looked up on Google Maps (so, using WGS84 / Pseudo-Mercator) and stored in a .csv file to create OGR geometries like Points, Lines or Polygons and their multi-versions. Then I will create a new empty georeferenced vector layer by means of a driver which will create the data source in which the empty vector layer will be stored. Then I will create the attribute fields for the empty layer, which will be stored in the layer definition. Then, I will create the features which will contain the previously created geometries and their attribute fields. Finally, I will insert the features into the new layer.

Code snippet 3: Reading point coordinates from a .csv file Suppose you have looked up your point coordinates on Google Maps or Google Earth and know what features you will have and what geometries you should use for your project. You should now create a .csv file (from converting either an EXCEL or a LIBRE OFFICE CALC file with "Save As" and choosing .csv format), in which you have three columns: the first one is the longitute (E/W), the second one is the latitude (N/S), and the third one represents the ID number of your feaures (from 1 to the last feature). I've created such a file with point coordinates from Google Maps [6]which we will use for this code snippet. You can inspect it at https://github.com/RoxanaTesileanu/multivariate_analyses/blob /master/DeepLearning/pointcoord.csv. Now, according to the point coordinates in the pointcoord.csv file, we're going to create 8 features for our project:

- the first group of points (ID 1) is for the first polygon which will be the representation of the first habitat patch (habitatPatch1) and is the Tampa Hill in Brasov,
- the second group of points (ID 2) is for the first road which will be the representation of a road in Racadau Area in Brasov,
- the third group of points (ID 3) is for the second road which will be the representation of a road in Carpatilor Area in Brasov,
- the fourth group of points (ID 4) is for the third road which will be the representation of a road in Noua Quarter in Brasov,
- the fifth group of points (ID 5) is for the second polygon which will be the representation of the second habitat patch (habitatPatch2) and is the Noua Forest Area,
- the sixth group of points (ID 6) is for the represenation of a GPS-track for an imaginary radio-collared bear individual,
- the seventh group of points (ID 7) is for a polygon which will be the representation of a third habitat patch (habitatPatch3) toward Postavaru Peak in Poiana Brasov,
- the eigth group of points (ID 8) is for the fourth polygon which will be the representation for our study area.

Of course, these features are created for educational purposes. I haven't included as many points as necessary for a detailed representation of the items. Also, if you include more features (additional roads, more GPS-tracks, more habitat patches, etc.) then the results should be useful for further research purposes. The aim of this tutorial is to show how you can construct your features from point coordinates and how to perform spatial analyses on them whatever number of features you consider appropriate for your own purposes and whatever number of point coordinates you use for their representation.

After this general presentation of the data we're going to use, we can continue and read the points into Scala [7] (parse them):

```
import scala.io._
```

```
val source = Source.fromFile("pointcoord.csv")
val data = source.getLines.map(_.split(",")).toArray
val dataHP1 = data.filter(_(2) == "1")
dataHP1.length
val dataRoad1 = data.filter(_(2) == "2")
dataRoad1.length
val dataRoad2 = data.filter(_(2) == "3")
dataRoad2.length
val dataRoad3 = data.filter(_(2) == "4")
dataRoad3.length
val dataHP2 = data.filter(_(2) == "5")
dataHP2.length
val dataGPSTrack = data.filter(_(2) == "6")
dataGPSTrack.length
val dataHP3 = data.filter(_(2) == "7")
dataHP3.length
val dataStArea = data.filter(_(2) == "8")
dataStArea.length
val pointsHP1 = dataHP1.map(i => (i(0).toDouble, i(1).toDouble))
pointsHP1.length
val pointsRoad1 = dataRoad1.map(i => (i(0).toDouble, i(1).toDouble))
val pointsRoad2 = dataRoad2.map(i => (i(0).toDouble, i(1).toDouble))
val pointsRoad3 = dataRoad3.map(i => (i(0).toDouble, i(1).toDouble))
val pointsHP2 = dataHP2.map(i => (i(0).toDouble, i(1).toDouble))
val pointsHP3 = dataHP3.mao(i => (i(0).toDouble, i(1).toDouble))
val pointsGPSTrack = dataGPSTrack.map(i => (i(0).toDouble, i(1).toDouble))
val pointsStArea = dataStArea.map(i => (i(0).toDouble, i(1).toDouble))
```

Code snippet 3: Explanation We read files in Scala with scala.io.Source. Because of that we import scala.io._ at the beginning of the parsing code. We then creat a value called source to store the data source into. Then, we create a value called data in which we access the lines of the .csv file. Each row of the .csv is a line. Because the .csv file is comma delimited, we split each line at "," and then we transform each line to an Array of strings. For each group of points we create a value in which we store them (i.e. dataHP1, dataRoad1). This is done by means of applying a filter on the whole data, to get only the points with the ID number we want. We then check the lenght of each group of data calling length on it (which shows how many Arrays (i.e. point coordinates) it contains. Because the data is provided as Array[String] we must convert the groups of data to Array[Double], which is done using the map function. If you encounter big problems in following this code snippet you can grasp to the book of Jason Swartz "Learning Scala" [8] and then build up your skills with the book of Marc Lewis "Introduction into Programming and Problem Solving Using Scala" [9] which is also accompanied by videos on youtube (https://www.youtube.com/playlist?list=PLLMXbkbDbVt9MIJ9DV4ps-_trOz WtphYO.

Code snippet 4: Creating points and multipoints, lines and multilines, and, polygons and multiploygons In this code snippet I will use the values of the point groups (i.e. pointsRoad1, pointsHP1, etc.) from code snippet 3 to build OGR geometries. In order to use the the point groups, I've stored the code from snippet 3 in an object called ReadPointCoordFromFile found in the readPointCoord.scala file available at the link https://github.com/RoxanaTesileanu/multivariate_analyses/blob/master/DeepLearning/readPointCoord.scala. You should download the .scala file and store it in the Scala project's directory, to make it available in REPL. Reload your project in SBT, restart the console to get to the Scala REPL and try out the following code lines:

```
:load readPointCoord.scala
import ReadPointCoordFromFile._
pointsRoad1
val currentPosition = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPoint)
currentPosition.AddPoint(pointsRoad1(0)._1, pointsRoad1(0)._2)
currentPosition
currentPosition.GetX
currentPosition.GetY
val multiPointHP1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiPoint)
val geomsForMP = for (p<- pointsHP1) yield new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPoint)
val zippedGeomsPointsHP1 = geomsForMP.zip(pointsHP1)
for (z <- zippedGeomsPointsHP1) ( (z._1).AddPoint((z._2)._1, (z._2)._2))
for (z <- zippedGeomsPointsHP1) multiPointHP1.AddGeometry(z._1)
pointsRoad1
val road1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad1) road1.AddPoint(p._1, p._2)
road1.GetGeometryCount
road1.AddPoint(pointsRoad1(0)._1, pointsRoad1(0)._2)
road1.GetGeometryCount
road1.IsEmpty
road1.GetPointCount
val road2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad2) road2.AddPoint(p._1, p._2)
road2.GetPointCount
val road3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad3) road3.AddPoint(p._1, p._2)
road3.GetPointCount
```

```
val gpsTrack1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p<- pointsGPSTrack) gpsTrack1.AddPoint(p._1, p._2)
val multiLineLines = Array(road1, road2, road3, gpsTrack1)
val multiLineEx = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiLineString)
for (l <- multiLineLines)  multiLineEx.AddGeometry(l)
val habitatPatch1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP1) habitatRing.AddPoint(p._1, p._2)
habitatRing.GetPointCount
habitatPatch1.AddGeometry(habitatRing)
habitatPatch1.CloseRings()
habitatPatch1.IsValid
val habitatPatch2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP2) habitatRing2.AddPoint(p._1, p._2)
habitatRing2.GetPointCount
habitatPatch2.AddGeometry(habitatRing2)
habitatPatch2.CloseRings()
val habitatPatch3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP3) habitatRing3.AddPoint(p._1, p._2)
habitatRing3.GetPointCount
habitatPatch3.AddGeometry(habitatRing3)
habitatPatch3.CloseRings()
val stArea = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val ringStArea = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p <- pointsStArea) ringStArea.AddPoint(p._1, p._2)
stArea.AddGeometry(ringStArea)
stArea.CloseRings()
stArea.IsValid
val multiPolygonEx = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiPolygon)
val multiPolyPolys = Array(habitatPatch1, habitatPatch2, habitatPatch3, stArea)
for (poly <- multiPolyPolys) multiPolygonEx.AddGeometry(poly)
```

Code snippet 4: Explanation In the above code we've created several OGR geometries:

- a point called currentPosition,
- a multipoint called multiPointHP1,
- a series of lines called road1, road2, road3, and gpsTrack1,
- a multiline called multiLineEx,

- a series of polygons called habitatPatch1, habitatPatch2, habitatPatch3, stArea, and
- a multipolygon called multiPolygonEx.

In general we can create an OGR geometry by creating a new empty geometry which will be populated with points by calling the AddPoint() function on them. The multi-geometry versions can be created by adding geometries to an empty multi-geometry by calling AddGeometry() on it. The polygons require a ring of points. In the above examples we only have one ring for each polygon. We could also create two rings (an inner and an outer ring) in order to create polygons with holes. For more details see [2].

Next, we will return to vector layers. In the following code snippet we will create a vector layer called RoadsAndGPSTracks,using the multiline geometry created in code snippet 4. For georeferencing we will use the WGS84 / Pseudo-Mercator SRS for which we've already created an instance in code snippet 2 called googleMapSR.

Code snippet 5: Creating a georeferenced vector layer Every new layer needs a name, a spatial reference and a geoemtry type. The new vector layer we will create in this code snippet will be called RoadsAndGPSTracks, it will be in WGS84 / Pseudo-Mercator and will be of type multiline. The layer RoadsAndGPSTracks will be created on a data source called "rdgps", which will be on its turn created by means of a ESRI shapefile driver. After we create the empty vector layer we will populate it with features. Each feature will contain one geometry (so, one road) and its attribute fields (i.e. ID, urban quarter, ect.). For this we have to create the fields and set them to the features. Finally, we're going to insert the features into the new layer.

Because we're going to use the all pieces of code created up to this point, I've grouped them into a package called GeospatialScala. You can download the directory and place it into the src/main/scala directory of your Scala project (https://github.com/RoxanaTesileanu/multivariate_analyses/tree/master/Deep Learning/src/main/scala/com/mai. The first line of the .scala source files specifies the package name. You must change it to "package GeospatialScala" if you have kept the src/main/scala directory structure.

```
import ReadPointCoordFromFile._
import CreateGeoms._
org.gdal.ogr.ogr.RegisterAll()
val driver = org.gdal.ogr.ogr.GetDriverByName("ESRI Shapefile")
val ds = driver.CreateDataSource("rdgps")
val googleMapSR = new org.gdal.osr.SpatialReference()
googleMapSR.ImportFromEPSG(3857)
val lyr = ds.CreateLayer("RoadsAndGPSTracks", googleMapSR,
```

```
org.gdal.ogr.ogrConstants.wkbMultiLineString)
val fd1 = new org.gdal.ogr.FieldDefn("Type", org.gdal.ogr.ogrConstants.OFTString)
val fd2 = new org.gdal.ogr.FieldDefn("ID", org.gdal.ogr.ogrConstants.OFTInteger)
val newLyr = ds.GetLayer(0)
newLyr.CreateField(fd1)
newLyr.CreateField(fd2)
val usedDfn = newLyr.GetLayerDefn()
val feat1 = new org.gdal.ogr.Feature(usedDfn)
feat1.SetGeometry(road1)
feat1.SetField("Type", "road")
feat1.SetField("ID", 1)
newLyr.CreateFeature(feat1)
val feat2 = new org.gdal.ogr.Feature(usedDfn)
feat2.SetGeometry(road2)
feat2.SetField("Type", "road")
feat2.SetField("ID", 2)
newLyr.CreateFeature(feat2)
val feat3 = new org.gdal.ogr.Feature(usedDfn)
feat3.SetGeometry(road3)
feat3.SetField("Type", "road")
feat3.SetField("ID", 3)
newLyr.CreateFeature(feat3)
val feat4 = new org.gdal.ogr.Feature(usedDfn)
feat4.SetGeometry(gpsTrack1)
feat4.SetField("Type", "gpstrack")
feat4.SetField("ID", 4)
newLyr.CreateFeature(feat4)
newLyr.GetFeatureCount
ds.FlushCache
```

Code snippet 5: Explanation In the above code snippet, we've created an empty georeferenced layer called lyr. Next, we've created the field templates for two fields (type and ID), and inserted them into the layer stored by the data source. Afterwards, we've created the four features of to be inserted in the layer, namely: feat1, feat2, feat3, feat4 in which we've stored the line geometries previously created (road1, road2, road3, road4, gpsTrack1). For each feature we've set not only the geometry used but also the field attributes, and inserted them into the extracted layer. We've written the changes to disk with FlushCache() called on the data source. You can see now that a new subdirectory appeared in your Scala project's directory called rdgps which contains the shapefile (a .dbf file, a .prj file, a .shp file and a .shx file). For more details see [2].

2.3.3 Reprojecting OGR geometries and vector layers

Let's suppose we want to reproject our previously created layer from WGS84 / Pseudo-Mercator (for which all points were in WGS 84/ Pseudo-Mercator) to WGS84 unprojected.

2.4 Overlay analyses

2.5 Proximity analyses

2.6 Writing vector data

2.7 Reading raster data

2.8 Pixels resizing

2.9 Moving window analyses

2.10 Map algebra

3 Spatial data analysis

3.1 Introduction into spatial data analysis

In its essence geostatistics adds georeferenced spatial information to the vector of recorded variables of each individual observation, which represents a spatial location. It further adds into models the space dependent random error. In this document I state that the classical multivariate techniques like multiple regression analysis and multidimensional scaling, can include this spatial perspective into modeling by treating the X and Y coordinates as "classical" variables, thus reducing the space dependent random error to the "classical" random error of multivariate statistics.

Geostatistics is a departure from classical statistics just because of the sentence that "it takes spatial autocorrelation of observations into account when predicting values for new points". It is not more than that to it. Maybe the most important fact is that geospatial analysis doesn't treat variables as we are used to in classical statistics but uses individual observations (i.e. individual points) and investigates the relationships between them from a spatial perspective [10], [11], [12], [13], [14]. It is adding space as a variable in the vector of recorded variables of each individual observation/point. The highlight of individual points is actually like in object-based classical multivariate statistics. It is treating space as an autocorrelated variable across a series of individual points, and letting all the other variables be "classical".

For me the most important moment in this introductory phase is to make you realize that in geostatistics we don't talk about classical samples of observations, where we concentrate on variables, but instead in geostatistics we concentrate on pairs of observations (i.e. of points). We compute covariances for such pairs, not for the whole sample as in classical statistics. We don't have weights for entire variables, we have weights for individual points caring those values of the variables studied. It is I believe very important the moment when you understand this. Spatial models treat individual points in ways similar to treating individual variables in classical statistics. But we must be aware these are individual points we are talking about, and we very much use distance measures for objects (like the Euclidean distance) like in the multivariate object-based classical statistics.

That being said I think I can use the same matrix calculations as in classical object-based multivariate analyses (where we use objects to predict values for variables), i.e. a n by n MATRIX OF DISSIMILARITIES BETWEEN OBJECTS by means of which we derive variables as LINEAR COMBINATIONS OF THE OBJECTS (Q-mode analyses) - see [15]. And of course classical variable-based multivariate analyses are equally possible - see [15] and [16]. The example from Quinn and Keough (2002) at the multiple regression chapter, where the study of Paruelo and Lauenroth is presented in which they've modeled the relative abundance of C3 plants against longitude and latitude is an implementation of this perspective [17]. If the spatial analysis includes a random error with spatial dependence, then why not include in the model the X and Y coordinates as two separate variables and make the random error spatial independent? Adding appropriate variables to a model is the approach used in multivariate statistics to reduce the unexplained variation [15] , [16].

Maybe spatial analysis is just classical multivariate analysis (variable- or object-based, or combined); the important thing is to include X and Y variables in the model.

This doesn't mean I give up the "spatial perspective". I will still use the X-Y coordinate plane to inspect how the residuals from fitted linear models are located. Eventually, delineate more than one target population. And reevaluate the sampling design based on these preliminary conclusions.

I will also use the classification of Cressie (1993) [10] which delineates three types of geospatial analyses:

- on continuous surfaces (raster) (using variable-based multivariate techniques)
- on discrete spatial features based on multiple points (lines, polygons) (using variable- and object-based multivatiate techniques)
- on discrete spatial features based on individual points (points) (using object-based multivariate techniques).

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] OSGeo, “GDAL Java API.” [Online]. Available: http://gdal.org/java/

[2] C. Garrard, *Geoprocessing with Python*. Shelter Island: Manning Publications Co., 2016.

[3] S. Safavi, “Installing GDAL/OGR on Ubuntu,” 2015. [Online]. Available: http://www.sarasafavi.com

[4] Canonical, “UbuntuGIS.” [Online]. Available: https://wiki.ubuntu.com/UbuntuGIS

[5] R. Tesileanu, “Using Linux as a development platform for Scala projects,” 2017. [Online]. Available: https://www.researchgate.net/publication/319260791_Using_Linux_as_a_development_platform_for_Scala_projects

[6] Google, “Google Maps.” [Online]. Available: https://maps.google.com/

[7] EPFL, “Scala Standard Library 2.12.0,” 2017. [Online]. Available: http://www.scala-lang.org/api/2.12.0/scala/index.html

[8] J. Swartz, *Learning Scala*. Sebastopol: O’Reilly, 2015.

[9] M. C. Lewis and L. L. Lacher, *Introduction to Programming and Problem Solving Using Scala*. CRC Press, 2017.

[10] N. A. C. Cressie, *Statistics for Spatial Data*. Wiley, 1993.

[11] R. Webster and M. A. Oliver, *Geostatistics for Environmental Scientists*, second edition ed. Wiley, 2007.

[12] E. H. Isaaks and R. M. Srivastava, *Applied Geostatistics*. New York: Oxford University Press, 1989.

[13] T. Hengl, *A Practical Guide to Geostatistical Mapping*, 2009.

[14] K. Johnston, Ver Hoef, Jay M., Krivoruchko, Konstantin, and Lucas, Neil, “Using ArcGis Geostatistical Analyst,” 2003.

[15] G. Quinn and M. Keough, *Experimental design and data analysis for biologists*. Cambridge: Cambridge University Press, 2002.

[16] J. D. Carroll and P. E. Green, *Mathematical Tools for Applied Multivariate Analysis*, revised ed. San Diego: Academic Press, 1997.

[17] J. Paruelo and Laeunroth, W.K., “Relative abundance of plant functional types in grasslands and shrublands of North America,” *Ecological Applications*, vol. 6, pp. 1212–1224, 1996.

Appendix A Point coordinates for code snippet 3

The following point coordinates were looked up on Google Maps (WGS84 / Pseudo-Mercator) and were used in section 2.3.2 to create the example vector layer. Please note they are for educational purposes.

```
//points for Tampa Hill, BV, entered clockwise
45.631118, 25.583995
45.632863, 25.586184
45.633955, 25.586698
45.634345, 25.587706
45.636370, 25.589595
45.636730, 25.589423
45.639431, 25.593006
45.642450, 25.598735
45.642083, 25.603038
45.640676, 25.607061
45.639738, 25.607404
45.638925, 25.607131
45.636318, 25.604551
45.634559, 25.602963
45.633002, 25.602048
45.630514, 25.597860
45.630139, 25.596379
45.629801, 25.596057
45.629043, 25.596454
45.627693, 25.595639
45.627115, 25.594705
45.626380, 25.595060
45.625577, 25.594877
45.626220, 25.592055
45.626215, 25.589502
45.627325, 25.589545
45.628361, 25.589137
45.629576, 25.587421
45.631077, 25.583880
```

// road Valea Racadau, BV
45.612195, 25.588475
45.614445, 25.590527
45.620019, 25.594370
45.623190, 25.594629
45.625304, 25.595018
45.626392, 25.595363
45.626890, 25.595169
45.627222, 25.595623
45.628853, 25.597199
45.628929, 25.597285
45.629729, 25.599574
45.629442, 25.600114
45.630016, 25.601280
45.630590, 25.603785
45.631979, 25.606290
45.633806, 25.609572
45.637007, 25.611947
45.636780, 25.614906
45.636146, 25.618339

//road zona Carpatilor, BV
45.636539, 25.618857
45.633428, 25.620369
45.632975, 25.619160
45.631299, 25.620088
45.631888, 25.623327
45.630529, 25.624105
45.629789, 25.622442
45.627494, 25.622463
45.628023, 25.618209

//road zona spre Noua, BV
45.628627, 25.624364
45.624866, 25.629071
45.627011, 25.633951
45.623432, 25.635506
45.622359, 25.634275
45.619671, 25.631036
45.619626, 25.624752
45.620547, 25.622507

//points for Padurea Noua, BV, entered clockwise
45.612222, 25.589546
45.621849, 25.595210
45.626304, 25.597557
45.629890, 25.602105
45.632456, 25.609955
45.633898, 25.610253
45.635462, 25.612225
45.636364, 25.615346
45.635233, 25.616744
45.631935, 25.617357
45.629894, 25.621671
45.627750, 25.621804
45.628823, 25.617216
45.627631, 25.617099
45.626265, 25.618057
45.626477, 25.619801
45.625147, 25.621939
45.625546, 25.625259
45.623579, 25.626994
45.623868, 25.629202
45.622561, 25.632379
45.620556, 25.630997
45.620293, 25.626782
45.622734, 25.621884
45.621789, 25.618871
45.619452, 25.621083
45.618350, 25.625433
45.616194, 25.625733
45.613602, 25.628569
45.611492, 25.629284
45.602445, 25.618537
45.606375, 25.590345

//imaginary GPS-track
45.617797, 25.612945
45.620919, 25.618952
45.618454, 25.608328
45.622381, 25.619041
45.620180, 25.603591
45.617171, 25.585708
45.622910, 25.586664
45.633693, 25.596202

```
//points for Saua Gorita spre Postavaru
45.610059, 25.586784
45.621180, 25.560753
45.627951, 25.569719
45.627187, 25.577856
45.627582, 25.579325
45.630980, 25.583281
45.631979, 25.606290
45.629479, 25.587688
45.628346, 25.589346
45.626449, 25.589949
45.625579, 25.594658
45.615251, 25.590175


// points for the study area, entered clockwise
45.623683, 25.562561
45.643940, 25.599744
45.637329, 25.623779
45.622708, 25.636663
45.614645, 25.633913
45.599965, 25.615302
```

Appendix B Scala source file: readPointCoord.scala

```
package com.mai.GeospatialScala
import scala.io._
object ReadPointCoordFromFile {

val source = Source.fromFile("pointcoord.csv")
val data = source.getLines.map(_.split(",")).toArray
val dataHP1 = data.filter(_(2) == "1")
dataHP1.length
val dataRoad1 = data.filter(_(2) == "2")
val dataRoad2 = data.filter(_(2) == "3")
dataRoad1.length
dataRoad2.length
val dataRoad3 = data.filter(_(2) == "4")
dataRoad3.length
val dataHP2 = data.filter(_(2) == "5")
dataHP2.length
```

```
val dataGPSTrack = data.filter(_(2) == "6")
dataGPSTrack.length
val dataHP3 = data.filter(_(2) == "7")
dataHP3.length
val dataStArea = data.filter(_(2) == "8")
dataStArea.length
val pointsHP1 = dataHP1.map(i => (i(0).toDouble, i(1).toDouble))
pointsHP1.length
val pointsRoad1 = dataRoad1.map(i => (i(0).toDouble, i(1).toDouble))
val pointsRoad2 = dataRoad2.map(i => (i(0).toDouble, i(1).toDouble))
val pointsRoad3 = dataRoad3.map(i => (i(0).toDouble, i(1).toDouble))
val pointsHP2 = dataHP2.map(i => (i(0).toDouble, i(1).toDouble))
val pointsHP3 = dataHP3.map(i => (i(0).toDouble, i(1).toDouble))
val pointsGPSTrack = dataGPSTrack.map(i => (i(0).toDouble, i(1).toDouble))
val pointsStArea = dataStArea.map(i => (i(0).toDouble, i(1).toDouble))
}
```

Appendix C Scala source file: createGeoms.scala

```
package com.mai.GeospatialScala
import ReadPointCoordFromFile._


object CreateGeoms {
val currentPosition = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPoint)
currentPosition.AddPoint(pointsRoad1(0)._1, pointsRoad1(0)._2)
currentPosition.GetX
currentPosition.GetY
val multiPointHP1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiPoint)
val geomsForMP = for (p<- pointsHP1) yield new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPoint)
val zippedGeomsPointsHP1 = geomsForMP.zip(pointsHP1)
for (z <- zippedGeomsPointsHP1) ( (z._1).AddPoint((z._2)._1, (z._2)._2))
for (z <- zippedGeomsPointsHP1) multiPointHP1.AddGeometry(z._1)
val road1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad1) road1.AddPoint(p._1, p._2)
road1.AddPoint(pointsRoad1(0)._1, pointsRoad1(0)._2)
val road2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad2) road2.AddPoint(p._1, p._2)
val road3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p <- pointsRoad3) road3.AddPoint(p._1, p._2)
val gpsTrack1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLineString)
for (p<- pointsGPSTrack) gpsTrack1.AddPoint(p._1, p._2)
val multiLineLines = Array(road1, road2, road3, gpsTrack1)
```

```
val multiLineEx = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiLineString)
for (l <- multiLineLines) multiLineEx.AddGeometry(l)
val habitatPatch1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing1 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP1) habitatRing1.AddPoint(p._1, p._2)
habitatPatch1.AddGeometry(habitatRing1)
habitatPatch1.CloseRings()
val habitatPatch2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing2 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP2) habitatRing2.AddPoint(p._1, p._2)
habitatPatch2.AddGeometry(habitatRing2)
habitatPatch2.CloseRings()
val habitatPatch3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val habitatRing3 = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p<- pointsHP3) habitatRing3.AddPoint(p._1, p._2)
habitatPatch3.AddGeometry(habitatRing3)
habitatPatch3.CloseRings()
val stArea = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbPolygon)
val ringStArea = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbLinearRing)
for (p <- pointsStArea) ringStArea.AddPoint(p._1, p._2)
stArea.AddGeometry(ringStArea)
stArea.CloseRings()
val multiPolygonEx = new org.gdal.ogr.Geometry(org.gdal.ogr.ogrConstants.wkbMultiPolygon)
val multiPolyPolys = Array(habitatPatch1, habitatPatch2, habitatPatch3, stArea)
for (poly <- multiPolyPolys) multiPolygonEx.AddGeometry(poly)
}
```

Appendix D Scala source file: createVLayer.scala

```
package com.mai.GeospatialScala

    import ReadPointCoordFromFile._
import CreateGeoms._


object CreateVLayer {


org.gdal.ogr.ogr.RegisterAll()
val driver = org.gdal.ogr.ogr.GetDriverByName("ESRI Shapefile")
val ds = driver.CreateDataSource("rdgps")
val googleMapSR = new org.gdal.osr.SpatialReference()
googleMapSR.ImportFromEPSG(3857)
```

```
val lyr = ds.CreateLayer("RoadsAndGPSTracks", googleMapSR, org.gdal.ogr.ogrConstants.wkbMultiLineStrin
val fd1 = new org.gdal.ogr.FieldDefn("Type", org.gdal.ogr.ogrConstants.OFTString)
val fd2 = new org.gdal.ogr.FieldDefn("ID", org.gdal.ogr.ogrConstants.OFTInteger)
val newLyr = ds.GetLayer(0)
newLyr.CreateField(fd1)
newLyr.CreateField(fd2)
val usedDfn = newLyr.GetLayerDefn()
val feat1 = new org.gdal.ogr.Feature(usedDfn)
feat1.SetGeometry(road1)
feat1.SetField("Type", "road")
feat1.SetField("ID", 1)
newLyr.CreateFeature(feat1)
val feat2 = new org.gdal.ogr.Feature(usedDfn)
feat2.SetGeometry(road2)
feat2.SetField("Type", "road") feat2.SetField("ID", 2)
newLyr.CreateFeature(feat2)
val feat3 = new org.gdal.ogr.Feature(usedDfn)
feat3.SetGeometry(road3)
feat3.SetField("Type", "road")
feat3.SetField("ID", 3)
newLyr.CreateFeature(feat3)
val feat4 = new org.gdal.ogr.Feature(usedDfn)
feat4.SetGeometry(gpsTrack1)
feat4.SetField("Type", "gpstrack")
feat4.SetField("ID", 4)
newLyr.CreateFeature(feat4)
newLyr.GetFeatureCount
ds.FlushCache

    }
```