Oracle® Cloud

Using Oracle Spatial AI on Autonomous AI Database Serverless

F91895-03
October 2025

Oracle Cloud Using Oracle Spatial AI on Autonomous AI Database Serverless,

F91895-03

Primary Author: Lavanya Jayapalan

Contributors: David Lapp, Siva Ravada, Qingyun (Jeffrey) Xie, Jayant Sharma, Hector Alejandro Saucedo Briseno, David Esparza

Contents

Preface

Topics:

- Audience
- Related Resources
- Conventions

Using Oracle Spatial AI on Autonomous AI Database Serverless describes how to use Oracle Spatial Artificial Intelligence (AI) on Autonomous AI Database Serverless.

Audience

This document is intended for data scientists, spatial developers, Geographic Information System (GIS) users, and business users.

Related Resources

For more information, see these Oracle resources:

- Oracle Machine Learning Notebooks
- Oracle Machine Learning Notebooks Interactive Tour
- Oracle Machine Learning for Python User's Guide
- Python API Reference for Oracle Spatial AI

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

About Oracle Spatial AI

Learn about Oracle Spatial Artificial Intelligence (AI).

Topics:

What is Oracle Spatial AI?

In general, Geospatial Artificial Intelligence (GeoAI) or Spatial AI refers to geospatial machine learning and deep learning capabilities that gather usable information and intelligence, which enable users to detect, track, discover, classify, predict, and analyze location related business events, geospatial objects, and ground features on the Earth.

Geospatial data is everywhere, and most events and business data are associated with location. Location plays a critical role in affecting environment, events, and businesses. Therefore, analyzing and applying location and related information to gather useful intelligence for various applications is important.

The following lists a few use cases where Spatial AI helps organizations to understand better the business opportunities, environmental impacts, or operational risks and gain valuable insights to make informed decisions:

- Analyzing patterns of cancer and epidemic disease such as cholera, SARS, and Covid-19.
- Finding hot spots of crime to help police to make staffing and patrolling decisions.
- Identifying patterns of car accidents or pedestrian deaths to help optimize arrangements of red lights and road networks.
- Predicting house prices based on census data and location information and helping to choose a home considering the home's proximity to economic opportunities, schools, health care, and roadways for commutes.

The current release of Oracle Spatial AI provides geospatial machine learning algorithms for analyzing and modeling geospatial vector data and location related events. It provides geospatial machine learning techniques, end-to-end workflow, and related APIs.

Spatial AI is integrated with Oracle Machine Learning (OML) and is deployed with OML4Py on Oracle Autonomous AI Database Serverless cloud service. This implies that you can access Spatial AI through OML interfaces and services on your Autonomous AI Database Serverless instance.

You can leverage this product to prepare and analyze data stored in Oracle Spatial and Oracle Cloud Infrastructure (OCI) Object Storage, train spatial machine learning models, and apply the models in a variety of applications. For example, you can apply clustering techniques to identify spatial patterns of events, detect hot spots, cold spots, anomalies and outliers. You can also apply spatial regression and classification techniques to analyze spatial data, predict house prices, and classify poverty levels.

Real-World Applications of Oracle Spatial AI

Oracle Spatial AI can be used to add value in several application areas.

- **Spatial Data Analysis and Modeling**
  Geospatial data scientists and application users can leverage Oracle Spatial AI to prepare and analyze data stored in Oracle Spatial and OCI object storage, train spatial machine learning models, and apply these models in a variety of applications. For example, users can use clustering, regionalization, and anomaly detection techniques to identify spatial patterns and anomalies. Users can also apply spatial regression and classification techniques to do predictive analysis.
- **Enterprise Applications**
  Enterprise applications include HCM, ERP, CRM, and industry solutions for utilities, defense, and public sectors. Most business data have a spatial component and therefore spatial analyzing and modeling capabilities adds significant value to those applications. For example, HR systems can leverage clustering to analyze grievances, safety violations, and disciplinary hot spots. Utility applications may train models to detect electric vehicle charging locations from their heavy load.
- **OML Functionalities**
  Oracle Spatial AI is delivered as part of OML. It fully leverages OML functionalities and complements OML by adding spatial modeling capabilities. Spatial AI, together with OML, enhances the user experience for data scientists and, in particular, geospatial users.

Spatial Machine Learning Concepts and Algorithms

Learn about the Spatial AI machine learning concepts.

Spatial Machine Learning is machine learning on spatial and location data to gather usable information and intelligence for various applications.

The majority of data are associated with location. Location and location relationships affect business and event outcomes. This kind of impact of location is called a spatial effect. There are two types of spatial effects, namely, Spatial Dependence and Spatial Heterogeneity. Spatial machine learning incorporates these spatial effects by taking into account the location data (in addition to business or attribute data) to improve the analytical and predictive models.

One approach of spatial machine learning is to use traditional machine learning models by integrating spatially explicit independent variables generated using spatial feature engineering operations. Oracle Spatial AI provides some Spatial Feature Engineering algorithms for this purpose. However, this approach may be limited by what spatially engineered features are used, as these may not be able to fully cover the spatial effects in most application scenarios.

Besides using traditional machine learning models involving spatial feature engineering, Spatial AI allows you to create specialized machine learning models designed to fully incorporate spatial information and spatial relationships or spatial effects (such as spatial dependence and spatial heterogeneity).

In summary, the following table lists the spatial machine learning techniques and the corresponding algorithms supported by Spatial AI. Using these techniques, you can build spatial machine learning models or generate spatially engineered features for use in traditional machine learning models to detect patterns and make predictions.

Techniques	Algorithms
Spatial Feature Engineering	• Spatial Lag Transformer • Categorical Lag Transformer • Spatial Coordinates Transformer • Spatial Imputer
Spatial Clustering	• LISA Hotspot • DBSCAN with Regionalization • Agglomerative with Regionalization
Spatial Anomaly Detection	• Local Outlier Factor (LOF)
Spatial Regression	• Spatial Cross-Regressive Model (SLX) • Spatial Lag Model (or Spatial Autoregressive (SAR) model) • Spatial Error Model (SEM) • Geographical Regressor (GR) • Geographically Weighted Regression (GWR) • Spatial Regimes (OLS_Regimes) • Spatial Fixed Effects (SFE)
Spatial Classification	• SLX Classifier • GWR Classifier • Geographical Classifier

Spatial Dependence

Spatial dependence means that a variable’s values at different locations are related to or affected by each other depending on their distances. The closer the distance, the more similar the values of the variable, and conversely.

This is according to Tobler's first law of geography, *Everything is related to everything else, but near things are more related than distant things*. It reflects the fact that the characteristics of the observations are affected by their spatial arrangement, and the values of observations are related to each other through distance.

Spatial dependence is also called or measured as spatial autocorrelation. A typical example of spatial dependence is house prices in a district. The more expensive houses are closer to each other, and the price of a house sold would affect the selling prices of its neighbors. Spatial dependence is one major type of spatial effects a spatial machine learning model needs to take into consideration when it exists.

Spatial Heterogeneity

Spatial heterogeneity refers to the uneven distribution of a variable’s values or systematic variation of outcomes across space.

Spatial heterogeneity means that parts of the machine learning model may vary systematically with geography, that is, the parameters or error terms of the model may change according to the location. The error term change of a model, or the presence of variance in the residuals, is caused by spatial heteroskedasticity, a special type of heterogeneity. For example, the climate and weather might change dramatically across different climate zones, affecting forestation/ deforestation and crop yields. If spatial heterogeneity exists as a spatial effect, then it needs to be considered by spatial machine learning models.

In the real world, spatial dependence may sometimes play a bigger role in affecting other independent variables or dependent variables (outcomes). But at other times, spatial heterogeneity may play a bigger role than spatial dependence. However, both play a role in many scenarios. This implies that for a specific use case you may choose different machine learning algorithms for different applications, depending upon the spatial effect.

Oracle Spatial AI takes into consideration the spatial relationships and spatial effects in all its spatial machine learning algorithms. For that purpose, spatial weight is a required parameter for all algorithms. Spatial weight is how spatial relationships are quantified. Spatial effects can be tested using statistics, such as Autocorrelation/Moran's I, Lagrange Multipliers, and Koenker-Basset Test.

See Spatial Weights, Spatial Autocorrelation, and Metrics for Spatial Regression for more information.

Spatial Feature Engineering

Spatial feature engineering refers to using geographic information to "construct" new data or develop additional information from geographic information.

For example, given a dataset of metro stations, generate metrics for the number of restaurants and theaters within several distances. Or, given a neighborhood, generate the average house price surrounding a specific house or location. The results then are usually either numerical or categorical variables. These are also called spatially explicit independent or exogenous variables in the spatial modeling context. The purpose of spatial feature engineering is to generate those new features from spatial data, which can be treated as extra independent variables and be directly fed into general machine learning algorithms (without modification of the algorithms) for analysis and predictions.

Machine learning based on spatial feature engineering reflects that those processes are not the same everywhere geographically. This is one type of spatial machine learning. However, depending on the application cases, this may not fully consider more intrinsic spatial relationships or neighborhood effects. Thus an application may require some specialized machine learning algorithms provided by Spatial AI.

You can leverage Oracle Spatial database functionalities to engineer new features based on spatial data. In addition, the following three new feature engineering methods are supported - Spatial Lag Transformer, Categorical Lag Transformer, and Spatial Coordinates Transformer.

Spatial Clustering and Regionalization

Spatial clustering is a fundamental method of geographical analysis that detects patterns from location data.

The following lists a few use cases where you can apply spatial clustering:

- Finding crime hotspots to help police make staffing and patrolling decisions.
- Identifying patterns of car accidents or pedestrian deaths to help optimize arrangements for red lights and road networks.

Spatial clustering consists of labeling the observations of a dataset, so that observations with the same label share common characteristics spatially. Clustering is widely used to provide insights into the geographic structure of complex spatial data. LISA Hotspot is a spatial clustering algorithm that fully considers spatial dependence.

Regionalization is a special kind of clustering to group observations, which are similar not only in their statistical attributes, but also in their spatial location. Observations are grouped so that each spatial cluster, or region, is spatially-coherent as well as data-coherent. DBSCAN with Regionalization and Agglomerative with Regionalization are two such clustering algorithms.

Spatial Anomaly Detection

Anomaly detection finds outliers and novelties, defined as observations that are significantly different from the others.

Outlier detection estimators try to fit the regions where the training data is the most concentrated, ignoring the deviant observations. Novelty detection finds whether a new observation is an outlier, in which case an outlier is also called a novelty. However, novelty detection assumes no outliers in the training data.

An example use case can be analyzing all environmental or traffic monitoring sensor data to find out the anomaly that could lead to identifying dysfunctional sensors.

Spatial anomaly detection identifies observations that are geographically isolated using spatial weights with standard anomaly detection methods (see the Local Outlier Factor anomaly detection method). In addition, Spatial Clustering and Regionalization algorithms can also be used to detect outliers and analyze anomalies.

Spatial Regression

Spatial regression is a particular type of regression that introduces space or geographical context into the statistical framework of regression for prediction or inferring causal relationships.

Although spatial regression can be done to some extent through traditional regression algorithms based on spatial feature engineering, its main task and focus is to provide unique regression algorithms that consider spatial relationships such as spatial dependence or spatial heterogeneity. Some example use cases include predicting house prices based on census data and location information, or finding a home considering the property’s proximity to economic opportunities, schools, health care, and roadways for commutes.

The following spatial regression algorithms are supported:

- Spatial Cross-Regressive Model (SLX)
- Spatial Lag Model (SAR)
- Spatial Error Model (SEM)
- Geographical Regressor (GR)
- Geographically Weighted Regression (GWR)
- Spatial Regimes (OLS_Regimes)
- Spatial Fixed Effects (SFE)

Spatial Classification

Spatial classification is a particular type of classification that introduces space or geographical context into the statistical framework of a classification to assign an object to a class in a set of categories.

Although spatial classification can be done to some extent through traditional classification algorithms based on spatial feature engineering, its main task and focus is to provide unique classification algorithms considering spatial dependence or spatial heterogeneity.

The following describes a few use cases where you can apply spatial classification to predict emergency responses and help in the decision-making process:

- Analyzing and predicting the crime rate based on zones and related demographic data
- Analyzing and predicting the severity of air pollution based on industrialization status and population density across a geographic region

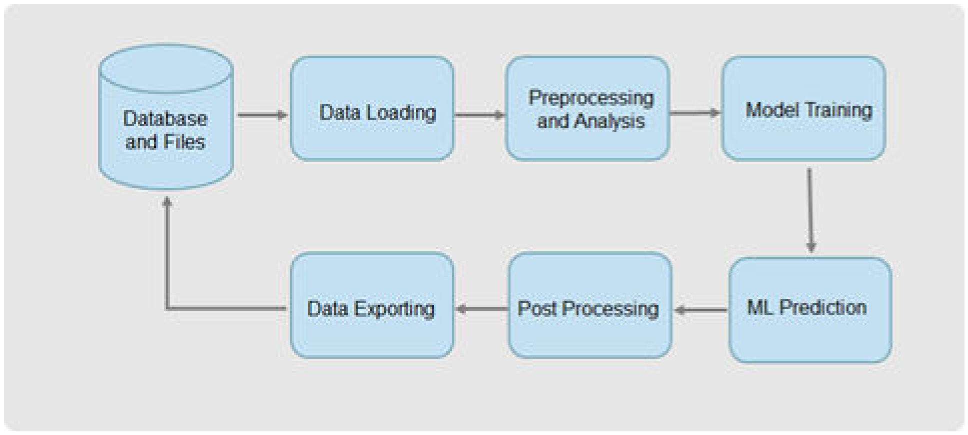
The following three spatial classification algorithms are supported:

- SLX Classifier
- GWR Classifier
- Geographical Classifier

Spatial AI Modeling Workflow

Learn about the Spatial AI modeling workflow for both supervised and unsupervised machine learning.

The following flowchart shows the typical workflow of Spatial AI modeling and analysis.



As seen in the preceding figure, the workflow comprises the following steps - data loading, preprocessing, training and evaluating ML models, predicting using the trained model, post-processing the results, and exporting the results back into database or files in other storage.

The workflow applies to both supervised (regression, classification, and anomaly detection) and unsupervised (clustering and regionalization) machine learning . However, in the latter case, the training and prediction actions are merged together into one step.

The workflow actions shown in the preceding figure are supported by the following components:

- **Data Access:** This component helps for reading from and writing into one of the following data sources - Autonomous AI Database Serverless, OCI Object Storage, or local files. However, note that the main storage system is the Autonomous AI Database Serverless database. A single proxy spatial data frame is used for both data loading and data exporting.
  See Access Spatial Data for more information.
- **Preprocessing:** This component prepares, processes, and augments source data. It includes filling missing values, scaling data, engineering new features, and splitting the source dataset into training dataset, validation dataset, and test dataset.

See Preprocess Spatial Data for more information.

- **Spatial Analysis:** This component provides basic spatial analytics. These are then used together with or to facilitate machine learning modeling to achieve better results or gain better insights.
  See Perform Spatial Analysis for more information.
- **AI Machine Learning Algorithms:** This component is at the core of Spatial AI and provides algorithms and techniques for descriptive and predictive analysis.
  Each algorithm provides methods that allows you to create or train models and make predictions.
  - **Regression, classification, and anomaly detection algorithms:** These are supervised learning algorithms where each algorithm has a fit method for training, a score method for evaluation, and a predict method for prediction.
    See Apply Spatial Regression, Apply Spatial Classification, and Apply Spatial Anomaly Detection for more information.
  - **Clustering and regionalization algorithms:** These are unsupervised learning algorithms where each algorithm has a fit method for both training and clustering.
    See Apply Spatial Clustering for more information.

  In addition to training and applying each model individually, this component also provides adaptive spatial regression. This tool enables you to automatically search and evaluate different regression algorithms and find out the best algorithm for a specific application.
- **Post-processing:** This component generally includes cleaning the results, removing redundancy, georeferencing spatial data, converting data formats, and generating and storing useful spatial layers.
  As the Spatial AI machine learning is applied only with spatial vector data, the results are relatively simple without much need of post-processing. The resulting data can be directly stored back in the database using the data access component. You can then further process the data through Oracle Spatial functionalities.

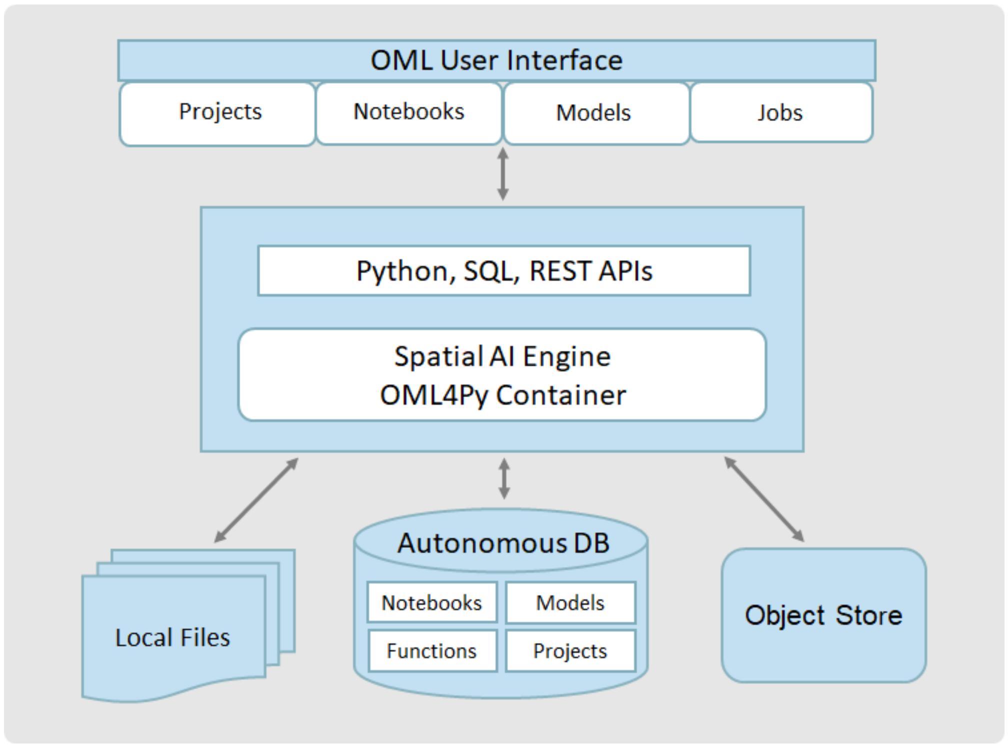
  See Run Post-Processing Tasks for more information.
- **Spatial pipeline:** The pipeline provides the capabilities to organize and simplify spatial machine learning workflow. The spatial pipeline extends the existing `scikit-learn` pipeline to take spatial information such as geometry data and spatial weights.
  See Work with Spatial Pipeline for more information.

Integration of Spatial AI with OML

Oracle Spatial AI is a Python library deployed with Oracle Machine Learning (OML) and runs in Oracle AI Database (or earlier Oracle Database versions) spawned OML4Py servers in the Autonomous AI Database Serverless environment.

Spatial AI is integrated with OML, mainly OML Notebooks and OML4Py. It is therefore considered as an extension to the OML product with added spatial modeling capabilities.

The following diagram shows the architecture of Oracle Spatial AI integrated with OML on Autonomous AI Database Serverless.



You can interact with the database using the secure OML connection. Besides Autonomous AI Database, Oracle Spatial AI can also access spatial data from OCI Object Storage or the local file system.

You can access Oracle Spatial AI through the OML User Interface (see Get Started with Oracle Machine Learning User Interface). Using this interface you can perform the following:

- Create and manage projects and notebooks.
- Schedule and monitor model training and prediction jobs.
- Publish Python functions and models through the OML4Py Embedded Execution REST and SQL APIs to leverage the scalability and high-performance of Autonomous AI Databases (see Oracle Machine Learning for Python User's Guide).

See Get Started with Oracle Spatial AI for more information.

Python API for Spatial AI

You can perform the spatial machine learning workflow using the Python API provided by Oracle Spatial AI.

The Python API includes the following major packages:

- **`oraclesai.data`:** Provides a unified structure for data access (read and write) to database and object storage as well as data processing in the machine learning workflow.
- **`oraclesai.preprocessing`:** Provides functionalities for data preprocessing, feature engineering, and label preparation.
- **`oraclesai.analysis`:** Provides some basic data analysis, particularly enable users to leverage Oracle Spatial functionality through Python.
- **`oraclesai.clustering`:** Provides spatial clustering techniques and algorithms.

- **`oraclesai.outliers`:** Provides spatial anomaly detection techniques and algorithms.
- **`oraclesai.regression`:** Provides spatial regression techniques and algorithms.
- **`oraclesai.pipeline`:** Provides spatial pipeline for chaining the processing steps in a machine learning workflow.
- **`oraclesai.classification`:** Provides spatial classification techniques and algorithms.

See *Python API Reference for Oracle Spatial AI* for more details on the packages.

Notebook Examples

Spatial AI provides over 20 sample notebooks which demonstrates how to use the spatial machine learning techniques in different use case scenarios.

These notebooks are self-contained and hosted on the OML User Interface (UI). You can directly access and run the example notebooks using the OML UI on your Autonomous AI Database Serverless instance.

Task	Notebooks
Data Access	• OML4Py Spatial AI PAR Object Store
Spatial Feature Engineering	• OML4Py Spatial AI Categorical Lag Transformer • OML4Py Spatial AI Scoord Transformer • OML4Py Spatial AI Spatial Imputer • OML4Py Spatial AI Spatial Lag Transformer
Spatial Clustering	• OML4Py Spatial AI Agglomerative Clustering and Regionalization • OML4Py Spatial AI DBSCAN Accidents • OML4Py Spatial AI Hotspot Clustering
Spatial Anomaly Detection	• OML4Py Spatial AI LOF Accidents
Spatial Regression	• OML4Py Spatial AI Geographical Regressor • OML4Py Spatial AI GWR Regressor • OML4Py Spatial AI OLS Regressor • OML4Py Spatial AI Spatial Lag and Error Regressors • OML4Py Spatial AI Spatial Fixed Effects Regressor
Spatial Classification	• OML4Py Spatial AI Geographical Classifier • OML4Py Spatial AI SLX Classifier
Spatial Analysis	• OML4Py Spatial AI Exploratory Analysis • OML4Py Spatial AI Spatial Operations
OML4Py	• OML4Py Spatial AI Embedded Execution for SQL • OML4Py Spatial AI SpatialDataFrame to OML DataFrame • OML4Py Spatial AI Save Load Run

About Python Libraries in Oracle Spatial AI

Oracle Spatial AI is deployed with OML4Py which comes installed with Oracle Autonomous AI Database Serverless.

The OML4Py installation includes Python, additional required Python libraries, and the OML4Py server components. A Python interpreter is included with Oracle Machine Learning Notebooks in Autonomous AI Database. See About the Python Components and Libraries in OML4Py for the list of Python libraries included in OML4Py.

Also, note the following for Spatial AI:

- **Python:** The Python installation used is 3.12.0 version.
- **Required Python Libraries for Oracle Spatial AI:**
  - `SQLAlchemy 2.0.15`
  - `contextily 1.3.0`
  - `geopandas 1.0.1`
  - `shapely 2.0.3`
  - `fiona 1.10b2`
  - `pyproj 3.6.1`
  - `esda 2.5.1`
  - `libpysal 4.9.2`
  - `mgwr 2.2.1`
  - `spglm 1.1.0`
  - `spreg 1.4.2`

  The preceding libraries are included with Python on your Oracle Autonomous AI Database Serverless instance.

2 Get Started with Oracle Spatial AI

Oracle Spatial AI is a Python package automatically deployed with OML4Py.

The following steps enable you to get started with Oracle Spatial AI:

1. Provision an Autonomous AI Database Serverless instance.

   See Provision Autonomous AI Database for more information.

2. Obtain your Oracle Machine Learning (OML) user credentials.

   You can request your Service Administrator to create and provide access to your OML account.

3. Access the OML user interface (UI) on your Autonomous AI Database instance.

   See Access Oracle Machine Learning User Interface for more information.

4. Create or open a notebook, and connect to the Python interpreter.

   See Use the Python Interpreter in a Notebook Paragraph for more information.

   Note

   Installing OML4Py automatically installs Spatial AI. When you connect to a Python interpreter in a notebook paragraph from the OML UI, an OML4Py container gets created on Oracle Cloud Infrastructure (OCI), and the Spatial AI package gets installed together with other OML4Py packages automatically. See Oracle Machine Learning for Python User's Guide for more information on OML4Py.

   You can now interact with Spatial AI through Oracle Machine Learning Notebooks.

3 Access Spatial Data

Spatial AI can retrieve spatial data from different data sources for analysis and machine learning modeling.

Spatial AI can work with spatial data stored in database tables (as geometry layers), or access spatial data in OCI Object Storage, or from your local file system (in their original file formats, such as Shapefiles and GeoJSON files).

Topics:

About Accessing Data in Spatial AI

Spatial data refers to layers of geometries, such as points, lines, and polygons.

The geometries record the location and shape of spatial objects and are associated with other types of data for analysis (refer to *Oracle Spatial Developer's Guide* for more information). Spatial AI API provides the `SpatialDataFrame` class, a data structure that unifies how spatial data is accessed in spatial analysis and machine learning workflows.

A `SpatialDataFrame` instance is created by calling the `create()` class method and passing in a reference to the data, which is called a dataset. A dataset refers to a data source and contains the connection and location of the source data.

The following table lists the four types of datasets (or data sources) that are supported.

Data Source	Description
`DBSpatialDataset`	A reference to a database table with a geometry layer.
`FileSpatialDataset`	A reference to a directory or file in a spatial format within a local file system.
`PARObjStoreSpatialDataset`	A reference to a folder or object in a spatial format located in OCI Object Store containing a Pre-Authenticated Request URL.
`GeoDataFrameDataset`	A reference to an existing `GeoDataFrame`.

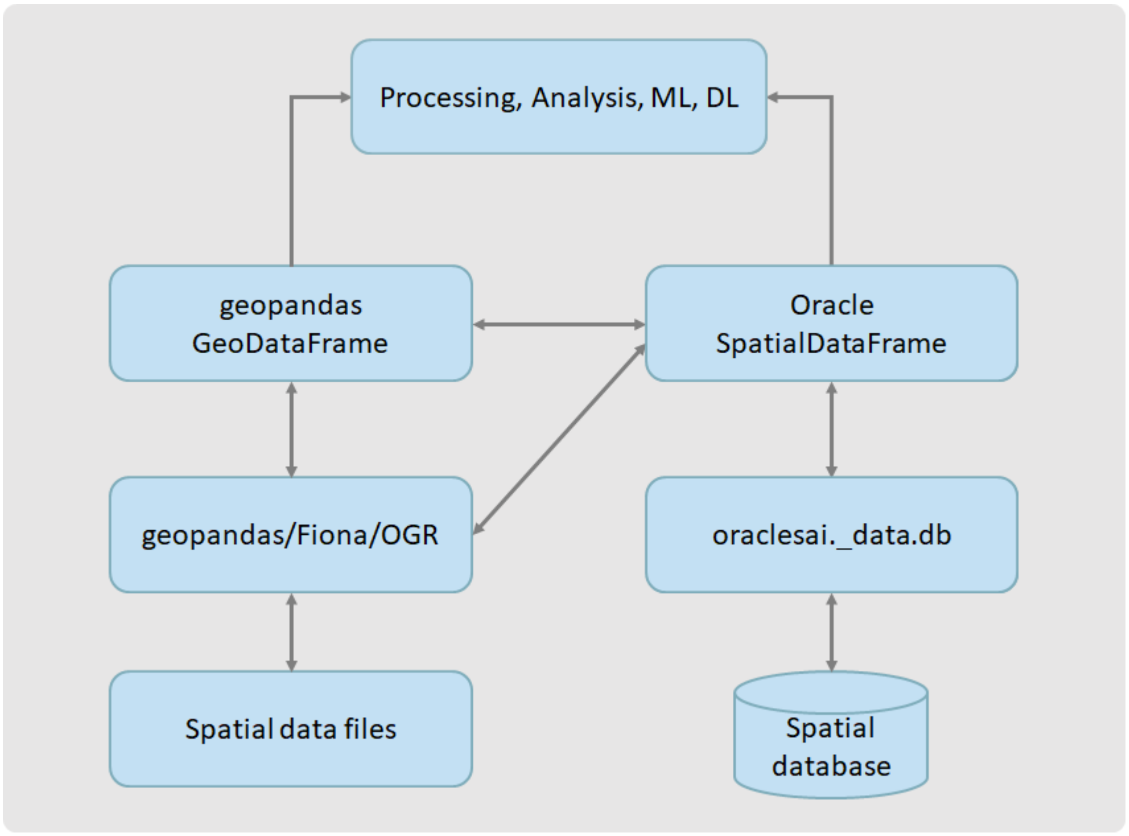
The following code example shows how to create an instance of `SpatialDataFrame` using `DBSpatialDataset` as data source to reference the database table *la_block_groups*.

```
import oml
from oraclesai import SpatialDataFrame, DBSpatialDataset

block_groups = 
SpatialDataFrame.create(DBSpatialDataset(table='la_block_groups', 
schema='oml_user'))
```

Data Access Workflow

Learn about the spatial data access workflow for the supported data sources.

The following figure shows the spatial vector data model and its internal data access workflow:



As seen in the preceding figure, creation of `SpatialDataFrame` depends on the data source:

- **Spatial database:** `SpatialDataFrame` is created by Spatial AI itself.
- **Files:** `SpatialDataFrame` is created through GeoPandas/Fiona/OGR.

You can convert a `SpatialDataFrame` to `GeoPandas GeoDataFrame` and its columns to `NumPy` arrays when more complex in-memory analysis is required or when the data needs to be passed to a standard library such as `scikit-learn` or `PySAL`. Conversely, you can also convert a GeoPandas `GeoDataFrame` to a `SpatialDataFrame`.

All Oracle Spatial AI API algorithms accept `SpatialDataFrame` as input. Some algorithms can also accept `GeoPandas GeoDataFrame` or `NumPy` arrays.

Data Manipulation with SpatialDataFrame

Review the properties and methods available in the `SpatialDataFrame` class for data manipulation.

Properties and Methods	Description
`columns`	Returns a list of column names.
`index_columns`	Returns a list of columns that represent the index; these columns are not part of the property columns.
`shape`	Returns a tuple with the dimensionality of the current instance.
`head`	Returns the first rows of the data.
`add_column`	Adds a new column to the current instance. The new column must have the same number of rows as the current instance.
`drop`	Returns a new instance of `SpatialDataFrame` with the specified columns removed.
`dropna`	Removes rows containing missing values and returns a new instance of `SpatialDataFrame`.
`iterrows`	Iterates through all the rows of the current instance.
`create`	Creates a `SpatialDataFrame` instance based on the dataset type.
`as_geo_data_frame`	Returns a GeoPandas `GeoDataFrame` based on the current instance.
`get_values`	Returns the data of the current instance as a NumPy array.
`write`	Writes the data of the current instance into the destination specified in the parameter dataset.
`sort_values`	Returns a new instance of `SpatialDataFrame` sorted by the values of the specified columns.

The `SpatialDataFrame` class also functions as the Python API for Oracle Spatial database enabling Python access to Oracle Spatial functionalities and in-database spatial analysis.

See Perform Spatial Analysis for more information.

4

Preprocess Spatial Data

You can prepare, process, and augment source data for spatial analysis using Oracle Spatial AI.

You can use the processing methods provided by Spatial AI for filling missing values, scaling data, engineering new features, and splitting the source dataset into training, validation, and test datasets.

Note that data preprocessing may vary for different scenarios. Also, some methods work only for numerical data, and the others work only for categorical variables. However, the machine learning algorithms work with numbers, and the preprocessing methods aim to provide "good" data to the algorithms to produce meaningful results.

The following preprocessing methods are implemented in Oracle Spatial AI.

Topics:

Categorical Lag Transformer

The categorical lag is used for categorical variables and represents the most common value in the neighborhood.

For example, given a feature representing a property type (such as house, apartment, townhouse, and so on), the categorical lag is the most common property in the surroundings.

This is also a feature engineering method which computes categorical lag values that can be directly used to train any machine learning models. The `CategoricalLagTransformer` class computes the categorical lag of a given training data and changes the value of an observation for its categorical lag. It transforms an observation's value with the most common value in the neighborhood.

An instance of this class takes the `spatial_weights_definition` parameter, which defines the relationship between the neighboring observations.

The main methods of the class are described in the following table.

Method	Description
`fit`	Calculates the spatial weights of the training data using the algorithm associated with the `spatial_weights_definition` parameter and the geometry column.

Method	Description
`transform`	Returns the most common value from each location's neighbors. By defining the `use_fit_lag` parameter, the method can use the neighbors from the training set, or the data passed into the `transform` method. The output is a NumPy array.
`fit_transform`	Calls the `fit` and `transform` methods in sequence with the training set.

See the CategoricalLagTransformer class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the `CategoricalLagTransfomer` method to transform the values from the *`INCOME_CLASS`* feature for the most common value of the corresponding neighbors.

The *`INCOME_CLASS`* column has four categories: `High`, `Medium-High`, `Medium-Low`, `Low`. These represent the income level for a specific observation. The target variable (*`MEDIAN_INCOME`*) and the `geometry` column are not part of the output.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.preprocessing import CategoricalLagTransformer
import pandas as pd

# Create a categorical variable based in the median income
labels=['Low', 'Medium-Low', 'Medium-High', 'High']
block_groups_extended = block_groups.add_column("INCOME_CLASS",
pd.qcut(block_groups["MEDIAN_INCOME"].values, [0, 0.25, 0.5, 0.75, 1],
labels=labels).tolist())

# Define the variables of the training data
X = block_groups_extended[["MEDIAN_INCOME", "INCOME_CLASS", "geometry"]]
print(f">> Original data:\n {X['INCOME_CLASS'].values[:10]}")

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=20)

# Create an instance of CategoricalLagTransformer
categorical_lag_transformer = CategoricalLagTransformer(weights_definition)

# Transforms the training data with the categorical lag
X_categorical_lag = categorical_lag_transformer.fit_transform(X,
y='MEDIAN_INCOME', geometries='geometry')

# Displays the transformed data
print(f"\n>> Transformed data:\n {X_categorical_lag[:10, :]}")
```

The resulting output is a NumPy array with a single column, representing the categorical lag of the *`INCOME_CLASS`* column. Note that both the target variable (*`MEDIAN_INCOME`*) and the geometries are not part of the output.

```
>> Original data:
 ['Medium-Low' 'Medium-High' 'Medium-High' 'High' 'High' 'High' 'High'
 'High' 'Medium-High' 'Medium-Low']
```

```
>> Transformed data:
 [['High']
 ['High']
 ['High']
 ['High']
 ['High']
 ['High']
 ['High']
 ['Medium-High']
 ['Medium-High']
 ['Medium-High']]
```

Spatial Lag Transformer

The spatial lag of a particular feature reflects the average value of that feature in the neighborhood around each observation.

For example, in a given neighborhood, the spatial lag of the house price is the average house price surrounding a specific house or location. This is a feature engineering method which computes spatial lag values that can be directly used to train any machine learning models.

The `SpatialLagTransformer` class calculates the spatial lag of training data and changes the value of an observation to its spatial lag. In other words, it changes an observation's value to the average value of its neighbors.

To create an instance of `SpatialLagTransformer`, it is necessary to define the `spatial_weights_definition` parameter, which establishes the relationship between neighboring locations.

The main methods of the class are described in the following table.

Method	Description
`fit`	Computes the spatial lag for all the features in the training set.
`transform`	Changes the spatial lag value depending on the `use_fit_lag` parameter. If `use_fit_lag=True`, then it calculates the spatial lag from the training set. Otherwise, it computes the spatial lag from the data passed into the `transform` method. The function returns a NumPy array.
`fit_transform`	Calls the `fit` and `transform` methods in sequence with the training data.

See the SpatialLagTransformer class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the `SpatialLagTransformer` method to change the *`MEAN_AGE`* and *`HOUSE_VALUE`* features values to determine their spatial lag values. Note that the *`MEDIAN_INCOME`* feature is ignored since it is defined as the target variable. The `geometry` feature is used to calculate the spatial lag, but it is not part of the output from the transformer.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.preprocessing import SpatialLagTransformer

# Define the variables
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "geometry"]]
```

```
# Print original data
print(f">> Original data:\n {X[['MEAN_AGE', 'HOUSE_VALUE']].get_values()
[:5]}")

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=5)

# Create an instance of SpatialLagTransformer
spatial_lag_transformer =
SpatialLagTransformer(spatial_weights_definition=weights_definition)

# Print the transformed data
X_spatial_lag = spatial_lag_transformer.fit_transform(X, y="MEDIAN_INCOME",
geometries="geometry")
print(f"\n>> Transformed data:\n {X_spatial_lag[:5, :]}")
```

The resulting output is a NumPy array with the spatial lag of the *`MEAN_AGE`* and *`HOUSE_VALUE`*.

```
>> Original data:
 [[4.75847626e+01 4.56300000e+05]
 [3.88231812e+01 8.36300000e+05]
 [4.78076096e+01 1.12630000e+06]
 [4.65636330e+01 9.60400000e+05]
 [5.11550865e+01 1.01090000e+06]]

>> Transformed data:
 [[4.03809292e+01 6.23460000e+05]
 [3.95882790e+01 8.20100000e+05]
 [4.69466225e+01 1.22280000e+06]
 [4.25439751e+01 1.04664000e+06]
 [4.43390564e+01 1.14368000e+06]]
```

Spatial Coordinates Transformer

The `SCoordTransformer` class takes input data containing a geometry column with geometries and produces a NumPy array containing the centroids of the geometries, which represent the x and y coordinates.

This transformer can be used to pass location information directly to a model.

The main methods of the class are described in the following table.

Method	**Description**
`fit`	Not yet implemented as it does not perform any calculations with the training data.
`transform`	Returns the XY coordinates of the geometries. In case of non-point spatial objects (such as lines and polygons), it returns the centroids of the geometries.
`fit_transform`	Calls the `fit` and `transform` methods sequentially with the training data.

See the SCoordTransformer class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *block_groups* SpatialDataFrame and the SCoordTransformer class to obtain the centroid's coordinates from a SpatialDataFrame. The geometries are specified in the geometry column.

```
from oraclesai.preprocessing import SCoordTransformer

# Define the variables of the training data
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "geometry"]]

# Use a referenced coordinate system
X = X.to_crs("epsg:3857")

# Print the given data
print(f">> Original data:\n {X['geometry'].head(5)}")

# Transform the data with the SCoordTransformer
coordinates = SCoordTransformer().fit_transform(X)

# Print the transformed data
print(f"\n>> Transformed data:\n {coordinates[:5, :]}")
```

The resulting output consists of the centroids of the geometries.

```
>> Original data:
                                              geometry
0  POLYGON ((-13175658.713 4010761.859, -13174935...
1  POLYGON ((-13175749.772 4004714.769, -13174771...
2  POLYGON ((-13179169.173 4002635.119, -13178970...
3  POLYGON ((-13177695.971 4003360.046, -13177503...
4  POLYGON ((-13177368.803 4002939.500, -13176993...

>> Transformed data:
 [[-13174765.1034151    4010231.26409032]
 [-13175173.61624862   4003637.47437617]
 [-13178654.77968995   4002868.5566815 ]
 [-13176298.82436636   4002826.86495246]
 [-13176753.58959072   4002684.55714192]]
```

Spatial Imputer

The SpatialImputer class allows us to fill the missing value of an observation using data from its neighbors.

According to Tobler's law, closer things are more related than the distant ones. Therefore, the goal is to leverage spatial weights to compute the missing values.

The following table describes the parameters of the SpatialImputer class.

Parameter	Description
spatial_weights_definition	Defines the relationship between the neighboring locations. It is necessary to retrieve information from the neighbors.
missing_values	All occurrences of missing_values will be imputed

Parameter	Description
`strategy`	By default, `SpatialImputer` uses “*mean*” to fill in the missing values. In other words, the weighted average from the neighboring observations replaces the missing values. The other options are "*median*", "*maximum*", and "*minimum*".

The `SpatialImputer` class is a transformer, and its main methods are described the following table.

Method	Description
`fit`	Calculates the spatial lag from the training data.
`transform`	Returns a NumPy array with the data parameters passed according to the specified `strategy`. It determines whether to use the neighbors from the training set by defining the `use_fit_lag` parameter.
`fit_transform`	Calls the `fit` and `transform` methods sequentially with the training data.

See the SpatialImputer class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` that was created earlier and performs the following:

1. Adds the missing values in the *`INTERNET`* column.
2. Defines the spatial weights using the K-Nearest Neighbors method.
3. Calls the fit_transform method of the SpatialImputer to fill in the missing values of the training set.

Note that the target column (*`MEDIAN_INCOME`*) and the column geometry are not part of the output.

```
import random
import numpy as np
from oraclesai import GeoDataFrameDataset
from oraclesai.preprocessing import SpatialImputer
from oraclesai.weights import KNNWeightsDefinition

random.seed(32)
block_groups_missing_df = block_groups.as_geodataframe()

# Assign missing values randomly to the internet column
ix = [row for row in range(block_groups.shape[0])]
for row in random.sample(ix, int(round(.1*len(ix)))):
    block_groups_missing_df.loc[row, "INTERNET"] = np.nan

# Create a SpatialDataFrame with the data containing missing values
block_groups_missing_pdf = 
SpatialDataFrame.create(GeoDataFrameDataset(block_groups_missing_df))

# Define the variables of the model
X = block_groups_missing_pdf[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", 
"INTERNET", "geometry"]]
```

```
# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Print the total number of missing values
print(f"Missing Values Before Imputation =
{np.sum(np.isnan(X.get_values()))}")

# Create an instance of SpatialImputer
spatial_imputer = SpatialImputer(missing_values=np.nan,
spatial_weights_definition=weights_definition)

# Fill the missing values of the training data
X_imputed = spatial_imputer.fit_transform(X, y="MEDIAN_INCOME")

# Print the total number of missing values (0 is expected)
print(f"Missing Values After Imputation = {np.sum(np.isnan(X_imputed))}")
```

The resulting output shows the number of missing values before and after imputation.

```
Missing Values Before Imputation = 344
Missing Values After Imputation = 0
```

Splitting Datasets

Spatial AI provides two ways of splitting a dataset into different subsets.

The following sections describe both the supported methods for splitting a dataset:

Using the `SpatialDataFrame.split` Function

A `SpatialDataFrame` can be split into two or more subsets by calling the `split` method. The `split` method takes a tuple containing the size ratio of each subset.

The number of elements contained in the ratio tuples dictates the number of subsets `SpatialDataFrames` returned.

See the SpatialDataFrame.split method in *Python API Reference for Oracle Spatial AI* for more information.

The following example splits the given `SpatialDataFrame` into train, test, and validation subsets, each containing 50%, 30%, and 20% of the number of elements of the original `SpatialDataFrame`, respectively.

```
# Print the size of the SpatialDataFrame defined as X
print(f"\n>> X (shape):\n {X.shape}")

# Split X into smaller SpatialDataFrames
X_train, X_test, X_validation = X.split(ratio=(0.5, 0.3, 0.2))

# Print the size of the resulting datasets
print(f"\n>> X_train (shape):\n {X_train.shape}")
print(f"\n>> X_test (shape):\n {X_test.shape}")
print(f"\n>> X_validation (shape):\n {X_test.shape}")
```

The output for the preceding example is:

```
>> X (shape):
(3437, 5)

>> X_train (shape):
(1718, 5)

>> X_test (shape):
(1031, 5)

>> X_validation (shape):(688, 5)
```

Using the `spatial_train_test_split` Function

The `spatial_train_test_split` function receives an instance of the `SpatialDataFrame` class and splits it into the training and test subsets.

Each subset is divided into the explanatory variables `X`, geometries, and target variable `y`. `X` is a vector of (`n`-samples * `n`-features), while geometry and `y` are vectors of `n`-samples. The training subsets can then be further split into training and validation subsets `SpatialDataFrame`by calling the same function.

See the spatial_train_test_split function in *Python API Reference for Oracle Spatial AI* for more information.

The following example splits the data stored in the *`block_groups`* `SpatialDataFrame` into two variables. *`X_train`* contains 90% of the original data, and *`X_test`* contains the remaining 10%. The proportion is indicated in the *`test_size`* parameter.

```
from oraclesai.preprocessing import spatial_train_test_split

# Define variables
X = block_groups_missing_pdf[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE",
"INTERNET", "geometry"]]

# Print the size of the data
print(f"\n>> X (shape):\n {X.shape}")

# Split the data into training and test sets, using 10% for testing
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.1)

# Print the size of the training and test sets
print(f"\n>> X_train (shape):\n {X_train.shape}")
print(f"\n>> X_test (shape):\n {X_test.shape}")
```

The code prints the original size of the data and the size of the two subsets from the split. The number of features in both subsets remains the same after the split.

```
>> X (shape):
 (3437, 5)

>> X_train (shape):
 (3093, 5)
```

```
>> X_test (shape):
 (344, 5)
```

5

Perform Spatial Analysis

Learn about the Oracle Spatial AI modules that help you to run spatial analysis.

This component contains the following five modules:

Topics:

Oracle Spatial Based Analysis

Spatial AI provides a basic Python API for geometry data in Oracle Spatial database.

This enables Spatial database-based analysis using the Python API provided by the `oraclesai.data` package which is implemented in the `SpatialDataFrame` class. It provides data access to Oracle Spatial and also data processing functionalities from Oracle Spatial.

These analyses include basic spatial operations, such as geometry computation, spatial query, aggregation, summary, join, and optimization. They are provided by Python calls to the Oracle Spatial database, which also enables in-database processing.

If the dataset is from the database, then the methods of the `SpatialDataFrame` class will push all the operations to Oracle Spatial and do in-database processing. Otherwise, all the operations are executed in memory.

The following table describes some of the techniques for spatial analysis.

Operation	Description
`aggregate`	Operation is executed over columns, and the supported functions are `count`, `avg`, `sum`, and `mbr`. The result is a `SpatialDataFrame` object.
`area`	Returns a new `SpatialDataFrame` instance containing the areas of each geometry.
`buffer`	Constructs a buffer around each geometry. The result contains the same rows and columns but with the geometry buffered.
`combine`	Combines the geometries from the current `SpatialDataFrame` instance with the geometries from the parameter `tgt`. The result is a `SpatialDataFrame` object with the combined geometries.
`crs`	Returns the coordinate reference system associated to the geometry.

Operation	Description
`groupby`	Involves splitting a `SpatialDataFrame` object by the given criteria, applying a function, and combining the results.
`length`	Returns a new `SpatialDataFrame` object with the lengths of the geometries.
`merge`	Joins two instances of `SpatialDataFrame` by comparing the joining keys. The result is another `SpatialDataFrame` with columns from both the instances. It allows to specify the joining keys and how the join is executed. It is also possible to define the geometry column for the resulting object.
`nearest_neighbors`	Returns a `SpatialDataFrame` object with observations that are closer to a given location, which can be either a shapely geometry or another instance of `SpatialDataFrame`. For the latter, the result will contain observations containing information from both `SpatialDataFrame` objects.
`relate`	Executes a primary spatial filter based on a specified spatial operator from {`anyinteract`, `inside`, `contains`, `equal`, `coveredby`, `on`, `covers`, `overlapbyintersect`, `overlapbydisjoint`}. It returns a new `SpatialDataFrame` instance.
`spatial_join`	Joins two instances of `SpatialDataFrame` by a specified interaction defined in the `spatial_op` parameter, which can contain any spatial interaction supported by the relate operation. The resulting object contains data from both input objects and the geometry from the calling instance is used as the geometry in the result.
`total_bounds`	The `total_bounds` property calculates the minimum bounding rectangle enclosing all the data. It returns the result as a tuple with the following elements `(min_x, min_y, max_x, max_y)`.
`to_crs`	Returns a new `SpatialDataFrame` object with the geometries in the specified CRS
`within_distance`	Returns a `SpatialDataFrame` object containing only observations located within a certain distance from a query window specified as a shapely query window or another.
`distance`	Returns a new `SpatialDataFrame` with the distance between the geometries of the current instance and the parameter `qry_win`.

See the SpatialDataFrame class in *Python API Reference for Oracle Spatial AI* for more information.

The following example creates an instance of `SpatialDataFrame` containing the name and location of schools in Los Angeles. It uses a *`DBSpatialDataset`* to get the data from the *`schools`* table.

```
import oml
from oraclesai import SpatialDataFrame, DBSpatialDataset

schools = SpatialDataFrame.create(DBSpatialDataset(table='schools',
schema='oml_user'))
```

Then, using the *`block_groups`* `SpatialDataFrame`, the `within_distance` and the `groupby` operations, the example computes the number of schools within two kilometers of each block group, and stores the result in another `SpatialDataFrame`. The index comes from the column *`GEOID`*.

```
schools_counts = block_groups.within_distance(schools,
distance=2000).groupby('GEOID').aggregate(count={'GEOID': 'SCHOOLS_COUNT'})
print(schools_counts["SCHOOLS_COUNT"])
```

By printing the resulting `SpatialDataFrame`, note the *`SCHOOLS_COUNT`* column, which indicates the number of schools within two kilometers from a block group.

```
      SCHOOLS_COUNT                                         geometry
0                 5  POLYGON ((-118.29131 34.26285, -118.29132 34.2...
1                 5  POLYGON ((-118.30075 34.25961, -118.30229 34.2...
2                 4  POLYGON ((-118.30076 34.26321, -118.30075 34.2...
3                 4  POLYGON ((-118.30320 34.27333, -118.30275 34.2...
4                 5  POLYGON ((-118.29069 34.27071, -118.29078 34.2...
...             ...                                              ...
3412             15  POLYGON ((-118.34312 33.99558, -118.34343 33.9...
3413             13  POLYGON ((-118.34930 33.99942, -118.34929 33.9...
3414             14  POLYGON ((-118.34432 33.99074, -118.34486 33.9...
3415             26  POLYGON ((-118.25165 34.08038, -118.25124 34.0...
3416             22  POLYGON ((-118.51849 34.18498, -118.51849 34.1...
```

Spatial Colocation Analysis

Spatial colocation measures and analyzes relationships between point features of different classes from the same spatial layer and, most often, from different spatial layers.

A typical example is determining whether different restaurants, such as McDonald's and Chipotle, are colocated. Further analysis is needed to identify whether McDonald's restaurants are colocated with metro stations and shopping centers and how they relate to population density and income levels. These help companies in site selection, site optimization, and also to minimize costs.

Colocation analysis is a tool that measures proximity patterns between two categories of point features, A and B, using the Local Colocation Quotient (LCLQ) statistic. For each feature of the Category of Interest (category A), it calculates its LCLQ score.

- Points of category A with a LCLQ score greater than one are more likely (than random) to have points of category B within their neighborhood.
- Points of category A with a LCLQ score less than one are less likely (than random) to have points of category B within their neighborhood.

- A point with a LCLQ score equal to one indicates that the proportion of categories within its neighborhood represents the proportion of the categories throughout the entire study area.

The LCLQ score indicates if a feature point is colocated, isolated, or undefined. The following table describes the possible scenarios.

LCLQ Type	Description
Colocated - Significant	The LCLQ score is greater than `1` with either a p-value less than `0.05` or a p-value of `None`.
Colocated – Not Significant	The LCLQ score is greater than `1` with a p-value equal to or greater than `0.05`.
Isolated - Significant	The LCLQ score is equal to or less than `1` with either a p-value less than `0.05` or a p-value of `None`.
Isolated – Not Significant	The LCLQ score is equal to or less than `1` with a p-value equal to or greater than `0.05`.
Undefined	The feature point did not have any neighbors from the neighboring category.

The colocation relationship is not symmetric. The LCLQ scores calculated when comparing category A to category B will be different than the LCLQ scores calculated when comparing category B to category A.

Some of the parameters required to execute colocation analysis are described in the following table.

Parameters	Description
`feature_data`	Data with point features.
`spatial_weights_definition`	Defines the relationship between neighboring locations. It is necessary to retrieve information from the neighbors.
`interest_category`	Two values that indicates the field and value of the category of interest. If the `interest_category` and `neighbor_category` parameters are defined, then the colocation analysis is executed using these values from the data specified in `feature_data`.
`neighbor_category`	Two values that indicates the field and value of the neighboring category.
`neighbor_feature_data`	If defined, ignores the `interest_category` and `neighbor_category` parameters. The category of interest is the point features in `feature_data`, while the other category is the point features defined in this parameter.
`n_permutations`	The number of permutations used to calculate the significance level of the colocation quotient scores. If `None`, then the significance level is not computed, and `None` is returned. Increasing the number of permutations also increases the processing time.
`is_time_window_analysis`	A Boolean parameter indicating if time-window analysis is required.
`interest_time_window`	Indicates the field, `start-time`, and `end-time` of the category of interest.

Parameters	Description
`neighbor_time_widow`	Indicates the field, `start-time`, and `end-time` of the neighboring category.

The result of the colocation analysis consists of a Pandas `DataFrame` with the following columns:

Column	Description
`row_index`	The observation's index in the `DataFrame` containing the category of interest.
`lclq`	The LCLQ score.
`t_stat`	The t-statistic associated with the LCLQ score.
`p_value`	The p-value associated with the LCLQ score, indicating the significance level of the LCLQ score.
`lclq_type`	The colocation type (as described in the earlier table).

See the spatial_colocation_analysis function in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`schools`* `SpatialDataFrame` and splits the data into two instances of `SpatialDataFrame` - `X` and `Y`, which represent two different classes and then executes a colocation analysis between the two classes. It requires to define the spatial weights since colocation analysis uses neighboring locations to calculate the LCLQ scores.

```
from oraclesai.weights import KernelBasedWeightsDefinition
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.analysis import spatial_colocation_analysis

# Split the data to create two different classes.
X, Y, _, _, _, _ = spatial_train_test_split(schools, y="", test_size=0.3, 
random_state=32)

# Define spatial weights
spatial_weights_definition = KernelBasedWeightsDefinition(k=25, fixed=False, 
function="gaussian")

# Execute colocation analysis between the two classes
colocation_analysis = spatial_colocation_analysis(X, 
spatial_weights_definition, neighbor_feature_data=Y, n_permutations=20)

# Print the result
print(colocation_analysis[:10])
```

The preceding code prints the results of the colocation analysis for the first ten observations in X, which contains the LCLQ score and the significance level.

```
   row_index      lclq    t_stat   p_value             lclq_type
0          2  1.054835       NaN       NaN  COLOCATED_SIGNIFICANT
1          3  0.948375  4.358899  0.000338   ISOLATED_SIGNIFICANT
2          4  0.944819  4.358899  0.000338   ISOLATED_SIGNIFICANT
3          6  1.113996  4.358899  0.000338  COLOCATED_SIGNIFICANT
```

```
4           7  1.013247       NaN       NaN  COLOCATED_SIGNIFICANT
5           9  0.999595 -4.358899  0.000338   ISOLATED_SIGNIFICANT
6          10  1.078993 -4.358899  0.000338  COLOCATED_SIGNIFICANT
7          11  1.012721       NaN       NaN  COLOCATED_SIGNIFICANT
8          14  0.978001  4.358899  0.000338   ISOLATED_SIGNIFICANT
9          20  0.961042 -4.358899  0.000338   ISOLATED_SIGNIFICANT
```

Spatial Weights

Spatial weights are used to quantify spatial relationships for analysis and modeling in Spatial AI.

The spatial weights are represented as a weighted graph, where each observation in a dataset represents a node. If two nodes are neighbors in a geographic context, then there is an edge between them with a weight associated with it. In some cases, the weight is binary, indicating that both nodes are spatially connected; in other cases, the weight is based on the distance between the nodes. As it is a common practice to work with sparse graphs, the spatial weights are usually implemented with an adjacency list.

Many machine learning algorithms and spatial analytics, such as spatial autocorrelation statistics and regionalization algorithms, rely on spatial weights in Spatial AI. The following table describes the spatial weights supported in Spatial AI.

Spatial Weights	Description
`KNNWeightsDefinition`	It is based on proximity and defines a spatial relationship of K-nearest neighbors.
`KernelBasedWeightsDefinition`	It uses a kernel function to define spatial relationships, and it is a decay function, which means that closer neighbors have larger values while further neighbors have smaller ones. • The `bandwidth` parameter is the distance used for the kernel function. • The `fixed` parameter indicates if the bandwidth is constant for all the observations. • The `function` parameter is the kernel that will be used; this can be one of the following values: {*`'triangular', 'uniform', 'quadratic', 'gaussian'`*}. • The parameter `k` is required to estimate the bandwidth, representing the number of neighbors for each observation.
`DistanceBandWeightsDefinition`	Uses the distance between two nodes as the weight for the edge connecting them. If the binary parameter is `True`, then the weight is `1` for all the neighbors at a distance less than the threshold value. Otherwise, it uses the decay parameter `alpha` to estimate the weight. By default, it uses the Euclidean distance metric.
`RookWeightsDefinition`	The relationship is based on contiguity, where two geometries are neighbors if they share at least one edge.
`QueenWeightsDefinition`	This definition is based on contiguity. Two geometries with a common vertex are neighbors with a weight of `1`.

The type of spatial weights depends on the problem scenario to be resolved. In the case of spatial weights not based on contiguity, the data must be in a projected referenced system. See the oraclesai.weights module in *Python API Reference for Oracle Spatial AI* for more information.

The following example shows how to create an instance of the `SpatialWeights` class from the *`block_groups`* `SpatialDataFrame` using the `KNNWeightsDefinition` and `SpatialWeights.create` functions.

```
from oraclesai.weights import SpatialWeights, KNNWeightsDefinition

spatial_weights = SpatialWeights.create(block_groups, 
KNNWeightsDefinition(k=5))

print(f"neighbors' indexes: {spatial_weights.neighbors[0]}")
print(f"neighbors' weights: {spatial_weights.weights[0]}")
```

The preceding code prints the neighbors' indexes of the first observation and their weights. In this case, all the weights are binary. The neighbors and their weights can be referenced using the `neighbors` and `weights` properties respectively.

```
neighbors' indexes: [2806, 81, 1717, 80, 1916]
neighbors' weights: [1.0, 1.0, 1.0, 1.0, 1.0]
```

Spatial Autocorrelation

Spatial autocorrelation is a metric that measures the relationship of a variable in a particular location with the same variable in other locations.

In practice, it is commonly calculated using neighboring observations to identify if the location influences the variable's value.

A positive spatial autocorrelation of a particular variable indicates the similarity of that variable among neighboring observations, meaning that similar values tend to be together. For instance, you can consider house prices as an example because the location influences a house price, causing neighboring houses to have similar prices.

A negative spatial autocorrelation indicates that neighboring observations have dissimilar values, causing a checkerboard pattern or the presence of spatial variance across the region. This is less common in social phenomena. One example is the distribution of supermarkets of different brands, or of hospitals. To avoid direct competition, the distribution of the supermarks or hospitals should be away from each other to have a better spatial coverage. In other words, it follows a pattern of negative spatial dependence.

Global Spatial Autocorrelation

The global spatial autocorrelation is a way to measure the overall trend followed by the values of a certain variable across different locations.

The Moran's I statistic is a common way to calculate the global spatial autocorrelation and is defined by the following formula.

$$I = \frac{n}{\sum_i \sum_j W_{ij}} \frac{\sum_i \sum_j W_{ij} Z_i Z_j}{\sum_i Z_i^2}$$

In the preceding formula, `n` is the number of observations, `W` is the spatial weights matrix, and `Z` is the standardized variable of interest.

A positive value of Moran’s I statistic indicates the presence of clusters where similar values tend to be together, reflecting the effect of spatial dependence. In contrast, a negative value of Moran’s I statistic suggests the presence of a checkerboard pattern or spatial variance where neighboring observations have dissimilar values, reflecting the effect of spatial heterogeneity.

Oracle Spatial AI provides the `MoranITest.create` function as part of `oraclesai.analysis`, which calculates the Moran’s I statistic for a given variable of a dataset.

See the MoranITest class in *Python API Reference for Oracle Spatial AI* for more information.

The following code uses the `MoranITest.create` function to calculate the Moran’s I statistic of the *`MEDIAN_INCOME`* column from the `SpatialDataFrame` *`block_groups`*. The class uses spatial weights to obtain the values from neighboring locations, which must be passed as a parameter, along with the dataset and the column of interest.

```
from oraclesai.analysis import MoranITest
from oraclesai.weights import SpatialWeights, KNNWeightsDefinition

spatial_weights = SpatialWeights.create(block_groups["geometry"].values,
KNNWeightsDefinition(k=5))

moran_test = MoranITest.create(block_groups, spatial_weights,
column_name="MEDIAN_INCOME")

print(f"Moran's I = {moran_test.i}")
print(f"p-value = {moran_test.p_value}")
```

The preceding code prints the Moran’s I statistic and its p-value. The positive value of the statistic indicates the presence of clustering where locations of similar income tend to be together.

```
Moran's I = 0.652331479721869
p-value = 0.001
```

Local Spatial Autocorrelation

The local spatial autocorrelation measures the relationships between each observation and its surroundings.

Although Moran’s I statistic helps understand the overall behavior of a variable across different locations in the whole dataset, it does not explain the relationship between a specific observation and its surroundings. For example, a positive value of Moran’s I statistic indicates the presence of clusters but it does not indicate where the clusters are or explain the degree of similarity of a variable in a given location with its neighbors. The local spatial autocorrelation fills this gap as it explain the relationship between a specific observation and its surroundings.

The Local Moran's I statistic represents the local spatial autocorrelation of a specific observation and is given by the following formula.

$$I_i = \frac{nZ_i}{\sum_j Z_j^2} \sum_j W_{ij} Z_j$$

The Moran's I statistic is the sum of all local Moran's I divided by the spatial weights.

A positive local Moran's I statistic indicates similarity with neighboring locations; it can be either a high value surrounded by high values or a low value surrounded by low values.

A negative local Moran's I statistic represents variance with neighboring locations; it can be either a high value around low values or a low value surrounded by high values. This metric helps to identify outliers in the dataset.

The `LocalMoranITest.create` function inside `oraclesai.analysis` calculates the local Moran's I statistic of each observation in a dataset.

See the LocalMoranITest class in *Python API Reference for Oracle Spatial AI* for more information.

The following code uses the `LocalMoranITest.create` function to calculate the local spatial autocorrelation for the *`MEDIAN_INCOME`* column of each observation in the *`block_groups`* dataset. The class uses spatial weights to obtain the values from neighboring locations and compute the local Moran's I.

```
from oraclesai.analysis import LocalMoranITest
from oraclesai.weights import SpatialWeights, KNNWeightsDefinition

spatial_weights = SpatialWeights.create(block_groups["geometry"].values,
KNNWeightsDefinition(k=5))

local_moran_test = LocalMoranITest.create(block_groups, spatial_weights,
column_name="MEDIAN_INCOME")

print(f"Local Moran's I: {local_moran_test.i_list[:10]}")
print(f"p-values: {local_moran_test.p_values[:10]}")
```

The preceding code prints the Local Moran's I and the corresponding p-value of the first ten observations of the dataset.

```
Local Moran's I: [-0.09208001 -0.16105385  0.34887379  2.13410581
2.53000192  0.96564933
  0.77039582  1.04246212 -0.01040734 -0.11960612]
p-values: [0.28  0.069 0.011 0.011 0.001 0.054 0.023 0.102 0.342 0.19 ]
```

Metrics for Spatial Regression

Data analysis is essential to build better machine learning models, particularly for spatial regression. Some common tasks involve analyzing multicollinearity, normal distribution bias, nonstationarity or heterogeneity, and spatial autocorrelation.

After training a regression model, a variety of statistics-based metrics are available to assess the model results. This helps you to choose the best spatial model for the task at hand. The following table describes some of these statistics which you can access in the `oraclesai.metrics` module. All the methods receive a spatial regression model as a parameter.

Metric	Description
`koenker_bassett`	It is helpful to identify the presence of variance in the residuals, which can be caused by spatial heteroskedasticity, a particular type of heterogeneity.
`lm_error`	Lagrange multiplier test to identify if a regression algorithm that includes the spatial lag over the error term is needed.
`lm_lag`	Lagrange multiplier test to identify if a regression algorithm that includes the spatial lag over the target variable is required.
`rlm_error`	Robust Lagrange multiplier test for Spatial Error model.
`rlm_lag`	Robust Lagrange multiplier test for Spatial Lag model.
`moran_res`	Test correlation between residuals and the spatial lag of residuals. A positive and significant value indicates the presence of spatial clustering, where regions with similar values tend to be together, reflecting the effect of spatial dependence. A negative and significant value indicates the presence of spatial variance or the checkerboard pattern, reflecting the effect of spatial heterogeneity.
`log_likelihood`	Returns the log-likelihood of the regression model. It is a way to measure the model fit.
`aic`	The Akaike Information Criteria, AIC, estimates the amount of information loss by the model.
`jarque_bera`	It is a test for normality in the residuals of a spatial regression model.
`vif`	The variance inflation factor (VIF) is helpful to detect multicollinearity. Multicollinearity happens when a spatial regression model has a correlation between the explanatory variables. It measures how much the variance of an estimated regression coefficient increases because of collinearity. Sometimes, features with high multicollinearity with another feature should be removed from the spatial model.

See the oraclesai.metrics module in *Python API Reference for Oracle Spatial AI* for more information.

6

Apply Spatial Clustering

Learn and apply the different machine learning algorithms for spatial clustering using Oracle Spatial AI.

Topics:

About Spatial Clustering

Clustering allows you to identify patterns and understand the data distribution.

General clustering is the task of assigning a label to an observation so that observations with the same label share common characteristics. The same applies to spatial clustering algorithms, with the difference that these algorithms add a spatial context, so not only observations with the same label share common properties but they also are geographically connected.

This allows us to identify patterns and understand the data distribution. The same applies to spatial clustering algorithms, with the difference that these algorithms add a spatial context, so not only observations with the same label share common properties but they also are geographically connected.

LISA Hotspot

Local Indicators of Spatial Association (LISAs) are widely used to identify geographical clusters as well as finding geographical outliers. This clustering approach is called LISA Hoptspot clustering.

Possible use cases for this spatial clustering include finding hot spots of crime to help police to make staffing and patrolling decisions, identifying patterns of car accidents or pedestrian deaths to help optimize arrangements of red lights and road networks.

The LISA Hotspot clustering algorithm does local autocorrelation analysis and summarizes the co-variation between observations and their immediate surroundings. It allows us to identify areas of high values (hot spots) and areas of low values (cold spots). For each region, there are four different labels representing each of the quadrants.

1. HH (High-High). A High value surrounded by high values.
2. LH (Low-High). A low value surrounded by high values.
3. LL (Low-Low). A low value surrounded by low values.
4. HL (High-Low). A high value around low values.

The LISA Hotspot clustering algorithm computes a local Moran's I (that is, LISA) for each location.

- A location with a positive local Moran's I statistic indicates the presence of neighbors with similar values (either high or low values), representing hot or cold spots.
- A location with a negative local Moran's value indicates neighbor locations with different values; it can be a high value surrounded by low values or a low value surrounded by high values, representing spatial outliers.

The `LISAHotspotClustering` class implements the LISA Hotspot clustering, and the following table describes its parameters.

Parameters	Description
`column`	A number that indicates a column, with which the algorithm uses the data associated to run the LISA Hotspot clustering method. If the column is not defined, the algorithm expects a dataset with a single column.
`spatial_weights_definition`	Defines the relationship between neighboring locations. It is required to retrieve information from the neighbors.
`max_p_value`	Used to label regions with a p-value below this threshold value. For regions with p-values equal to or greater than this threshold value, the algorithm assigns the label `-1`.
`supported_quadrants`	A list indicating that only observations from these quadrants are labeled. Values indicate quadrant location: `1 (High-High)`, `2 (Low-High)`, `3 (Low-Low)`, `4 (High-Low)`. The remaining observations are assigned a label `-1`.
`seed`	Ensures reproducibility of conditional randomization.
`n_jobs`	The maximum number of concurrently running jobs.

Once an instance of `LISAHotspotClustering` is created, the clustering algorithm is executed by calling the `fit` method. The label assigned to each observation can be retrieved with the `labels_` property. The following table describes the main properties available from this class.

Parameters	Description
`labels_`	The label assigned to each observation. Labels indicate quadrant location: `1 (High-High)`, `2 (Low-High)`, `3 (Low-Low)`, `4 (High-Low)`. Depending on the parameters passed to the object, observations can have a label `-1`.
`regions_`	A dictionary representing observations with the same label that are geographically connected according to the spatial weights.
`Is`	The Local Moran's I of each observation.
`ps`	The p-value of the Local Moran's I of each observation.

See the LISAHotspotClustering class in *Python API Reference for Oracle Spatial AI* for more information.

The following example executes a hot spot analysis over the *`MEDIAN_INCOME`* column of the *`block_groups`* `SpatialDataFrame`. It does not require defining a target variable when passing the training data to the `fit` method. The `labels_` property returns the label assigned to each observation, with hot spots marked with `1`, cold spots with `3`, and outliers represented by `2` and `4`.

```
from oraclesai.weights import DistanceBandWeightsDefinition
from oraclesai.clustering import LISAHotspotClustering
```

```
# Define variables and CRS
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'geometry']].to_crs('epsg:3857')

# Create an instance of LISAHotspotClustering
lisa_model = LISAHotspotClustering(column="MEDIAN_INCOME",
max_p_value=0.05,
spatial_weights_definition=DistanceBandWeightsDefinition(threshold=2500))

# Train the model
lisa_model.fit(X)

# Print the labels
print(f"labels = {lisa_model.labels_[:10]}")
```

The program prints the labels of the first ten observations.

```
labels = [ 2  2  1  1  1  1 -1 -1 -1 -1]
```

DBSCAN with Regionalization

DBSCAN is a density-based clustering technique capable of finding clusters of different shapes and sizes from a large amount of data.

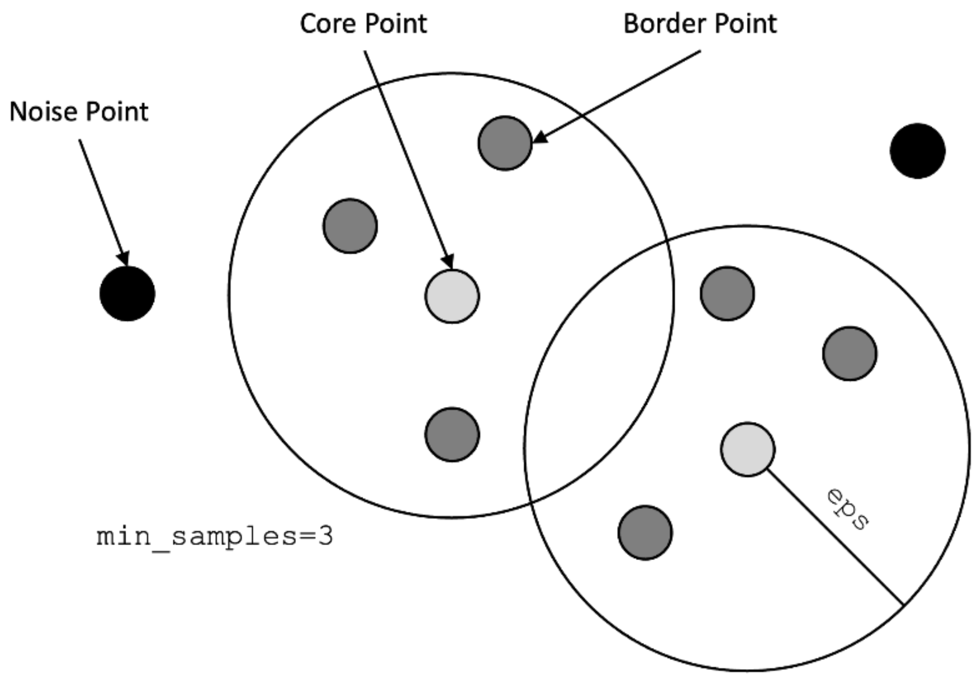
This algorithm does not require the number of clusters as a parameter. Instead, it uses the following parameters.

- `min_samples`: The minimum number of points required for a region to be considered a cluster.
- `eps`: The distance threshold for searching points in the neighborhood of a point.

The algorithm starts at any point. If at least `min_samples` points are within a radius of `eps`, then all the points in the neighborhood are considered part of the same cluster. The process is then repeated for all the points in the neighborhood. There are three types of points or observations:

- **Core Point:** At least has a `min_samples` number of points in its neighborhood within the radius `eps`.
- **Border Point:** It is reachable from a core point, but there are fewer than `min_samples` number of points within its neighborhood.
- **Noise Point:** It is neither a core point nor a border point. It is a point that is not reachable from any core points.

The following image is an example displaying the different types in the DBSCAN algorithm.



Standard DBSCAN clustering does not fully consider the observation's spatial location. When this algorithm is applied on spatial data, it often results in data points of a cluster dispersed across spatial regions. Regionalization is used to provide a spatial context to the DBSCAN algorithm, this way, observations of the same cluster are similar not only in their attributes, but also in their spatial location.

The DBSCAN algorithm with regionalization performs the following steps:

1. Creates an instance of the `DBScanClustering` class specifying the parameters: `min_samples`, `eps`, and `spatial_weights_definition`.
2. Calls the `fit` method passing the data as parameter to train the model.
3. The `labels_ property` indicates the label assigned to each observation. Noise points are labeled with -1. Use the labels and the location of each observation to visualize the clusters in a map.

If you do not provide the `eps` parameter, it is estimated automatically (see [1] for more details on `eps` estimation method). The initial `eps` value is estimated by:

- Calculating the Euclidean distance between each pair of neighboring locations using the K-nearest neighbors approach, where the value of k is equal to `min_samples`.
- Obtaining the distance to the nearest neighbor for each observation and sorting the distances in ascending order.
- Plotting the sorted distances to form an elbow curve.
- The estimated value of `eps` is the distance associated with the elbow's location, which is represented by the furthest point from the line that crosses the first and last points.

See the DBScanClustering class in *Python API Reference for Oracle Spatial AI* for more information.

The following code fits a DBSCAN model with training data from the *`block_groups`* `SpatialDataFrame`. The goal is to identify geographic areas with similar characteristics.

The clustering model is the final step of a spatial pipeline, which contains a preprocessing step to standardize the data. The geometry column is not considered a feature but it is used to compute the spatial weights.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.clustering import DBScanClustering
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define variables and CRS
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'geometry']].to_crs('epsg:3857')

# Create an instance of DBScanClustering
reg_dbscan = DBScanClustering(eps=0.9,
                              min_samples=5,

spatial_weights_definition=KNNWeightsDefinition(k=30))

# Add the model into a Spatial Pipeline with a preprocessing step
reg_dbscan_pipeline = SpatialPipeline([('scale', StandardScaler()),
('clustering', reg_dbscan)])

# Train the model
reg_dbscan_pipeline.fit(X)

# Print the labels
print(f"labels =
{reg_dbscan_pipeline.named_steps['clustering'].labels_[:20]}")
```

The preceding code prints the label assigned to the first 20 observations using the DBSCAN algorithm with regionalization.

```
labels = [ 0  0  0  0  0  0  0 -1  0  0  0  0  0 -1  0  0  0  0  0  0]
```

Agglomerative with Regionalization

Agglomerative clustering performs a hierarchical clustering using a bottom up approach.

In agglomerative clustering, initially there is one cluster for each observation. In each iteration, the two closest clusters are merged. The algorithm continues until any one of the following stopping criteria applies:

- Reaches a certain number of clusters.
- The distance between two clusters is larger than a certain threshold.

Standard Agglomerative clustering does not fully consider the observation's spatial location. When this algorithm is applied on spatial data, it often results in data points of a cluster dispersed across spatial regions. Regionalization is used to provide a spatial context to the agglomerative algorithm. By defining spatial weights, agglomerative with regionalization includes a spatial constraint in the clustering algorithm, so elements of the same cluster share common characteristics and are geographically connected.

See the AgglomerativeClustering class in *Python API Reference for Oracle Spatial AI* for more information.

The following table describes some of the properties of the `AgglomerativeClustering` class.

Parameters	Description
`n_clusters_`	The algorithm stops when reaching the specified number of clusters.
`distance_threshold`	The algorithm stops if the distance between the two closest clusters is greater than this value. If this parameter is defined, then it requires to set `n_clusters=None`.
`spatial_weights_definition`	Defines the relationship between neighboring locations. It is required to retrieve information from the neighbors. Only *KNN* and *DistanceBand* weights are supported.
`linkage`	The strategy followed to identify the distance between two clusters. The options are: • *`ward:`* The variance between two clusters. • *`average:`* The distance between the average of two clusters. • *`complete`*`:` The maximum distance between a pair of points from two distinct clusters. • *`single`*`:` The minimum distance between a pair of points from two distinct clusters.
`affinity`	The metric used to compute the distance.
`n_jobs`	The maximum number of concurrently running jobs.

The following code uses the *`block_groups`* `SpatialDataFrame` and `AgglomerativeClustering` to identify locations sharing common characteristics according to certain features. It uses regionalization to keep the clusters geographically connected.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.clustering import AgglomerativeClustering
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define training features
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'geometry']]

# Use geodetic reference systems to calculate distances.
X = X.to_crs('epsg:3857')
# Create an instance defining stopping criteria and spatial weights
reg_agglomerative = AgglomerativeClustering(n_clusters=6,
spatial_weights_definition=KNNWeightsDefinition(k=5))

# Create a spatial pipeline with preprocessing and clustering steps.
agglomerative_pipeline = SpatialPipeline([('scale', StandardScaler()),
('clustering', reg_agglomerative)])

# Train the model
agglomerative_pipeline.fit(X)

# Print the labels associated with each observation
print(f"labels =
{agglomerative_pipeline.named_steps['clustering'].labels_[:20]}")
```

The output are the labels associated to the first 20 observations of the data.

```
labels = [1 4 4 4 4 0 0 0 1 1 1 1 1 1 1 1 0 2 0 2]
```

K-Means

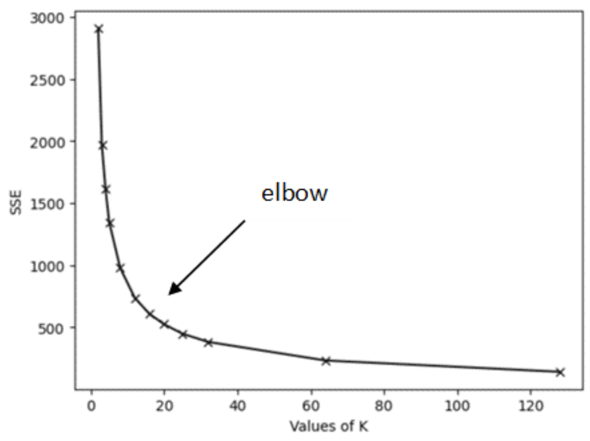
K-means is a widely used clustering algorithm. It is based on proximity.

It starts with a set of cluster centroids, and each observation is assigned to the closest centroid. Then, a cluster's centroid is updated with the average of the observations assigned to it. The process continues until the centroids are no longer updated, or until a maximum number of iterations is reached.

The `KMeansClustering` class does not support regionalization. This means that elements of the same clusters can be geographically disconnected. However, you can use K-means as a benchmark for comparison purposes. It can directly take an instance of `SpatialDataFrame` as input parameter for modeling, even though the spatial information is not leveraged. It can then be incorporated into the Spatial Pipeline.

The K-Means algorithm requires defining the number of clusters with the `n_clusters` parameter. But if this is not known, the `KMeansClustering` class provides two methods to estimate the number of clusters. The user can specify any of the following methods in the `init_method` parameter.

- **The Elbow method:** This strategy runs the K-Means algorithm for different values of K and keeps track of the sum squared error (SSE) for each one. By plotting the errors against the values of K, the optimal value of K is given by the graph's "*elbow*" as shown in the following figure:



- **The Silhouette method:** This method runs the K-Means algorithm for different values of K and measures the Silhouette score for each run. It returns the value of K associated with the greatest score. The Silhouette score measures how well each observation lies within its cluster. The measure is in the range `[-1, 1]`, and it can be interpreted as follows:
  - A silhouette coefficient near `1` indicates that the observation is far from the neighboring clusters.
  - A value of `0` suggests that the observation is close to or on the decision boundary between two adjacent clusters.

  – A negative value indicates that the observation might have been assigned to the wrong cluster.

If the `n_clusters` parameter is not defined, then the algorithm uses the elbow method to estimate it. See the KMeansClustering class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`blocks_groups`* `SpatialDataFrame` and the `KMeansClustering` class to identify clusters based on specific features.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.clustering import KMeansClustering
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define training features
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'geometry']]

# Create an instance of KMeansClustering with K=5
kmeans_model = KMeansClustering(n_clusters=5)

# Create a spatial pipeline with preprocessing and clustering steps.
kmeans_pipeline = SpatialPipeline([('scale', StandardScaler()),
('clustering', kmeans_model)])

# Train the model
kmeans_pipeline.fit(X)

# Print the labels associated with each observation
print(f"labels = {kmeans_pipeline.named_steps['clustering'].labels_[:20]}")
```

The output consists of the labels of the first 20 observations.

```
labels = [3 1 3 1 3 3 1 1 3 3 2 2 1 3 2 2 1 1 1 1]
```

7

Apply Spatial Anomaly Detection

Learn and apply the Local Outlier Factor (LOF) algorithm for spatial anomaly detection using Oracle Spatial AI.

Topics:

About Spatial Anomaly Detection

Anomaly detection identifies outliers and novelties, defined as observations that are significantly different from the others.

Also, note the following:

- Outlier detection estimators identify regions where the training data is concentrated, ignoring the deviant observations.
- Novelty detection identifies whether a new or unseen observation is an outlier according to an already defined training set.

Spatial anomaly detection identifies geographically isolated observations using spatial weights with standard anomaly detection methods. Examples include analyzing all environmental or traffic monitoring sensor data to find anomalies, which can lead to identifying dysfunctional sensors.

Local Outlier Factor

Local Outlier Factor (LOF) measures the LOF score for each observation, representing the local deviation of the density of that observation concerning its neighbors.

The LOF score depends on how isolated an observation is with respect to the surrounding neighborhood. The larger the LOF score, the more isolated is the observation.

Using the k-nearest neighbors, the algorithm compares the local density of a sample to the local densities of its neighbors. Those samples with a significantly lower density than their neighbors are considered outliers.

In a spatial context, the LOF score helps to identify geographically isolated samples. For example, analyzing locations with a high concentration of car accidents and labeling isolated accident locations as outliers, which can be targeted for further examination.

The LOF method (see [2] for more details on the `LOF` method) consists of the following steps:

1. Define a method to measure the distance between two observations - according to either features or geography. If spatial weights are defined, then the distance comes from the corresponding weight, except for binary weights, where the distance is calculated based on the geometries.
2. Define a method that computes the k-distance of an observation. The k-distance is the distance to the furthermost neighbor or the K-th neighbor from KNN. If spatial weights are

defined, then the neighboring observations are obtained according to the spatial weights, and the distance between two observations comes from the corresponding weight, except for binary weights, where the distance is calculated based on their geometries.

3. Define a method to compute the reachability-distance between two observations.

$$RD(i,j) = \max(Kdistance(i), distance(i,j))$$

4. Compute the Local Reachibility Distance (LRD) of each observation according to the following formula, where Nb_i presents the neighbors of the i-th observation. If spatial weights are defined, then the neighbors of the i-th observation come from the spatial weights. Otherwise, the algorithm uses the k-nearest neighbors method.

$$LRD(i) = \frac{|Nb_i|}{\sum_{j \in Nb_i} RD(i,j)}$$

5. Compute the LOF score of each observation.

$$LOF(i) = \frac{\frac{\sum_{j \in Nb_i} LRD(j)}{LRD(i)}}{|Nb_i|}$$

Using `LocalOutlierFactor` for novelty detection requires setting the `novelty` parameter to *`True`* and calling the `predict` method to identify whether the unseen or new data are outliers.

See the LocalOutlierFactor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses a dataset based on the report of accidents in a city. It contains the location and severity of the car accidents. The example first creates an instance of `SpatialDataFrame` based on a database table.

```
from oraclesai import SpatialDataFrame, DBSpatialDataset
import oml

accidents_pdf = 
SpatialDataFrame.create(DBSpatialDataset(table='chicago_accidents', 
schema='oml_user'))
```

The goal is to identify outliers in accidents where people get injured based on location. The dataset contains a categorical variable, *`INJURY_RATING`*, that indicates the severity of the car accident. The example focuses on accidents with *`INJURY_RATING`* greater than or equal to 3.

The following code uses the `LocalOutlierFactor` to calculate the LOF score of each observation based on the neighboring locations according to the `spatial_weights_definition` parameter.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.outliers import LocalOutlierFactor
```

```
# Get records with INJURY_RATING >= 3
accidents_injury_pdf = accidents_pdf[accidents_pdf["INJURY_RATING"] >= 3]

# Keep columns INJURY_RATING and geometry, and use a geodetic coordinate
system
X = accidents_injury_pdf[["INJURY_RATING", "geometry"]].to_crs("epsg:3857")

# Create an instance of the LOF model defining the spatial weights
slof_model =
LocalOutlierFactor(spatial_weights_definition=KNNWeightsDefinition(k=20))

# Train the model
slof_model.fit(X)

# Get and print the LOF scores for each observation
slof_scores = -1 * slof_model.negative_outlier_factor_
print(slof_scores[:10])
```

The program prints the LOF score of the first 10 observations. Note that the `negative_outlier_factor_` property returns the negative LOF score.

```
[1.12403072 1.49269168 1.18196622 1.23728049 0.89957071 1.00487086
 1.03445893 0.98740889 1.01636585 1.00944292]
```

8

Apply Spatial Regression

Learn and apply Oracle Spatial AI machine learning algorithms for spatial regression.

Topics:

About Spatial Regression

Spatial regression consists of predicting the value of a continuous variable based on input data that is derived by identifying relationships between independent variables and a target variable while considering a geographical context.

For example, a house price is impacted by the prices of nearby houses, and so including this spatial effect in a regression model can help make more accurate predictions on the house price. Spatial regression is essential in geographic applications and the following lists a few more scenarios where it can be applied:

- Predict house prices based on census data and location information.
- Choose a house, considering its proximity to economic opportunities, schools, health care, and roadways for daily commutes.
- Predict the median income of a specific region based on neighboring locations.

Depending on the nature of the data and the task, you should choose one of the algorithms that works best for you. Oracle Spatial AI also provides tools to decide which algorithm to use and suggests the resulting machine learning algorithm that better fits the data.

Spatial Diagnostics Using OLS

The first step in spatial modeling is to do some spatial diagnostics, such as analyzing multicollinearity, normal distribution bias, spatial heterogeneity, and spatial dependence. You can do this using the Ordinary Least Squre (OLS) model.

The OLS algorithm fits a line that minimizes the Mean Squared Error (MSE) from the training set to predict new values. The formula of the ordinary linear regression is given as shown:

$$y_i = \alpha + \sum_{j=1}^{m} \beta_j X_{ij} + \epsilon_i,$$

In the preceding formula: α is the intercept or constant parameter, β is a vector of parameters that give us information about to what extent each variable is related to the target variable *y* and ε_i represents the error. The goal when training an OLS model is to estimate the parameters α and β to predict values of *y* for new values of *X*.

The `OLSRegressor` class in Spatial AI adds the `spatial_weights_definition` parameter in the general OLS model, which allows you to get spatial statistics after training the model. These statistics help identify the presence of spatial dependence or spatial heterogeneity and determine if another algorithm is required. So `OLSRegressor` intends to help user diagnose the data to see if there are any special spatial relationships, which in turn helps decide which spatial regression algorithm to use for the specific task. See Metrics for Spatial Regression for more information on the statistics.

The following table describes the main methods of the `OLSRegressor` class.

Method	Description
`fit`	Trains the OLS model from the given training data and obtains spatial statistics if the `spatial_weights_definition` parameter is specified.
`predict`	Uses the trained parameters to estimate the target variable of the given data.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	The R-squared statistic for the given data.

See the OLSRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and creates an OLS model defining the `spatial_weights_definition` parameter. After training the model, it calls the `predict` and `score` methods. Finally, the program prints a summary of the model containing the spatial statistics.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.regression import OLSRegressor
from oraclesai.weights import KNNWeightsDefinition

# Define the training and test set.
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "INTERNET",
"geometry"]]
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2)

# Create the OLSRegressor defining the spatial_weights
spatial_ols_model = OLSRegressor(KNNWeightsDefinition(k=10))

# Train the model and specify the target variable
spatial_ols_model.fit(X_train, "MEDIAN_INCOME")

# Print the predictions of the test set
ols_predictions_test =
spatial_ols_model.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {ols_predictions_test[:10]}")

# Print the R-squared score of the test set
ols_r2_score = spatial_ols_model.score(X_test, y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {ols_r2_score}")

# Prints a summary of the model
print(spatial_ols_model.summary)
```

The program output includes the following:

- The `predict` method returns the estimated values of the target variable over the test set.
- The `score` method returns the R-squared metric of the model from the test set.
- The `summary` property provides multiple statistics and the parameters associated with each explanatory variable. Also, it includes spatial statistics based on the `spatial_weights_definition` parameter.

```
>> predictions (X_test):
 [84333.95556955 88819.9988673  52445.40662329 66192.50638257
 66613.63752196 53802.16810985 65151.54020825 29424.26087764
 37296.49147829 85676.22038382]

>> r2_score (X_test):
 0.6009367861353069
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of
Observations:        2750
Mean dependent var  :  70051.6531                Number of
Variables   :          4
S.D. dependent var  :  40235.8666                Degrees of
Freedom    :        2746
R-squared           :      0.6385
Adjusted R-squared  :      0.6381
Sum squared residual:1608810557754.003                F-
statistic           :   1616.7374
Sigma-square        :585874201.658                Prob(F-
statistic)     :          0
S.E. of regression  :   24204.838                Log likelihood       :
-31659.426
Sigma-square ML     :585022021.001                Akaike info criterion :
63326.852
S.E of regression ML:  24187.2285                Schwarz criterion     :
63350.529

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     t-Statistic
Probability
------------------------------------------------------------------------------
```

```
------
            CONSTANT    -61472.5132881    3718.3646558     -16.5321368
0.0000000
            MEAN_AGE     798.8367637      94.5268152       8.4509011
0.0000000
         HOUSE_VALUE       0.0558167       0.0014696      37.9805274
0.0000000
            INTERNET    85961.1765144    3867.7192944      22.2252883
0.0000000
------------------------------------------------------------------------------
------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER          20.388

TEST ON NORMALITY OF ERRORS
TEST                             DF         VALUE           PROB
Jarque-Bera                       2          955.683           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF         VALUE           PROB
Breusch-Pagan test                3         1198.122           0.0000
Koenker-Bassett test              3          526.447           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF        VALUE           PROB
Moran's I (error)              0.2395         29.493           0.0000
Lagrange Multiplier (lag)         1          426.035           0.0000
Robust LM (lag)                   1            4.674           0.0306
Lagrange Multiplier (error)       1          854.940           0.0000
Robust LM (error)                 1          433.579           0.0000
Lagrange Multiplier (SARMA)       2          859.614           0.0000

================================ END OF REPORT
=====================================
```

Spatial Cross-Regressive Model

The Spatial Cross-Regressive (SLX) regression model executes a regular liner regression involving a feature engineering step to add features that provide a spatial context to the data.

This is according to Tobler's law that closer things are more related than distant things. The algorithm adds one or more columns with the spatial lag of certain features, representing the average from neighboring observations.

The `SLXRegressor` class requires the definition of the spatial weights with the `spatial_weights_definition` parameter to establish how the neighboring observations interact. The following table describes the main methods of the `SLXRegressor` class.

Method	Description
`fit`	The parameters of the `fit` method are the same for most of the regression algorithms, except for the `column_ids` parameter, which specify the columns that are used to compute the spatial lag. The algorithm estimates the parameters of the explanatory variables plus the parameters associated with those added with the spatial lag.
`predict`	The `predict` method calculates the spatial lag of the dataset using the same columns defined in the `fit` process and returns the value of the OLS equation evaluated in the extended dataset. By setting the `use_fit_lag=True` parameter, the algorithm calculates the spatial lag from the training set. This is helpful when the prediction dataset contains few observations.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the R-squared statistic for the given data. By setting the `use_fit_lag=True` parameter, the algorithm calculates the spatial lag from the training set. Otherwise, it computes the spatial lag from the provided data.

See the SLXRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the `SLXRegressor` class to train an SLX regression model with training data (`X_train`) using the `MEDIAN_INCOME` column as the target variable. The `MEAN_AGE`, `MEAN_EDUCATION_LEVEL`, and `HOUSE_VALUE` columns are used to calculate the spatial lag.

Using the test set (`X_test`), the code calls the `predict` and `score` methods to estimate the values of the target variable and the R-squared metric respectively.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SLXRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define the explanatory variables
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'INTERNET', 'geometry']]

# Define the training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Create a SXL Regressor model
slx_model = SLXRegressor(spatial_weights_definition=weights_definition)

# Add the model to a pipeline along with a preprocessing step
slx_pipeline = SpatialPipeline([('scale', StandardScaler()),
('slx_regression', slx_model)])

# Train the model
```

```
slx_pipeline.fit(X_train, "MEDIAN_INCOME",
slx_regression__column_ids=["MEAN_AGE", "MEAN_EDUCATION_LEVEL",
"HOUSE_VALUE"])

# Print the predictions with the test set
slx_predictions_test =
slx_pipeline.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {slx_predictions_test[:10]}")

# Print the score with the test set
slx_r2_score = slx_pipeline.score(X_test, y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {slx_r2_score}")
```

The program produces the following output:

```
>> predictions (X_test):
 [102070.14467552 103393.34495125  18080.13247972  28780.88885959
 166553.11466239  47847.19216301  97311.05264284  28621.06664768
  86030.99787827  18315.17778001]

>> r2_score (X_test):
 0.6520502048458249
```

Note that printing the property summary of the trained model displays new parameters which are associated with the spatial lag of the columns specified in the training process.

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of
Observations:        2750
Mean dependent var  :  69703.4815                Number of
Variables    :          8
S.D. dependent var  :  39838.5789                Degrees of
Freedom    :        2742
R-squared           :      0.6404
Adjusted R-squared  :      0.6395
Sum squared residual:1569034862694.453                F-
statistic           :    697.5148
Sigma-square        :572222779.976                Prob(F-
statistic)     :          0
S.E. of regression  :   23921.178                Log likelihood        :
-31625.004
Sigma-square ML     :570558131.889                Akaike info criterion :
63266.007
S.E of regression ML:  23886.3587                Schwarz criterion     :
63313.362

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     t-Statistic
Probability
```

```
------------------------------------------------------------------------------
------
             CONSTANT    69454.0719691     458.7116277     151.4111868
0.0000000
             MEAN_AGE    3407.3842392     632.8239483       5.3844110
0.0000001
MEAN_EDUCATION_LEVEL    11619.0976034    1254.9099676       9.2589093
0.0000000
          HOUSE_VALUE    20550.0723247     970.0583796      21.1843666
0.0000000
             INTERNET    10089.1251192     670.1690078      15.0545982
0.0000000
         SLX-MEAN_AGE     106.5803082     136.9729582       0.7781120
0.4365701
SLX-MEAN_EDUCATION_LEVEL    -995.5040769     172.6431756
-5.7662521       0.0000000
      SLX-HOUSE_VALUE       3.1809763     136.4013684       0.0233207
0.9813962
------------------------------------------------------------------------------
------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            9.435

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2        1258.500           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test                7        1083.843           0.0000
Koenker-Bassett test              7         436.439           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Moran's I (error)              0.2586        31.945           0.0000
Lagrange Multiplier (lag)         1        1044.952           0.0000
Robust LM (lag)                   1          55.266           0.0000
Lagrange Multiplier (error)       1         997.181           0.0000
Robust LM (error)                 1           7.495           0.0062
Lagrange Multiplier (SARMA)       2        1052.447           0.0000

================================ END OF REPORT
=====================================
```

Spatial Lag Model

The presence of spatial dependence indicates that values of observations are related to each other through distance, and the spatial lag model that includes this dependence is expected to perform better. The spatial lag model is also known as Spatial Autoregressive Model (SAR).

The SAR model considers spatial dependence over the target variable, meaning that the value of a region's target variable is related to its neighbors' target variable.

The SAR model includes the spatial lag of the dependent variable into the linear equation. This results in an extra parameter associated with the spatial lag of the dependent variable as shown in the following formula.

$$y_i = \alpha + \rho y_{lag-i} + \sum_{j=1}^{m} \beta_j X_{ij} + \epsilon_i,$$

The equation can be represented as shown:

$$y = \alpha + \rho W y + X\beta + \epsilon,$$

In the preceding equation, W is the standardized spatial weights matrix, and ρ is called the spatial autoregressive coefficient.

The `SpatialLagRegressor` class requires setting of the `spatial_weights_definition` parameter, which establishes the relationship between neighboring observations. The following table describes the main methods of the `SpatialLagRegressor` class.

Method	Description
`fit`	Trains the `SpatialLagRegressor` model from the given training data. The model includes a parameter for the spatial lag of the target variable.
`predict`	Uses the trained parameters, including the one associated with the spatial lag of the target variable, to estimate the target variable of the given data.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the R-squared statistic for the given data.

See the SpatialLagRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame`. It creates an instance of the `SpatialLagRegressor` class defining the `spatial_weights_definition` parameter. Then, it creates a spatial pipeline with a preprocessing step to standardize the data and applies the Spatial Lag model at the final step.

The model is trained using a training set (`X_train`) and the `MEDIAN_INCOME` column as the target variable. Finally, it calls the `predict` and `score` methods with a test set (`X_test`) to estimate the values of the target variables and obtain the model's R-Square score respectively.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialLagRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define features
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "INTERNET",
"geometry"]]

# Define training and test sets
```

```
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Create an instance of SpatialLagRegressor
spatial_lag_model =
SpatialLagRegressor(spatial_weights_definition=KNNWeightsDefinition(k=5))

# Add the model in a Spatial Pipeline with a preprocessing step
spatial_lag_pipeline = SpatialPipeline([("scaler", StandardScaler()),
("spatial_lag", spatial_lag_model)])

# Train the model with MEDIAN_INCOME as the target variable
spatial_lag_pipeline.fit(X_train, "MEDIAN_INCOME")

# Print the predictions with the test set
spatial_lag_predictions_test =
spatial_lag_pipeline.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {spatial_lag_predictions_test[:10]}")

# Print the R-squared metric with the test set
spatial_lag_r2_score = spatial_lag_pipeline.score(X_test, y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {spatial_lag_r2_score}")
```

The program produces the following output:

```
>> predictions (X_test):
 [ 92285.13545208 100551.0381313   30910.61123168  45166.3218764
 177515.68764358  44088.89962954  98205.35728383  27788.19879028
  72553.17695035  24875.81828048]

>> r2_score (X_test):
 0.6150829472253789
```

Spatial Error Model

The Spatial Error Model (SEM) introduces a spatial lag in the *error* term of the linear equation.

By adding the spatial lag of the residual, the neighbors' errors influence the observation error. This leads to an extra parameter to be associated with the spatial lag of the error term as shown in the following formula:

$$y_i = \alpha + \sum_{j=1}^{m} \beta_j X_{ij} + u_i$$

$$u_i = \lambda u_{lag-i} + \epsilon_i$$

$$u_{lag-i} = \sum_{k} W_{ik} u_k$$

In the preceding formula, W is the spatial weights matrix.

The SpatialErrorRegressor class implements the spatial error model, which requires the definition of the spatial_weights_definition parameter to use it. The following table describes the main methods of the SpatialErrorRegressor class.

Method	Description
fit	Trains the SpatialErrorRegressor model from the given training data. The model includes a parameter for the spatial lag of the error term.
predict	Uses the trained parameters, including the one associated with the spatial lag of the error term, to estimate the target variable of the given data.
fit_predict	Calls the fit and predict methods sequentially with the training data.
score	Returns the R-squared statistic for the given data.

See the SpatialErrorRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *block_groups* SpatialDataFrame. It creates an instance of the SpatialErrorRegressor class defining the spatial_weights_definition parameter which establishes the relationship between neighboring observations. Then, it adds the model in a spatial pipeline along with a preprocessing step to standardize the data. The model is trained using a training set (X_train) and the MEDIAN_INCOME column as the target variable. Finally, it calls the predict and score methods with the test set (X_test) to estimate the values of the target variable and the model's R-Square score respectively.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialErrorRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define features
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "INTERNET",
"geometry"]]

# Define training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Create an instance of SpatialErrorRegressor
spatial_error_model =
SpatialErrorRegressor(spatial_weights_definition=KNNWeightsDefinition(k=5))

# Add the model in a Spatial Pipeline along with a preprocessing step
spatial_error_pipeline = SpatialPipeline([("scaler", StandardScaler()),
("spatial_error", spatial_error_model)])

# Train the model with MEDIAN_INCOME as the target variable
spatial_error_pipeline.fit(X_train, "MEDIAN_INCOME")

# Print the predictions with the test set
spatial_error_predictions_test =
spatial_error_pipeline.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {spatial_lag_predictions_test[:10]}")
```

```
# Print the R-squared metric with the test set
spatial_error_r2_score = spatial_error_pipeline.score(X_test,
y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {spatial_error_r2_score}")
```

The program produces the following output:

```
>> predictions (X_test):
 [ 92285.13545208 100551.0381313   30910.61123168  45166.3218764
 177515.68764358  44088.89962954  98205.35728383  27788.19879028
  72553.17695035  24875.81828048]

>> r2_score (X_test):
 0.635646418630968
```

Note that printing the property summary of the trained model displays an extra `lambda` parameter. This parameter is associated with the spatial lag of the error term.

```
REGRESSION
----------
SUMMARY OF OUTPUT: MAXIMUM LIKELIHOOD SPATIAL ERROR (METHOD = FULL)
-------------------------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of
Observations:        2750
Mean dependent var  :  69703.4815                Number of
Variables   :           4
S.D. dependent var  :  39838.5789                Degrees of
Freedom    :        2746
Pseudo R-squared    :      0.6285
Sigma-square ML     :472895616.755                Log likelihood       :
-31440.423
S.E of regression   :   21746.163                Akaike info criterion :
62888.846
                                                 Schwarz criterion     :
62912.523

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     z-Statistic
Probability
------------------------------------------------------------------------------
------
            CONSTANT    70397.9327157     855.6991730      82.2694878
0.0000000
            MEAN_AGE    4337.6721310     537.9090592       8.0639507
0.0000000
         HOUSE_VALUE    20927.8165549     706.2614165      29.6318276
0.0000000
            INTERNET    10643.3244395     580.3422845      18.3397363
0.0000000
              lambda       0.5152500       0.0215703      23.8869736
0.0000000
------------------------------------------------------------------------------
```

```
------
================================ END OF REPORT
=====================================
```

Geographical Regressor

A `GeographicalRegressor` is a spatial machine learning algorithm used to perform regression by leveraging the existing `scikit-learn` regression algorithms which are used to create both, a global model containing all the observations from the training data and a local model for each observation and by summing the weighted results of the global model and the local model.

Both, `GeographicalRegressor` and `GeographicalClassifier` extend the Geographical Random Forest algorithm by allowing the use of various underlying machine learning algorithms besides Random Forest and supporting parallelism in the training of local models, ensuring robust and scalable performance. See [4] for more information on the Geographical Random Forest algorithm.

This algorithm is useful when there is a high degree of spatial heterogeneity in the data. This implies that the data behaves differently in different geographical regions. Hence, it will be hard to fit a single model appropriately. This algorithm supports any `scikit-learn` regression algorithms. That means, for different applications, you can specify the underneath regression algorithm, which includes random forest, support vector, gradient boosting, and decision tree.

The `GeographicalRegressor` class implements this regressor. You can specify the `scikit-learn` regression algorithm when you create an instance of this class. First, a global model is built using the parameters provided at creation time. If the spatial relationship is not specified (by providing `SpatialWeightsDefinition` or bandwidth information), it is first computed. After the spatial relation is defined, several local models are built. The prediction is performed by locating the local model closer to the data to be predicted and summing the weighted results of the global model and the local model. Specifically, the returned prediction is calculated as follows:

```
local_model_prediction * local_weight + global_model_prediction * (1.0 -
local_weight)
```

In the preceding calculation, `local_weight` can be a default or specified value.

When using the `GeographicalRegressor` class, you can specify a `scikit-learn` regression algorithm to be used when you create an instance of this class. The following table describes the principal methods of this class.

Method	**Description**
`fit`	First, the global model is built using the parameters provided at creation time. If the spatial relationship is not specified (either by the `spatial_weights_definition` or the `bandwidth` parameter), it is internally computed. Then, several local models are trained.
`predict`	The prediction is executed by locating the local model closer to the data to be predicted and summing the weighted results of the global and local models. The returned prediction is calculated as follows: `local_model_prediction * local_weight + global_model_prediction * (1.0 - local_weight)` In the preceding calculation, `local_weight` is a parameter that can specified by the user.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.

Method	Description
`score`	Returns the R-squared statistic for the given data.

See the GeographicalRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`houses_full`* `SpatialDataFrame`, with a connection to the `la_median_house_values` database table. This table contains housing information from the city of Los Angeles. The *`houses_full`* instance will be used in this example.

```
from oraclesai import SpatialDataFrame, DBSpatialDataset
import oml

houses_full =
SpatialDataFrame.create(DBSpatialDataset(table='la_median_house_values',
schema='oml_user'))
houses_full = houses_full.to_crs('epsg:3857')
```

The code then performs the following steps:

1. Defines the target variable (*`HOUSE_VALUE_MEDIAN`*) and the explanatory variables for the regression model.
2. Splits the data into training and test sets.
3. Defines the spatial weights and creates an instance of `GeographicalRegressor`, which runs multiple Random Forest regressors locally.
4. Calls the `predict` and `score` methods to estimate the target variable of the test set and the R-squared statistic from the same test set.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import SpatialWeights, KNNWeightsDefinition
from sklearn.ensemble import RandomForestRegressor
from oraclesai.regression import GeographicalRegressor

# Define explanatory variables
feature_columns = [
    'BEDROOMS_TOTAL',
    'EDU_LEVEL_SCORE_MEDIAN',
    'POPULATION_DENSITY',
    'ROOMS_TOTAL',
    'COMPLETE_PLUMBING_PERC',
    'COMPLETE_KITCHEN_PERC',
    'HOUSE_AGE_MEDIAN',
    'RENTED_PERC',
    'UNITS_TOTAL'
]

# Define the target variable
target_column = 'HOUSE_VALUE_MEDIAN'

# Select a subset of columns
houses = houses_full[[target_column] + feature_columns]
```

```
# Remove rows with null values
houses = houses.dropna()

# Define the training and test sets
X_train, X_test, y_train, y_test, geom_train, geom_test =
spatial_train_test_split(houses,

     y=target_column,

     test_size=0.33,

     numpy_result=True,

     random_state=32)

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=3)
train_weights = SpatialWeights.create(geom_train, weights_definition)
test_weights = SpatialWeights.create(geom_test, weights_definition)

# Create an instance of GeographicalRegressor
grf_model = GeographicalRegressor(model_cls=RandomForestRegressor,
n_estimators=10, random_state=32)

# Train the model
grf_model.fit(X_train, geometries=geom_train, y=y_train,
spatial_weights=train_weights)

# Print the predictions with the test set
grf_predictions_test = grf_model.predict(X_test,
geometries=geom_test).flatten()
print(f"\n>> predictions (X_test):\n {grf_predictions_test[:10]}")

# Print the score with the test set
grf_r2_score = grf_model.score(X_test, y_test, geometries=geom_test)
print(f"\n>> r2_score (X_test):\n {grf_r2_score}")
```

The output of the program is as follows:

```
>> predictions (X_test):
 [622135.  422560.  426457.5 749530.  925412.5 469420.  526467.5 880195.
 460922.5 421930. ]

>> r2_score (X_test):
 0.6774993920744854
```

Geographically Weighted Regression

The Geographically Weighted Regression (GWR) model is used in the presence of spatial heterogeneity, which can be identified as a sign of regional variation.

The GWR model creates a local linear regression model for every observation in the dataset. It incorporates the target and explanatory variables from the observations within their neighborhood, allowing the relationships between the independent and dependent variables to vary by locality.

The following shows the equation for the GWR model:

$$y_j^{(i)} = W_{i,j} \left(\beta_0^{(i)} + \sum_{k=1} \beta_k^{(i)} X_{j,k} + \epsilon^{(i)} \right)$$

In the preceding equation, W is the spatial weights matrix, $y_j{}^{(i)}$ is the estimation of the target variable for observation j at location i .

The `GWRRegressor` class trains local linear regressions for every sample in the dataset, incorporating the dependent and independent variables of locations falling within a specified bandwidth.

The following table describes the main methods of the `GWRRegressor` class.

Method	Description
`fit`	The algorithm requires a bandwidth, which can be set by the user with the `bandwidth` parameter or by specifying the `spatial_weights_definition` parameter. If the `bandwidth` parameter is defined, the algorithm ignores the bandwidth associated with the spatial weights. The bandwidth can be either a threshold distance or a value of k for the K-Nearest Neighbors method. If neither the `bandwidth` nor the `spatial_weights_definition` parameters are defined, then the bandwidth is estimated internally based on the geometries.
`predict`	To make predictions, GWR creates a model for each observation on the prediction set using neighboring observations from the training data. Then, it uses those models to estimate the target variable.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the R-squared statistic for the given data.

See the GWRRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the `GWRRegressor` to train a model to predict the target variable, `MEDIAN_INCOME`. It uses a training set to train the model and a test set to make predictions of the target variable and obtain the R-squared statistic.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import DistanceBandWeightsDefinition
from oraclesai.regression import GWRRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define target and explanatory variables
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'INTERNET', 'geometry']]

# Use a referenced coordinate system
```

```
X = X.to_crs("epsg:3857")

# Define training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.1, random_state=32)

# Define the spatial weights
weights_definition = DistanceBandWeightsDefinition(threshold=10000)

# Create an instance of GWR passing the spatial weights
gwr_model = GWRRegressor(spatial_weights_definition=weights_definition)

# Add the regressor to a pipeline along with a preprocessing step
gwr_pipeline = SpatialPipeline([('scale', StandardScaler()),
('gwr_regression', gwr_model)])

# Train the model specifying the target variable
gwr_pipeline.fit(X_train, "MEDIAN_INCOME")

# Print the predictions with the test set
gwr_predictions_test =
gwr_pipeline.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {gwr_predictions_test[:10]}")

# Print the score with the test set
gwr_r2_score = gwr_pipeline.score(X_test, y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {gwr_r2_score}")
```

The output of the program is shown is as shown:

```
>> predictions (X_test):
 [111751.58871802 123406.64795915  25850.4248602   23565.60954771
 180171.51825151  47052.37667604 118800.80714934  31067.07113894
  62079.81316461  30673.82128591]

>> r2_score (X_test):
 0.6942389040067138
```

The `summary` property includes statistics of the OLS and GWR models. As for the estimated parameters, it displays the average value from all the local models.

```
===========================================================================
Model type                                                         Gaussian
Number of observations:                                                3093
Number of covariates:                                                     5

Global Regression Results
---------------------------------------------------------------------------
Residual sum of squares:
1816309978579.363
Log-likelihood:                                                  -35614.052
AIC:                                                              71238.104
AICc:                                                             71240.132
BIC:
1816309953761.425
```

```
R2:                                                                0.635
Adj. R2:                                                           0.634

Variable                             Est.         SE  t(Est/SE)    p-value
------------------------------- ---------- ---------- ---------- ----------
X0                               69761.518    436.080    159.974      0.000
X1                                2555.817    564.452      4.528      0.000
X2                                5613.607    843.158      6.658      0.000
X3                               19204.921    602.745     31.862      0.000
X4                               10031.929    637.215     15.743      0.000

Geographically Weighted Regression (GWR) Results
---------------------------------------------------------------------------
Spatial kernel:                                                Fixed bisquare
Bandwidth used:                                                10000.000

Diagnostic information
---------------------------------------------------------------------------
Residual sum of squares:
1247690194588.343
Effective number of parameters (trace(S)):                         117.770
Degree of freedom (n - trace(S)):                                 2975.230
Sigma estimate:                                                  20478.262
Log-likelihood:                                                 -35033.321
AIC:                                                             70304.183
AICc:                                                            70313.751
BIC:                                                             71021.184
R2:                                                                  0.749
Adjusted R2:                                                         0.739
Adj. alpha (95%):                                                    0.002
Adj. critical t value (95%):                                         3.075

Summary Statistics For GWR Parameter Estimates
---------------------------------------------------------------------------
Variable                   Mean        STD        Min     Median        Max
-------------------- ---------- ---------- ---------- ---------- ----------
X0                    62341.157  12808.790 -66225.562  64262.819  94371.705
X1                     2998.233   3153.236 -12716.566   3338.876  18130.392
X2                    10539.611   7148.106  -7226.756   9336.382  70067.037
X3                    16577.403   9934.050  -9579.528  16819.683  47874.385
X4                     9771.744   4232.729   1656.213   9326.487  44417.212
===========================================================================
```

Spatial Regimes

In the spatial regimes algorithm, the regression equation parameters are estimated according to a categorical variable called regime.

This categorical variable can represent different things, such as a region in a spatial context. Neighborhoods, such as district or block names, can be used to define regimes. The model reflects spatial heterogeneity across regions, with different regions having their own regression models.

The `SpatialRegimesRegressor` class consists of linear regression models where the terms of the linear equation vary depending on the regime. The following table describes the main methods of the `SpatialRegimesRegressor` class.

Method	Description
`fit`	The `regime` parameter indicates the categorical variable used as regime. An OLS is run for each regime, obtaining a different set of parameters for each regime.
`predict`	To predict new values, the algorithm uses the parameters associated with the regimes of the prediction data.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the R-squared statistic for the given data. For each observation, it uses the model associated with the corresponding regime.

Even when the `SpatialRegimesRegressor` class does not consider spatial weights in the training process, it uses the `spatial_weights_definition` parameter to obtain spatial diagnostics.

See the SpatialRegimesRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the `SpatialRegimesRegressor` class. However, before executing the regression task, the example requires to define a categorical variable as regime. Then the functions split the geographical area of a `SpatialDataFrame` into a grid with a certain number of rows and columns, where each grid cell is represented by an integer number that will serve as the categorical variable.

```
import bisect

def get_cell_id(array_x, array_y, point, ncols):
    point_x, point_y = point.x, point.y
    grid_x = bisect.bisect_left(array_x, point_x) - 1
    grid_y = bisect.bisect_left(array_y, point_y) - 1

    return grid_y * ncols + grid_x

def create_grid(pdf_data, grid_column, nrows=2, ncols=2):
    min_x, min_y, max_x, max_y = pdf_data.total_bounds
    geometries = pdf_data["geometry"].values
    centroids = [geom.centroid for geom in geometries]

    step_x = (max_x - min_x) / ncols
    step_y = (max_y - min_y) / nrows

    split_x = [min_x + step_x * i for i in range(ncols + 1)]
    split_y = [min_y + step_y * i for i in range(nrows + 1)]

    column_values = []
    for centroid in centroids:
        column_values.append(get_cell_id(split_x, split_y, centroid, ncols))

    return pdf_data.add_column(grid_column, column_values)
```

Using the preceding functions, the following code:

1. Creates another instance of `SpatialDataFrame` with a categorical variable, `GRID_ID`, representing the grid cells that will serve as the regimes.
2. Stores the regimes into a separate variable and removes the categorical variable from the dataset.
3. Trains the `SpatialRegimesRegressor` model with the training set (`X_train`) by calling the `fit` method, setting the `regime` parameter, and using the `MEDIAN_INCOME` column as the target variable.
4. Calls the `predict` and `score` methods using the test set (`X_test`), to estimate the target variable and obtain the R-squared metric.

```
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialRegimesRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Create a categorical variable by splitting the geographic region in a grid
block_groups_grid = create_grid(block_groups, "GRID_ID", nrows=3, ncols=3)

# Define the explanatory variables
X = block_groups_grid[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'INTERNET', 'GRID_ID', 'geometry']]

# Define the training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Get the regime values
regimes_train = X_train["GRID_ID"].values.tolist()
regimes_test = X_test["GRID_ID"].values.tolist()

# Discard the categorical variable
X_train = X_train.drop("GRID_ID")
X_test = X_test.drop("GRID_ID")

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Create a Spatial Regimes Regressor model
spatial_regimes_model =
SpatialRegimesRegressor(spatial_weights_definition=weights_definition)

# Add the model to a spatial pipeline along with a preprocessing step
spatial_regimes_pipeline = SpatialPipeline([('scale', StandardScaler()),
('spatial_regimes', spatial_regimes_model)])

# Train the model using "MEDIAN_INCOME" as the target variable and specifying
the regime values
spatial_regimes_pipeline.fit(X_train, "MEDIAN_INCOME",
spatial_regimes__regimes=regimes_train)

# Print the predictions with the test set
spatial_regimes_predictions_test =
spatial_regimes_pipeline.predict(X_test.drop(["MEDIAN_INCOME"]),
spatial_regimes__regimes=regimes_test).flatten()
```

```
print(f"\n>> predictions (X_test):\n
{spatial_regimes_predictions_test[:10]}")

# Print the score with the test set
spatial_regimes_r2_score = spatial_regimes_pipeline.score(X_test,
y="MEDIAN_INCOME", spatial_regimes__regimes=regimes_test)
print(f"\n>> r2_score (X_test):\n {spatial_regimes_r2_score}")
```

The output of this program is as follows:

```
>> predictions (X_test):
 [ 99973.28903064 119316.0422925   21627.0522275   26862.24033126
 176529.76909922  55563.36270093 115297.87445691  33401.15374394
  63827.11873494  26992.92679579]

>> r2_score (X_test):
 0.67377148094271
```

Since the `spatial_weights_definition` parameter was set when creating the `SpatialRegimesRegressor` instance, the `summary` property of the trained model displays spatial statistics. Note that there is a set of parameters for each regime, as well as some spatial statistics, such as Moran's I and Lagrange Multipliers for spatial dependence.

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES - REGIMES
---------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of
Observations:        2750
Mean dependent var  :  69703.4815                Number of
Variables   :          40
S.D. dependent var  :  39838.5789                Degrees of
Freedom    :        2710
R-squared           :      0.6974
Adjusted R-squared  :      0.6930
Sum squared residual:1320270117156.439                F-
statistic           :    160.1405
Sigma-square        :487184545.076                Prob(F-
statistic)     :           0
S.E. of regression  :   22072.257                Log likelihood        :
-31387.645
Sigma-square ML     :480098224.421                Akaike info criterion :
62855.290
S.E of regression ML:  21911.1438                Schwarz criterion     :
63092.065

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     t-Statistic
Probability
------------------------------------------------------------------------------
------
          1_CONSTANT    67301.4371567    1953.6056568      34.4498578
```

```
0.0000000
          1_MEAN_AGE    -787.8377162    1485.8378441      -0.5302313
0.5959950
1_MEAN_EDUCATION_LEVEL    19399.3182180    3114.5763711       6.2285576
0.0000000
       1_HOUSE_VALUE    18607.2342406    1584.1781459      11.7456703
0.0000000
          1_INTERNET    13025.7000079    2370.5082392       5.4948976
0.0000000
          2_CONSTANT    70316.2663016    3128.3635757      22.4770122
0.0000000
          2_MEAN_AGE    4475.1151552    1602.7038604       2.7922283
0.0052714
2_MEAN_EDUCATION_LEVEL    6155.3917348    2436.3442043       2.5264869
0.0115775
       2_HOUSE_VALUE    8287.3366860    4847.3374558       1.7096678
0.0874418
          2_INTERNET    9610.2177802    1714.0106903       5.6068599
0.0000000
          3_CONSTANT    24528.5879950    5872.7675236       4.1766659
0.0000305
          3_MEAN_AGE    4605.8239137    1904.1647555       2.4188159
0.0156366
3_MEAN_EDUCATION_LEVEL    22124.7054269    5152.1353075       4.2942788
0.0000181
       3_HOUSE_VALUE    22528.7956619    1505.5002005      14.9643259
0.0000000
          3_INTERNET    22442.8115822    3672.8299785       6.1104956
0.0000000
          4_CONSTANT    60346.7138163    1011.7946534      59.6432424
0.0000000
          4_MEAN_AGE    2025.4934828    1131.5366834       1.7900378
0.0735594
4_MEAN_EDUCATION_LEVEL    12613.8139792    1879.7592801       6.7103347
0.0000000
       4_HOUSE_VALUE    15802.2959953    1094.1149414      14.4429944
0.0000000
          4_INTERNET    7544.7984901    1423.9963625       5.2983271
0.0000001
          5_CONSTANT    60570.6305539    1375.4910298      44.0356420
0.0000000
          5_MEAN_AGE    4004.8956000    1338.2798927       2.9925695
0.0027914
5_MEAN_EDUCATION_LEVEL    7093.5634835    1762.1713354       4.0254675
0.0000584
       5_HOUSE_VALUE    4973.1688262    2760.5550262       1.8015105
0.0717336
          5_INTERNET    5212.2336124    1092.1003496       4.7726691
0.0000019
          6_CONSTANT    74193.6261803    1593.8110537      46.5510802
0.0000000
          6_MEAN_AGE    8804.9736797    1830.8258733       4.8092906
0.0000016
6_MEAN_EDUCATION_LEVEL    -1282.6669985    2732.9823394      -0.4693287
0.6388725
       6_HOUSE_VALUE    24763.0330906    2724.0892923       9.0903896
```

```
0.0000000
          6_INTERNET    14378.1718270    2116.0137823       6.7949330
0.0000000
          7_CONSTANT    72053.1153887    1522.5496169      47.3239851
0.0000000
          7_MEAN_AGE    3957.0149819    1885.0370696      2.0991709
0.0358941
7_MEAN_EDUCATION_LEVEL    -1604.5557316    2759.9951560      -0.5813618
0.5610450
       7_HOUSE_VALUE    25077.4167626    3315.7621906       7.5630927
0.0000000
          7_INTERNET    11840.4394166    2062.1006321       5.7419309
0.0000000
          8_CONSTANT    58026.3709199    3699.6679150      15.6842107
0.0000000
          8_MEAN_AGE    4496.6200307    2673.0921045      1.6821792
0.0926493
8_MEAN_EDUCATION_LEVEL    17341.3083231    5737.4485722       3.0224773
0.0025306
       8_HOUSE_VALUE    35050.3546911    3390.1281391      10.3389469
0.0000000
          8_INTERNET    15125.8210946    3364.7884860       4.4953260
0.0000072
------------------------------------------------------------------------------
------
Regimes variable: unknown

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER          10.296

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2        1869.657           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test               39        1548.245           0.0000
Koenker-Bassett test             39         544.999           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Moran's I (error)              0.1497        19.689           0.0000
Lagrange Multiplier (lag)         1         174.856           0.0000
Robust LM (lag)                   1           1.572           0.2099
Lagrange Multiplier (error)       1         334.438           0.0000
Robust LM (error)                 1         161.155           0.0000
Lagrange Multiplier (SARMA)       2         336.010           0.0000


REGIMES DIAGNOSTICS - CHOW TEST
                 VARIABLE        DF        VALUE           PROB
                 CONSTANT         7         141.366           0.0000
              HOUSE_VALUE         7          74.722           0.0000
                 INTERNET         7          41.075           0.0000
                 MEAN_AGE         7          19.445           0.0069
```

```
     MEAN_EDUCATION_LEVEL         7          54.041            0.0000
              Global test        35         566.146            0.0000
================================ END OF REPORT
=====================================
```

Spatial Fixed Effects

The spatial fixed effects algorithm computes an intercept or constant parameter for each regime, while the other model parameters remain constant. It is a simplified version of the spatial regimes algorithm.

The `SpatialFixedEffectsRegressor` class consists of regression models where each model has a different constant parameter, one for each regime. The rest of the parameters of the models are the same. To predict new values, it gets the constant parameter for the corresponding regime internally and uses that parameter in the regression equation along with the other parameters. You can also pass in the `spatial_weights_definition` parameter to obtain spatial diagnostics for analyzing the input features and fine tune the model.

The following table describes the main methods of the `SpatialFixedEffectsRegressor` class.

Method	**Description**
`fit`	The `regime` parameter indicates the categorical variable used as regime. The intercept parameter of the linear equation is different for each regime, while the rest of the parameters remain constant.
`predict`	To predict new values, the algorithm gets the intercept of the linear equation from the corresponding regime (according to the `regime` parameter), and uses it along with the other parameters.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the R-squared statistic for the given data. For each observation, it uses the intercept associated with the corresponding regime, according to the `regime` parameter.

When creating an instance of the `SpatialFixedEffectsRegresssor` class, it is possible to define the `spatial_weights_definition` parameter to obtain spatial diagnostics after training the model.

See the SpatialFixedEffectsRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and the functions defined in Spatial Regimes to create the regimes by splitting the geographical area into a grid, where each cell represents a regime.

Then, trains the `Spatial Fixed Effects` model. Finally, using the test set, it calls the `predict` and `score` methods to estimate the target variable and the R-squared metric respectively.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialFixedEffectsRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Create a categorical variable by splitting the geographic region in a grid
block_groups_grid = create_grid(block_groups, "GRID_ID", nrows=3, ncols=3)
```

```
# Define the explanatory variables
X = block_groups_grid[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'INTERNET', 'GRID_ID', 'geometry']]

# Define the training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Get the regime values
regimes_train = X_train["GRID_ID"].values.tolist()
regimes_test = X_test["GRID_ID"].values.tolist()

# Discard the categorical variable
X_train = X_train.drop("GRID_ID")
X_test = X_test.drop("GRID_ID")

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Create a Spatial Fixed Effects Regressor model
sfe_model =
SpatialFixedEffectsRegressor(spatial_weights_definition=weights_definition)

# Add the model to a spatial pipeline along with a preprocessing step
sfe_pipeline = SpatialPipeline([('scale', StandardScaler()), ('sfe',
sfe_model)])

# Train the model using "MEDIAN_INCOME" as the target variable and specifying
the regimes
sfe_pipeline.fit(X_train, "MEDIAN_INCOME", sfe__regimes=regimes_train)

# Print the predictions with the test set
sfe_predictions_test = sfe_pipeline.predict(X_test.drop(["MEDIAN_INCOME"]),
sfe__regimes=regimes_test).flatten()
print(f"\n>> predictions (X_test):\n {sfe_predictions_test[:10]}")

# Print the score with the test set
sfe_r2_score = sfe_pipeline.score(X_test, y="MEDIAN_INCOME",
sfe__regimes=regimes_test)
print(f"\n>> r2_score (X_test):\n {sfe_r2_score}")
```

The program prints the predictions of the target variable of the first 10 observations, and the R-squared metric for the test set as shown:

```
>> predictions (X_test):
 [101512.84282764 109422.92724391  29615.01694646  29230.32429018
 162356.33498145  53108.14145735 105985.63259313  28588.56284749
  81056.36661461  19790.46314804]

>> r2_score (X_test):
 0.6701128016747615
```

The intercept values for each regime can be visualized using the `summary` property, and if the `spatial_weights_definition` parameter was defined when creating the regressor, the summary also includes spatial statistics, such as the Moran's I and Lagrange Multipliers for spatial lag and spatial error.

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES - REGIMES
---------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of 
Observations:        2750
Mean dependent var  :  69703.4815                Number of 
Variables   :          12
S.D. dependent var  :  39838.5789                Degrees of 
Freedom    :         2738
R-squared           :      0.6573
Adjusted R-squared  :      0.6559
Sum squared residual:1495203246049.754                F-
statistic           :    477.4024
Sigma-square        :546093223.539                Prob(F-
statistic)     :           0
S.E. of regression  :   23368.638                Log likelihood       :
-31558.731
Sigma-square ML     :543710271.291                Akaike info criterion :
63141.461
S.E of regression ML:  23317.5957                Schwarz criterion    :
63212.494

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     t-Statistic     
Probability
------------------------------------------------------------------------------
------
          1_CONSTANT    75646.5430042    1406.0974938      53.7989317
0.0000000
          2_CONSTANT    77794.0850074    1338.3185516      58.1282273
0.0000000
          3_CONSTANT    58981.5644323    1948.7462992      30.2664151
0.0000000
          4_CONSTANT    60320.9906786    1002.6995461      60.1585898
0.0000000
          5_CONSTANT    69884.3635458    1076.5155202      64.9171909
0.0000000
          6_CONSTANT    75355.5269590    1338.6764983      56.2910659
0.0000000
          7_CONSTANT    71531.4267958    1445.6625603      49.4800300
0.0000000
          8_CONSTANT    72960.0800416    1983.5523209      36.7825337
0.0000000
    _Global_MEAN_AGE    2989.5036511     583.1586204       5.1263988
0.0000003
_Global_MEAN_EDUCATION_LEVEL    6304.4360113     904.9392927
6.9666950       0.0000000
 _Global_HOUSE_VALUE    21452.9209086     664.4420803      32.2871196
0.0000000
```

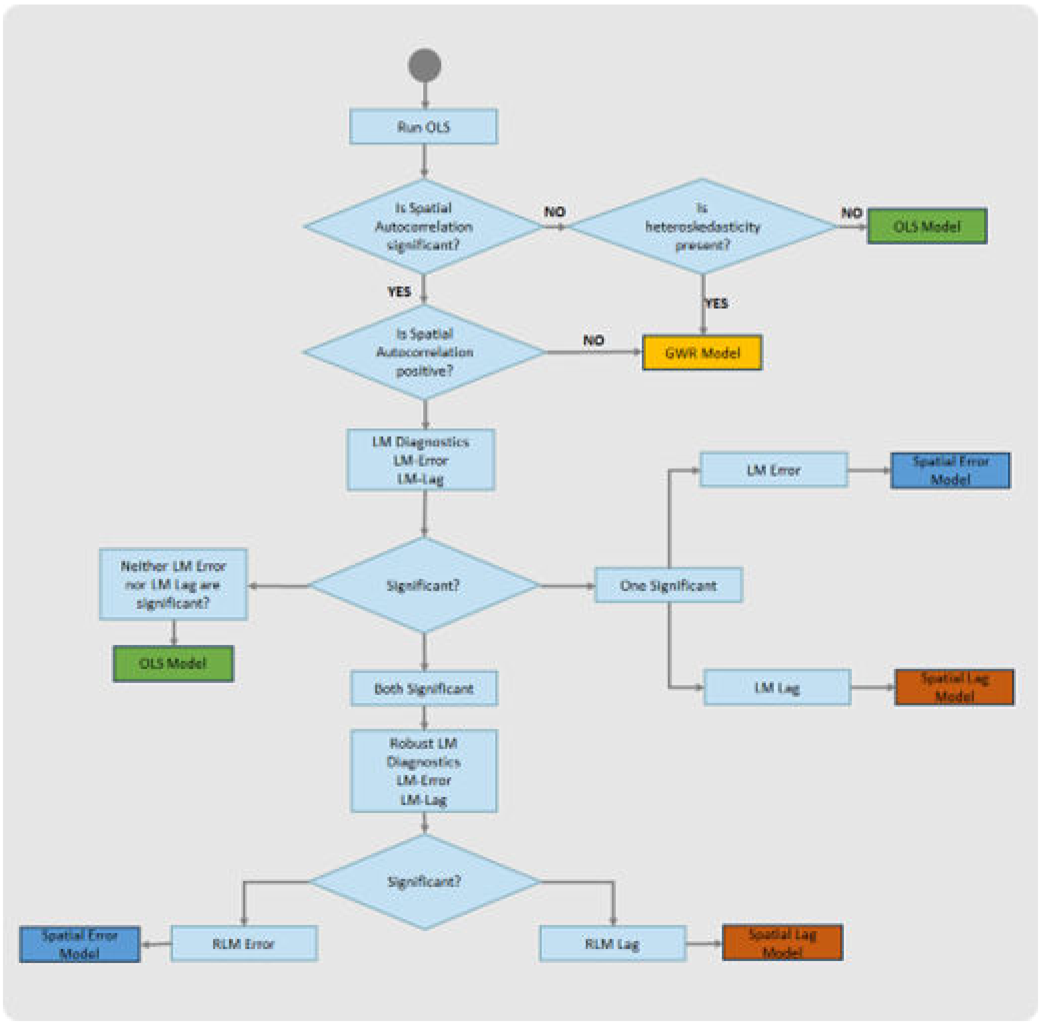
```
    _Global_INTERNET    8352.1786588     664.9940434      12.5597797
0.0000000
------------------------------------------------------------------------------
------
Regimes variable: unknown

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            4.274

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2        1415.811           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test               11        1252.140           0.0000
Koenker-Bassett test             11         486.455           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Moran's I (error)              0.2201        27.742           0.0000
Lagrange Multiplier (lag)         1         317.696           0.0000
Robust LM (lag)                   1           1.495           0.2214
Lagrange Multiplier (error)       1         722.582           0.0000
Robust LM (error)                 1         406.382           0.0000
Lagrange Multiplier (SARMA)       2         724.078           0.0000


REGIMES DIAGNOSTICS - CHOW TEST
                 VARIABLE        DF        VALUE           PROB
                 CONSTANT         7         184.738           0.0000
              Global test         7         184.738           0.0000
================================ END OF REPORT
=====================================
```

Adaptive Spatial Regression

The `AdaptiveSpatialRegressor` class consists of an automated approach that finds the regression algorithm that better fits the data. This is the best approach when you do not know which model to use.

The algorithm trains an `OLSRegressor` model specifying the `spatial_weights_definition` parameter to get the spatial diagnostics. Based on spatial statistics, it suggests the regression algorithm. You have to provide spatial weights definition when using this algorithm, otherwise, the algorithm recommends `OLSRegressor`.

The following figure shows the current workflow for choosing the best algorithm.


Run OLS
Is Spatial Autocorrelation significant?
NO
Is heteroskedasticity present?
NO
OLS Model
YES
YES
Is Spatial Autocorrelation positive?
NO
GWR Model
LM Diagnostics LM-Error LM-Lag
LM Error
Spatial Error Model
Neither LM Error nor LM Lag are significant?
Significant?
One Significant
OLS Model
Both Significant
LM Lag
Spatial Lag Model
Robust LM Diagnostics LM-Error LM-Lag
Significant?
Spatial Error Model
RLM Error
RLM Lag
Spatial Lag Model

From spatial diagnostics, the algorithm gets the Moran's I statistic. If the value is statistically significant, then it is interpreted as follows:

- A positive value of Moran's I statistic indicates the presence of spatial dependence, or spatial clustering, and an algorithm that includes this spatial dependence is preferred. Two algorithms that consider spatial dependence are `SpatialLagRegressor` and `SpatialErrorRegressor`. Depending on the Lagrange Multipliers obtained from spatial diagnostics, the algorithm selects one of them (see [3] for more detailed information about spatial regression diagnostics).
- If the Moran's I statistic is negative, then it indicates the presence of regional variance or spatial heteroskedasticity, and a local method such as `GWRRegressor` is more suitable.

In case the Moran’s I statistic is not statistically significant but the variability of the residuals is significant, then the algorithm selects the `GWRRegressor`.

See the SpatialAdaptiveRegressor class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and `SpatialAdaptiveRegressor` to train a model from a training set. Then, using a test set, the code estimates the target variable and gets the R-squared metric.

```
%python
```

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialAdaptiveRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define target and explanatory variables
X = block_groups[['MEDIAN_INCOME', 'MEAN_AGE', 'MEAN_EDUCATION_LEVEL',
'HOUSE_VALUE', 'INTERNET', 'geometry']]

# Define training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME",
test_size=0.2, random_state=32)

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=5)

# Create an instance of SpatialAdaptiveRegressor
spreg_model =
SpatialAdaptiveRegressor(spatial_weights_definition=weights_definition)

# Add the model to a spatial pipeline along with a preprocessing step
spreg_pipeline = SpatialPipeline([('scale', StandardScaler()),
('spreg_regression', spreg_model)])

# Train the model
spreg_pipeline.fit(X_train, "MEDIAN_INCOME")

# Print the selected model
print(f">> Algorithm chosen:
{spreg_pipeline.named_steps['spreg_regression'].model_type.__name__}")

# Print the predictions with the test set
spreg_predictions_test =
spreg_pipeline.predict(X_test.drop("MEDIAN_INCOME")).flatten()
print(f"\n>> predictions (X_test):\n {spreg_predictions_test[:10]}")

# Print the score with the test set
spreg_r2_score = spreg_pipeline.score(X_test, "MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {spreg_r2_score}")
```

The output of the program consists of the name of the algorithm chosen by `SpatialAdaptiveRegressor`, the predictions of the first 10 observations of the test set, and the R-squared metric of the test set.

```
> Algorithm chosen: ErrorModel

>> predictions (X_test):
 [101563.4135695  105231.46019748  24081.18722085  38529.02025428
 164280.78271333  50332.38349005 102590.59769969  27659.63416001
  81911.84382123  17657.93225933]

>> r2_score (X_test):
 0.6456845274014411
```

9 Apply Spatial Classification

Learn and apply the machine learning algorithms for spatial classification.

Topics:

About Spatial Classification

A spatial classification task consists of training a model that predicts the value of a target variable according to explanatory variables, where the target variable is categorical.

Spatial classification algorithms provide a spatial context to the already known classification algorithms. It is helpful when the variables are influenced by their geographic location.

The following lists a few use cases of spatial classification:

- Categorize the crime level of a particular zone.
- Identify the severity of air pollution across a geographic region and classify them into different zones for decision-making.
- Estimate the outcome of a political election in different locations.

SLX Classifier

The Spatial Cross-Regressive (SLX) classification algorithm executes logistic regression involving a feature engineering step to add features that provide a spatial context to the data.

The algorithm adds one or more columns with the spatial lag of certain features, representing the average from neighboring observations.

Using the `SLXClassifier` class requires defining the spatial weights with the `spatial_weights_definition` parameter, which establishes the interaction between neighboring observations.

For a multi-class classification problem, the algorithm uses the *one-vs-rest* strategy, training a model for each class. One common issue with this strategy is that it can result in an imbalanced dataset, where the proportion of elements of one class is much larger than the other. To handle this scenario, the `SLXClassifier` class provides the following two oversampling methods:

- **Random.** This method creates duplicates of random samples (with replacement) from the minority class.
- **Synthetic Minority Oversampling Technique (SMOTE).** This algorithm selects a random sample of the minority class, **a**, and from its k nearest neighbors, it selects a random neighbor, **b**. The vector **ab** is multiplied by a random number in the range [0, 1], and the

result is added to sample **a**, generating a new synthetic instance. See [5] for more information on SMOTE.

The following parameters specify the oversampling method and the number of new samples:

Parameter	Description
`balance_method`	The oversampling method. The default value is `None`, the other options are *random* and *smote*.
`balance_ratio`	A number in the range `[0, 1]` representing the desired ratio of observations from the minority class. A value of 1 will result in the same number of observations for both classes.

The following table describes the main methods of the `SLXClassifier` class.

Method	Description
`fit`	The parameters of the `fit` method are the same for most of the regression algorithms, except for the `column_ids` parameter, which specify the columns that are used to compute the spatial lag. The algorithm estimates the parameters of the explanatory variables plus the parameters associated with those added with the spatial lag.
`predict`	The `predict` method calculates the spatial lag of the dataset using the same columns defined in the fit process and returns the category with the highest probability according to Logistic Regression. By setting the `use_fit_lag=True` parameter, the algorithm calculates the spatial lag from the training set. This is helpful when the prediction dataset contains few observations.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the accuracy for the given data. By setting the `use_fit_lag=True`, the algorithm calculates the spatial lag from the training set. Otherwise, it computes the spatial lag from the provided data.

See the SLXClassifier class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and performs the following steps:

1. Creates a categorical variable, `INCOME_LABEL`, based on the `MEDIAN_INCOME` column, to use as the target variable.
2. Creates an instance of `SXLClassifier` specifying the `balance_method` and `balance_ratio` parameters.
3. Trains the model using a training set.
4. Prints the predictions from the model and the model's accuracy using the test set.

```
import numpy as np
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.classification import SLXClassifier
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define the categories for the target variable
labels=["low", "medium-low", "medium-high", "high"]
```

```
# The target variable comes from the column MEDIAN_INCOME
income_array = block_groups['MEDIAN_INCOME'].values

# Define constants to create the target variable
min_income = np.min(income_array)
max_income = np.max(income_array)
delta = (max_income - min_income) / 4

# Define a function that returns a category based on the median income
def get_label(income):
    if income <= min_income + delta:
        return "low"
    elif min_income + delta < income <= min_income + 2 * delta:
        return "medium-low"
    elif min_income + 2 * delta < income <= min_income + 3 * delta:
        return "medium-high"
    return "high"

# Create a new SpatialDataFrame with the target variable "INCOME_LABEL"
block_groups_extended = block_groups.add_column("INCOME_LABEL",
[get_label(income) for income in income_array])

# Define the target and explanatory variables
X = block_groups_extended[['INCOME_LABEL', 'MEAN_AGE',
'MEAN_EDUCATION_LEVEL', 'HOUSE_VALUE', 'INTERNET', 'geometry']]

# Split the data into training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="INCOME_LABEL",
test_size=0.2, random_state=32)

# Define the spatial weights
weights_definition = KNNWeightsDefinition(k=20)

# Create the instance of SLXClassifier
slx_classifier = SLXClassifier(spatial_weights_definition=weights_definition,
                                                random_state=15,
                                                balance_method="smote",
                                                balance_ratio=0.05)

# Add the model to a spatial pipeline along with a pre-processing step
classifier_pipeline = SpatialPipeline([('scale', StandardScaler()),
('classifier', slx_classifier)])

# Train the model specifying the target variable and the parameter column_ids
classifier_pipeline.fit(X_train, "INCOME_LABEL",
classifier__column_ids=["MEAN_AGE", "HOUSE_VALUE"])

# Print the predictions with the test set
slx_predictions_test =
classifier_pipeline.predict(X_test.drop("INCOME_LABEL")).flatten()
print(f"\n>> predictions (X_test):\n {slx_predictions_test[:10]}")

# Print the accuracy with the test set
slx_accuracy_test = classifier_pipeline.score(X_test, "INCOME_LABEL")
print(f"\n>> accuracy (X_test):\n {slx_accuracy_test}")
```

The output consists of the predictions of the first ten observations and the model's accuracy using the test set.

```
>> predictions (X_test):
 ['medium-low' 'medium-low' 'low' 'low' 'high' 'low' 'medium-low' 'low'
 'low' 'low']

>> accuracy (X_test):
 0.7438136826783115
```

The `summary` property displays the statistics of the trained model, or models in case of multi-class, along with the mean value of the estimated parameters.

```
Multi-Class Logistic Model Results
---------------------------------------------------------------------------
      label    deviance          llf         aic           bic       D2
adj_D2
       high  342.409525  -171.204763  356.409525 -21710.554931 0.553921
0.552958
        low 1855.672117  -927.836058 1869.672117 -20197.292339 0.498926
0.497844
 medium-low 2506.593561 -1253.296780 2520.593561 -19546.370895 0.249868
0.248249
medium-high  840.588033  -420.294016  854.588033 -21212.376424 0.357128
0.355741

Parameters (Average Results)
Variable                             Est.        STD        Min
Median        Max
------------------------------ ---------- ---------- ---------- ----------
----------
constant                           -3.740      4.007     -9.350
-3.432      1.256
MEAN_AGE                           -0.043      0.275     -0.345
-0.062      0.296
MEAN_EDUCATION_LEVEL                0.960      1.229     -1.077
1.472      1.974
HOUSE_VALUE                        -0.037      0.867     -1.175
-0.047      1.119
INTERNET                            0.652      1.369     -1.435
0.824      2.394
SLX-MEAN_AGE                        0.018      0.016     -0.006
0.022      0.035
SLX-HOUSE_VALUE                     0.001      0.026     -0.017
-0.012      0.047
```

Geographical Classifier

Similar to `GeographicalRegressor`, the `GeographicalClassifier` class trains a global model and multiple local models and predicts by combining the weighted results from both models.

By defining the `global_model` and `model_cls` parameters, you can specify the `scikit-learn` global and local classifiers respectively. The classifiers can be any `scikit-learn` classifiers, including Random Forest, Support Vector, Gradient Boosting, Decision Trees, and so on.

Both, `GeographicalClassifier` and `GeographicalRegressor` extend the Geographical Random Forest algorithm by allowing the use of various underlying machine learning algorithms besides Random Forest and supporting parallelism in the training of local models, ensuring robust and scalable performance. See [4] for more information on the Geographical Random Forest algorithm.

The following table describes the main methods of the `Geographical Classifier` class.

Method	Description
`fit`	First, the global model is built using the parameters provided at creation time. If the spatial relationship is not specified (either by the `spatial_weights_definition` or the `bandwidth` parameter), it is internally computed. Then, several local models are trained.
`predict`	The following steps describe the prediction method: 1. The prediction is executed by locating the local model closer to the observation to be predicted. 2. By using a weighted average of the predictions from the global and local model, the algorithm estimates a discrete range of values corresponding to classes, representing the probability of an observation belonging to each class. 3. The category associated with the highest probability represents the predicted value.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the model's accuracy for the given data.

See the Geographical Classifier class in *Python API Reference for Oracle Spatial AI* for more information.

The following code uses the `houses_full` `SpatialDataFrame`, containing housing information for the city of Los Angeles. The example performs the following steps:

1. Creates a categorical variable based on the `HOUSE_VALUE_MEDIAN` column.
2. Defines the training and test sets.
3. Creates an instance of `GeographicalClassifier`.
4. Trains the local model using the `RandomForestClassifier` from `scikit-learn`.
5. Calls the `predict` and `score` methods to estimate the target variable and the model's accuracy of a test set respectively.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import DistanceBandWeightsDefinition
from sklearn.ensemble import RandomForestClassifier
from oraclesai.classification import GeographicalClassifier

# Define explanatory variables
feature_columns = [
    'BEDROOMS_TOTAL',
    'EDU_LEVEL_SCORE_MEDIAN',
    'POPULATION_DENSITY',
    'ROOMS_TOTAL',
    'COMPLETE_PLUMBING_PERC',
    'COMPLETE_KITCHEN_PERC',
    'HOUSE_AGE_MEDIAN',
    'RENTED_PERC',
    'UNITS_TOTAL'
]

# The target variable will be built from this column
target_column = 'HOUSE_VALUE_MEDIAN'

# Select a subset of columns
houses = houses_full[[target_column] + feature_columns]

# Remove rows with null values
houses = houses.dropna()

# Define training and test sets
X_train, X_test, y_train, y_test, geom_train, geom_test = 
spatial_train_test_split(houses,

     y=target_column,

     test_size=0.33,

     numpy_result=True,

     random_state=32)

# Define constants to create a categorical variable
y = houses[target_column].values
y_mean = y.mean()
y_std = y.std()

# House prices below the mean minus 0.5 std are considered a low-value
# House prices above the mean plus 0.5 std are considered a high-value
mid_low_price =  y_mean - y_std * 0.5
mid_hi_price = y_mean + y_std * 0.5

# Define the function that generates the target variable based on the house 
value
def classify_house_value(house_value):
    if house_value < mid_low_price:
        return 0.0
    if house_value > mid_hi_price:
        return 2.0
    return 1.0

# Generate the target variable for the training and test sets
y_c_train = [classify_house_value(inc) for inc in y_train]
y_c_test = [classify_house_value(inc) for inc in y_test]

# Define the spatial weights
weights_definition = DistanceBandWeightsDefinition(threshold=2388.51)

# Create an instance of GeographicalClassifier
grfc_model = GeographicalClassifier(model_cls=RandomForestClassifier,
                                    n_estimators=10,
                                    local_weight=0.80,
spatial_weights_definition=weights_definition,
                                    random_state=32)
# Train the model
grfc_model.fit(X_train, y=y_c_train, geometries=geom_train, n_jobs=-1)

# Print the predictions with the test set
grfc_predictions_test = grfc_model.predict(X_test,
geometries=geom_test).flatten()
print(f"\n>> predictions (X_test):\n {grfc_predictions_test[:10]}")

# Print the score with the test set
grfc_accuracy = grfc_model.score(X_test, y_c_test, geometries=geom_test)
print(f"\n>> accuracy (X_test):\n {grfc_accuracy}")
```

The output consists of the predictions of the first 10 observations of the test set and the model's accuracy using the same test set.

```
>> predictions (X_test):
 [1 1 0 2 2 1 1 0 0 0]

>> accuracy (X_test):
 0.7343004295345901
```

GWR Classifier

The Geographically Weighted Regression (GWR) classifier is a binary classifier used in the presence of spatial heterogeneity, which can be identified as a sign of regional variation.

The algorithm creates a local classifier for every observation in the dataset by incorporating the target and explanatory variables from the observations within their neighborhood, allowing the relationships between the independent and dependent variables to vary by locality.

The classifier trains a logistic regression model for every sample in the dataset, incorporating the dependent and independent variables of locations falling within a specified bandwidth. The goal is to maximize the cross-entropy loss function defined as follows.

$$l = \sum_{i=1}^{n} \left(y_i \ln(h(x_i, \theta)) + (1 - y_i) \ln(1 - h(x_i, \theta)) \right)$$

In the preceding function, *y* is either `0` or `1`, the function *h* is the activation function for Logistic Regression, which is the Sigmoid function.

The following table describes the main methods of the `GWRClassifier` class.

Method	Description
`fit`	The algorithm requires a bandwidth, which can be set by the user with the `bandwidth` parameter or by specifying the `spatial_weights_definition` parameter. If the `bandwidth` parameter is defined, the algorithm ignores the bandwidth associated with the spatial weights. The bandwidth can be either a threshold distance or a value of k for the K-Nearest Neighbors method. If neither the `bandwidth` nor the `spatial_weights_definition` parameters are defined, then the bandwidth is estimated internally based on the geometries.
`predict`	To make predictions, GWR trains a model for each observation on the prediction set using neighboring observations from the training data. Then, it uses those models to estimate the target variable.
`fit_predict`	Calls the `fit` and `predict` methods sequentially with the training data.
`score`	Returns the model's accuracy for the given data.

See the GWRClassifier class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and performs the following steps:

1. Creates a categorical variable based on the `MEDIAN_INCOME` column to be used as the target variable.
2. Creates an instance of `GWRClassifier`.
3. Trains the model using a training set.
4. Prints the predictions from the model and the model's accuracy using the trained model.

```
import pandas as pd
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import DistanceBandWeightsDefinition
from oraclesai.classification import GWRClassifier
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Create a categorical variable, "INCOME_LABEL", based on the second quantile
of the median income
block_groups_extended = block_groups.add_column("INCOME_LABEL",
pd.qcut(block_groups['MEDIAN_INCOME'].values, [0, 0.5, 1], labels=[0,
1]).to_list())

# Set a referenced coordinate system
block_groups_extended = block_groups_extended.to_crs('epsg:3857')

# Define the target and explanatory variables
X = block_groups_extended[['INCOME_LABEL', 'MEAN_AGE',
'MEAN_EDUCATION_LEVEL', 'HOUSE_VALUE', 'INTERNET', 'geometry']]

# Define the training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="median_income",
test_size=0.2, random_state=32)
```

```
# Define the spatial weights definition
weights_definition = DistanceBandWeightsDefinition(threshold=15000)

# Create an instance of GWRClassifier
gwr_classifier = GWRClassifier(spatial_weights_definition=weights_definition)

# Add the model to a spatial pipeline along with a pre-processing step
classifier_pipeline = SpatialPipeline([('scale', StandardScaler()), ('gwr',
gwr_classifier)])

# Train the model specifying the target variable
classifier_pipeline.fit(X_train, "INCOME_LABEL")

# Print the predictions with the test set
gwr_predictions_test =
classifier_pipeline.predict(X_test.drop("INCOME_LABEL")).flatten()
print(f"\n>> predictions (X_test):\n {gwr_predictions_test[:10]}")

# Print the accuracy with the test set
gwr_accuracy_test = classifier_pipeline.score(X_test, "INCOME_LABEL")
print(f"\n>> accuracy (X_test):\n {gwr_accuracy_test}")
```

The output consists of the predictions of the first 10 observations and the model's accuracy using the test set.

```
>> predictions (X_test):
 [1 1 0 0 1 0 1 0 0 0]

>> accuracy (X_test):
 0.8384279475982532
```

The `summary` property includes statistics of a global logistic regression and the `GWRClassifier`. As for the estimated parameters, it displays the average value from all the local models.

```
===========================================================================
Model type                                                         Binomial
Number of observations:                                                2750
Number of covariates:                                                     5

Global Regression Results
---------------------------------------------------------------------------
Deviance:                                                          2088.938
Log-likelihood:                                                   -1044.469
AIC:                                                               2098.938
AICc:                                                              2098.960
BIC:                                                             -19649.694
Percent deviance explained:                                           0.452
Adj. percent deviance explained:                                      0.451

Variable                                   Est.         SE  t(Est/SE)    p-value
------------------------------- ---------- ---------- ---------- ----------
X0                                       -0.044      0.061     -0.717      0.473
X1                                        0.439      0.072      6.084      0.000
X2                                        0.685      0.104      6.603      0.000
X3                                        0.542      0.109      4.989      0.000
```

```
X4                                  1.298      0.092      14.088      0.000

Geographically Weighted Regression (GWR) Results
---------------------------------------------------------------------------
Spatial kernel:                                               Fixed bisquare
Bandwidth used:                                                    15000.000

Diagnostic information
---------------------------------------------------------------------------
Effective number of parameters (trace(S)):                            56.675
Degree of freedom (n - trace(S)):                                   2693.325
Log-likelihood:                                                     -888.994
AIC:                                                                1891.337
AICc:                                                               1893.765
BIC:                                                                2226.816
Percent deviance explained:                                          0.534
Adjusted percent deviance explained:                                 0.524
Adj. alpha (95%):                                                      0.004
Adj. critical t value (95%):                                           2.850

Summary Statistics For GWR Parameter Estimates
---------------------------------------------------------------------------
Variable                   Mean        STD        Min     Median        Max
-------------------- ---------- ---------- ---------- ---------- ----------
X0                       -0.020      0.846     -1.630     -0.140      3.328
X1                        0.512      0.325      0.020      0.385      2.156
X2                        0.931      0.665     -1.213      1.168      2.893
X3                        0.995      0.981     -0.615      0.834      6.249
X4                        1.190      0.356      0.324      1.119      2.531
===========================================================================
```

10
Work with Spatial Pipeline

Oracle Spatial AI provides spatial pipelining capabilities to organize and simplify spatial machine learning workflow.

Topics:

About Spatial Pipeline

The spatial pipeline extends the existing `scikit-learn` pipeline to include spatial information such as geometry data and spatial weights.

The `SpatialPipeline` class can easily chain together both spatial and non-spatial steps, and is composed of estimators. An estimator can be one of the following:

- **Transformer**: An estimator with the `fit` and `transform` methods that are described in the following table.

Method	Description
`fit`	The `fit` method computes statistics and other properties from the training data.
`transform`	The `transform` method applies the values calculated in the fit method to change the data.
`fit_transform`	Calls the `fit` and `transform` methods sequentially with the training data.

  One typical example of a transformer is the `StandardScaler`, which standardizes the data so that each feature has zero mean and unit variance. Usually, transformers are part of the pre-processing step in a pipeline.

- **Classifier/Regressor**: This estimator must be the last step in a pipeline. It can be either a regression or a classification task. The methods available in a pipeline correspond to those in the final step. In this case, it has the `fit`, `predict`, and `score` methods along with the other methods associated with the estimator. Usually, the pipeline goes through multiple transformers before reaching this estimator.
- **Composite Estimator:** These estimators can combine multiple estimators and can be chained with other estimators. For example, having a pre-processing pipeline to execute multiple transformations to the data and then making this pipeline part of another pipeline for a regression task. There are three composite estimators:

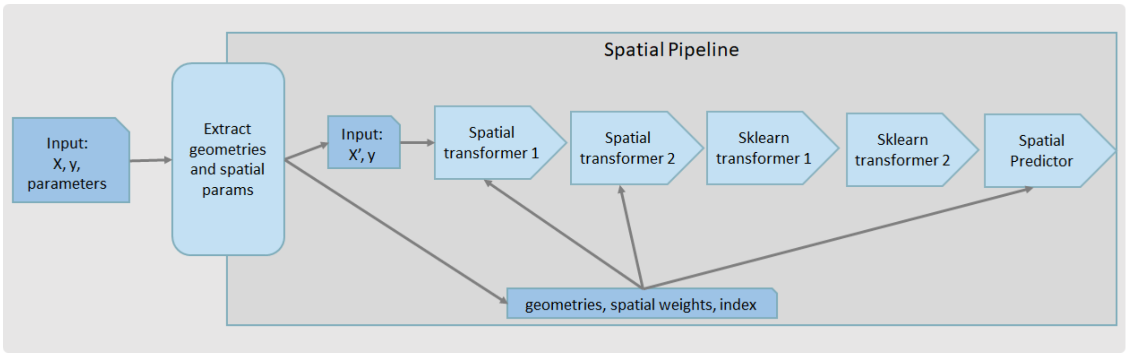
Estimator	Description
`SpatialPipeline`	A pipeline that includes spatial information.

Estimator	Description
`SpatialFeatureUnion`	Concatenate resulting columns (features) from different estimators to create a single input while sharing spatial information.
`SpatialColumnTransformer`	Selects a subset of columns (features) from the input and passes these columns to an estimator while sharing spatial information.

A spatial pipeline can take the same input as a regular `scikit-learn` pipeline plus the spatial information which is required by spatial processes (spatial transformers and spatial models or predictors). This additional spatial information can be divided into two categories:

- **Data location/geometries:** The geometry associated with each sample in the input data, `X`, is a vector of geometries. This vector can be embedded in `X` if `X` is either a geopandas `GeoDataFrame` or a `SpatialDataFrame`. It can also be defined in the parameter geometries.
- **Spatial parameters:** These are additional parameters used to provide context about geometries (CRS), describe/quantify spatial relationships (spatial weights definition, spatial weights objects), or help perform faster spatial searches (spatial index).

The following figure shows the data flow in a spatial pipeline.



As seen in the preceding figure, the input data comprising `X`, `y`, and (optionally) spatial parameters are received by the spatial pipeline. Note that the input `X` can be split into `X'` (non-spatial data) and geometries. Then, the spatial parameters and the geometries are extracted and passed to all the spatial steps in the pipeline.

Spatial Feature Union

The `SpatialFeatureUnion` estimator shares spatial properties with multiple transformers and concatenates the results.

The following table describes the main methods of the `SpatialFeatureUnion` class.

Method	Description
`fit`	Calls the `fit` method of all the transformers.
`transform`	Calls the `transform` method of each transformer and concatenate the results.
`fit_transform`	Fits all the transformers, transforms the data, and concatenates the results.

See the SpatialFeatureUnion class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *block_groups* `SpatialDataFrame` and `SpatialFeatureUnion` to concatenate the output from two transformers. The first is a `SimpleImputer` from `scikit-learn`, and the second is a `SpatialLagTransformer`. The dataset has three columns, excluding the geometries, so the final result contains six columns.

```
from oraclesai.pipeline import SpatialFeatureUnion
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.preprocessing import SpatialLagTransformer
from sklearn.impute import SimpleImputer

# Define training variables
X = block_groups[['MEAN_AGE', 'HOUSE_VALUE', "MEDIAN_INCOME", "geometry"]]

# Print X
print("=========================== X =================================")
print(X.get_values()[:5,:])

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=5)

# Define a Spatial Lag Transformer
spatial_lag_transformer =
SpatialLagTransformer(spatial_weights_definition=weights_definition)

# Use SpatialFeatureUnion to concatenate the output from all the transformers
slag_feature_union = SpatialFeatureUnion([("imputer", SimpleImputer()),
                                          ("spatial_lag",
spatial_lag_transformer)])

# Print the final result
print("\n=================== X transformed ============================")
print(slag_feature_union.fit_transform(X)[:5, :])
```

The first three columns of the transformed data represent the output from the `SpatialImputer`, and the other three represent the output from the `SpatialLagTransformer`.

```
=========================== X =================================
[[4.75847626e+01 4.56300000e+05 5.38280000e+04]
 [3.88231812e+01 8.36300000e+05 6.07240000e+04]
 [4.78076096e+01 1.12630000e+06 8.25380000e+04]
 [4.65636330e+01 9.60400000e+05 1.43661000e+05]
 [5.11550865e+01 1.01090000e+06 1.23977000e+05]]

=================== X transformed ============================
[[4.75847626e+01 4.56300000e+05 5.38280000e+04 4.03809292e+01
  6.23460000e+05 7.92068000e+04]
 [3.88231812e+01 8.36300000e+05 6.07240000e+04 3.95882790e+01
  8.20100000e+05 9.82008000e+04]
 [4.78076096e+01 1.12630000e+06 8.25380000e+04 4.69466225e+01
  1.22280000e+06 1.14899600e+05]
 [4.65636330e+01 9.60400000e+05 1.43661000e+05 4.25439751e+01
  1.04664000e+06 1.16867800e+05]
 [5.11550865e+01 1.01090000e+06 1.23977000e+05 4.43390564e+01
  1.14368000e+06 1.45833400e+05]]
```

Spatial Column Transformer

The `SpatialColumnTransformer` shares spatial information with multiple transformers, applying transformations to different columns and concatenating the results.

The following table describes the main methods of the `SpatialColumnTransformer` class.

Method	Description
`fit`	Calls the `fit` method of all the transformers.
`transform`	Calls the `transform` method of each transformer and concatenates the results.
`fit_transform`	Fits all the transformers, transforms the data, and concatenates the results.

See the SpatialColumnTransformer class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and `SpatialColumnTransformer` to concatenate the output from two different transformers. The first is a `SimpleImputer` from `scikit-learn`, and the second is a `SpatialLagTransformer` applied to `HOUSE_VALUE` and `MEDIAN_INCOME` columns. The final result contains five columns.

```
from oraclesai.pipeline import SpatialColumnTransformer
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.preprocessing import SpatialLagTransformer
from sklearn.impute import SimpleImputer

# Define training variables
X = block_groups[["MEAN_AGE", "HOUSE_VALUE", "MEDIAN_INCOME", "geometry"]]

# Print X
print("========================== X =================================")
print(X.get_values()[:5,:])

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=5)

# Define a Spatial Lag Transformer
spatial_lag_transformer =
SpatialLagTransformer(spatial_weights_definition=weights_definition)

# Use SpatialColumnTransformer to concatenate column subsets
slag_column_transformer = SpatialColumnTransformer([
    ("imputer", SimpleImputer(), ["MEAN_AGE", "HOUSE_VALUE",
"MEDIAN_INCOME"]),
    ("spatial_lag", spatial_lag_transformer, ["HOUSE_VALUE",
"MEDIAN_INCOME"])])

# Print the final result
print("\n================== X transformed ============================")
print(slag_column_transformer.fit_transform(X)[:5, :])
```

The first three columns of the transformed data represent the output of the first transformer, and the other two columns represent the output of the second transformer.

```
=========================== X ==================================
[[4.75847626e+01 4.56300000e+05 5.38280000e+04]
 [3.88231812e+01 8.36300000e+05 6.07240000e+04]
 [4.78076096e+01 1.12630000e+06 8.25380000e+04]
 [4.65636330e+01 9.60400000e+05 1.43661000e+05]
 [5.11550865e+01 1.01090000e+06 1.23977000e+05]]

=================== X transformed ============================
[[4.75847626e+01 4.56300000e+05 5.38280000e+04 6.23460000e+05
  7.92068000e+04]
 [3.88231812e+01 8.36300000e+05 6.07240000e+04 8.20100000e+05
  9.82008000e+04]
 [4.78076096e+01 1.12630000e+06 8.25380000e+04 1.22280000e+06
  1.14899600e+05]
 [4.65636330e+01 9.60400000e+05 1.43661000e+05 1.04664000e+06
  1.16867800e+05]
 [5.11550865e+01 1.01090000e+06 1.23977000e+05 1.14368000e+06
  1.45833400e+05]]
```

Spatial Pipeline

The `SpatialPipeline` class shares spatial information through a pipeline of transformers, other estimators, and a final estimator.

Note that the final estimator step of the pipeline is not optional in this case. A typical scenario consists of having a preprocessing pipeline in charge of different tasks, such as cleaning the data, filling missing values, and standardizing the data. Then, the preprocessing pipeline is part of another pipeline with a final estimator, either a regressor or a classifier.

The following table describes the main methods of the `SpatialPipeline` class.

Method	**Description**
`fit`	Calls the `fit` method of the pipeline transformers and the final estimator.
`fit_predict`	Calls the `fit` and `transform` methods of the pipeline transformer and the `fit` and `predict` methods of the final estimator.
`predict`	Calls the `transform` method of all the transformers in the pipeline and calls the `predict` method of the final estimator.

See the SpatialPipeline class in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and `SpatialColumnTransformer` to define a feature-engineering step, which creates new columns representing the spatial lag of specific columns. Then, the feature-engineering step is added into a `SpatialPipeline`, along with a pre-processing step that standardizes the data and a final estimator consisting of a spatial error regression model.

```
from oraclesai.pipeline import SpatialColumnTransformer
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.preprocessing import SpatialLagTransformer
from oraclesai.regression import SpatialErrorRegressor
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Define target and explanatory variables
X = block_groups[["MEAN_AGE", "HOUSE_VALUE", "MEDIAN_INCOME", "geometry"]]

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Define a Spatial Lag Transformer
spatial_lag_transformer =
SpatialLagTransformer(spatial_weights_definition=weights_definition)

# Create an instance of SpatialErrorRegressor
spatial_error_regressor =
SpatialErrorRegressor(spatial_weights_definition=weights_definition)

# Use SpatialColumnTransformer to concatenate column subsets
feature_engineering_step = SpatialColumnTransformer([
    ("imputer", SimpleImputer(), ["MEAN_AGE", "HOUSE_VALUE"]),
    ("spatial_lag", spatial_lag_transformer, ["HOUSE_VALUE"])])

# Create a pipeline with three steps: Feature-Engineering, Scaler, Regressor
regression_pipeline = SpatialPipeline([
    ("feature_engineering", feature_engineering_step),
    ("scaler", StandardScaler()),
    ("regressor", spatial_error_regressor)
])

# Train the model
regression_pipeline.fit(X, y="MEDIAN_INCOME")

# Print the score of the training set
print(f"r2_score = {regression_pipeline.score(X, y='MEDIAN_INCOME')}")
```

The output consists of the R-squared metric from the final estimator. The example calls the `score` method to run the transform methods of all the transformers in the pipeline.

```
r2_score = 0.5559292598577543
```

11

Work with Data Visualization

Oracle Spatial AI provides visualization functions compatible with `SpatialDataFrame`.

The two main functionalities are plotting geometries and clusters based on `Matplotlib`.

Topics:

Plot Geometries

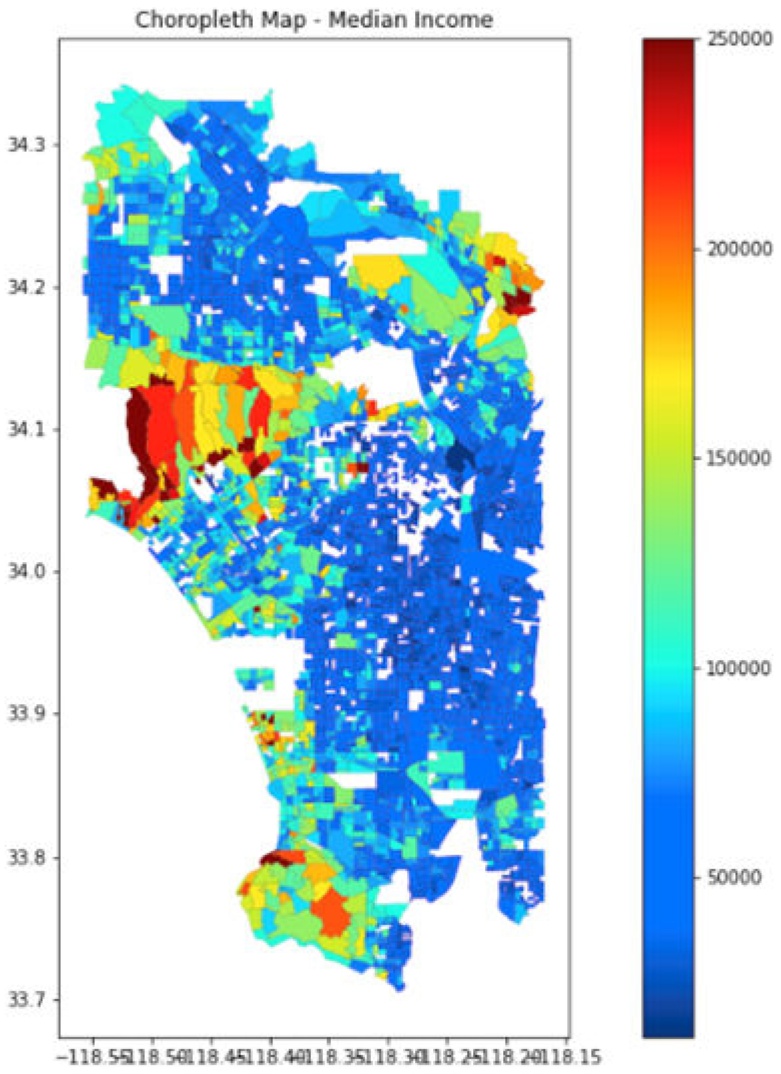
The `plot_geometries` function in `oraclesai.vis` can take the exact parameters of the `plot` function of a `GeoDataFrame`. Additionally, it supports data from a `SpatialDataFrame` or `GeoDataFrame` by specifying the `data` parameter.

See the plot_geometries function in *Python API Reference for Oracle Spatial AI* for more information.

The following example displays a map with the geometries in the *`block_groups`* `SpatialDataFrame` with the color representing the value of the `MEDIAN_INCOME` column.

```
import matplotlib.pyplot as plt
from oraclesai.vis import plot_geometries

fig, ax = plt.subplots(figsize=(15,10))

# Set the titles
ax.set_title('Choropleth Map - Median Income');

# Plot the choropleth map
plot_geometries(data=block_groups, ax=ax,
column=block_groups["MEDIAN_INCOME"].values, cmap=plt.get_cmap("jet"),
legend=True, edgecolor='black', linewidth=0.1 )
```

The output is as shown:



Plot Clusters

The `plot_clusters` function in `oraclesai.vis` allows you to associate geometries and labels, and display this association in a map.

The function works with data from `SpatialDataFrame` or `GeoDataFrame`.

The following table describes the main parameters of the `plot_clusters` function.

Parameter	Description
`X`	The data with the geometries. It can be either a `SpatialDataFrame` or a `GeoDataFrame`.
`labels`	The labels associated with `X`. Each observation in `X` has a label assigned to it.
`background_data`	A `SpatialDataFrame` or a `GeoDataFrame` with its geometries as background.
`crs`	Indicates the coordinate reference system to be used for plotting. By default, it uses `X.crs`.

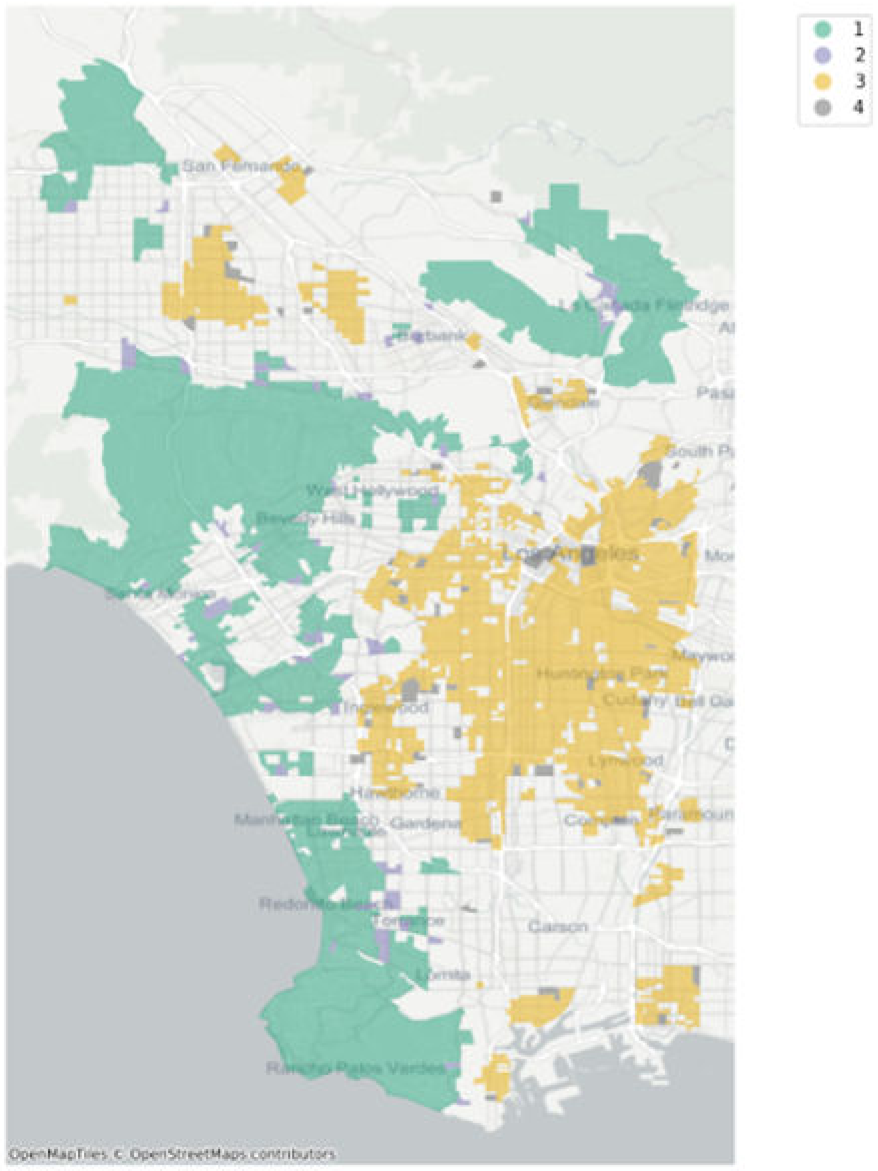
Parameter	Description
`with_bounds`	If `True`, each cluster is displayed with an enclosed polygon.
`with_noise`	If `False`, all the observations with the label –1 are ignored. Otherwise, the observations are assigned to a cluster.
`with_legend`	If `True`, a legend indicating the cluster's labels is displayed.
`with_basemap`	If `True`, the default basemap is displayed. If a `TileProvider` instance is passed, it uses it as the basemap. Alternatively, it can also take a dictionary with the `oracles.vis.add_basemap` parameters except `ax`.

See the plot_clusters function in *Python API Reference for Oracle Spatial AI* for more information.

The following example trains a clustering model with a `LISAHotspotClustering` instance using the `MEDIAN_INCOME` column. It then displays the geometries and the corresponding labels using the `plot_clusters` function in a map.

```
import matplotlib.pyplot as plt
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.clustering import LISAHotspotClustering
from oraclesai.vis import plot_clusters

X = block_groups["MEDIAN_INCOME"]

# Define spatial weights
weights_definition = KNNWeightsDefinition(k=10)

# Create an instance of LISAHotspotClustering
lisa_model = LISAHotspotClustering(max_p_value=0.05,

spatial_weights_definition=weights_definition)
# Train the model
lisa_model.fit(X)

fig, ax = plt.subplots(figsize=(12,12))
plot_clusters(X, lisa_model.labels_, with_noise=False, with_basemap=True,
cmap='Dark2', ax=ax)
```

The output is as shown:



Add a Basemap

You can use the `add_basemap` function to add a basemap to be displayed as a background map.

The basemap provider is specified in the `source` parameter which can be either a `xyzservices.TileProvider` object or an URL. If the `source` parameter is not defined, then it uses the default basemap.

Oracle Spatial AI already provides basemaps based on eLocation. The following basemaps are available through `oraclesai.vis.elocation`:

- `osm_positron` (default)
- `osm_bright`
- `osm_darkmatter`

- `world_map_mb`

See the add_basemap function in *Python API Reference for Oracle Spatial AI* for more details.

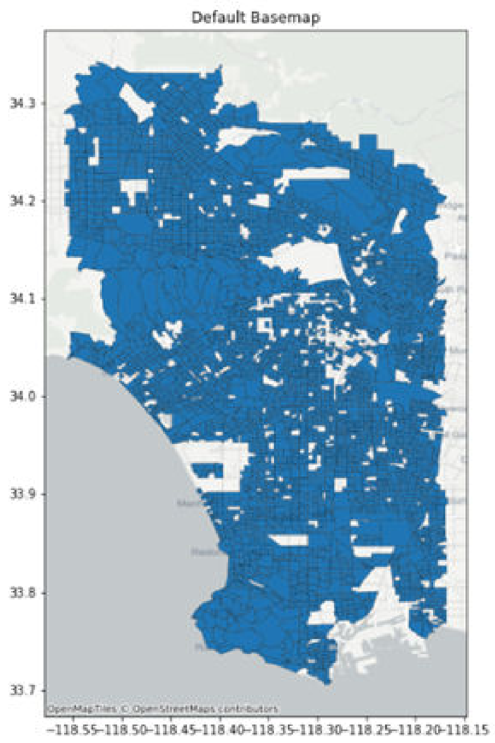
The following code displays the geometries of the *`block_groups`* `SpatialDataFrame` using two different basemaps.

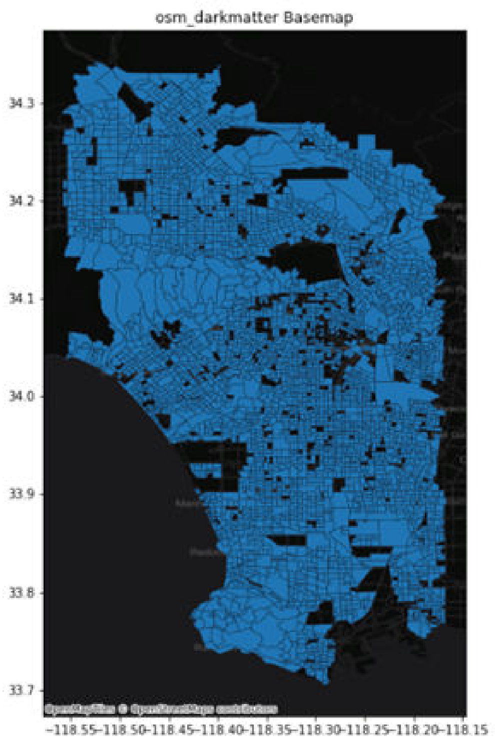
```
import matplotlib.pyplot as plt
from oraclesai.vis import plot_geometries, add_basemap, elocation

fig, ax = plt.subplots(1, 2, figsize=(15,10))

# Set the titles
ax[0].set_title('Default Basemap');
ax[1].set_title('osm_darkmatter Basemap');

plot_geometries(data=block_groups, with_basemap=True, ax=ax[0],
edgecolor='black', linewidth=0.2 )
plot_geometries(data=block_groups, ax=ax[1], edgecolor='black',
linewidth=0.2 )

add_basemap(ax=ax[1], source=elocation.osm_darkmatter, crs=block_groups.crs)
```

By defining the `with_basemap=True` parameter in the `plot_geometries` function, the default basemap is displayed (see the left image in the following figure). A different basemap can be added with the `add_basemap` function.

Alternatively, you can set the `with_basemap=elocation.osm_darkmatter` parameter in the `plot_geometries` function (see the right image in the following figure). In this case, you can omit calling the `add_basemap` function.

The following figure shows the background basemap added by both the preceding methods.


Default Basemap
34.3
34.2
34.1
34.0
33.9
33.8
33.7
OpenMapTiles © OpenStreetMaps contributors
−118.55 −118.50 −118.45 −118.40 −118.35 −118.30 −118.25 −118.20 −118.15


osm_darkmatter Basemap
34.3
34.2
34.1
34.0
33.9
33.8
33.7
OpenMapTiles © OpenStreetMaps contributors
−118.55 −118.50 −118.45 −118.40 −118.35 −118.30 −118.25 −118.20 −118.15

12 Run Post-Processing Tasks

You can run post-processing tasks using the `SpatialDataFrame` class to interact with the database tables and files.

Also, OML4Py provides the functionality to save and load models into a datastore.

Post-processing tasks in the Spatial AI workflow include the following:

- Storing the model's predictions or transformations as database tables or files.
  You can store data, such as features created as part of a feature engineering task or changes made as part of a preprocessing task, in a database using the `write` function in the `SpatialDataFrame` class.
- Saving a model to an OML4Py datastore.
  Your can store trained models, transformers, estimators, and Python objects in an OML4Py datastore.
- Loading a model from the OML4Py datastore.
  You can retrieve and use previously stored models, transformers, or Python objects that are available in an OML4Py datastore.

The post-processing tasks are explained in detail in the following sections:

Topics:

- Store Data into Database Tables or Files
- Save a Model to a OML4Py Datastore
- Load a Model from an OML4Py Datastore

Store Data into Database Tables or Files

Working on a machine learning task may involve data transformation activities (such as creating meaningful features, data cleaning, encoding categorical variables, and so on) to make the data more useful. You can then store these data changes in the database or files using the `write` method of the `SpatialDataFrame` class.

All transformations of a `SpatialDataFrame` do not have any direct effect in the database tables prior to calling the `write` method.

The following table describes the main parameters of the `write` function.

Parameter	Description
`dataset`	This parameter is an instance of `SpatialDataset`, and represents the resulting dataset.
`if_exists`	The options are `fail` and `replace`. Determines the action to take if the resulting dataset already exists.
`include_index`	If `True`, the index columns of the instance are written in the result.
`create_spatial_metadata`	If the spatial metadata needs to be created, then this value is set to `True` . Used only for Oracle Spatial database datasets.

Parameter	Description
`create_spatial_index`	This parameter is used only for Oracle Spatial database datasets, and it is `True` if a spatial index needs to be created.

See the SpatialDataFrame.write function in *Python API Reference for Oracle Spatial AI* for more information.

The following example uses the *`block_groups`* `SpatialDataFrame` and performs the following steps:

1. Adds a new column with a categorical variable called `INCOME_LABEL`.
2. Calls the `add_column` method which returns a new instance of `SpatialDataFrame` with the extended dataset.
3. Calls the `write` method to store the data in the database.
4. Loads the data from the recently created table in a `SpatialDataFrame` and verifies the newly added column.

```
# The column INCOME_LABEL is not in the dataset
if "INCOME_LABEL" not in block_groups.columns:
    print("The column INCOME_LABEL is not part of the columns of
block_groups")

# Create the variable "INCOME_LABEL" based on the median income
block_groups_extended = block_groups.add_column("INCOME_LABEL",
pd.qcut(block_groups['MEDIAN_INCOME'].values, [0, 0.5, 1], labels=[0,
1]).to_list())


# Store the extended data in the database
block_groups_extended.write(DBSpatialDataset(table='write_test'),
                            if_exists='replace',
                            create_spatial_index=True)

# Load the stored dataset in a new SpatialDataFrame
block_groups_new =
SpatialDataFrame.create(DBSpatialDataset(table='write_test'))

# The column INCOME_LABEL is contained in the dataset's columns
if "INCOME_LABEL" in block_groups_new.columns:
    print("The column INCOME_LABEL is contained in the columns of
block_groups_new")
```

The output confirms that the new column, `INCOME_LABEL`, is part of the dataset stored in the database.

```
The column INCOME_LABEL is not part of the columns of block_groups
The column INCOME_LABEL is contained in the columns of block_groups_new
```

Save a Model to a OML4Py Datastore

You can store spatial models and Python objects in the OML4Py datastore and use them for other tasks.

The following code shows how to save a model into the OML4Py datastore. The example uses a Python object, `my_model`, which represents a spatial estimator (such as a regressor, classifier, or some other estimator). The model is saved as `sai_ds` in a datastore with an empty description. Note that by setting `overwrite=True`, any existing datastore with the same name is replaced.

```
import oml
oml.ds.save({'spatial_model': my_spatial_model}, 'sai_ds', description='some 
description', overwrite=True)
```

See Save Objects to a Datastore in *Oracle Machine Learning for Python User's Guide* for more information.

Load a Model from an OML4Py Datastore

You can call the `oml.ds.load` function to load a spatial model from its datastore and use it to solve different problems.

The following code loads the object associated with the string *spatial_model* from the `sai_ds` datastore. Note that by setting `to_globals=False`, the `oml.ds.load` function returns a dictionary containing pairs of object names and values.

```
import oml

ds_objects = oml.ds.load('sai_ds', objs=['spatial_model'], to_globals=False)
my_spatial_model = ds_objects['spatial_model']
```

See Load Saved Objects From a Datastore in *Oracle Machine Learning for Python User's Guide* for more information.

You can list all the objects saved in a datastore as shown:

```
print(oml.ds.dir())
```

13

Use Spatial AI with OML Embedded Python Execution

Learn to use Spatial AI with OML Embedded Python Execution.

Topics:

About Embedded Python Execution

Embedded Python execution is a feature of Oracle Machine Learning for Python (OML4Py) that allows users to invoke user-defined Python functions directly in a database instance.

See Embedded Python Execution in *Oracle Machine Learning for Python User's Guide* for more information.

To demonstrate how to use embedded execution, the following example prepares a spatial regression model. This model is used in the subsequent topics in this chapter which describe how to invoke a Python function using embedded execution, including creating a function that uses that model to make predictions, and then using that function for embedded execution from Python, SQL and REST.

The example steps are as follows:

1. Defines the regressor model using the *block_groups* `SpatialDataFrame` and `SpatialErrorRegressor`.
2. Creates a Spatial Pipeline with a pre-processing step to standardize the data and the regressor as the last step.
3. Trains the model using `MEDIAN_INCOME` as the target variable.

```
from oraclesai.preprocessing import spatial_train_test_split
from oraclesai.weights import KNNWeightsDefinition
from oraclesai.regression import SpatialErrorRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Define variables
X = block_groups[["MEDIAN_INCOME", "MEAN_AGE", "HOUSE_VALUE", "geometry"]]

# Define training and test sets
X_train, X_test, _, _, _, _ = spatial_train_test_split(X, y="MEDIAN_INCOME", 
test_size=0.2, random_state=32)
```

```
# Create instance of SpatialErrorRegressor
spatial_error_model =
SpatialErrorRegressor(spatial_weights_definition=KNNWeightsDefinition(k=5))

# Add the model into a Spatial Pipeline along with a pre-processing step
spatial_error_pipeline = SpatialPipeline([("scaler", StandardScaler()),
("spatial_error", spatial_error_model)])

# Train the model with MEDIAN_INCOME as the target variable
spatial_error_pipeline.fit(X_train, "MEDIAN_INCOME")
```

Once the model is trained, save the model into an OML4Py datastore.

```
oml.ds.save({'spatial_error': spatial_error_pipeline},
    'spatial_error_ds', description='some description',
    overwrite=True)
```

Store a Function for Embedded Execution

You can store a user-defined Python function for embedded execution.

The following Python code creates a function that receives prediction data. The function loads the trained model from the OML4Py datastore and returns the result by calling the `predict` method with the prediction data.

You need to register the function for embedded execution with OML by using the `oml.script.create` method. Note that this function is enclosed within triple quotes.

```
func = """def error_model_predict_(X):
    import oml
    objs = oml.ds.load('spatial_error_ds', objs=['spatial_error'],
to_globals=False)
    error_model = objs['spatial_error']
    pred = error_model.predict(X)
    return pred.tolist()"""

oml.script.create("errorModelPredict", func, is_global=True, overwrite=True)
```

The `oml.script.create` function adds a user-defined Python function to the OML script repository. The `is_global` parameter specifies whether to create a global Python function or if it is available only to the current user. The `overwrite` parameter specifies whether to overwrite the Python function if it already exists.

Call an Embedded Function from Python

You must use the `oml.do_eval` function to run a user-defined Python function.

The following example calls the `errorModelPredict` function, passing the test data without the target variable.

```
spatial_error_predictions = oml.do_eval(func='errorModelPredict',
X=X_test.drop("MEDIAN_INCOME"))
print(spatial_error_predictions[:10])
```

The code prints the first ten predictions for the test data.

```
[[85565.81571662657], [88769.98209276547], [46010.46116330226],
[61275.919165868865], [163674.5321011373], [40178.55663104116],
[89290.25850064949], [47908.54834079923], [83884.02318889851],
[50495.29040429841]]
```

Call an Embedded Function with SQL and REST APIs

You can call an embedded function with SQL and REST APIs.

Perform the following steps:

1. Get an access token before calling OML embedded execution API from SQL or REST.

   As a prerequisite, note the following information for your ADB environment:

   - **`tenant_name:`** Tenancy ID
   - **`database_name:`** Name of the database
   - **`user_name:`** OML username
   - **`password:`** Password for the OML user
   - **`host:`** Root domain

   Perform a REST request to get an access token. The REST request can be done using different approaches. For example, the following code shows how to get a token using a REST call through Python.

   ```
   import json
   import requests
   import warnings
   import os

   token=None

   response = requests.post(
           f"https://{host}:443/omlusers/tenants/{tenant_name}/databases/
   {database_name}/api/oauth2/v1/token",
           headers={"Content-Type": "application/json", "Accept":
   "application/json"},
           json={"grant_type": "password", "username": username, "password":
   password},
           verify=False)
   token = response.json()["accessToken"]
   print(f"token='{token}'")
   ```

2. Call an Embedded Python Function from SQL.

An access token must be set always before performing a call to OML Embedded Execution from SQL. Set the access token in the token store through SQL or PL/SQL and the `pyqSetAuthToken` function.

```
exec pyqSetAuthToken('<access-token>');
```

Call the OML's `pyqEval` function which then calls the user-defined Python function in a SQL query.

The following code uses the `pyqEval` function to call the `errorModelPredict` function that was previously created. The function also passes the `X` parameter consisting of a single observation.

```
select *
    from table(pyqEval(
        par_lst => '{"X": [[30.6005898, 342200.000]]}',
        out_fmt => 'JSON',
        scr_name => 'errorModelPredict'
        ));
```

The result from the preceding code consists of the predicted median income for the given observation.

```
NAME    VALUE
    [[48228.470695050346]]
```

3. Call an Embedded Python Function from REST.

   Make a successful REST request by passing the Spatial AI function-specific parameters within the `parameters` field as a JSON string.
   The following examples use CURL to send a request that calls the `errorModelPredict` function with the parameter X containing a single observation. Note that an access token must first be obtained. In this example, the access token is set in the *`token`* environment variable and is passed in the request.

```
curl -i -k -X POST --header 'Authorization: Bearer ${token}' \
--header 'Content-Type: application/json' --header 'Accept: application/
json' \
-d '{  "oml_connect": true, "parameters": "{\"X\": [[30.6005898,
342200.000]]}" }' \
"${host}:8080/oml/tenants/${tenant_name}/databases/${database_name}/api/py-
scripts/v1/do-eval/errorModelPredict"
```

   The following shows a sample response that includes the request status code and the output of the function representing the estimated income value for the given observation.

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1;mode=block
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
Content-Security-Policy: frame-ancestors 'none'
```

```
Set-Cookie: JSESSIONID=node0nyjijo5nrw2swfj850bvbauc43.node0; Path=/oml;
Secure; HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json
Content-Length: 32

{"result":[[48228.470695050346]]}
```

Predefined Spatial Functions Available from OML Embedded Python Execution

Spatial AI provides some pre-defined spatial functions. You can call the `register_sai_scripts` function (in `oraclesai.oml`) to register the pre-defined spatial functions for embedded execution.

The following code registers the pre-defined spatial functions for embedded execution into the script repository and lists them using the `oml.script.dir` function.

```
import oml
from  oraclesai.oml import register_sai_scripts

# register for all the users and overwrite if already registered
register_sai_scripts(is_global=True, overwrite=True)

# list registered scripts
oml.script.dir(sctype='all')[['name']]
```

Note the `errorModelPredict` function (defined in Store a Function for Embedded Execution) in the list of registered functions along with the other pre-defined functions.

```
                                     name
0                              clustering
1  compute_global_spatial_autocorrelation
2   compute_local_spatial_autocorrelation
3                 compute_spatial_weights
4                      create_spatial_lag
5                       errorModelPredict
```

The following table lists the parameters that are required for all the pre-defined spatial functions.

Parameter	Description
`oml_connect`	This parameter must always be `true` since all the pre-defined spatial functions require a connection to the database.
`table`	The name of a database table.

The rest of the parameters may vary depending on the spatial function.

The following table describes each one of the pre-defined spatial functions for embedded execution.

Spatial Functions	Description
compute_spatial_weights	This function computes the spatial weights for the given spatial table and stores a `SpatialWeights` object in the data store according to the `save_weights_as` parameter.
compute_global_spatial_autocorrelation	Computes the Moran's I statistic for the given spatial table and column. The function returns the value of the Moran's I statistic, its z-value, and its p-value.
compute_local_spatial_autocorrelation	Calculates the Local Moran's I statistic of all the observations from the given spatial table. The function returns a table containing the Local Moran's I statistic for each row, along with the z-value and p-value.
create_spatial_lag	Computes the spatial lag for the given column of the provided spatial table. The function returns a table with the calculated spatial lag for each row from the input table.
clustering	Executes a clustering algorithm with the data from the given spatial table, using only the specified columns or all the columns if the `columns` parameter is not provided. Available clustering methods are `DBSCAN`, `AGGLOMERATIVE`, and `KMEANS`.

All the predefined spatial functions support computing the spatial weights and storing them in a datastore for later use. The goal of these functions is to execute common tasks involving spatial information. You can always add more functions for specific purposes as described in Store a Function for Embedded Execution.

compute_spatial_weights

Format

```
compute_spatial_weights(table, weights_def, save_weights_as,
    spatial_col=None, crs=None)
```

Parameters

The parameters for this pre-defined function are described in the following table.

Parameter	Description
`table`	Name of the spatial table.
`weights_def`	Defines the relationship between neighboring locations. This is passed as a json object specifying the type of the weights definition and its parameters. Each parameter is defined in detail in the API Reference documentation. The following lists the supported types and parameters: • **KNN**: `[k]` • **Kernel**: `[bandwidth, fixed, k, function]` • **DistanceBand**: `[threshold, p, alpha, binary]` • **Queen** • **Rook**

Parameter	Description
`save_weights_as`	Specifies how the object is stored in the data store. The value is a json file that determines the parameters of `oml.ds.save`. The supported parameters are: `[ds_name, obj_name, overwrite_ds, append, overwrite_obj, grantable, compression]`. Some parameter names slightly differ from those in the `oml.ds.save` function. The parameter `overwrite_obj` is used to indicate whether an already existing object should be replaced with the current object.
`spatial_col`	Specifies the column containing the geometries. The column can be specified in the table's metadata. If not specified, the column name is retrieved from the table.
`crs`	Specifies the Coordinate Reference System. If not specified, it is inferred from the table.

Example

The following code calculates the spatial weights of the dataset from the table specified in the `table` parameter, using the strategy defined in the `weights_def` parameter. This example uses the K-nearest neighbor approach with `K=4`. The result is saved into the spatial data store with the object name `la_bg_knn4`.

```
select *
    from table(
        pyqEval(
            par_lst => '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "weights_def": {"type": "KNN", "k": 4},
                "save_weights_as": {"ds_name": "spatial", "obj_name":
"la_bg_knn4", "append": true, "overwrite_obj": true}
            }',
            out_fmt => 'XML',
            scr_name =>  'compute_spatial_weights'
        )
    );
```

compute_global_spatial_autocorrelation

Format

```
compute_global_spatial_autocorrelation(table, column, weights=None,
weights_def=None,
    save_weights_as=None, spatial_col=None, crs=None)
```

Parameters

The parameters for this pre-defined function are described in the following table.

Parameter	Description
`table`	Name of the spatial table.
`column`	The column of interest whose values are used to compute the global autocorrelation index.

Parameter	Description
`weights`	Required when trying to use spatial weights already stored in the data store. Internally it calls the function `olm.ds.load`. The supported parameters are `ds_name` and `obj_name`, indicating the data store name and object name, respectively.
`weights_def`	Required if the parameter `weights` is not specified. Establishes the relationship between neighboring locations. This is passed as a json object specifying the type of the weights definition and its parameters. Each parameter is defined in detail in the API Reference documentation. The following lists the supported types and parameters: • **KNN**: `[k]` • **Kernel**: `[bandwidth, fixed, k, function]` • **DistanceBand**: `[threshold, p, alpha, binary]` • **Queen** • **Rook**
`save_weights_as`	Only used if `weights_def` is specified. Specifies how the spatial weights are stored in the data store. The value is a json file that determines the parameters of `oml.ds.save`. The supported parameters are: `[ds_name, obj_name, overwrite_ds, append, overwrite_obj, grantable, compression]`. Some parameter names slightly differ from those in the `oml.ds.save` function. The parameter `overwrite_obj` is used to indicate whether an already existing object should be replaced with the current object.
`spatial_col`	Specifies the column containing the geometries. The column can be specified in the table's metadata. If not specified, the column name is retrieved from the table.
`crs`	Specifies the Coordinate Reference System. If not specified, it is inferred from the table.

Example

The following example shows how to calculate Moran's I statistic from a specific table column using spatial weights already saved in a data store. It uses the `median_income` column and the spatial weights obtained from the code example in compute_spatial_weights.

```
select *
    from table(
        pyqEval(
            par_lst => '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "column": "median_income",
                "weights": {"ds_name":"spatial", "obj_name": "la_bg_knn4"}
            }',
            out_fmt => '{ "I": "NUMBER", "expected_I": "NUMBER",  "p_value":
"NUMBER", "z_value": "NUMBER" }',
            scr_name => 'compute_global_spatial_autocorrelation'
        )
    );
```

The output contains the following fields:

- The value of Moran's I statistic.

- The expected value under normality assumption.
- The p-value.
- The z-value.

The preceding example result will be similar to:

```
I    expected_I    p_value    z_value
0.6658882028    -0.0002910361    0.001    58.1778030148
```

If the spatial weights are not previously saved in a datastore, it is possible to calculate the Moran's I statistic and the spatial weights according to the `weights_def` parameter. The following code calculates Moran's I statistic of the `MEDIAN_INCOME` column and uses the Queen strategy (two observations are neighbors if they share at least a common vertex) to calculate the spatial weights, which are stored in the `spatial` datastore with the object name `la_bg_queen`.

```
select *
    from table(
        pyqEval(
            '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "column": "median_income",
                "weights_def": {"type":"Queen"},
                "save_weights_as": {"ds_name":"spatial", "obj_name":
"la_bg_queen", "append": true, "overwrite_obj": true}
            }',
            '{ "I": "NUMBER", "expected_I": "NUMBER",  "p_value": "NUMBER",
"z_value": "NUMBER" }',
            'compute_global_spatial_autocorrelation'
        )
    );
```

The output of the Moran's I statistic - the expected value under normality assumption, the p-value, and the z-value are as shown.

```
I    expected_I    p_value    z_value
0.6765793161    -0.0002910361    0.001    64.9421284293
```

You can list all the objects in a datastore using the `oml.ds.describe` function. The following code lists all the objects in the `spatial` datastore.

```
oml.ds.describe(name='spatial')
```

The output consists of all the objects in the `spatial` datastore, containing the previously created `la_bg_knn4` and `la_bg_queen` objects.

```
object_name         class    size  length  row_count  col_count
0    la_bg_knn4  OMLDSWrapper  696002       1          1          1
1   la_bg_queen  OMLDSWrapper  295285       1          1          1
```

compute_local_spatial_autocorrelation

Format

```
compute_local_spatial_autocorrelation(table, column, result_table=None,
key_column=None,
    weights=None, weights_def=None, save_weights_as=None, spatial_col=None,
crs=None)
```

Parameters

The parameters for this pre-defined function are described in the following table.

Parameter	Description
`table`	Name of the spatial table.
`column`	The column of interest whose values are used to compute the local autocorrelation index.
`result_table`	If specified, it calls the function `oml.create` to store a pandas `DataFrame` containing the results of the local autocorrelation in a table with the specified name.
`key_column`	If specified, the specified column is added to the resulting `DataFrame`. Otherwise, a column with the DataFrame's index is attached to the result.
`weights`	Required when trying to use spatial weights already stored in the data store. Internally it calls the function `olm.ds.load`. The supported parameters are `ds_name` and `obj_name`, indicating the data store name and object name, respectively.
`weights_def`	Required if the parameter `weights` is not specified. Establishes the relationship between neighboring locations. This is passed as a json object specifying the type of the weights definition and its parameters. Each parameter is defined in detail in API Reference documentation. The following lists the supported types and parameters: • **KNN**: `[k]` • **Kernel**: `[bandwidth, fixed, k, function]` • **DistanceBand**: `[threshold, p, alpha, binary]` • **Queen** • **Rook**
`save_weights_as`	Only used if `weights_def` is defined. Specifies how the spatial weights are stored in the data store. The value is a json file that determines the parameters of `oml.ds.save`. The supported parameters are: `[ds_name, obj_name, overwrite_ds, append, overwrite_obj, grantable, compression]`. Some parameter names slightly differ from those in the `oml.ds.save` function. The parameter `overwrite_obj` is used to indicate whether an already existing object should be replaced with the current object.
`spatial_col`	Specifies the column containing the geometries. The column can be specified in the table's metadata. If not specified, the column name is retrieved from the table.
`crs`	Specifies the Coordinate Reference System. If not specified, it is inferred from the table.

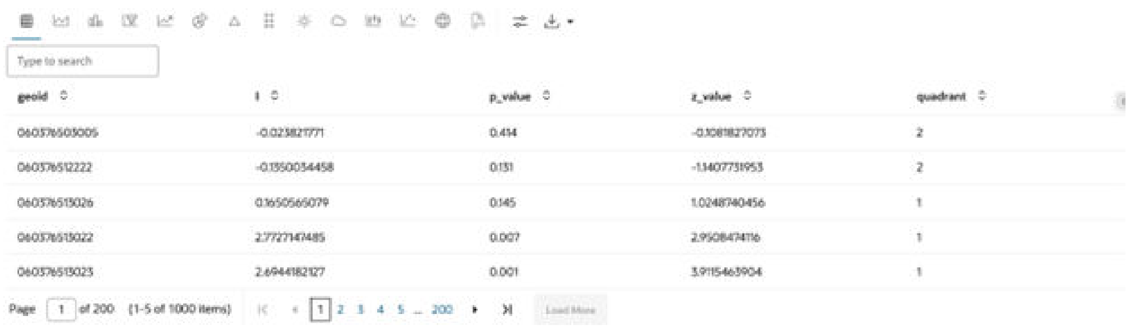
Example

The following example calculates the local Moran's I statistic for each row in a table from a specific column and uses the spatial weights already saved in a data store. It uses the `median_income` column and the spatial weights from the `spatial` data store with the object name `la_bg_knn4`, corresponding to the spatial weights calculated with the K-nearest neighbors method with K=4.

```
select *
    from table(
        pyqEval(
            '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "key_column": "geoid",
                "column": "median_income",
                "weights": {"ds_name":"spatial", "obj_name": "la_bg_knn4"}
            }',
            '{ "geoid": "VARCHAR2(50)", "I": "NUMBER", "p_value": "NUMBER",
"z_value": "NUMBER", "quadrant": "NUMBER" }',
            'compute_local_spatial_autocorrelation'
        )
    );
```

For each row in the table, the result contains the following:

- The local Moran's I statistic.
- The p-value.
- The z-value.
- The belonging quadrant.
    1. A high value surrounded by high values.
    2. A low value around high values.
    3. A low value surrounded by low values.
    4. A high value around high values.



Type to search

geoid	I	p_value	z_value	quadrant
060376503005	-0.023821771	0.414	-0.1081827073	2
060376512222	-0.1350034458	0.131	-1.1407731953	2
060376513026	0.1650565079	0.145	1.0248740456	1
060376513022	2.7727147485	0.007	2.9508474116	1
060376513023	2.6944182127	0.001	3.9115463904	1

Page 1 of 200 (1-5 of 1000 items) 1 2 3 4 5 ... 200 Load More

create_spatial_lag

Format

```
create_spatial_lag(table, column, strategy='mean', result_table=None,
key_column=None,
    weights=None, weights_def=None, save_weights_as=None, spatial_col=None,
crs=None)
```

Parameters

The parameters for this pre-defined function are described in the following table.

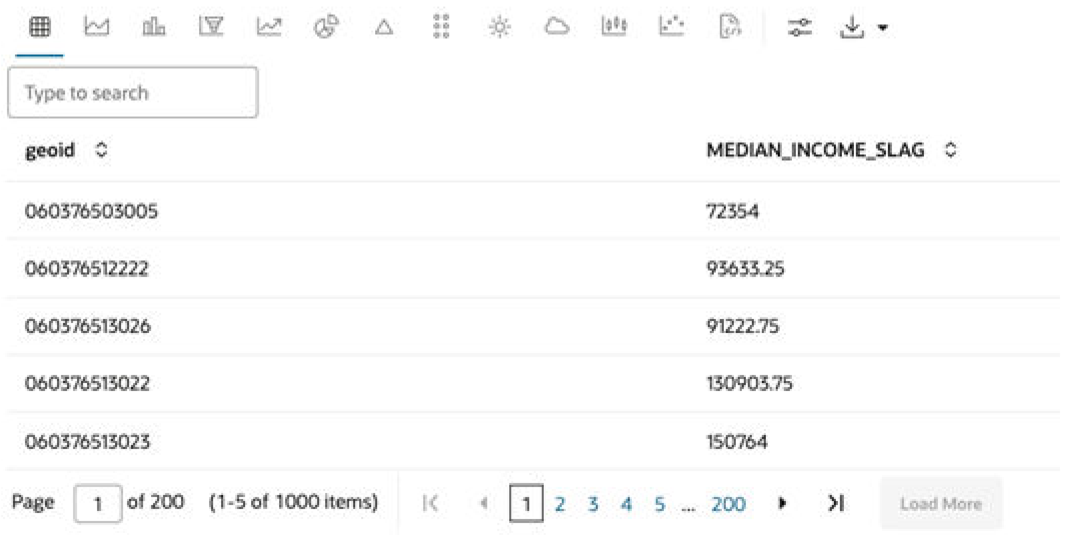
Parameter	Description
`table`	Name of the spatial table.
`column`	The column of interest whose values are used to compute the spatial lag.
`strategy`	The strategy to be used to calculate the spatial lag; there are two possible choices: *mean* and *median*.
`result_table`	If specified, it calls the function `oml.create` to store a pandas `DataFrame` containing spatial lag in a table with the specified name.
`key_column`	If specified, the specified column is added to the resulting `DataFrame`. Otherwise, a column with the index of the `DataFrame` is attached to the result.
`weights`	Required when trying to use spatial weights already stored in the data store. Internally it calls the function `olm.ds.load`. The supported parameters are `ds_name` and `obj_name`, indicating the data store name and object name, respectively.
`weights_def`	Required if the parameter `weights` is not defined. Establishes the relationship between neighboring locations. This is passed as a json object specifying the type of the weights definition and its parameters. Each parameter is defined in detail in the API Reference documentation. The following lists the supported types and parameters: • **KNN**: `[k]` • **Kernel**: `[bandwidth, fixed, k, function]` • **DistanceBand**: `[threshold, p, alpha, binary]` • **Queen** • **Rook**
`save_weights_as`	Only used if `weights_def` is defined. Specifies how the spatial weights are stored in the data store. The value is a json file that determines the parameters of `oml.ds.save`. The supported parameters are: `[ds_name, obj_name, overwrite_ds, append, overwrite_obj, grantable, compression]`. Some parameter names slightly differ from those in the `oml.ds.save` function. The parameter `overwrite_obj` is used to indicate whether an already existing object should be replaced with the current object.
`spatial_col`	Specifies the column containing the geometries. The column can be specified in the table's metadata. If not specified, the column name is retrieved from the table.
`crs`	Specifies the Coordinate Reference System. If not specified, it is inferred from the table.

Example

The following code calculates the spatial lag of a specific column according to given spatial weights. For each row, it calculates the average value of a particular column from neighboring locations. It uses the `median_income` column and spatial weights from a datastore.

```
select *
    from table(
        pyqEval(
            '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "key_column": "geoid",
                "column": "median_income",
                "weights": {"ds_name":"spatial", "obj_name": "la_bg_knn4"}
            }',
            '{ "geoid": "VARCHAR2(50)", "MEDIAN_INCOME_SLAG": "NUMBER" }',
            'create_spatial_lag'
        )
    );
```

The result contains the average income from neighboring locations for each row. Note that the index comes from the `key_column` parameter, which is the `geoid` column in this case.


Type to search

geoid	MEDIAN_INCOME_SLAG
060376503005	72354
060376512222	93633.25
060376513026	91222.75
060376513022	130903.75
060376513023	150764

Page 1 of 200 (1-5 of 1000 items) 1 2 3 4 5 ... 200 Load More

clustering

Format

```
clustering(table, method, scale=True, key_column=None, columns=None, 
weights=None, weights_def=None,
   save_weights_as=None, spatial_col=None, crs=None, to_crs=None, plot=None, 
**kwargs)
```

Parameters

The parameters for this pre-defined function are described in the following table.

Parameter	Description
`method`	A string specifying the clustering algorithm to execute. The options are: *DBSCAN*, *KMEANS* ,and *AGGLOMERATIVE*.
`scale`	If specified, it calls the function `oml.create` to store a pandas `DataFrame` containing spatial lag in a table with the specified name.
`key_column`	If defined, the specified column is added to the resulting pandas `DataFrame`. Otherwise, a column with the index of the `DataFrame` is attached to the result.
`columns`	An array of strings indicating the features that form the training set.
`weights`	Required when trying to use spatial weights already stored in the data store. Internally it calls the function `olm.ds.load`. The supported parameters are `ds_name` and `obj_name`, indicating the data store name and object name, respectively.
`weights_def`	Required if the parameter `weights` is not specified. Establishes the relationship between neighboring locations. This is passed as a json object specifying the type of the weights definition and its parameters. Each parameter is defined in detail in the API Reference documentation. The following lists the supported types and parameters: • **KNN**: `[k]` • **Kernel**: `[bandwidth, fixed, k, function]` • **DistanceBand**: `[threshold, p, alpha, binary]` • **Queen** • **Rook**
`save_weights_as`	Only used if `weights_def` is defined. Specifies how the spatial weights are stored in the data store. The value is a json file that determines the parameters of `oml.ds.save`. The supported parameters are: `[ds_name, obj_name, overwrite_ds, append, overwrite_obj, grantable, compression]`. Some parameter names slightly differ from those in the `oml.ds.save` function. The parameter `overwrite_obj` is used to indicate whether an already existing object should be replaced with the current object.
`spatial_col`	Specifies the column containing the geometries. The column can be specified in the table's metadata. If not specified, the column name is retrieved from the table.
`crs`	Specifies the Coordinate Reference System. If not specified, it is inferred from the table.
`to_crs`	If specified, the Coordinate Reference System will change to the specified value.
`plot`	A dictionary specifying the properties of the Plot Clusters function. If defined, a plot showing the resulting clusters is included in the response.

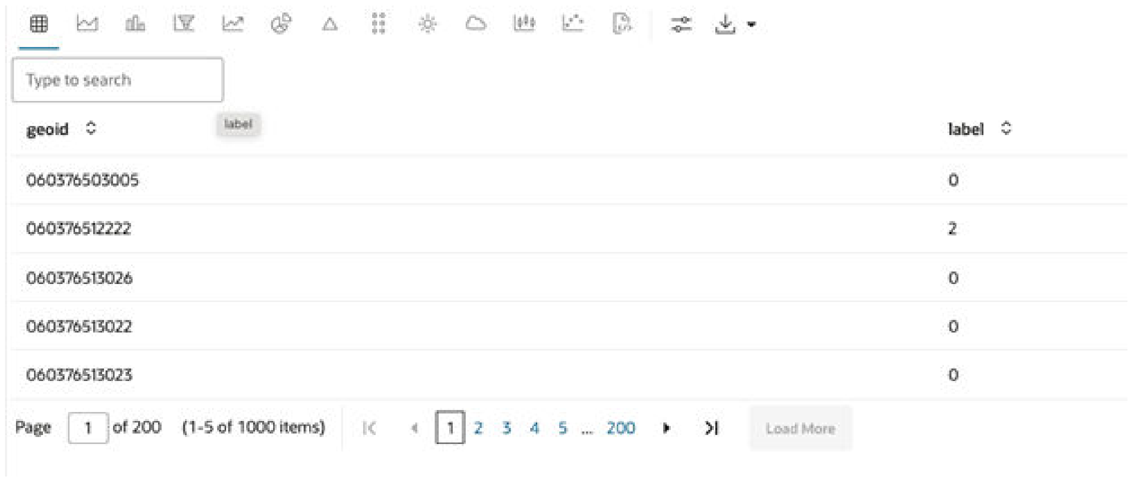
Example

This example shows how to run the agglomerative with regionalization algorithm over a given dataset, specifying the number of clusters and the type of spatial weights.

The clustering algorithm is set in the `method` parameter, while the number of clusters and the spatial weights are defined in the `n_clusters` and `weights_def` parameters respectively. The features considered for clustering are specified in the `columns` parameter.

```
select *
    from table(
        pyqEval(
            '{
                "oml_connect": true,
                "table": "oml_user.la_block_groups",
                "columns": ["median_income"],
                "method": "AGGLOMERATIVE",
                "n_clusters": 6,
                "key_column": "geoid",
                "weights_def": {"type": "Queen"}
            }',
            '{ "geoid": "VARCHAR2(50)", "label": "NUMBER" }',
            'clustering'
        )
    );
```

The result contains the index column specified in the `key_column` parameter and the labels of each row, indicating to which cluster they belong.

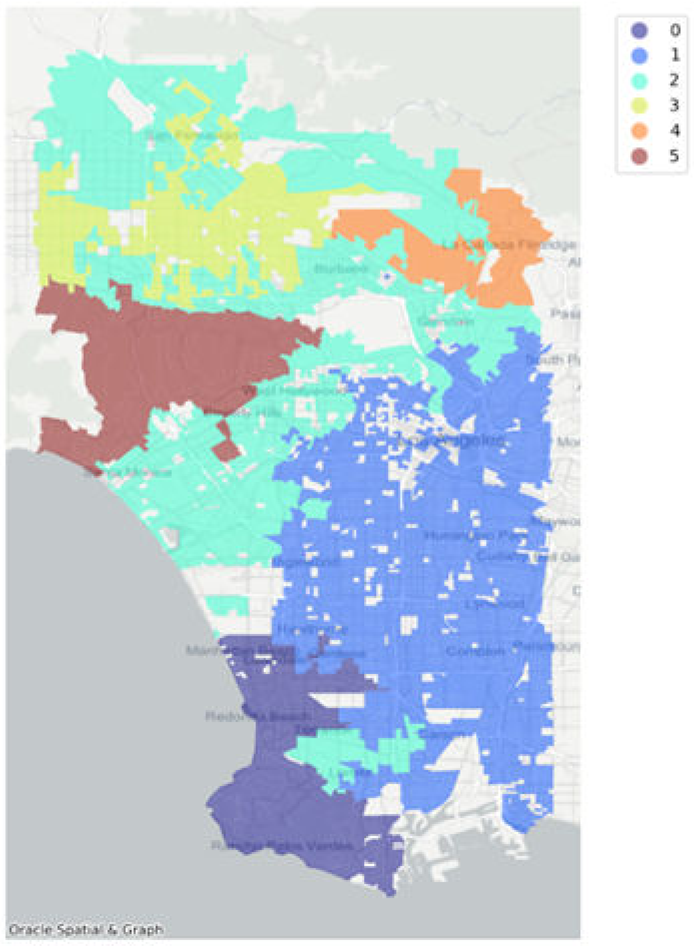


geoid	label
060376503005	0
060376512222	2
060376513026	0
060376513022	0
060376513023	0

You can visualize the clusters using the select IMAGE clause and the `oml_graphics_flag` parameter set to `true`. In the following code, the `plot` parameter indicates that it uses a basemap as background. Also, note that the output format (`out_fmt`) is set to *PNG*.

```
select IMAGE
    from table(
        pyqEval(
            par_lst => '{
            "oml_connect": true,
            "oml_graphics_flag": true,
            "table": "oml_user.la_block_groups",
            "columns": ["median_income"],
            "method": "AGGLOMERATIVE",
            "n_clusters": 6,
                "key_column": "geoid",
                "weights_def": {"type": "Queen"},
                "plot": {"with_basemap": true}
            }',
            out_fmt => 'PNG',
            scr_name => 'clustering'
        )
);
```

The result is a map with the observations colored according to the cluster they are assigned. Note that there are six clusters as specified in the `n_clusters` parameter. By defining spatial weights, the agglomerative clustering algorithm executes regionalization. This means that observations assigned to the same cluster share common characteristics and are geographically connected.



14 Review Use Cases for Using Spatial AI

Review the different use cases which show the end-to-end Spatial AI processes such as loading the data, understanding the data, analyzing the data, training a model, visualizing the results, and finalizing with post-processing tasks, such as saving a model in a datastore and creating functions for embedded Python execution.

Topics:

Spatial Regression Use Case Scenario

This use case aims to predict the median income of certain regions in the city of Los Angeles according to certain features from the Los Angeles Income Census dataset.

The dataset is stored in the `la_block_groups` table in the database. Based on spatial analysis and statistics, the example trains a regression model to estimate the median income.

During the exercise, different statistics are calculated to find out which spatial model is more suitable for the given data. `AdaptiveSpatialRegressor` internally computes all those statistics and suggests the best model. However, to show the analysis capabilities of Oracle Spatial AI, the exercise goes through different steps to find the best regression model.

The following steps enable you to solve a regression task using an OML notebook.

Load the Data

Perform the following steps to load the data:

1. Create an instance of `SpatialDataFrame`.

   The census dataset is stored in the `la_block_groups` table in the database. To load it into Python, use a `DBSpatialDataset` and create an instance of `SpatialDataFrame`.

   ```
   import oml
   from oraclesai import SpatialDataFrame, DBSpatialDataset

   block_groups =
   SpatialDataFrame.create(DBSpatialDataset(table='la_block_groups',
        schema='oml_user'))
   ```

   The dataset contains information about different regions in the city of Los Angeles, and features such as `median_income` and `house_value` provide information about each region's income. Other features provide demographic information about gender, race, and age.

2. Review the variables (shown in the following table) of the `SpatialDataFrame` instance and define the columns that represent the target variable, the explanatory variables, and the geometries.

Variable	Description
MEDIAN_INCOME	The target variable representing the median income.
MEAN_AGE	The average age.
MEAN_EDUCATION_LEVEL	Score based on the different education levels listed in the Census table.
HOUSE_VALUE	Median value of houses in the region.
PER_WHITE	Proportion of the white population in the region.
PER_BLACK	Proportion of the black population in the region.

The following code selects a subset of columns from the SpatialDataFrame instance.

```
X = block_groups[['MEDIAN_INCOME',
                  'MEAN_AGE',
                  'MEAN_EDUCATION_LEVEL',
                  'HOUSE_VALUE',
                  'INTERNET',
                  'geometry']]
```

3. Define the training, validation, and test sets.
   a. Split the data into training and test sets using the spatial_train_test_split function from oreaclesai.preprocessing. Assign 20% of the data for testing.

   ```
   from oraclesai.preprocessing import spatial_train_test_split

   X_train_valid, X_test, _, _, _, _ = spatial_train_test_split(X, 
   y="MEDIAN_INCOME", 
       test_size=0.2, random_state=32)
   ```

   b. Split the remaining 80% of the data again to create the training and validation sets, using 10% for validation and the rest for training. The validation set is helpful to evaluate the model's performance before using it with the test set.

   ```
   X_train, X_valid, _, _, _, _ = spatial_train_test_split(X_train_valid, 
   y="MEDIAN_INCOME", 
       test_size=0.1, random_state=32)
   ```

Explore the Data

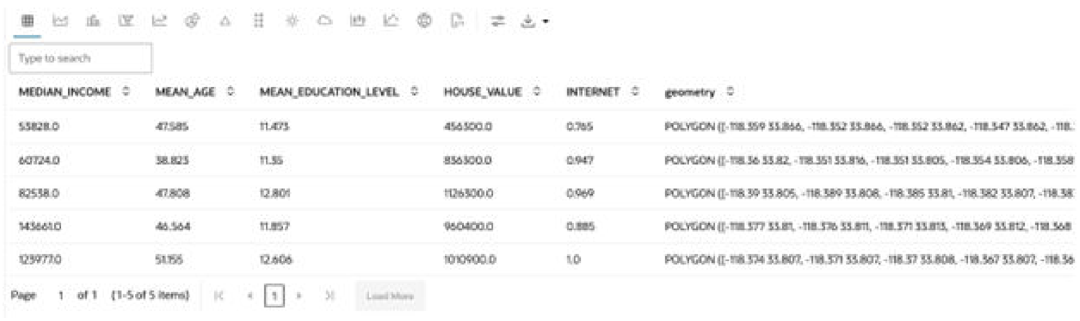
Exploring the data helps you to understand the variables individually and how they interact.

Perform the following steps to explore the data:

1. Understand the data by visualizing the first five observations of the training set using the head method.

   ```
   from oraclesai import enable_geodataframes
   enable_geodataframes(z)

   z.show(X_train.head())
   ```

   The output is as shown:



Type to search

MEDIAN_INCOME	MEAN_AGE	MEAN_EDUCATION_LEVEL	HOUSE_VALUE	INTERNET	geometry
53828.0	47.585	11.473	456300.0	0.765	POLYGON ((-118.359 33.866, -118.352 33.866, -118.352 33.862, -118.347 33.862, -118.
60724.0	38.823	11.35	836300.0	0.947	POLYGON ((-118.36 33.82, -118.351 33.816, -118.351 33.805, -118.354 33.806, -118.358
82538.0	47.808	12.801	1126300.0	0.969	POLYGON ((-118.39 33.805, -118.389 33.808, -118.385 33.81, -118.382 33.807, -118.38
143661.0	46.564	11.857	960400.0	0.885	POLYGON ((-118.377 33.81, -118.376 33.811, -118.371 33.813, -118.369 33.812, -118.368
123977.0	51.155	12.606	1010900.0	1.0	POLYGON ((-118.374 33.807, -118.371 33.807, -118.37 33.808, -118.367 33.807, -118.36

Page 1 of 1 (1-5 of 5 items)

2. Define spatial weights to understand the influence of each variable in neighboring locations (by establishing the relationship between neighboring locations).

   Use the K-Nearest Neighbor approach, which indicates that for each observation, the nearest K observations are considered neighbors.

   ```
   from oraclesai.weights import KNNWeightsDefinition

   weights_definition = KNNWeightsDefinition(k=10)
   ```

3. Calculate the spatial lag to study the interaction with the neighboring locations.

   The spatial lag of an observation represents the average value of a certain feature among its neighbors. For example, the average house value across neighboring locations.

   The following code calculates the spatial lag for all the variables in the training set, except the geometries.

   ```
   from oraclesai.preprocessing import SpatialLagTransformer

   X_spatial_lag = 
   SpatialLagTransformer(weights_definition).fit_transform(X_train)
   ```

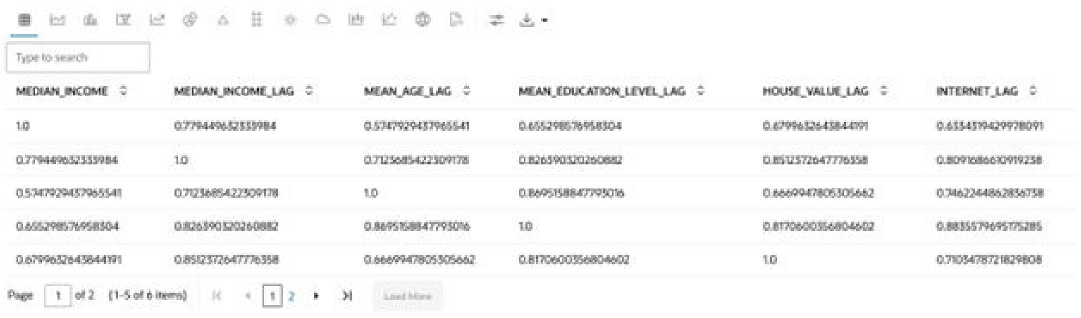
   According to Tobler's first law of geography, *everything is related to everything else, but near things are more related than distant things.* To understand the relation between features in a specific location, use the correlation between a feature and its spatial lag. For example, a strong positive correlation between the median income and the average income from neighboring locations could indicate an influence on the median income from its neighbors.

   The following code displays the correlation matrix of the spatial lag variables and the target variable, where the spatial lag variables have the suffix `_LAG`.

   ```
   import numpy as np
   import pandas as pd

   # Append the target variable to the spatial lag variables
   X_target_spatial_lag = np.append(X_train["MEDIAN_INCOME"].get_values(), 
   X_spatial_lag, 1)

   # Create a Pandas' DataFrame
   columns = ["MEDIAN_INCOME", "MEDIAN_INCOME_LAG", "MEAN_AGE_LAG", 
   "MEAN_EDUCATION_LEVEL_LAG", "HOUSE_VALUE_LAG", "INTERNET_LAG"]
   X_target_spatial_lag_df = pd.DataFrame(data=X_target_spatial_lag,
   ```

```
columns=columns)

z.show(X_target_spatial_lag_df.corr())
```

The output is as shown:



MEDIAN_INCOME	MEDIAN_INCOME_LAG	MEAN_AGE_LAG	MEAN_EDUCATION_LEVEL_LAG	HOUSE_VALUE_LAG	INTERNET_LAG
1.0	0.7794496632335984	0.5747929437965541	0.655298576958304	0.6799632643844191	0.6334319429978091
0.7794496632335984	1.0	0.7123685422309178	0.826390320260882	0.8512372647776358	0.8091686610919238
0.5747929437965541	0.7123685422309178	1.0	0.8695158847793016	0.6669947805305662	0.7462244862836738
0.655298576958304	0.826390320260882	0.8695158847793016	1.0	0.8170600356804602	0.8835579695175285
0.6799632643844191	0.8512372647776358	0.6669947805305662	0.8170600356804602	1.0	0.7103478721829808

4. Measure the influence of neighbors.

   There is a strong positive correlation between the target variable (`MEDIAN_INCOME`) and its spatial lag (`MEDIAN_INCOME_LAG`). This indicates that locations with similar income tend to be together, which is an indicator of spatial dependence.
   To confirm the presence of spatial dependence, calculate the Moran's I statistic, which measures spatial autocorrelation.

   - A positive and significant value indicates the presence of spatial clustering, where regions with similar values (high or low) tend to be together, reflecting the effect of spatial dependence.
   - A negative and significant value indicates the presence of spatial variance or the checkerboard pattern, reflecting the effect of spatial heterogeneity.

   ```
   from oraclesai.analysis import MoranITest
   from oraclesai.weights import SpatialWeights
   # Create spatial weights from definition
   spatial_weights = SpatialWeights.create(X_train["geometry"].values,
   weights_definition)

   # Run the Moran's I test
   moran_test = MoranITest.create(X_train, spatial_weights,
   column_name="MEDIAN_INCOME")

   # Print the Moran's I and the p-value
   print("Moran's I = ", moran_test.i)
   print("p_value = ", moran_test.p_value)
   ```

   The Moran's I statistic is positive and significant, confirming the presence of spatial dependence in the target variable.

   ```
   Moran's I =  0.5744827266749303
   p_value =  0.001
   ```

5. Get the spatial statistics.

Some spatial statistics become available by running the OLS model with spatial diagnostics. To get spatial diagnostics, it is required to define the spatial weights when creating the instance of `OLSRegressor`.

```
from oraclesai.regression import OLSRegressor

ols_model = OLSRegressor(weights_definition).fit(X_train, "MEDIAN_INCOME")
```

Obtain the Moran's I statistic from the model's residuals using the `moran_res metric` from `oraclesai.metrics`.

```
from oraclesai.metrics import moran_res

morans_i, _, p_value = moran_res(ols_model)

print(f"Moran's I = {morans_i}")
print(f"p_value = {p_value}")
```

The positive and significant value of Moran's I statistic of the residuals indicates the presence of spatial dependence in the residuals, which means that the prediction error of an observation is similar to the prediction error of its neighbors.

```
Moran's I = 0.2594180201084295
p_value = 9.432690077796932e-203
```

The two regression models , Spatial Lag Model and Spatial Error Model, include the effect of spatial dependence in their regression equation.

Use the Lagrange Multipliers tests from the spatial diagnostics of the trained OLS model to choose the best model for the data. The Lagrange Multiplier tests for Spatial Lag and Spatial Error are part of `oraclesai.metrics`.

```
from oraclesai.metrics import lm_lag, lm_error, rlm_lag, rlm_error

print(f"Lagrange Multiplier (lag): {lm_lag(ols_model)}")
print(f"Robust LM (lag): {rlm_lag(ols_model)}")
print(f"Lagrange Multiplier (error): {lm_error(ols_model)}")
print(f"Robust LM (error): {rlm_error(ols_model)}")
```

Use the robust tests when Lagrange Multiplier tests are significant for Spatial Lag and Spatial Error. Both robust tests are significant, but the value of the statistic for spatial error is much larger, indicating that the Spatial Error model is a better fit for the data.

```
Lagrange Multiplier (lag): (357.8764476978743, 8.165543828650201e-80)
Robust LM (lag): (10.656323334376838, 0.001096952308135397)
Lagrange Multiplier (error): (904.2345462924114, 1.178375337257614e-198)
Robust LM (error): (557.0144219289139, 3.750470342578867e-123)
```

Train the Model

The Spatial Error model introduces a spatial lag in the error term of the regression equation. By adding the spatial lag in the residual, the neighbors' errors influence the observation error.

The following code creates an instance of `SpatialErrorRegressor` and trains the model using a Spatial Pipeline with a preprocessing step to standardize the data.

```
from oraclesai.regression import SpatialErrorRegressor
from oraclesai.pipeline import SpatialPipeline
from sklearn.preprocessing import StandardScaler

# Create the instance of SpatialErrorRegressor
spatial_error_model =
SpatialErrorRegressor(spatial_weights_definition=weights_definition)

# Add the regressor to a spatial pipeline along with a pre-processing step
spatial_error_pipeline = SpatialPipeline([("scaler", StandardScaler()),
("spatial_error", spatial_error_model)])

# Train the Spatial Error model
spatial_error_pipeline.fit(X_train, "MEDIAN_INCOME")
```

The `summary` property of a regressor displays different statistics of the model and the estimated parameters. The following code gets the trained model and prints its summary.

```
# Get the trained model
error_model_fit = spatial_error_pipeline.named_steps["spatial_error"]

# Print the summary of the trained model
print(error_model_fit.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: MAXIMUM LIKELIHOOD SPATIAL ERROR (METHOD = FULL)
-------------------------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :     dep_var                Number of
Observations:        2475
Mean dependent var  :  69640.3568                Number of
Variables   :           5
S.D. dependent var  :  39961.9492                Degrees of
Freedom    :        2470
Pseudo R-squared    :      0.6285
Sigma-square ML     :454661980.170                Log likelihood        :
-28246.730
S.E of regression   :   21322.804                Akaike info criterion :
56503.460
                                                 Schwarz criterion     :
56532.530

------------------------------------------------------------------------------
------
            Variable     Coefficient       Std.Error     z-Statistic
Probability
------------------------------------------------------------------------------
------
            CONSTANT    70782.1082416    1248.2789978      56.7037564
```

```
0.0000000
            MEAN_AGE    2575.5035983     588.8525955       4.3737662
0.0000122
MEAN_EDUCATION_LEVEL    11051.5768223    1050.1057765      10.5242511
0.0000000
         HOUSE_VALUE    19081.0829838     814.4699114      23.4276094
0.0000000
            INTERNET    7640.9119411     682.4557729      11.1962009
0.0000000
              lambda       0.6563181       0.0239453      27.4090149
0.0000000
-------------------------------------------------------------------------------
------
================================ END OF REPORT
=====================================
```

Evaluate the Model

Although the `score` method of a regressor returns the R-squared metric, there are other metrics in `oraclesai.metrics` that are helpful to evaluate a model. For instance, the Akaike Information Criteria (AIC), which measures the amount of information lost by the model.

```
from oraclesai.metrics import aic

print(f"AIC: {aic(error_model_fit)}")

score_train = spatial_error_pipeline.score(X_train, y="MEDIAN_INCOME")
print(f"r2_score (X_train): {score_train}")

score_valid = spatial_error_pipeline.score(X_valid, y="MEDIAN_INCOME")
print(f"r2_score (X_valid): {score_valid}")
```

Having a validation set helps to evaluate the model before making predictions with the test set. Also, it can be used to determine if the model is overfitted or underfitted. The preceding code shows the following metrics.

```
AIC: 56503.460498197666
r2_score (X_train): 0.6212791699175433
r2_score (X_valid): 0.6417600931041549
```

Score

You can call the `predict` method to make predictions with the test dataset, and the `score` method to obtain the R-squared metric with the test dataset.

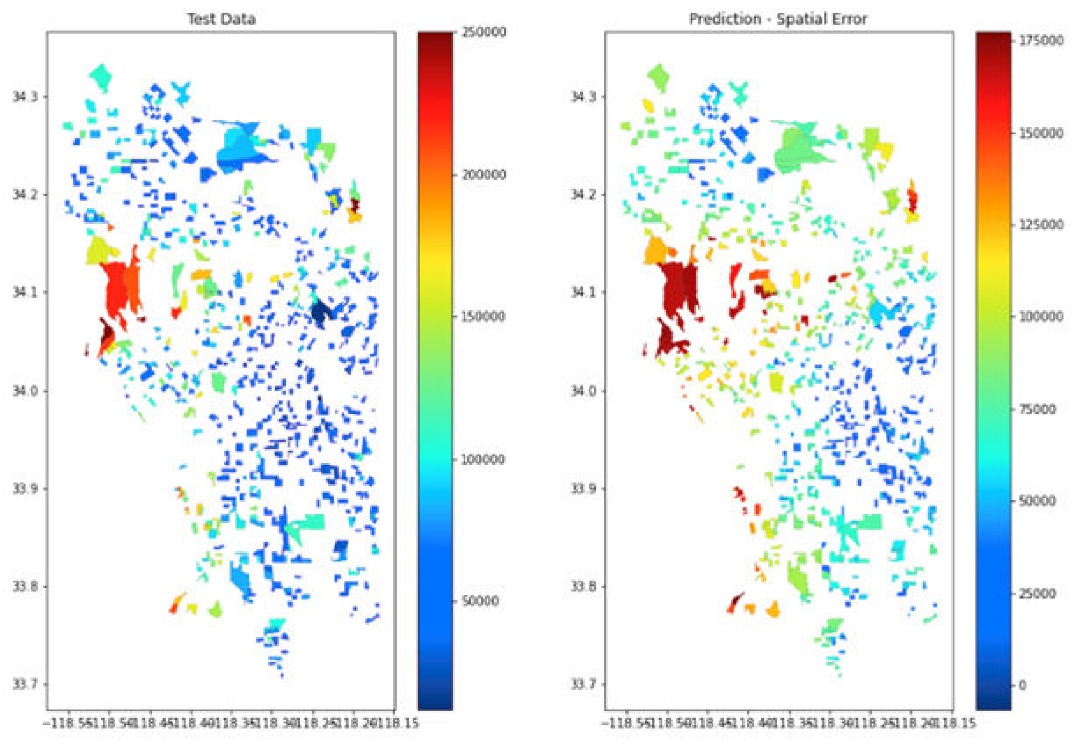
```
predictions_test =
spatial_error_pipeline.predict(X_test.drop(["MEDIAN_INCOME"])).flatten()
print(f"\n>> predictions (X_test):\n {predictions_test[:10]}")

score_test = spatial_error_pipeline.score(X_test, y="MEDIAN_INCOME")
print(f"\n>> r2_score (X_test):\n {score_test}")
```

The output is as shown:

```
>> predictions (X_test):
 [103705.9560757  107611.13674597  22112.24223308  37592.05306079
 170447.29190844  49590.69485066 104998.38030099  25865.98085974
  83318.68789415  15481.54002089]

>> r2_score (X_test):
 0.6433383305587677
```

The following code displays the test set and the model's predictions in a map.

```
import matplotlib.pyplot as plt
from oraclesai.vis import plot_geometries

fig, ax = plt.subplots(1, 2, figsize=(15,10))

# Set plot's labels and titles
ax[0].set_title('Test Data');
ax[1].set_title('Prediction - Spatial Error');

# Plot the choropleth map
plot_geometries(data=X_test, ax=ax[0], column=X_test["MEDIAN_INCOME"].values,
cmap=plt.get_cmap("jet"), legend=True, edgecolor='black', linewidth=0.1 )
plot_geometries(data=X_test, ax=ax[1], column=predictions_test,
cmap=plt.get_cmap("jet"), legend=True, edgecolor='black', linewidth=0.1 )
```



Run the Post-Processing Steps

Perform the following post-processing steps once the model is trained:

1. Save the model in a datastore.

   The following code save the model in an OML datastore as part of the post-processing step. The Spatial Pipeline is stored in the datastore named `sai_regressor_ds`, containing a dictionary with object names and Python objects.

   ```
   import oml

   oml.ds.save({'spatial_error_pipeline': spatial_error_pipeline},
      'sai_regressor_ds', description='some description', overwrite=True)
   ```

2. Load the model from a datastore.

   Use the `oml.ds.load` function to load the model from a datastore into Python for predictions by specifying the name of the datastore and the name of the Python object with the trained model.

   ```
   ds_objs = oml.ds.load('sai_regressor_ds', objs=['spatial_error_pipeline'],
   to_globals=False)
   error_model_loaded = ds_objs['spatial_error_pipeline']
   ```

3. Create and store a user-defined Python function that makes predictions with the trained model given a prediction set.

   The following code creates a Python user-defined function (UDF) that loads the trained model from a datastore and uses it to make predictions with a given prediction set. The UDF is then registered with OML using `oml.script.create`.

   ```
   udf = """def error_model_pipeline_predict_(X):
       import oml
       ds_objs = oml.ds.load('sai_regressor_ds',
   objs=['spatial_error_pipeline'], to_globals=False)
       error_model_pipeline = ds_objs['spatial_error_pipeline']
       pred = error_model_pipeline.predict(X)
       return pred.tolist()"""

   oml.script.create("errorModelPipelinePredict", udf, is_global=True,
   overwrite=True)
   ```

4. Run a Python UDF with SQL.

   The following code uses `pyqEval` to run the Python UDF `errorModelPipelinePredict` in SQL by passing the `X` parameter consisting of a single observation.

   ```
   select *
       from table(pyqEval(
           par_lst => '{"X": [[30.6005898, 12.1, 342200.000, 0.8]]}',
           out_fmt => 'JSON',
           scr_name => 'errorModelPipelinePredict'
           )
       );
   ```

The Moran's I statistic is positive and significant, confirming the presence of spatial dependence in the target variable.

The response shows estimated median income for the given observation.

```
NAME    VALUE
    [[67428.20461759513]]
```

Spatial Clustering Use Case Scenario

This use case identifies hots spots, colds spots, and outliers of the median income in the city of Los Angeles according to the Los Angeles Income Census dataset.

Based on spatial analysis and statistics, it trains a clustering algorithm based on the median income that finds hot spots, cold spots, and outliers.

This example shows two ways to identify hot spots, cold spots, and outliers. The first consists of executing a series of spatial analysis tasks, while the other consists of using the `LISAHotspotClustering` class.

The following steps enable you to get started on this use case using OML notebook.

Load the Data

The census dataset is stored in the `la_block_groups` table of the database. You can load it into Python using a `DBSpatialDataset` to create an instance of `SpatialDataFrame`.

```
import oml
from oraclesai import SpatialDataFrame, DBSpatialDataset

block_groups =
SpatialDataFrame.create(DBSpatialDataset(table='la_block_groups',
schema='oml_user'))
```

The dataset contains information about different regions in the city of Los Angeles. Features such as *`median_income`* and *`house_value`* provide information about each region's income. Other features provide demographic information about gender, race, and age.

Explore the Data

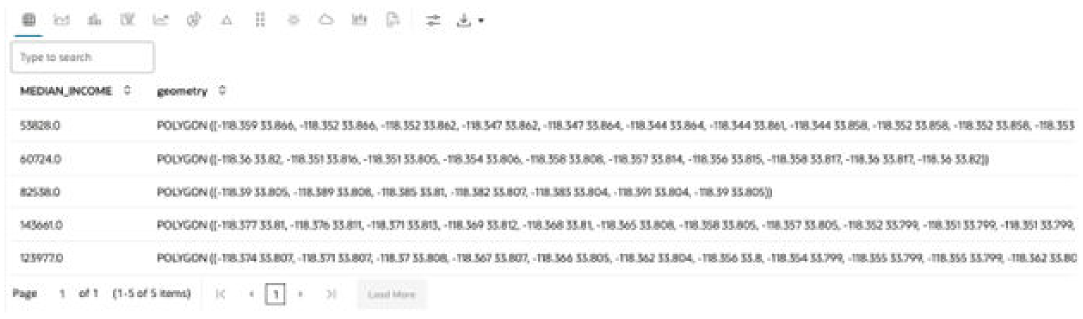
Exploring the data helps you to understand the variables individually and how they interact.

Perform the following steps to explore the data:

1. Understand the data by visualizing the first observations of the training set using the `head` method. The following example uses the `median_income` column .

   ```
   from oraclesai import enable_geodataframes
   enable_geodataframes(z)

   X = block_groups["MEDIAN_INCOME"]
   z.show(X.head())
   ```

   The output is as shown:



Type to search

MEDIAN_INCOME	geometry
53828.0	POLYGON ((-118.359 33.866, -118.352 33.866, -118.352 33.862, -118.347 33.862, -118.347 33.864, -118.344 33.864, -118.344 33.861, -118.344 33.858, -118.352 33.858, -118.352 33.858, -118.353
60724.0	POLYGON ((-118.36 33.82, -118.351 33.816, -118.351 33.805, -118.354 33.806, -118.358 33.808, -118.357 33.814, -118.356 33.815, -118.358 33.817, -118.36 33.817, -118.36 33.82))
82558.0	POLYGON ((-118.39 33.805, -118.389 33.808, -118.385 33.81, -118.382 33.807, -118.383 33.804, -118.391 33.804, -118.39 33.805))
143661.0	POLYGON ((-118.377 33.81, -118.376 33.811, -118.371 33.813, -118.369 33.812, -118.368 33.81, -118.365 33.808, -118.358 33.805, -118.357 33.805, -118.352 33.799, -118.351 33.799, -118.351 33.799,
123977.0	POLYGON ((-118.374 33.807, -118.371 33.807, -118.37 33.808, -118.367 33.807, -118.366 33.805, -118.362 33.804, -118.356 33.8, -118.354 33.799, -118.355 33.799, -118.355 33.799, -118.362 33.80

Page 1 of 1 (1-5 of 5 items) 1 Load More

2. Define spatial weights to understand the behavior of each variable in neighboring locations (by establishing the relationship between the neighboring locations).

   Use the K-Nearest Neighbor approach, which indicates that for each observation the nearest K observations are considered neighbors.

   ```
   from oraclesai.weights import KNNWeightsDefinition

   # Define spatial weights
   weights_definition = KNNWeightsDefinition(k=10)
   ```

3. Calculate the global spatial autocorrelation.

   The Moran's I statistic is a measure of spatial autocorrelation. It computes the global spatial autocorrelation if applied on the whole dataset.

   - A positive and significant value indicates the presence of spatial clustering, where regions with similar values tend to be together, reflecting the effect of spatial dependence.
   - A negative and significant value indicates the presence of spatial variance or the checkerboard pattern, reflecting the effect of spatial heterogeneity.

   The following code calculates the spatial lag for all the variables in the training set, except the geometries.

   ```
   from oraclesai.analysis import MoranITest
   from oraclesai.weights import SpatialWeights

   # Create spatial weights from definition
   spatial_weights = SpatialWeights.create(X["geometry"].values,
   weights_definition)

   # Run the Moran's I test
   moran_test = MoranITest.create(X, spatial_weights,
   column_name="MEDIAN_INCOME")

   # Print the Moran's I and the p-value
   print("Moran's I = ", moran_test.i)
   print("p_value = ", moran_test.p_value)
   ```

   The output of the program is as shown:

   ```
   Moran's I =  0.6086540661785302
   p_value =  0.001
   ```

A positive and significant value indicates the presence of Spatial Dependence, represented by clusters of observations with similar income. However, it does not indicate the location of such clusters.

4. Calculate the local spatial autocorrelation.

   Use the Local Indicators of Spatial Association (LISA) method to find the clusters. The algorithm calculates the Local Moran's I statistic for each observation.

   - A location with a positive local Moran's I statistic indicates the presence of neighbors with similar values (either high or low values), representing hot or cold spots.
   - A location with a negative local Moran's value indicates neighbor locations with different values; it can be a high value surrounded by low values or a low value surrounded by high values, representing spatial outliers.

   The `LocalMoranITest` class calculates each observation's local spatial autocorrelation index based on a specific feature and spatial weights. The following code prints the local autocorrelation index and p-values of the first ten observations in the dataset.

```
from oraclesai.analysis import LocalMoranITest

# Run the Local Moran's I test
local_moran_test = LocalMoranITest.create(X, spatial_weights, 
column_name="MEDIAN_INCOME")

# Print the Local Moran's I and  p-values
print("Local Moran's I = ", local_moran_test.i_list[:10])
print("p_values = ", local_moran_test.p_values[:10])
```

   The output of the code is as shown:

```
Local Moran's I =  [-0.28929661 -0.24813967  0.53874783  2.50789083  
2.59829807  0.96529687
  0.62729663  0.79068262 -0.00862826 -0.11777731]
p_values =  [0.025 0.003 0.001 0.001 0.001 0.019 0.015 0.088 0.336 0.119]
```

Train the Model

You can use the `LISAHotspotClustering` class for identifying clusters and outliers. It internally performs all the analysis done so far, calculating the Local Moran's I for each observation in the dataset and assigns them to the corresponding quadrant.

Depending on the build parameters, some observations can be labeled with –1 (undefined). For example, in the following code, setting the `max_p_value=0.05` parameter causes all observations with a p-value greater than 0.05 to be labeled with –1 in order to keep only statistically significant values.

```
from oraclesai.clustering import LISAHotspotClustering

# Create an instance of LISAHotspotClustering
lisa_model = LISAHotspotClustering(column="MEDIAN_INCOME",
                                   max_p_value=0.05,

spatial_weights_definition=weights_definition)

# Train the model
```

```
lisa_model.fit(X)

# Print the labels
lisa_labels = lisa_model.labels_
print(f"labels = {lisa_labels[:10]}")
```

The output of the program are the labels or quadrants assigned to the first ten observation of the training set.

```
labels = [ 2  2  1  1  1  1  1 -1 -1 -1]
```

Hot spots are labeled with the number 1, while cold spots are labeled with the number 3. To identify only hot and cold spots, perform the following.

```
import numpy as np

hotcold_labels = np.where(lisa_labels % 2 != 0, lisa_labels, -1)
```

A spatial outlier is an observation with a value different from its neighbors. These are represented with the label 2 and 4. To identify the spatial outliers, run the following code.

```
outlier_labels = np.where(lisa_labels % 2 == 0, lisa_labels, -1)
```

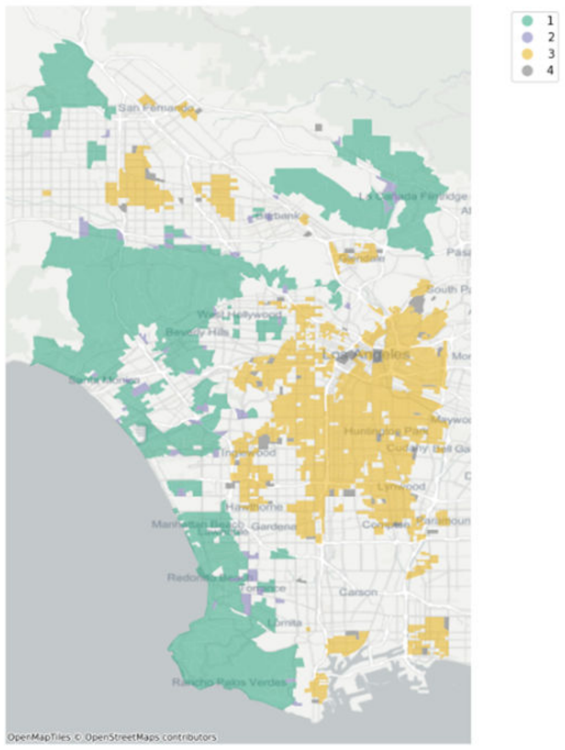
Visualize the Result

Oracle Spatial AI implements a plotting functionality for clusters in the `plot_clusters` function. The following code passes the training data and the labels as parameters. The `with_noise=False` parameter avoids displaying the observations labeled with –1. The `with_basemap=True` parameter sets a basemap as background.

```
from oraclesai.vis import plot_clusters

fig, ax = plt.subplots(figsize=(12,12))

plot_clusters(X, lisa_labels, with_noise=False, with_basemap=True,
cmap='Dark2', ax=ax)
```

The result consists of those observations with a local Moran's I statistically significant, colored according to their quadrant. Quadrants are defined as follows:

1. A high value surrounded by high values (hot spots).
2. A low value around high values (outliers).
3. A low value surrounded by low values (cold spots).
4. A high value around high values (outliers).



Run the Post-Processing Steps

Perform the following post-processing steps once the model is trained:

1. Store the trained model in a datastore for later use and to avoid training the model again.

   Use the `oml.ds.save` function specifying the name of the datastore and the name/object pair as shown in the following code:

   ```
   oml.ds.save({'lisa_model': lisa_model}, 'spatial_ai_ds',
   description='Hotspot Clustering for Median Income', overwrite=True)
   print(oml.ds.dir())
   ```

   Identify the recently created one from the directory of datastores.

   ```
           datastore_name  ...                          description
   0      agglomerative_ds  ...
   1   dbscan_accidents_ds  ...
   2      sai_regressor_ds  ...                     some description
   3               spatial  ...                                 None
   4         spatial_ai_ds  ...  Hotspot Clustering for Median Income
   5      spatial_error_ds  ...                     some description

   [6 rows x 5 columns]
   ```

2. Load the model from a datastore.

Use the `oml.ds.load` function to load the model from a datastore by specifying the name of the datastore and the name of the Python object with the trained model.

```
ds_objs = oml.ds.load('spatial_ai_ds', objs=['lisa_model'],
to_globals=False)
lisa_model_loaded = ds_objs['lisa_model']

print(lisa_model_loaded._labels[:10])
```

After loading the trained clustering model from a datastore, obtain the labels assigned to each observation with the `_labels` property. The preceding code prints the labels from the first ten observations:

```
[ 2  2  1  1  1  1  1 -1 -1 -1]
```

3. Create and store a user-defined Python function (UDF) that loads the trained model from a datastore and returns the labels assigned to the training data.

   The UDF is registered with OML using `oml.script.create`.

```
udf = """def get_lisa_labels_():
    import oml
    ds_objs = oml.ds.load('spatial_ai_ds', objs=['lisa_model'],
to_globals=False)
    lisa_model = ds_objs['lisa_model']

    return lisa_model._labels.tolist()"""

oml.script.create("lisaLabels", udf, is_global=True, overwrite=True)
```

4. Run a Python UDF with SQL.

   The following code uses `pyqEval` to run the Python UDF `lisaLabels` in SQL.

```
select *
    from table(pyqEval(
        par_lst => '{}',
        out_fmt => 'JSON',
        scr_name => 'lisaLabels'
    )
);
```

The response is the labels assigned to all the observations by the clustering algorithm. For simplicity, the following output shows only the first ten labels.

```
[2,2,1,1,1,1,1,-1,-1,-1,…]
```

A
Additional References

This section lists the additional references used in this book.

1. *Rahmah, N., and Sitanggang, I. S., 2016*. Determination of Optimal Epsilon (Eps) Value on DBSCAN Algorithm to Clustering Data on Peatland Hotspots in Sumatra. IOP Conf. Ser.: Earth Environ. Sci. 31 012012.
2. *Breunig, M. M., Kriegel, H., Ng, R. T. and Sander, J., 2000*. LOF: Identifying Density-Based Local Outliers. Proc. ACM SIGMOD 2000 Int. Conf. On Management of Data, Dalles, TX.
3. *Anselin, L., 2007*. Spatial Regression Analysis in R - A Workbook.
4. *Georganos, S., Grippa, T., Gadiaga, A. N., Linard, C., Lennert, M., Vanhuysse, S., Mboga, N., Wolff, E. and Kalogirou, S., 2021*. Geographical random forests: a spatial extension of the random forest algorithm to address spatial heterogeneity in remote sensing and population modelling. Geocarto International, 36(2), 121–136.
5. *Walker, B., 2019*. Synthetic Minority Over-sampling Technique (SMOTE) from Scratch. Medium.