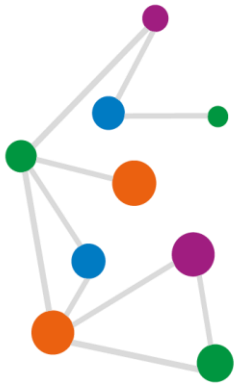


EIS
Exploration
Information
System

D 3.4: EIS Toolkit Beta Release

User Manual and Technical Specifications

Version 3.0

Lead Beneficiary: GISPO
04 / 2024

Niko Aarnio [1], Timo Aarnio[1], Miikka Kallio [1], Mika Sorvoja[1], Johanna Torppa[2], Bijal Chudasama[2], Ina Storch[3], Peggy Hielscher[3], Michael Steffen[3], Andreas Knobloch[3]

[1]GISPO
[2]GTK
[3]Beak Consultants GmbH

Funded by
the European Union

Disclaimer

The content of this report reflects only the author's view. The European Commission is not responsible for any use that may be made of the information it contains.

Funded by
the European Union

Document information

Grant Agreement / Proposal ID	101057357
Project Title	Exploration Information System
Project Acronym	EIS
Scientific Coordinator	Vesa Nykänen (vesa.nykanen@gtk.fi) – GTK
Project starting date (duration)	1 May 2022 (36 months)
Related Work Package	WP 3
Related Task(s)	Task 3.3
Lead Organisation	GISPO
Contributing Partner(s)	GTK, BEAK, UTU
Due Date	31.10.2023
Submission Date	09.11.2023, Revision: 06.12.2023, Revision: 05.04.2024
Dissemination level	PU

History

Date	Version	Submitted by	Reviewed by	Comments
26.10.2023	0.1	Andreas Knobloch		Template
06.11.2023	0.2	Timo Aarnio		Draft
08.11.2023	0.9	Andreas Knobloch		Revision
09.11.2023	1.0	Hafsa Munia		Submission
06.12.2023	2.0	Andreas Knobloch		Revision
07.12.2023	2.0	Hafsa Munia		Submission
04.04.2024	3.0	Andreas Knobloch		Revision
05.04.2024	3.0	Hafsa Munia		Submission

Funded by
the European Union

Table of Contents

Funded by
the European Union

List of figures

Abbreviations and Acronyms

Acronym	Description
WP	Work Package
GIS	Geographic Information System
EIS	Exploration Information System
GUI	Graphical User Interface

Summary

The Deliverable D3.4 provides a description of the contents and general structure of the EIS Toolkit beta release.

Keywords

Software Design, Mineral Prospectivity Modelling, Mineral Predictive Mapping, QGIS, Artificial Intelligence, Toolkit, Wizard, Python, Beta release

1. Executive summary

This document describes the first public beta release of the "EIS Toolkit", which is a standalone Python library for conducting Mineral Prospectivity Mapping. The toolkit utilizes industry-standard libraries for well-defined tasks and provides an interface for integrations both with the upcoming "QGIS EIS Plugin" and other software.

A common way to use this kind of a library is via Jupyter Notebooks, which is an interactive environment for running Python software and a sample of such use is included with this deliverable in **Appendix 3** both as a runnable/interactive notebook file and a static export that visualizes the workflow and its results. The interactive version works as a user manual for end users.

The technical documentation for "EIS Toolkit" is automatically generated and is described in chapter 4 and an export of the documentation (also in beta stage) is included with this deliverable in the **Appendix 4**.

As the toolkit is still in development phase it is not feature complete meaning that new functionalities (see Deliverable D3.3) will be added and current implementations will be further optimized with the help of feedback from users.

This and future releases and related documentation can be downloaded from "EIS Toolkit" GitHub repository which is located at: https://github.com/GispoCoding/eis_toolkit.

Funded by
the European Union

2. Introduction

2.1. General remarks

This document will provide background information about the beta release of "EIS Toolkit". The development is still in beta phase so not all functionality is included or fully optimized. The documentation for EIS Toolkit is currently in two forms:

- a Jupyter Notebook that guides the user through an example workflow (User Guide) (see **Appendix 3**) and
- a Technical Specification document that includes documentation for the actual EIS Toolkit functionality (see **Appendix 4**).

2.2. EIS WP3 and Task 3.3

The main objective of WP3 is the development of a GIS (Geographical Information System)-based Exploration Information System (EIS) for predictive mapping of mineral resources. EIS does not have a strict definition but can be characterized as an environment for performing data analysis and modelling, for managing data and other information, and for representing results in various forms.

In Task 3.3, a library of Python functions for EIS is implemented. We call this library the "EIS Toolkit", and it is a comprehensive collection of independent functions relevant for performing mineral prospectivity analysis related tasks. These tasks include mainly predictive mapping and data integration via mathematical modelling, but also some general data processing and analysis. Also, tools for evaluation of the goodness of the models and modelling results are included for efficient decision making in the identification and prioritization of exploration targets. The tools in the EIS toolkit are not specific for any specific type of mineral system, as same the same data processing, modelling and model validation methods are applicable to a number of mineral system types. The idea of the EIS toolkit is to bring together special functions needed for performing, and also to facilitate, mineral prospectivity analysis.

The "EIS Toolkit" contains implementations of existing and new algorithms (from Deliverable D3.3), and new functions can be added even after the EIS project. Emphasis is given to exploring the applicability of modern machine learning methods, such as convolutional neural networks, the use of which in mineral prospectivity modelling is still at its infancy. Existing Python libraries have been reviewed and are used to minimize the coding effort.

The structure of and interfaces to the functions included in the library have been planned (in Deliverable D3.1) so that the functions can be smoothly integrated to other software, such as the "EIS QGIS Plugin. The EIS QGIS Plugin (will be submitted later in the project through Deliverables 3.6 and 3.7) provides the platform (or the exploration information system) for entire mineral prospectivity analysis workflow, taking also into account the specific features of different mineral system types.

2.3. Purpose of the EIS Toolkit and user requirements

The main purpose of EIS Toolkit is:

- to collect tools needed in different phases of mineral potential analysis into a single Python library, to avoid searching through different libraries for suitable functions.

The EIS Toolkit was designed in strong connection with WP2 and WP6 to answer the following specific needs of the end users:

- to process several typical types of input data to represent mineral system proxies
- to transform proxies into a unified form for modeling
- to prepare a training set for machine learning from different forms of input data
- to evaluate input data, generated proxy data and mineral potential results

The developed tools are based on the defined proxies by EIS WP2, as defined in the Deliverable D2.1 “Preliminary list of pathfinders and vectors for mineral systems”.

Furthermore, EIS WP6 conducted a market analysis of potential end users and how the EIS Toolkit and EIS Wizard can be exploited most effectively. The results have been presented in EIS Deliverable D6.3 “Updated Version of the Communication, Exploitation and Dissemination Plan”.

In addition, in the frame of the EIS Toolkit design phase at the beginning of the project, the EIS Deliverable D3.1 “Design Report – EIS Toolkit & QGIS EIS Wizard” by EIS WP3 includes a brief market analysis of existing software packages and the user requirements.

3. Components of the EIS Toolkit and related questions by the users to be answered

The functions within the EIS Toolkit are designed to be universally applicable across all mineral system types, without specific components tailored to individual types. Instead, the mineral system type is considered in selecting the data that the functions are applied to.

Together with WP2 - through their submitted Deliverable D2.1 (Preliminary list of pathfinders and vectors for mineral systems) - a proxy table was generated for each of the three types of mineral systems considered in the EIS project. A proxy is the geochemical, geophysical or geological feature that is related to the mineral system and is the input to the prospectivity modelling process.

Proxies are obtained from measurements and observations through several geo-processing steps using ***pre-processing tools***. Example processing steps for the IOCG type mineral system are shown in Appendices 1 and 2. **Appendix 1** contains the four main workflows for processing the data. **Appendix 2** shows which workflow is connected with each of the proxies. In addition to the main workflows outlined in Appendix 1, certain proxies may require additional attributes such as density or distance to neighboring features to be calculated by the user. Alternatively, derivatives of raster data may need to be computed or raster files may need to be re-projected.

Exploratory analysis tools are available to assist in selecting representative threshold values for these tasks.

Once proxies are generated, further ***data transformations*** are necessary based on the chosen prediction tool for mineral prospectivity modeling. The ***training data tools*** module facilitates the generation or processing of training data, if required, while ***validation tools*** allow for the evaluation of the final prediction results.

All these functionalities are encompassed within the EIS Toolkit and are organized into highlighted modules to cover the entire workflow for mineral prospectivity modeling.

Funded by
the European Union

Figure 1. **The mineral prospectivity modelling (MPM) workflow (blue boxes) and EIS toolkit functions related to each phase of the workflow (black boxes). Yellow boxes provide a general description of tools needed in each phase of MPM.**

While the EIS Toolkit itself does not implement mineral system types or guide users through a Mineral Predictive Mapping (MPM) workflow, its contents have been carefully selected to support a comprehensive MPM workflow. This section aims to provide a structured approach to mineral prospectivity analysis and justify the necessity of each tool in satisfying user needs at each step. The tools are listed in an order that users are likely to follow, as defined within the EIS QGIS Plugin, although users may navigate between steps as needed.

Each tool included in the EIS Toolkit serves a specific purpose within the mineral prospectivity analysis workflow. The categories and tools presented here are intended to align with the concrete needs of users at each step. While the reasons for inclusion of each individual tool are not explicitly described in this section due to the beta release status of the EIS Toolkit, the tools are justified implicitly by addressing the need for each tool category. The categorization of tools may differ slightly within the toolkit itself, but this structure aims to provide clarity and guidance for users.

3.1. Pre-processing tools

Related User Questions:

- **How do I extract important information from my raw data?**
- **How do I produce datasets efficiently using a mineral system approach?**

EIS Tools:

- **Distance computation (Euclidean distance to vector features)**
- **Distance to anomaly (Euclidean distance to anomalous raster pixels)**
- **Density computation (density of vector features)**
- **Interpolation (vector features)**
    - **Inverse distance weighting (IDW)**
    - **Kriging interpolation (ordinary, universal)**

The resulting proxy datasets, defined in EIS as either distance raster or density raster, remain closely connected to the original phenomena or observations while being in a consistent format suitable for further processing in modeling tasks.

The Distance to anomaly workflow was identified as a common method for generating proxy data from raw raster data, particularly in geophysics. In this workflow, point data can be interpolated (optional if not already a raster file) and the raster is then reclassified with a user defined trash hold value defining anomalous values. Then, distances to anomalies are computed.

3.2. Exploratory data analysis tools

Related User Questions:

- **How do I explore the data I have?**
- **How do I select datasets for proxy processing or modeling?**
- **How can I define anomalous values in my data?**

EIS Tools:

- **Distribution plots: Histogram, KDE (kernel density estimation), ECDF (empirical cumulative distribution functions)**
- **Relational plots: Scatterplot, Lineplot, Pairplot**

- **Categorical plots: Boxplot, Barplot**
- **Other plots: Heatmap, Parallel coordinates plot**
- **Statistics: Descriptive statistics (Minimum, maximum, mean, quantiles, standard deviation, skewness), Chi-square test, Correlation matrix, Covariance matrix, Shapiro-Wilk Normality test**
- **Exploratory tools: PCA (Principal component analysis), Autoencoder model for image segmentation, Self-organizing map (SOM), K-means, DBSCAN (Density-Based Spatial Clustering of Applications with Noise), Feature importance**

The contents of exploratory data analysis have been selected with the mindset of providing a comprehensive set of different plots, statistics and methods to satisfy user needs. As such, not all tools will be needed for each analysis and some users might not be interested in using certain tools available. However, the categories and some of the tools can be highlighted here to be essential inclusions for EIS Toolkit.

Various plots are a powerful and standard way to inspect patterns in data. In practice, thresholds, correlations and magnitudes are often explored. Distribution plots are applicable for both vector and raster data, which makes them powerful and frequently used at all stages of analysis. Especially histograms are commonly used, as it is the most well-known distribution plot method. As an interesting and newer method to study data with many attributes or modalities, parallel coordinates plot was implemented.

Along with plotting, statistics are commonly used to understand the fundamental properties of data. Some statistics can be directly used for selecting a parameter value, such as threshold, for some tool such as Distance to anomaly in proxy processing phase, while other offer more interpretive – but necessary – information for decision-making. All included statistics were seen as commonly needed in MPM workflows.

The set of exploratory methods is a varied one and includes tools that might call for individual justification. Some of these tools can be used to support preparing proxies, while some offer an alternative approach to data processing. PCA is one of the most commonly used methods to reduce dimensionality of data, and in MPM can be used for example on a geochemical dataset with many different attributes. The output of PCA could be used to either inspect patterns in data, or be used directly in modeling later. Autoencoder for image segmentation, SOM, K-means clustering and DBSCAN are all primarily methods to inspect spatial patterns in the data, and together form a good supplementary set of data exploration. Feature importance is a method designed to support modeling by ranking the relevance of datasets in machine learning modeling.

3.3. Model data preparation tools

Related User Questions:

- **What data requirements does my chosen model have?**
- **What further proxy preprocessing is needed for optimal results?**
- **How do I address potential issues like data imbalance in my model?**

EIS Tools:

- **Machine learning data preparation: Balancing, Sampling, One-hot encoding**
- **General data preparation: Splitting datasets, Unifying raster, Resampling, Re-projecting, Clipping**
- **Data transformations: Reclassifying, Binarizing, Clip, Normalizing, Transforming, CoDa (compositional data transforms)**

Model data preparation consists of tools that are relevant for specific models and tools that the user might need or want to apply before proceeding to prediction phase. Data balancing, sampling and one-hot encoding are examples of model-specific methods that used to prepare data for machine learning methods. While data

balancing is not required but might improve the performance of the model, one-hot encoded data is strictly required for correct interpretation of categorical data. Scaling all datasets to same range, using same scaling method, is often necessary for modeling. The included data transformation functions satisfy this need in a very comprehensive way, offering multiple different methods. Perhaps the most commonly used one of these is min-max scaling data into range [0, 1], also called data normalization. To produce data that can be fed into Fuzzy overlay model, several of these methods can be utilized to produce the desired memberships (gaussian, large, linear, near, power, small). Of the listed tools under General data preparation, Unify raster is probably the most useful one to ensure all datasets are matching in their shape and size. Many of the other tools of the category are used internally by Unify raster, but made available for users that want to preprocess step-by-step.

3.4. Prediction/modelling tools

Related User Questions:

- **How can I predict mineral deposit locations?**
- **What machine learning models are suitable for my task?**
- **How do I evaluate and apply trained models?**

EIS Tools:

- **Train/Test various machine learning models: Random forest, Gradient boosting, Logistic regression, MLP, CNN**
- **Apply trained models**
- **Data-driven methods: Fuzzy overlay, Weights of evidence**

The prediction methods included in EIS Toolkit can be grouped into two categories: machine learning methods and data-driven methods. Fuzzy overlay and Weights of evidence constitute the data-driven methods and the rest are machine learning modeling related tools. As both Fuzzy overlay and Weights of evidence are commonly used methods in ML modeling, they were a natural inclusion. For all machine learning methods where applicable, both classifier and regressor models were included to cover all possible prediction tasks, even if classifiers are more direct to predict mineral occurrences. Of the machine learning methods, especially Random forest models have been widely applied in MPM contexts. On the other end, CNN models have only recently been built for mineral prediction and therefore represent the more innovative side of EIS Toolkit, bringing emerging and modern tools for advanced MPM. The rest of the machine learning models fall somewhere in-between these two in respect to their popularity, and were included to offer a comprehensive set of methods to choose from.

3.5. Evaluation tools

Related User Questions:

- **Is my model effectively predicting mineral deposits?**
- **How can I assess the performance of my model?**
- **Should I adjust my model's parameters or data?**

EIS Tools:

- **Classifier metrics (Accuracy, Precision, Recall, F-score)**
- **Regressor metrics (MAE, MSE, RMSE, R2)**
- **Area under curve (AUC)**

- **Prediction area (PA) curve**
- **Receiver Operating Characteristic (ROC) curve**
- **Confusion matrix**
- **Loss function (for neural networks)**

Evaluation methods allow users to assess the effectiveness of trained models using a variety of metrics and visualization tools. Both classifier and regressor metrics are available, along with ROC and PA curves for classification tasks and loss functions for neural networks. ROC curve has been included, since it can be a useful tool to when assessing one or more classifiers at varying classification thresholds and has been used in MPM contexts. For neural networks, classifier and regressor evaluation methods can be applied, but the loss function provides additional insight into the training performance.

3.6. Additional tools

- **Cell-based association**
- **Extract shared lines of vector features**
- **Rasterize vector**
- **Reproject vector**
- **Calculate vector feature geometry**
- **Create constant raster**
- **Extract values from raster**
- **Unique combinations in raster**
- **Surface derivatives**
- **Filters**
- **Replace data/no data**
- **Harmonize no data**

Additional tools not directly aligned with the previous steps are included in the toolkit to provide alternative or supplementary methods for data processing and analysis. These tools offer insights into data characteristics and address specific user needs, enhancing the toolkit's versatility and utility. Examples include tools for rasterizing vectors, reprojecting data, and computing surface derivatives.

By following the structured approach outlined in this section and leveraging the functionalities provided by the EIS Toolkit, users can efficiently preprocess data, generate proxies, train models, and evaluate predictions for mineral prospectivity mapping, ensuring a robust and informed decision-making process.

4. User manual

This chapter outlines how to use "EIS Toolkit", what are the system requirements, how it can be installed and a Jupyter Notebook for interactive use with guidance. Additionally, a static export of the Jupyter Notebook is provided that visualizes an example workflow when done inside Jupyter Notebook.

4.1. System requirements

The only system requirement for installing the "EIS Toolkit" through the Python package is to have a supported version of Python installed on the system. The "EIS Toolkit" supports all Python 3.9.x and 3.10.x versions. Setting up a development environment requires cloning the source code from GitHub, and additional requirements based on preferences. The environment can be set up using either Docker, Poetry or Conda. Note that setting up the development environment with Conda requires Python 3.9.

4.2. Installation guide

There are multiple ways to set up the "EIS Toolkit". The simplest and most common way is to install the "EIS Toolkit" Python package through pip. This can be achieved with a single command. The package automatically installs all the necessary Python libraries, and the "EIS Toolkit" will be ready to use. The "EIS Toolkit" Python package can be downloaded from the GitHub repository under releases.

The "EIS Toolkit" can also be installed by setting up the development environment. Clone the source code from GitHub, and follow the preferred instructions found under instructions/ directory:

- Setting up with Docker.
- Setting up with Poetry.
- Setting up with Conda.

4.3. Jupyter Notebook description

JupyterLab is included as a development dependency for testing purposes. It can be used for example to visualize data and test "EIS Toolkit" functions. The notebooks are found under the notebooks/ directory. You can import and use the toolkit's functions in these notebooks.

There is a notebook that contains general usage instructions for running and modifying Jupyter Notebooks and another one for testing that dependencies to other python packages work. At the time of writing (November 2023), there are a few notebooks to showcase some of the toolkit's implemented functions and a notebook to demonstrate the workflow (please see Appendix 1).

Instructions to run JupterLab can be found in GitHub.

5. Technical specifications

The "EIS Toolkit" encompasses a comprehensive suite of functions designed to facilitate efficient and effective mineral exploration. These functions are meticulously organized into distinct modules, each serving a specific purpose within the exploration workflow. Technical specifications for the "EIS Toolkit" functions are automatically generated from docstrings, which are extensively documented within the source code. Below is an overview of the key modules.

For further details on the technical specifications and functionalities of each module, please refer to **Appendix 4**

In the following sections, the numbers in brackets refer to the chapters / section numbers in the Technical Specifications in Appendix 4.

5.1. Conversions module

The Conversions Module in the "EIS Toolkit" provides essential functionalities for converting different types of geospatial data, facilitating seamless integration and analysis within the exploration workflow.

CSV to Geo Data Frame Conversion (3.1):

This function enables users to read CSV files containing geospatial data and convert them into Geo Data Frames. It supports the transformation of point data represented by X and Y coordinates or geometric shapes specified in Well-Known Text (WKT) format. Users can also define the target Coordinate Reference System (CRS) for the resulting Geo Data Frame.

Raster to Data Frame Conversion (3.2):

The Raster to Data Frame conversion function allows users to convert raster datasets into Pandas Data Frames. Users can select specific bands from multi-band raster or utilize all bands for conversion. Additionally, pixel coordinates (row, col) can be optionally included in the Data Frame for each pixel of the raster, providing valuable spatial context to the data.

By utilizing these conversion functionalities, users can seamlessly transform and prepare diverse geospatial datasets for further analysis and exploration within the "EIS Toolkit."

5.2. Exploratory analyses module

The Exploratory Analyses Module within the "EIS Toolkit" encompasses a diverse range of functions aimed at uncovering insights and relationships within geospatial data, facilitating informed decision-making in mineral exploration endeavors.

DBSCAN Clustering (4.1):
DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is utilized to identify clusters within geospatial data based on density. By specifying parameters such as maximum distance and minimum samples, users can effectively identify spatial patterns indicative of mineralization occurrences.

Descriptive Statistics (4.2):
This function computes descriptive statistics from both vector and raster data, providing insights into the distribution and variability of geospatial attributes essential for exploration analysis and decision-making.

Feature Importance Evaluation (4.3):
Evaluate the importance of features derived from geospatial data using machine learning models. By assessing the contribution of each feature to model performance, users can prioritize exploration targets effectively.

K-means Clustering (4.4):
Perform K-means clustering on geospatial data to partition it into distinct clusters. This aids in identifying spatially coherent regions or anomalies associated with mineralization occurrences.

Plot Parallel Coordinates (4.5):
Generate parallel coordinates plots to visualize multivariate relationships within geospatial datasets. This aids in identifying patterns and trends across multiple variables, enhancing understanding of mineralization indicators and exploration targets.

Principal Component Analysis (PCA) (4.6):
Conduct PCA to reduce the dimensionality of geospatial datasets while preserving essential information. Visualization of principal components assists in identifying spatial patterns and relationships crucial for mineral exploration.

Statistical (Hypothesis) Testing (4.7):
Compute statistical tests such as the Chi-square test for independence to assess relationships between categorical variables. Additionally, functions are available for computing correlation matrices, covariance matrices, and testing for normality, providing further insights into the underlying structure and characteristics of geospatial data.

Through the utilization of these exploratory analyses functions, users can gain valuable insights into the spatial characteristics of geological features and mineral occurrences, facilitating more informed and effective decision-making in mineral exploration endeavors.

5.3. Prediction module

The Prediction Module in the "EIS Toolkit" equips users with powerful tools for predicting mineral occurrences and evaluating prospectivity.

Fuzzy Overlay Operations (5.1):
This suite of functions enables users to perform various fuzzy overlay operations on raster data. By utilizing fuzzy logic techniques such as AND, OR, product, sum, and gamma overlay, users can analyze multi-dimensional geospatial data to identify areas of interest and assess mineral prospectivity.

Weights of Evidence Analysis (5.2):
These functions facilitate the calculation of weights of evidence, allowing users to assess the spatial associations between input data and mineral deposits. By determining posterior probabilities and weights of spatial association, users can make informed decisions regarding mineral exploration targets and planning.

Through the capabilities offered by the Prediction Module, users can leverage fuzzy logic and weights of evidence

5.4. Raster processing module

The Raster Processing module offers a comprehensive suite of functions for efficiently manipulating raster data, ensuring its compatibility and suitability for various analytical tasks. These tools enable users to address diverse raster data processing needs, including data validation, spatial analysis, and format standardization. By providing a robust set of functionalities, the module empowers users to seamlessly manage and optimize raster data for subsequent analysis and modeling tasks, enhancing the efficiency and accuracy of geospatial workflows.

Check Raster Grids (6.1):
This function ensures consistency in gridding among input raster, optionally checking for matching bounds.

Clipping (6.2):
Clips a raster with polygon geometries, extracting specific regions of interest.

Create Constant Raster (6.3):
Generates a constant raster based on user-defined parameters, offering flexibility in defining extent and coordinate system.

Extract Values from Raster (6.4):
Extracts raster values using point data to a Data Frame, facilitating data extraction for analysis.

Re-projecting (6.5):
Re-projects a raster to match a given coordinate reference system (CRS), ensuring compatibility between datasets.

Resampling (6.6):
Resamples a raster according to a specified resolution, maintaining data integrity during transformation.

Snapping (6.7):
Aligns a raster to a reference grid raster, ensuring alignment for consistent analysis.

Unifying (6.8):
Unifies given raster relative to a base raster by re-projecting, resampling, aligning, and optionally clipping them. This operation ensures consistency in grid properties and extents across multiple raster.

Windowing (6.9):
Extracts a window from a raster centered at specified coordinates, allowing for localized analysis. Padding with no data values is applied if the window extends beyond the raster bounds.

5.5. Training data tools module

The Training Data Tools module equips users with essential functionalities for preparing training datasets, particularly in the context of machine learning tasks. By offering these tools, the module facilitates the creation of balanced and well-prepared training datasets, crucial for achieving optimal model performance and accuracy in machine learning applications

Class Balancing (7.1):
With this function users can address class imbalance issues by applying the SMOTETomek resampling method, ensuring a more representative distribution of classes in the training data. It adjusts the class distribution by

oversampling the minority class and under-sampling the majority class to mitigate class imbalance issues. Users can specify various parameters such as the sampling strategy and random state to customize the resampling process. The function returns the resampled feature matrix and target labels, ensuring a more balanced dataset for training machine learning models

5.6. Transformations module

The Transformations module provides a suite of functions for preprocessing raster data, enabling users to apply various transformations to prepare their data for modeling and analysis. These functions include binarization, clipping, normalization, logarithmic transformation, sigmoid transformation, and winsorization, offering flexibility in data preprocessing to suit different modeling needs. Whether it's transforming data into binary format, clipping values based on specific thresholds, or normalizing data to a standard scale, these tools empower users to preprocess raster data effectively before further analysis or modeling tasks.

Binarize (8.1):
This function binarizes raster data based on a given threshold. It replaces values less than or equal to the threshold with 0 and values greater than the threshold with 1.

Clip (8.2):
The clip transform function clips raster data based on specified upper and lower limits. It replaces values below the lower limit and above the upper limit with provided values.

Linear (8.3):
Min-max-scaling normalizes raster data based on a specified new range. It transforms the data into the new interval defined by the provided minimum and maximum values.

Z-Score Normalization (8.3):
This function normalizes raster data based on the mean and standard deviation. The resulting data will have a mean of 0 and a standard deviation of 1.

Logarithmic (8.4):
The log-transform function performs a logarithmic transformation on the provided raster data. It replaces negative values with a specified no data value and applies the logarithmic transformation based on the chosen base.

Sigmoid (8.5):
The sigmoid-transform transforms data into a sigmoid shape based on a specified new range. It uses parameters such as minimum, maximum, slope, and center to perform the transformation.

Winsorize (8.6):
The winsorize function winsorizes raster data based on specified percentile values. It replaces values outside the specified percentile ranges with the nearest values within the range.

5.7. Validation module

The Validation Module in the "EIS Toolkit" provides essential tools for evaluating the performance and reliability of predictive models and data analyses.

Calculate AUC (9.1):
This function computes the area under the curve (AUC), which is a crucial metric for assessing the performance of classification models. By providing X and Y values corresponding to false positive rate and true positive rate, respectively, users can obtain a quantitative measure of model performance.

Calculate Base Metrics (9.2):
The Calculate Base Metrics function computes true positive rate, proportion of area, and false positive rate values for different thresholds. It leverages mineral deposit locations and mineral prospectivity maps to evaluate model performance comprehensively.

Get P-A Plot Intersection Point (9.3):
This function calculates the intersection point for prediction rate and area curves in a prediction-area (P-A) plot. By analyzing threshold values along with true positive rate and proportion of area values, users can identify optimal thresholds for model evaluation.

Plot Correlation Matrix (9.4):
The Plot Correlation Matrix function generates a heatmap visualization of the correlation matrix. This visualization aids in understanding the relationships between variables, providing insights into potential multicollinearity and the overall structure of the data.

Plot Prediction-Area (P-A) Curves (9.5):
This function plots prediction-area (P-A) curves, which are valuable for evaluating mineral prospectivity maps and evidential layers. By visualizing true positive rate and proportion of area values against threshold values, users can assess the performance of predictive models.

Plot Rate Curve (9.6):
The Plot Rate Curve function generates success rate, prediction rate, or ROC curves based on provided X and Y values. This visualization helps users understand the trade-offs between true positive rate and false positive rate, aiding in model selection and evaluation.

5.8. Vector processing module

The Vector Processing Module in the "EIS Toolkit" offers a range of functionalities for handling and analyzing vector data, facilitating geospatial analysis tasks.

Cell-Based Association (10.1):
This function initializes a Cell-Based Association (CBA) matrix from vector files. It calculates the mesh based on geometries contained in the file and cell size specified by the user. Users can add multiple vector datasets to the matrix, incorporating targeted shapes and attributes for comprehensive analysis.

Distance Computation (10.2):
The Distance Computation function calculates the distance from raster cells to the nearest geometry in a given set of vector data. This tool is valuable for proximity analysis and understanding spatial relationships between raster and vector datasets.

IDW (10.3):
The Inverse Distance Weighted (IDW) interpolation function performs spatial interpolation on vector data, generating a raster output. Users can specify the target column containing values for interpolation, resolution of the output raster, and the power parameter to control the rate at which weights decrease with distance.

Kriging Interpolation (10.4):
This function implements Kriging interpolation on input vector data, producing a raster output with interpolated values. Users can select the variogram model, coordinates type, and method (ordinary or universal) for Kriging interpolation, providing flexibility in spatial analysis tasks.

Rasterize Vector (10.5):
Transforming vector data into raster format is facilitated by the Rasterize Vector function. Users can specify parameters such as resolution, value column, default value, and merge strategy to customize the rasterization process according to their analysis requirements.

Re-project Vector (10.6):
Re-projecting vector data to match a given coordinate reference system (CRS) is achieved with the Re-project Vector function. Users can specify the target CRS using its EPSG code, ensuring consistency and compatibility across geospatial datasets.

Vector Density (10.7):
The Vector Density function computes the density of geometries within a raster grid. Users can define parameters such as resolution, base raster profile, buffer value, and statistic (e.g., density).

6. Initial testing

6.1. Listing of initial testers

The initial testing is conducted by a small group of voluntary testers within the project who have not been actively involved in the development of "EIS Toolkit". The initial testers are:

- Ina Storch (Beak),
- Roberto De La Rosa (Beak),
- Alex Vella (BRGM),
- Martiya Sadeghi (SGU),
- Patrick Casey (SGU),
- Sinovuyo Busakwe (UTU) and
- Gonzalo Ares (CSIC).

Testing is supported by other project members, primarily developers of "EIS Toolkit".

6.2. Listing and description of initial test data for testers

In the first stage of testing, test data consists of two parts: small (< 1 MB) geospatial datasets uploaded in the GitHub repository and a set of IOCG data uploaded in cloud. The small datasets found in GitHub are meant only to test that the tools work without crashes and errors and cannot be used to perform a full mineral prospectivity mapping workflow. In contrast, the IOCG dataset was made available to perform full test analyses from beginning to end using "EIS Toolkit", and to enable identifying tools needing performance optimization.

The IOCG dataset consists of the following files:

- Deposits
  - Deposit occurrences datasets
- Geochemical data
  - Geochemical dataset
  - Interpolated copper concentration dataset
  - Interpolated cobalt concentration dataset
  - Interpolated iron concentration dataset
- Geological data
  - Tectonic setting dataset
  - Structures dataset
  - Lithology dataset
  - Target area dataset
  - Distances to structures dataset
  - Tectonic setting dataset
- Geophysical data
  - AEM datasets

Funded by
the European Union

    - Gravity datasets
    - Magnetic dataset
    - Radiometric dataset
- Discretized data (data prepared for Weights of evidence modeling)

6.3. Guidelines for testers to report bugs and problems

Development of "EIS Toolkit" is done in GitHub, and all initial testers with a GitHub account should use repository of "EIS Toolkit" to report found bugs and to send modification suggestions. The most practical way to do this is to use *GitHub issues*, which are widely used in programming projects for reporting bugs and development related tasks.

The "EIS Toolkit" GitHub page has basic instructions for developing, testing and using the toolkit, and the initial testers should refer to guidance found in there. In addition, a more detailed "Getting started with testing" document was sent to all initial testers.

In the initial testing phase, the possibility to report bugs and other findings via email was given to testers. However, this will not be possible in the future if the tester group expands.

Funded by
the European Union

7. Listing of tools that will be implemented in next 6 months until final release

The beta release of the "EIS Toolkit" includes most of the planned tools, but not all. Data preprocessing is almost complete, but some exploration and modeling functions are still being developed and refined.

The following planned tools are not included in the beta release:

- Surface derivatives (aspect, slope, curvature)
- Distance to anomaly
- Calculate vector geometry
- Extract shared lines
- Autoencoder
- Self-organizing map
- CoDa
- Training data sampler
- MLP
- CNN
- Random forests
- Model uncertainty estimation
- Model performance estimation
- Hyperparameter optimization.

Funded by
the European Union

8. Conclusion

The first beta release of the "EIS Toolkit" has been completed and the work continues with expanding and optimizing the toolkit's functionality and with the implementation of "QGIS EIS Plugin", which will include a graphical user-interface for conducting Mineral Prospectivity Mapping with QGIS.

Testing of the "EIS Toolkit" was already started before the beta release and revealed some challenges that require further analysing.

The role of testing will further increase after the first release of the "QGIS EIS Plugin", and the plan is to gather a testing group and let them test the plugin individually.

After the testing phase, reported issues will be addressed before the final release, by the end of April 2024 – with the submission of Deliverable D3.5.

Funded by
the European Union

9. References

Nykänen, V., Lahti, I., Niiranen, T., & Korhonen, K. (2015). Receiver operating characteristics (ROC) as validation tool for prospectivity models — A magmatic Ni–Cu case study from the Central Lapland Greenstone Belt, Northern Finland. Ore Geology Reviews, 71, 853-860. doi:10.1016/j.oregeorev.2014.09.007

Funded by
the European Union

Appendix 1: Main workflows for proxy processing

Funded by
the European Union

Preprocess

PROXY TYPE

User action

User input

Input/Output

Background functionality

1. Distance to points, polygons or lines having specific categorical attribute valules

Point shape or csv file, polygon shape file, or line shape file

If csv input: define the northing and easting attributes

Select the attribute and give its values of interest (allow multiple conditions)

2. Distance to points, polygons or lines representing attribute values above certain threshold

Point shape or csv file, polygon shape file or line shape file

If csv input: define the northing and easting attributes

Select the attributes of interest

OR

Give the threshold for selecting high values

Run "select by attributes" using ">threshold" or "==value"

Calculate euclidean distance to the selected features

Distance raster

Values of the attributes of interest

Explore 1.

Threshold

3. Distance to grid cells representing values above a specific threshold

Raster

OR

Give the threshold for binarizing values

Binarize raster as 1=high and 0=low

OR

Polygonize areas covered by cell values of 1

Calculate euclidean distance to the polygons

Distance raster

OR

Spatial filters, functional transformations

OR

Binary raster

Explore 1.

Threshold

3.1 Distance to location above a specific threshold. Input data: point

4. Value, interpolated from point data

Raster

OR

Give the threshold for binarizing values

Binarize raster as 1=high and 0=low

OR

Polygonize areas covered by cell values of 1

Calculate euclidean distance to the polygons

Distance raster

OR

Spatial filters, functional transformations

OR

Binary raster

Explore 1.

Threshold

4. Value, interpolated from point data

Point shape or csv file

If csv input: define the northing and easting attributes

Select the attribute of interest

Select interpolation method and parameters

Interpolate

Raster

Spatial filters, functional transformations

Appendix 2: EIS proxy table for IOCG vectors

Funded by
the European Union

Mineral system Component	Mappable ingredient	Type of data	Format	Proxy	Processing workflow
Source	Albitized country rocks (Metal and ligand source)	Geological map	Polygon	Distance to albitized country rocks	1
		Mineralogy (outcrop observation data)	1) Point 2) Polygon		1
		Mineralogy (drilling data)	Point		1
	felsic (meta)volcanic rocks and subvolcanic rocks (Metal and fluid and Heat source;	Geological map	Polygon	Distance to felsic (meta)volcanic rocks and subvolcanic rocks	1
	Hydrothermal alteration of metavolcanic and granitoid rocks (Metal and ligand Source)	(1) Lithogeochemistry	Raster	Distance to high concentrations of Na, K or Mg±Fe	4 + 3
		(2) Bottom of till geochemistry	Point		2
		EM	Raster	Distance to high conductivity anomalies	3
	Fe-rich rocks (magmatic hydrothermal fluids source and metal source)	(1) Lithogeochemistry	Point/ Raster	Distance to high Fe concentrations	2/ 4+3
		(2) Magnetic Data	Raster Point	distance to Fe-oxides mapped from high magnetic anomalies	1 or 3, but processing is needed to get from TMI to the Fe oxides, if not directly proportional
		(3) Gravity Data	Raster Point	distance to Fe-oxides mapped from high density anomalies	1 or 3, but processing is needed to get from density to the Fe oxides, if not directly proportional
		(4) Geological Map	Polygon	Distance to Fe-oxides (includes both magnetite and hematite minerals; or any other Fe-oxides related mineral of the interest of the users)	1
		(5) Mineralogy	Point		1
	Magmatic intrusions (Magmatic hydrothermal fluids, heat sources)	(1) Geological map	Polygon	Distance to magmatic intrusion of the relevant age (age from geology and radiometric data; distance from geophysical data)	1 or 3, if the polygons have the age information already
		(2) Radiometric ages	Point		
	Alkaline intrusions (Fluid Source)	(1) Lithogeochemistry	Point	Distance to rock units displaying alkaline magma signature	1&3 attribute value for alkaline magma signature needed
		(2) Geology	Polygon		1&3 attribute value for alkaline magma signature needed
		(3) Isotope composition	Point		1&3 attribute value for alkaline magma signature needed
	Metamorphic facies	1) Geology (metamorphic rock distributions)	Polygon	Greenschist to mid-Amphibolite facies metamorphosed host rocks	1
		(2) Metamorphic indicator mineral	Point		
	Tectonic setting (everything)	1) Geology	Polygon	Syn- to late-orogenic back arc closure related intrusions.	1
		(2) lithogeochemistry (lithotectonic rock associations)	Point		
	Magmatic rocks (Hydrothermal fluids Source)	Lithogeochemistry	Point	Distance to high Fe3O4 concentration	3.1
	Co-REE anomaly (Metal Source)	Lithogeochemistry	Point	Distance to high concentrations of Co-REEs	3.1
		mineralogy	Point		
	I and A-type felsic igneous rocks; (Fluid and energy Source)	Lithogeochemistry	Point	Distance to magmatism contemporary with mineralisation (within 5 km buffer)	1 calculate buffer
		Geology	1) Polygon 2) Lines		
	Major structures	Geology	Lines	Distance to structures	1
		Gravity worms	Lines	Distance to high order structrues, interpretated from gravity worms	1
	Lower order structures	Geology	Lines	Distance to lower order structures	1
		Gravity worms	Lines	Distance to lower order structrues, interpretated from pseudo gravity worms	1

Active/structural Pathways	Fluid flow (fault network) and deposition	Structure map	Poly line	Distance to kinks, offsets and breaks (interpretation from magnetic data, e.g. gradient)	1
	Fluid flow (fault network) and deposition	Structure map	Poly line	Distance to high density structure (interpretation of high density structures derived from geological map or airborne VLF)	2 attribute needed: density of structures
	Regional scale structure, crustal domain boundary; 75 km buffer	Structural Geology	Poly line	Distance to crustal boundary (interpretation from Solid geology, seismic reflection, passive seismic, granity, magnetic data)	1
	Local scale structure; 2.5 km buffer	Geology	Shape file	Distance to intensity of structure	2
Depositional processes	Na-Ca Alteration	Bedrock & drill hole observations	Point	Distance to altered rocks	2
	K-alteration	Bedrock & drill hole observations	Point	Distance to altered rocks	2
		Till geochemistry	Point/raster	K-anomalies in till	3.1
	FeOx alteration	Bedrock & drill hole observations	Point	Distance to altered rocks	2
		Geophysics Mag	Raster Polygons	Magnetic highs (Or: distance to Fe-oxides mapped from high magnetic anomalies)	3.1 or 1
		Till geochemistry	Point Raster	Elevated V in till	2 3.1
	Geochemical signal	Till geochemistry	Point/raster	Elevated Ba, Co, Cu, P, Au in till Elevated Au, Ba, Co, Cu, La, P in till	3.1 or 3
		Radiometric U, Th	Raster	Radiometric U and/or Th anomalies	3
	Element deposition/mineralisation, marker horizon	Airborne Magnetic data	Raster	Distance to high-magnetic field anomalies	3
	Chemical trap, replacement	Geological map	Polygons (shp)	Presence of carbonate horizons in metavolcanic rocks	1
	Elemental deposition/mineralisation	Lithogeochemistry	Csv (table) Point	Distance to high concentrations of REE, Fe	3.1
	Magmatic crystallization	Geological map	Polygons (shp)	Massive magnetite Massive magnetite and vonsenite	1
	Trap	Lithogeochemistry	CSV Points (shp)	Ca-Fe metasomatism, Distance high Ca-Fe molar proportions relative to Na, K, and Mg	2 attribute needed: ratio of (Ca+Fe) to (Na + K + Mg)
	Mineralization footprint	(1) Geology (Alteration minerals, e.g.; Albite) (2) Mineralogy (3) Lithogeochemistry	Polygon Polygon Point	Distance to rock units displaying - sodic alteration signature - potassic and/or calcic alteration signature	1
		(1) Lithogeochemistry (2) Petrography/Geology (3) Stable isotope ratios	Point Point Polygons Point	Low Ti signature in Fe-oxides	3.1 1 3.1
		(1) geology (2) Lithogeochemistry	Points Point	Distance to (Sub) economic Cu-Au concentrations	1 3.1
Mineralisation, remobilisation, modification	Remobilisation	structural geology data	Polylines (shp)	Distance to kinks, offsets and breaks picked from magnetic data	2
	Remobilisation	Local-scale fault structure (large faults, high strain intensity) Local-scale fault structure (shearzones with strain measured)	Polylines (shp)	Distance to structures of high density/intensity Distance to structures	2 attribute for fault size, shear zone, strain, intensity needed
	Till geochemical anomalies	Till geochemistry	Csv (table) Points (shp)	Distance to elevated REE, Fe, Cu, Co, Au, Bi, Mo anomalies	3.1 repeat for each element

Appendix 3: EIS Toolkit – Workflow Demonstration

Funded by
the European Union

This Jupyter notebook serves as a first user guide to beta testers of EIS Toolkit. It shows how to import and call various tools of EIS Toolkit and includes steps of a simple MPM workflow.

Note that to run this notebook without modifications you need to have the test data (and under correct folder structure). One can always change the filepaths and use their own data, granted that the data is in right the format and is meaningful geospatial data. However, the primary function of this notebook is not to provide a template for conducting MPM workflows, but instead serve as example and showcase some of the tools of EIS Toolkit.

1. Imports and filepath definitions

In [2]:
```
import rasterio
import geopandas as gpd
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from rasterio.io import MemoryFile
from rasterio.plot import show
import tempfile

import sys
sys.path.insert(0, "..")

from eis_toolkit.exploratory_analyses.basic_plots_seaborn import histogram,
from eis_toolkit.exploratory_analyses.descriptive_statistics import descript
from eis_toolkit.exploratory_analyses.pca import compute_pca

from eis_toolkit.prediction.fuzzy_overlay import gamma_overlay
from eis_toolkit.prediction.weights_of_evidence import weights_of_evidence_c

from eis_toolkit.raster_processing.extract_values_from_raster import extract
from eis_toolkit.raster_processing.unifying import unify_raster_grids

from eis_toolkit.transformations.sigmoid import _sigmoid_transform
from eis_toolkit.transformations.linear import _min_max_scaling

from eis_toolkit.vector_processing.idw_interpolation import idw
from eis_toolkit.vector_processing.distance_computation import distance_comp
from eis_toolkit.vector_processing.rasterize_vector import rasterize_vector
```

In [3]:
```
# Filepaths
AEM_inphase_fp = "../tests/data/local/workflow_demo/IOCG_AEM_Inph_.tif"
AEM_quad_fp = "../tests/data/local/workflow_demo/IOCG_AEM_Quad.tif"
AEM_ratio_fp = "../tests/data/local/workflow_demo/IOCG_EM_ratio.tif"
Magn_AS_fp = "../tests/data/local/workflow_demo/Mag_DGRF_AS_FFT_ers_PCS_tif_

till_geochem_fp = "../tests/data/local/workflow_demo/Geochemical_Data/Vector
structures_fp = "../tests/data/local/workflow_demo/Geological_Data/IOCG_CLB_
```

```
lithology_fp = "../tests/data/local/workflow_demo/Geological_Data/IOCG_CLB_L
known_occurances_fp = "../tests/data/local/workflow_demo/Deposits_Occurrence
```

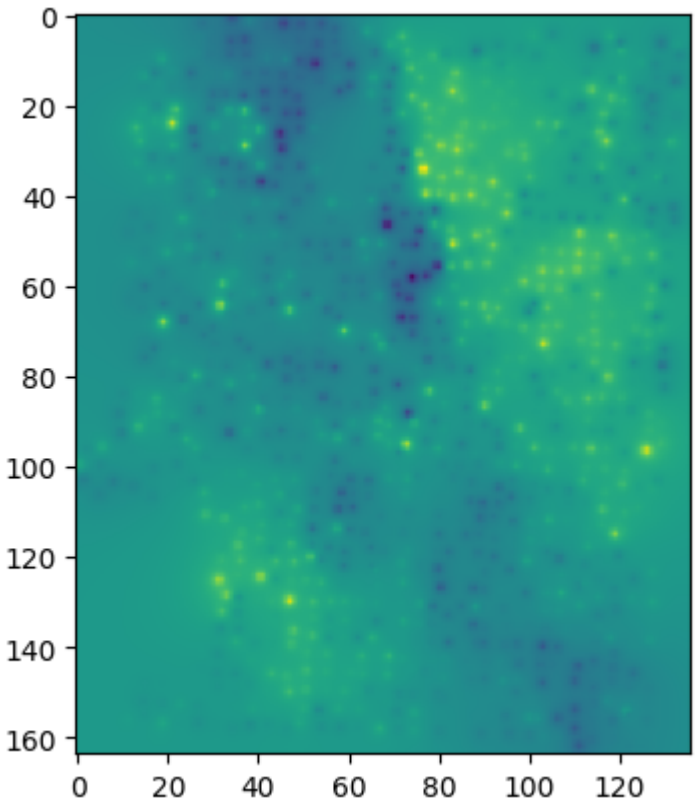
2. Preprocess data

In [4]:
```
# Take AEM_inphase raster profile and grid shape as the base/target
with rasterio.open(AEM_inphase_fp) as AEM_inphase:
    raster_profile = AEM_inphase.profile
    raster_extent = AEM_inphase.bounds
    raster_pixel_size = AEM_inphase.transform[0]
```

In [ ]:
```
# Preprocess geochemical data
till_geochem = gpd.read_file(till_geochem_fp)

columns_to_process = ["Fe_ppm_511", "Co_ppm_511", "Cu_ppm_511"]

interpolation_results = {}
interpolation_extent = (raster_extent[0], raster_extent[2], raster_extent[1]

# Log transform concentrations and interpolate
# NOTE: EIS Toolkit implements transforms only for rasters at the moment, th
for column in columns_to_process:
    new_col_name = column[:2] + "_log"
    till_geochem[new_col_name] = np.log(till_geochem[column])

    interpolation_results[new_col_name] = idw(till_geochem, new_col_name, (5
```

In [6]:
```
# Visualize copper interpolated
plt.imshow(interpolation_results["Cu_log"][0])
```

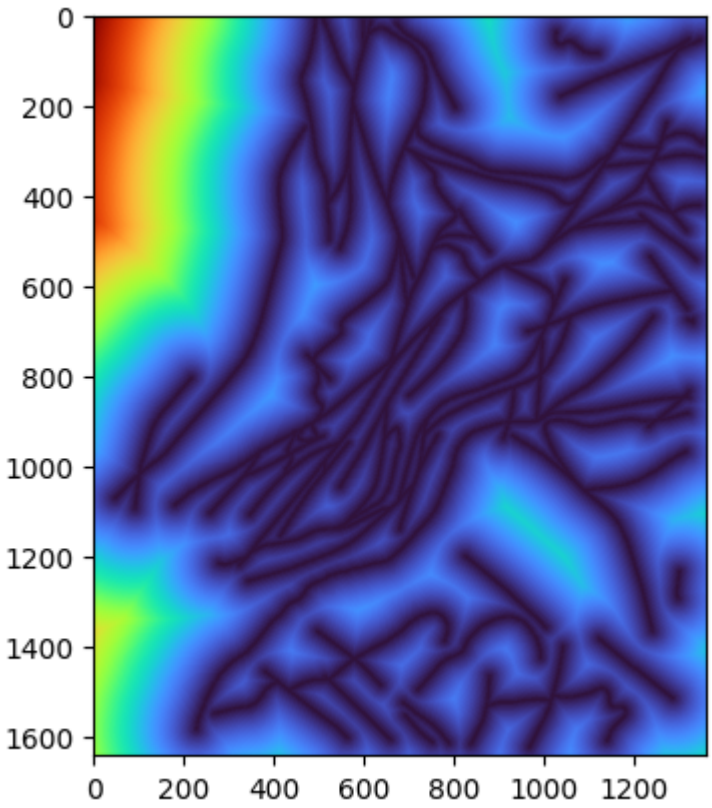
Out[6]:
```
<matplotlib.image.AxesImage at 0x7f54f25a69e0>
```



In [60]:
```
# Preprocess geological data
structures = gpd.read_file(structures_fp)

distances_to_stuctures = distance_computation(raster_profile, structures)
```

In [7]:
```
# Visualize distances to structures
plt.imshow(distances_to_stuctures, cmap="turbo")
```

Out[7]:
```
<matplotlib.image.AxesImage at 0x7f12b8324400>
```



In [8]:
```
# Rasterize lithology
lithology = gpd.read_file(lithology_fp)
rasterized_lithology, lithology_out_meta = rasterize_vector(lithology, base_
```

In [9]:
```
# Extract raster values at deposits
known_occurances = gpd.read_file(known_occurances_fp)

# Open rasters with context manager, save lithology data to temporary raster
with \
    MemoryFile() as memfile, \
    rasterio.open(AEM_inphase_fp) as AEM_inphase, \
    rasterio.open(AEM_quad_fp) as AEM_quad, \
    rasterio.open(AEM_ratio_fp) as AEM_ratio, \
    rasterio.open(Magn_AS_fp) as Magn_AS:

    with tempfile.NamedTemporaryFile() as tmpfile:
        with rasterio.open(tmpfile.name, "w", **raster_profile) as dest:
            dest.write(rasterized_lithology, 1)
        with rasterio.open(tmpfile.name) as lithology_raster:
            raster_list = [AEM_inphase, AEM_quad, AEM_ratio, Magn_AS, lithol
            col_names = ["AEM_inphase", "AEM_quad", "AEM_ratio", "Magn_AS",
            raster_values_at_deposits = extract_values_from_raster(raster_li

            unified_rasters = unify_raster_grids(AEM_inphase, raster_list[1:

raster_values_at_deposits
```

Out[9]:

	x	y	AEM_inphase	AEM_quad	AEM_ratio	Magn_AS	Lith
0	370953.0	7502821.0	-408.969513	369.960022	-1.059486	17.158161	
1	371340.0	7496345.0	3889.745605	2384.806152	1.817769	21.545200	
2	385205.0	7497926.0	-339.839050	187.980026	-1.730280	14.112287	
3	369187.0	7490345.0	1468.369385	1285.413208	1.034502	20.598907	
4	368175.0	7489722.0	537.748718	324.102142	1.903474	43.523659	
5	350043.0	7489418.0	-6.511929	625.734375	0.025980	72.489517	
6	363838.0	7492354.0	-435.950928	248.234833	-1.672447	8.311082	
7	364672.0	7482684.0	-210.937698	1754.664917	-0.309453	31.408430	
8	396649.0	7501845.0	-373.599243	440.641846	-1.192332	12.663096	
9	372344.0	7492293.0	-885.269836	675.116272	-2.274024	8.680225	
10	371994.0	7491258.0	619.958496	578.203674	1.160073	25.673422	
11	368240.0	7487137.0	-259.313599	617.076233	-0.381119	58.577671	
12	373779.0	7504766.0	27.086342	577.757690	-0.123563	5.490668	

3. Explore data

Explore rasters

In [10]:

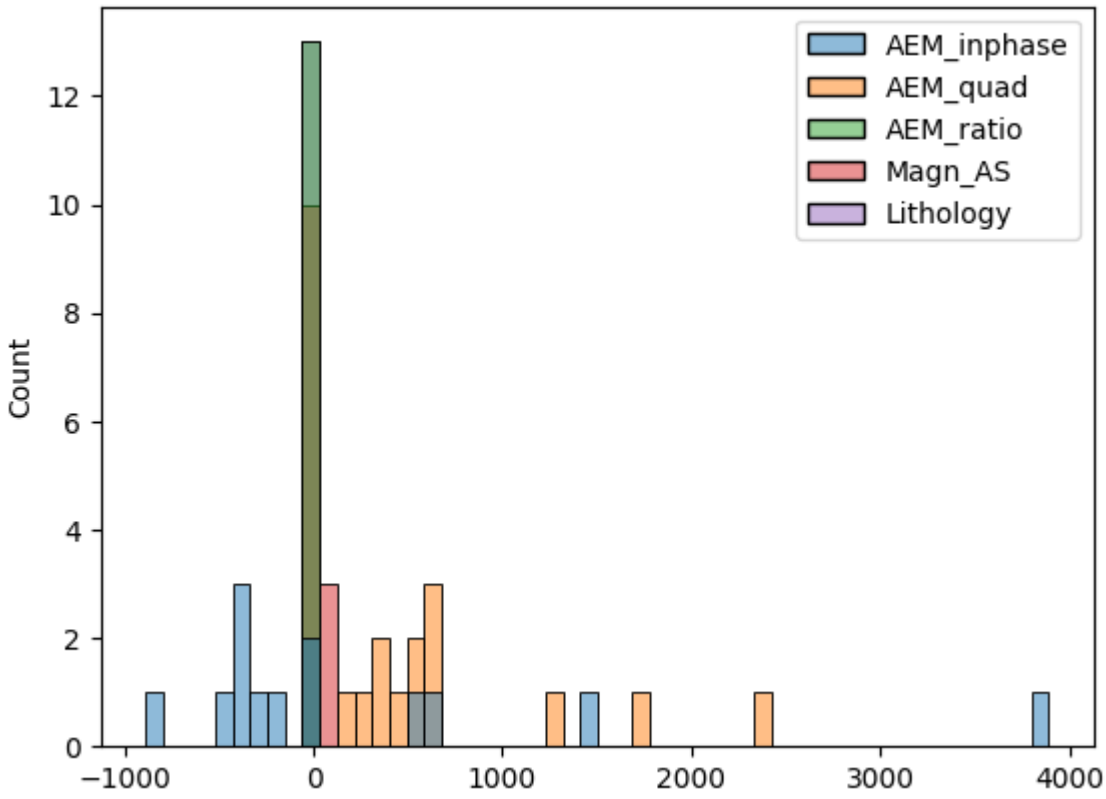
```
# Calculate basic statistics of rasters at deposit locations
inphase_stats_at_points = descriptive_statistics_dataframe(raster_values_at_
quad_stats_at_points = descriptive_statistics_dataframe(raster_values_at_dep
ratio_stats_at_points = descriptive_statistics_dataframe(raster_values_at_de
magn_stats_at_points = descriptive_statistics_dataframe(raster_values_at_dep

stats_dataframe = pd.DataFrame(
    {
        'inphase':pd.Series(inphase_stats_at_points),
        'quad':pd.Series(quad_stats_at_points),
        'ratio':pd.Series(ratio_stats_at_points),
        'magn':pd.Series(magn_stats_at_points),
    }
)
stats_dataframe
```

Out[10]:

	inphase	quad	ratio	magn
min	-885.269836	187.980026	-2.274024	5.490668
max	3889.745605	2384.806152	1.903474	72.489517
mean	278.655135	774.591645	-0.215454	26.171717
25%	-373.599243	369.960022	-1.192332	12.663096
50%	-210.937698	578.203674	-0.309453	20.598907
75%	537.748718	675.116272	1.034502	31.408430
standard_deviation	1191.602431	623.131335	1.315682	19.687020
relative_standard_deviation	4.276262	0.804464	-6.106545	0.752225
skew	2.079341	1.489026	0.191099	1.145725

In [11]:
```
# Plot histogram of raster values at deposit locations
histogram(raster_values_at_deposits.drop(["x", "y"], axis=1))
```

Out[11]: <Axes: ylabel='Count'>

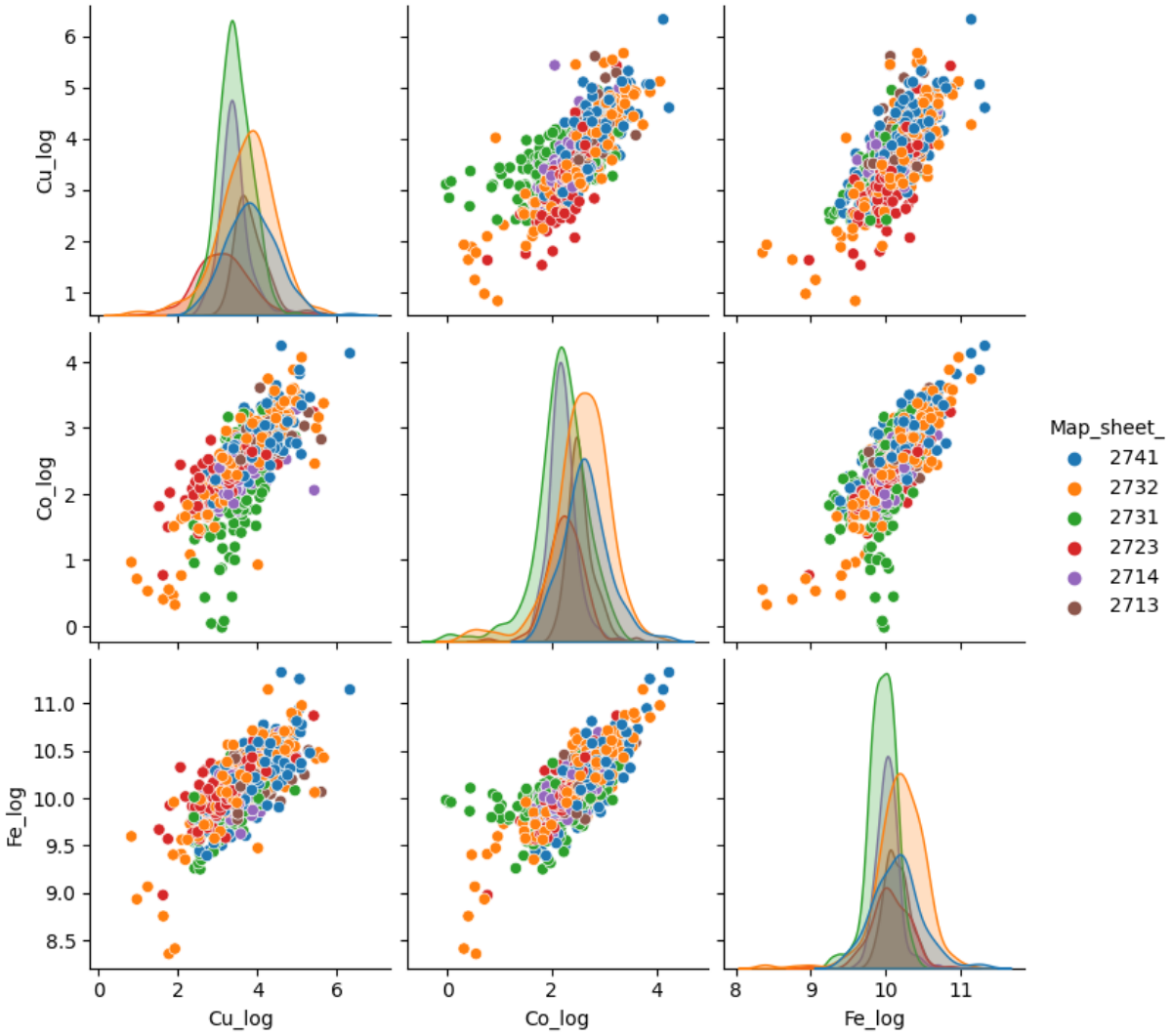


Explore geochemical data (vector data)

In [23]:
```
# Plot pairplot of various log transformed concentrations
pairplot(till_geochem[["Cu_log", "Co_log", "Fe_log", "Map_sheet_"]], hue="Ma
```

```
/home/niko/.local/lib/python3.10/site-packages/seaborn/axisgrid.py:118: User
Warning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

Out[23]: <seaborn.axisgrid.PairGrid at 0x7f54e86b0820>



In [42]:
```
# Define cmap to use in the following visualizations
cmap = plt.get_cmap('jet')
```

Explore raster data with PCA

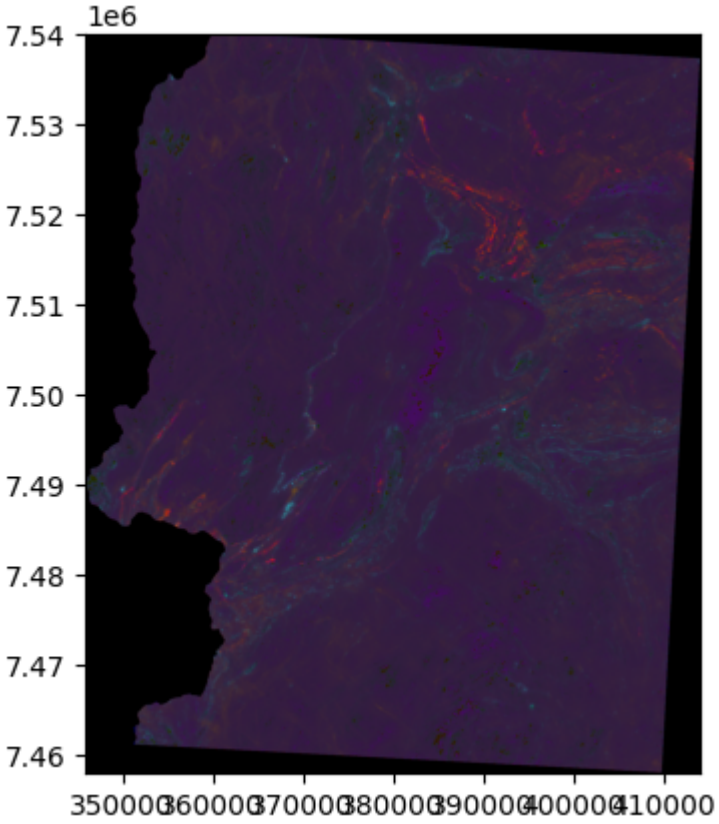
In [161…
```
# Preprocess nodata before modeling (for these rasters, min value is the noc
arrays_to_stack = []

for raster_array, meta in unified_rasters:
    raster_array[raster_array == np.nanmin(raster_array)] = np.nan
    arrays_to_stack.append(raster_array[0])

stacked_arrays = np.stack(arrays_to_stack)
```

In [170…
```
# Compute PCA for the input rasters
out_array, explained_variances = compute_pca(stacked_arrays, 3)
explained_variances
```

Out[170… array([0.41700809, 0.32194577, 0.16958726])

In [168…
```
# Visualize PCA outputs

# Scale each band
scaled_bands = []
for band in range(out_array.shape[0]):
    scaled_band = _min_max_scaling(out_array[band], (0, 255))
    scaled_bands.append(scaled_band)

# Stack scaled bands back together
scaled_pca_output = np.stack(scaled_bands)

# Display the RGB image
show(scaled_pca_output.astype(np.uint8), transform=raster_profile["transform
```

```
/tmp/ipykernel_83370/1271570867.py:13: RuntimeWarning: invalid value encount
ered in cast
  show(scaled_pca_output.astype(np.uint8), transform=raster_profile["transfo
rm"])
```



Out[168… <Axes: >

4. Fuzzy logic modeling

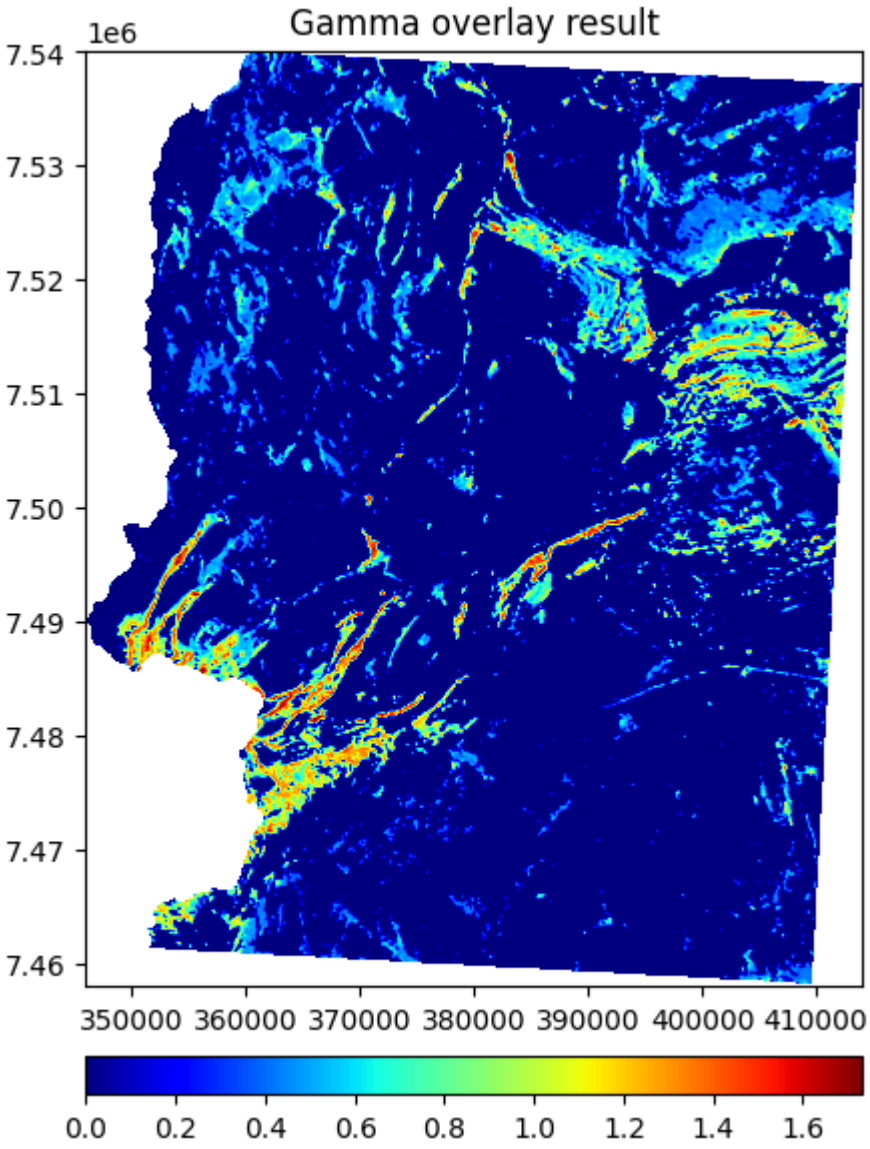
In [53]:
```
# Transform data before fuzzy overlay
arrays_to_stack = []
for raster_array, meta in unified_rasters:
    raster_array[raster_array == np.nanmin(raster_array)] = np.nan
```

```
    out_array = _sigmoid_transform(raster_array, (0, 1), 1, True)
    arrays_to_stack.append(out_array[0])

transformed_arrays_for_fuzzy_overlay = np.stack(arrays_to_stack)
```

In [54]:
```
# Compute gamma overlay
overlay_result = gamma_overlay(transformed_arrays_for_fuzzy_overlay, 0.5)
```

In [55]:
```
# Plot gamma ovelay result
fig, ax = plt.subplots(1, 1, figsize = (5, 9))

ax.set_title("Gamma overlay result")
clrbar = plt.imshow(overlay_result, cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(overlay_result, ax = ax, transform = raster_profile["transform"], cmap=
```

Out[55]: <Axes: title={'center': 'Gamma overlay result'}>

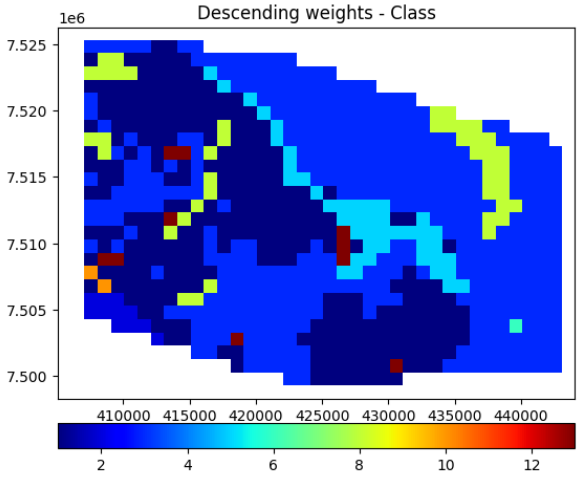


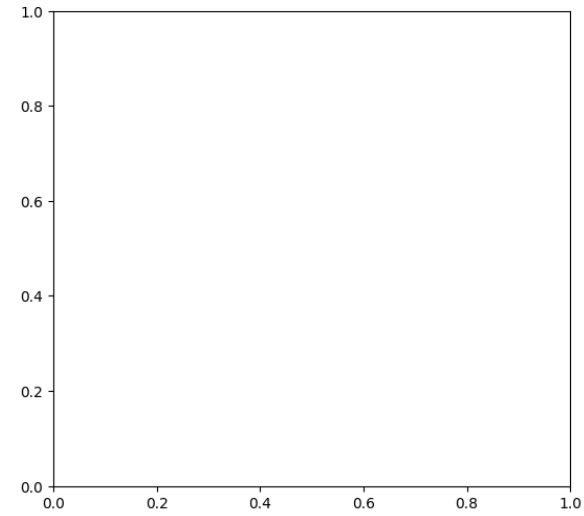
5. Weights of evidence modeling

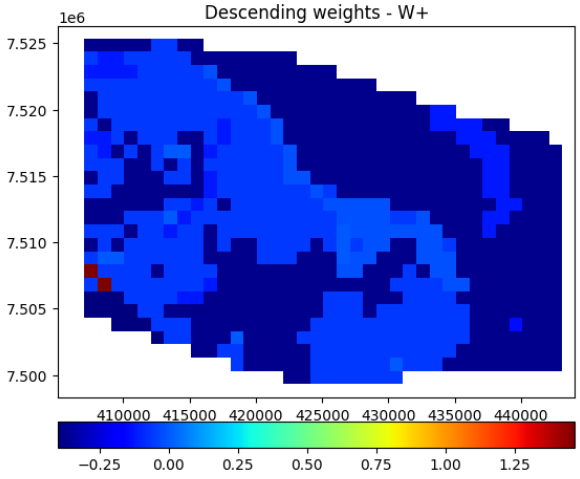
In [56]:
```
# Calculate weights
with rasterio.open("../tests/data/remote/wofe/wofe_evidence_raster.tif") as
    deposits = gpd.read_file("../tests/data/remote/wofe/wofe_deposits.shp")

    weights_desc, arrays_desc, out_meta, deposit_pixels, evidence_pixels = w
        evidential_raster=evidence_raster,
        deposits=deposits,
        weights_type='descending',
        studentized_contrast_threshold=1
    )

    weights_asc, arrays_asc, _, _, _ = weights_of_evidence_calculate_weights
        evidential_raster=evidence_raster,
        deposits=deposits,
        weights_type='ascending',
        studentized_contrast_threshold=1
    )
```

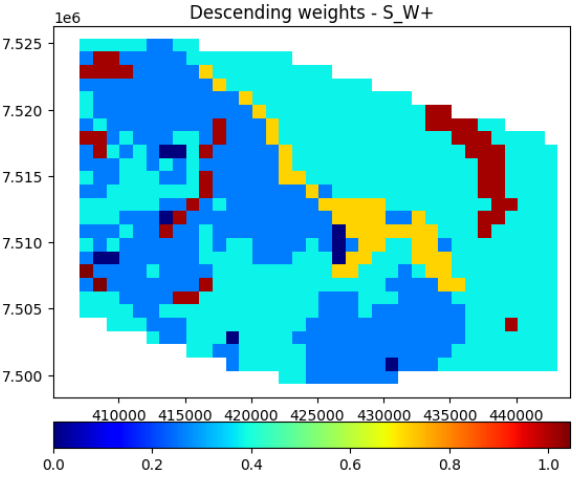
In [57]:
```
# Plot descending weights
fig, axs = plt.subplots(3, 2, figsize = (14, 20))

axs[0, 0].set_title("Descending weights - Class")
clrbar = axs[0, 0].imshow(arrays_desc["Class"], cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(arrays_desc["Class"], ax = axs[0, 0], transform = out_meta["transform"]

axs[1, 0].set_title("Descending weights - W+")
clrbar = axs[1, 0].imshow(arrays_desc["W+"], cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(arrays_desc["W+"], ax = axs[1, 0], transform = out_meta["transform"], c

axs[1, 1].set_title("Descending weights - S_W+")
clrbar = axs[1, 1].imshow(arrays_desc["S_W+"], cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(arrays_desc["S_W+"], ax = axs[1, 1], transform = out_meta["transform"],

axs[2, 0].set_title("Descending weights - Generalized W+")
clrbar = axs[2, 0].imshow(arrays_desc["Generalized W+"], cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(arrays_desc["Generalized W+"], ax = axs[2, 0], transform = out_meta["tr

axs[2, 1].set_title("Descending weights - Generalized S_W+")
clrbar = axs[2, 1].imshow(arrays_desc["Generalized S_W+"], cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(arrays_desc["Generalized S_W+"], ax = axs[2, 1], transform = out_meta["
```

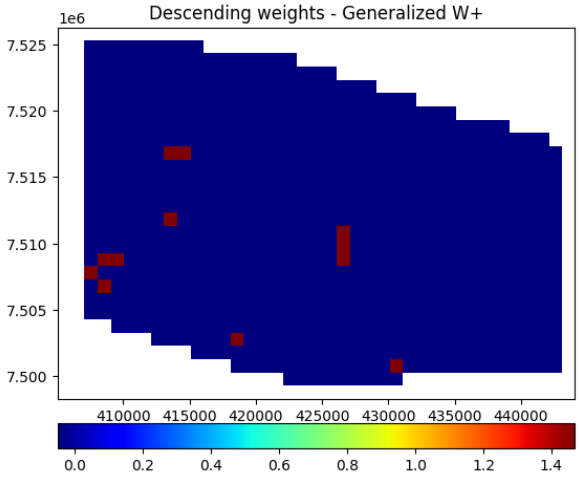
Out[57]:
```
<Axes: title={'center': 'Descending weights - Generalized S_W+'}>
```

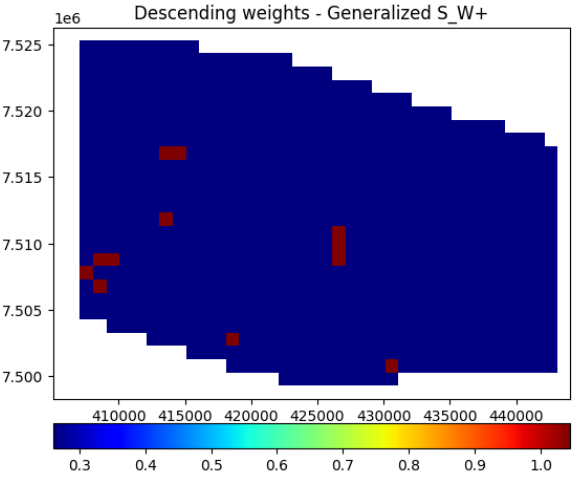










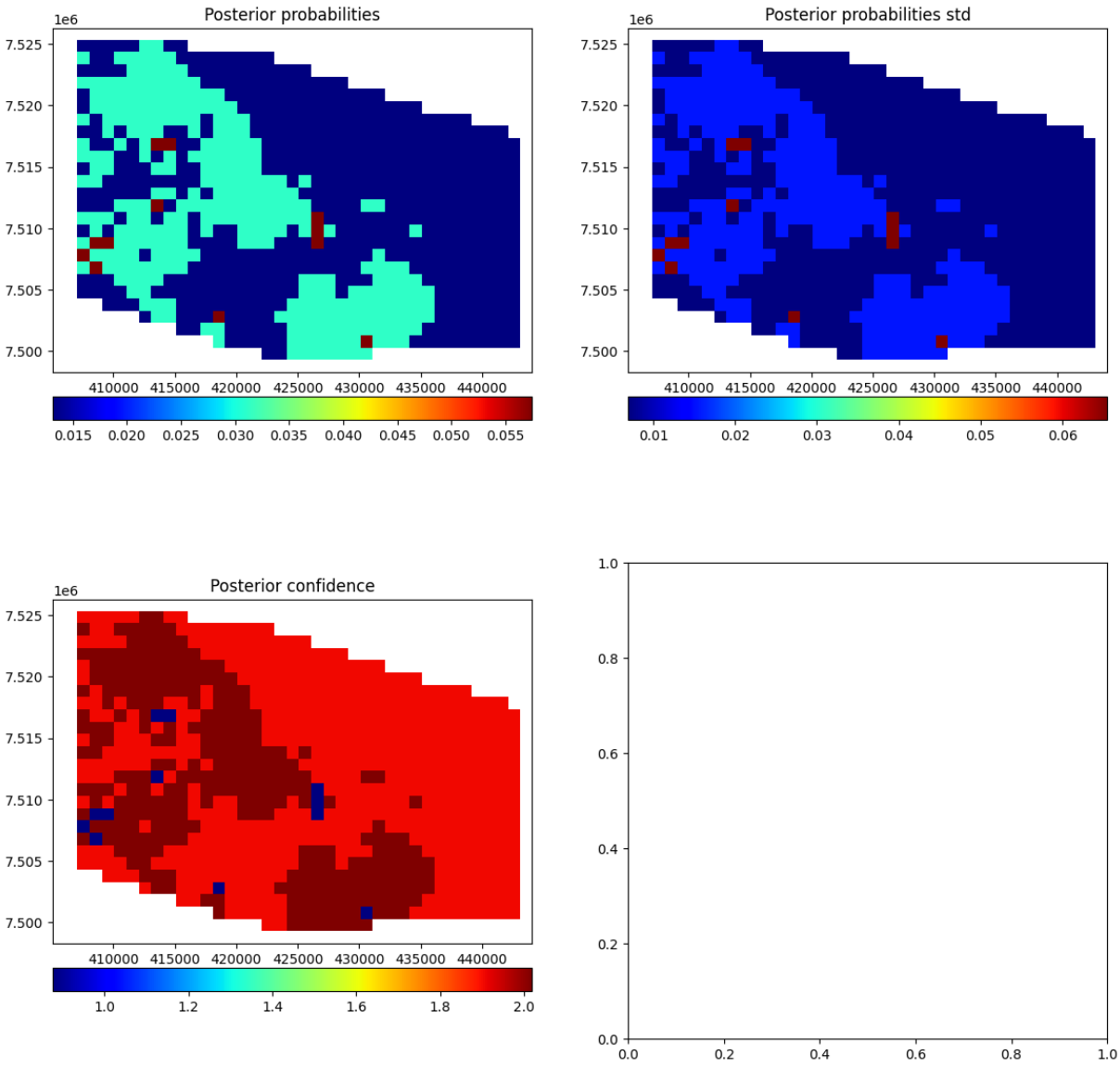


In [58]:
```
# Calculate posterior probabilities / responses
posterior_array, posterior_array_std, posterior_confidence = weights_of_evic
```

In [59]:
```
# Plot posterior probabilities weights

fig, axs = plt.subplots(2, 2, figsize = (14, 14))
```

```
axs[0, 0].set_title("Posterior probabilities")
clrbar = axs[0, 0].imshow(posterior_array, cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(posterior_array, ax = axs[0, 0], transform = out_meta["transform"], cma

axs[0, 1].set_title("Posterior probabilities std")
clrbar = axs[0, 1].imshow(posterior_array_std, cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(posterior_array_std, ax = axs[0, 1], transform = out_meta["transform"],

axs[1, 0].set_title("Posterior confidence")
clrbar = axs[1, 0].imshow(posterior_confidence, cmap=cmap)
plt.colorbar(clrbar, orientation="horizontal", pad = 0.05)
show(posterior_confidence, ax = axs[1, 0], transform = out_meta["transform"]
```

Out[59]: <Axes: title={'center': 'Posterior confidence'}>



Appendix 4: EIS Toolkit – Technical Specifications

Funded by
the European Union

EIS Toolkit

Technical specifications

None

Made by Gispo Ltd.

Table of contents

1. General

This is the documentation site of the eis_toolkit python package. Here you can find documentation for each module. The documentation is automatically generated from docstrings.

Development of eis_toolkit is related to EIS Horizon EU project.

2. Dependency licenses

Name	Version	License
protobuf	3.19.4	3-Clause BSD License
tensorboard-plugin-wit	1.8.1	Apache 2.0
absl-py	1.2.0	Apache Software License
flatbuffers	1.12	Apache Software License
ghp-import	2.1.0	Apache Software License
google-auth	2.11.0	Apache Software License
google-auth-oauthlib	0.4.6	Apache Software License
google-pasta	0.2.0	Apache Software License
grpcio	1.48.1	Apache Software License
importlib-metadata	4.12.0	Apache Software License
keras	2.9.0	Apache Software License
libclang	14.0.6	Apache Software License
requests	2.28.1	Apache Software License
rsa	4.9	Apache Software License
tenacity	8.2.2	Apache Software License
tensorboard	2.9.1	Apache Software License
tensorboard-data-server	0.6.1	Apache Software License
tensorflow	2.9.2	Apache Software License
tensorflow-estimator	2.9.0	Apache Software License
tensorflow-io-gcs-filesystem	0.26.0	Apache Software License
watchdog	2.1.9	Apache Software License
packaging	21.3	Apache Software License; BSD License
python-dateutil	2.8.2	Apache Software License; BSD License
affine	2.3.1	BSD
cligj	0.7.2	BSD
geopandas	0.11.1	BSD
Fiona	1.8.21	BSD License
Jinja2	3.1.2	BSD License
Markdown	3.3.7	BSD License
MarkupSafe	2.1.1	BSD License
Pygments	2.13.0	BSD License
Shapely	1.8.4	BSD License
Werkzeug	2.2.2	BSD License
astunparse	1.6.3	BSD License
click	8.1.3	BSD License

Name	Version	License
click-plugins	1.1.1	BSD License
cycler	0.11.0	BSD License
gast	0.4.0	BSD License
h5py	3.7.0	BSD License
idna	3.3	BSD License
joblib	1.1.0	BSD License
kiwisolver	1.4.4	BSD License
mkdocs	1.3.1	BSD License
numpy	1.23.2	BSD License
oauthlib	3.2.0	BSD License
pandas	1.4.4	BSD License
patsy	0.5.2	BSD License
pyasn1	0.4.8	BSD License
pyasn1-modules	0.2.8	BSD License
rasterio	1.3.2	BSD License
requests-oauthlib	1.3.1	BSD License
scikit-learn	1.1.2	BSD License
scipy	1.9.1	BSD License
statsmodels	0.13.2	BSD License
threadpoolctl	3.1.0	BSD License
wrapt	1.14.1	BSD License
eis-toolkit	0.1.0	European Union Public Licence 1.2 (EUPL 1.2)
Pillow	9.2.0	Historical Permission Notice and Disclaimer (HPND)
opt-einsum	3.3.0	MIT
snuggs	1.4.7	MIT
GDAL	3.4.3	MIT License
Keras-Preprocessing	1.1.2	MIT License
PyYAML	6.0	MIT License
attrs	22.1.0	MIT License
cachetools	5.2.0	MIT License
charset-normalizer	2.1.1	MIT License
fonttools	4.37.1	MIT License
mergedeep	1.3.4	MIT License
mkdocs-material	8.4.2	MIT License
mkdocs-material-extensions	1.0.3	MIT License

Name	Version	License
munch	2.5.0	MIT License
plotly	5.14.0	MIT License
pymdown-extensions	9.5	MIT License
pyparsing	3.0.9	MIT License
pyproj	3.3.1	MIT License
pytz	2022.2.1	MIT License
pyyaml_env_tag	0.1	MIT License
setuptools-scm	6.4.2	MIT License
six	1.16.0	MIT License
termcolor	1.1.0	MIT License
tomli	2.0.1	MIT License
urllib3	1.26.12	MIT License
zipp	3.8.1	MIT License
certifi	2022.6.15	Mozilla Public License 2.0 (MPL 2.0)
matplotlib	3.5.3	Python Software Foundation License
typing_extensions	4.3.0	Python Software Foundation License

3. Conversions

3.1 Convert csv to geodataframe

`csv_to_geodataframe(csv, indexes, target_crs)`

Read CSV file to a GeoDataFrame.

Usage of single index expects valid WKT geometry. Usage of two indexes expects POINT feature(s) X-coordinate as the first index and Y-coordinate as the second index.

Parameters:

Name	Type	Description	Default
`csv`	`Path`	Path to the .csv file to be read.	required
`indexes`	`Sequence[int]`	Index(es) of the geometry column(s).	required
`target_crs`	`int`	Target CRS as an EPSG code.	required

Returns:

Type	Description
`GeoDataFrame`	CSV file read to a GeoDataFrame.

Source code in `eis_toolkit/conversions/csv_to_geodataframe.py`

```
79  @beartype
80  def csv_to_geodataframe(
81      csv: Path,
82      indexes: Sequence[int],
83      target_crs: int,
84  ) -> geopandas.GeoDataFrame:
85      """
86      Read CSV file to a GeoDataFrame.
87
88      Usage of single index expects valid WKT geometry.
89      Usage of two indexes expects POINT feature(s) X-coordinate as the first index and Y-coordinate as the second index.
90
91      Args:
92          csv: Path to the .csv file to be read.
93          indexes: Index(es) of the geometry column(s).
94          target_crs: Target CRS as an EPSG code.
95
96      Returns:
97          CSV file read to a GeoDataFrame.
98      """
99
100     data_frame = _csv_to_geodataframe(
101         csv=csv,
102         indexes=indexes,
103         target_crs=target_crs,
104     )
105     return data_frame
```

3.2 Convert raster to dataframe

`raster_to_dataframe(raster, bands=None, add_coordinates=False)`

Convert raster to Pandas DataFrame.

If bands are not given, all bands are used for conversion. Selected bands are named based on their index e.g., band_1, band_2,...,band_n. If wanted, image coordinates (row, col) for each pixel can be written to dataframe by setting add_coordinates to True.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Raster to be converted.	required
`bands`	`Optional[Sequence[int]]`	Selected bands from multiband raster. Indexing begins from one. Defaults to None.	`None`
`add_coordinates`	`bool`	Determines if pixel coordinates are written into dataframe. Defaults to False.	`False`

Returns:

Type	Description
`DataFrame`	Raster converted to a DataFrame.

Source code in `eis_toolkit/conversions/raster_to_dataframe.py`

```
33  @beartype
34  def raster_to_dataframe(
35      raster: rasterio.io.DatasetReader,
36      bands: Optional[Sequence[int]] = None,
37      add_coordinates: bool = False,
38  ) -> pd.DataFrame:
39      """Convert raster to Pandas DataFrame.
40
41      If bands are not given, all bands are used for conversion. Selected bands are named based on their index e.g.,
42      band_1, band_2,...,band_n. If wanted, image coordinates (row, col) for each pixel can be written to
43      dataframe by setting add_coordinates to True.
44
45      Args:
46          raster: Raster to be converted.
47          bands: Selected bands from multiband raster. Indexing begins from one. Defaults to None.
48          add_coordinates: Determines if pixel coordinates are written into dataframe. Defaults to False.
49
50      Returns:
51          Raster converted to a DataFrame.
52      """
53
54      data_frame = _raster_to_dataframe(
55          raster=raster,
56          bands=bands,
57          add_coordinates=add_coordinates,
58      )
59      return data_frame
```

4. Exploratory analyses

4.1 DBSCAN

```
dbscan(data, max_distance=0.5, min_samples=5)
```

Perform DBSCAN clustering on the input data.

Parameters:

Name	Type	Description	Default
`data`	`GeoDataFrame`	GeoDataFrame containing the input data.	required
`max_distance`	`float`	The maximum distance between two samples for one to be considered as in the neighborhood of the other. Defaults to 0.5.	`0.5`
`min_samples`	`int`	The number of samples in a neighborhood for a point to be considered as a core point. Defaults to 5.	`5`

Returns:

Type	Description
`GeoDataFrame`	GeoDataFrame containing two new columns: one with assigned cluster labels and one indicating whether a point is a core point (1) or not (0).

Raises:

Type	Description
`EmptyDataFrameException`	The input GeoDataFrame is empty.
`InvalidParameterException`	The maximum distance between two samples in a neighborhood is not greater than zero or the number of samples in a neighborhood is not greater than one.

Source code in `eis_toolkit/exploratory_analyses/dbscan.py`

```python
@beartype
def dbscan(data: gdp.GeoDataFrame, max_distance: float = 0.5, min_samples: int = 5) -> gdp.GeoDataFrame:
    """
    Perform DBSCAN clustering on the input data.

    Args:
        data: GeoDataFrame containing the input data.
        max_distance: The maximum distance between two samples for one to be considered as in the neighborhood of
            the other. Defaults to 0.5.
        min_samples: The number of samples in a neighborhood for a point to be considered as a core point.
            Defaults to 5.

    Returns:
        GeoDataFrame containing two new columns: one with assigned cluster labels and one indicating whether a
            point is a core point (1) or not (0).

    Raises:
        EmptyDataFrameException: The input GeoDataFrame is empty.
        InvalidParameterException: The maximum distance between two samples in a neighborhood is not greater
            than zero or the number of samples in a neighborhood is not greater than one.
    """

    if data.empty:
        raise EmptyDataFrameException("The input GeoDataFrame is empty.")

    if max_distance <= 0:
        raise InvalidParameterValueException(
            "The input value for the maximum distance between two samples in a neighborhood must be greater than zero."
        )

    if min_samples <= 1:
        raise InvalidParameterValueException(
            "The input value for the minimum number of samples in a neighborhood must be greater than one."
        )

    dbscan_gdf = _dbscan(data, max_distance, min_samples)

    return dbscan_gdf
```

4.2 Descriptive statistics

`descriptive_statistics_dataframe(input_data, column)`

Generate descriptive statistics from vector data.

Generates min, max, mean, quantiles(25%, 50% and 75%), standard deviation, relative standard deviation and skewness.

Parameters:

Name	Type	Description	Default
`input_data`	`Union[DataFrame, GeoDataFrame]`	Data to generate descriptive statistics from.	required
`column`	`str`	Specify the column to generate descriptive statistics from.	required

Returns:

Type	Description
`dict`	The descriptive statistics in previously described order.

Source code in `eis_toolkit/exploratory_analyses/descriptive_statistics.py`

```
@beartype
def descriptive_statistics_dataframe(input_data: Union[pd.DataFrame, gpd.GeoDataFrame], column: str) -> dict:
    """Generate descriptive statistics from vector data.

    Generates min, max, mean, quantiles(25%, 50% and 75%), standard deviation, relative standard deviation and skewness.

    Args:
        input_data: Data to generate descriptive statistics from.
        column: Specify the column to generate descriptive statistics from.

    Returns:
        The descriptive statistics in previously described order.
    """
    if column not in input_data.columns:
        raise InvalidColumnException
    data = input_data[column]
    statistics = _descriptive_statistics(data)
    return statistics
```

`descriptive_statistics_raster(input_data)`

Generate descriptive statistics from raster data.

Generates min, max, mean, quantiles(25%, 50% and 75%), standard deviation, relative standard deviation and skewness.

Parameters:

Name	Type	Description	Default
`input_data`	`DatasetReader`	Data to generate descriptive statistics from.	required

Returns:

Type	Description
`dict`	The descriptive statistics in previously described order.

Source code in `eis_toolkit/exploratory_analyses/descriptive_statistics.py`

```
@beartype
def descriptive_statistics_raster(input_data: rasterio.io.DatasetReader) -> dict:
    """Generate descriptive statistics from raster data.

    Generates min, max, mean, quantiles(25%, 50% and 75%), standard deviation, relative standard deviation and skewness.

    Args:
        input_data: Data to generate descriptive statistics from.

    Returns:
        The descriptive statistics in previously described order.
    """
    data = input_data.read().flatten()
    statistics = _descriptive_statistics(data)
    return statistics
```

4.3 Feature importance

`evaluate_feature_importance(classifier, x_test, y_test, feature_names, number_of_repetition=50, random_state=0)`

Evaluate the feature importance of a sklearn classifier or linear model.

Parameters:

Name	Type	Description	Default
`classifier`	`BaseEstimator`	Trained classifier.	required
`x_test`	`ndarray`	Testing feature data (X data need to be normalized / standardized).	required
`y_test`	`ndarray`	Testing target data.	required
`feature_names`	`Sequence[str]`	Names of the feature columns.	required
`number_of_repetition`	`int`	Number of iteration used when calculate feature importance (default 50).	`50`
`random_state`	`int`	random state for repeatability of results (Default 0).	`0`

Return: A dataframe composed by features name and Importance value The resulted object with importance mean, importance std, and overall importance Raises: InvalidDatasetException: When the dataset is None.

Source code in `eis_toolkit/exploratory_analyses/feature_importance.py`

```python
@beartype
def evaluate_feature_importance(
    classifier: sklearn.base.BaseEstimator,
    x_test: np.ndarray,
    y_test: np.ndarray,
    feature_names: Sequence[str],
    number_of_repetition: int = 50,
    random_state: int = 0,
) -> tuple[pd.DataFrame, dict]:
    """
    Evaluate the feature importance of a sklearn classifier or linear model.

    Parameters:
        classifier: Trained classifier.
        x_test: Testing feature data (X data need to be normalized / standardized).
        y_test: Testing target data.
        feature_names: Names of the feature columns.
        number_of_repetition: Number of iteration used when calculate feature importance (default 50).
        random_state: random state for repeatability of results (Default 0).
    Return:
        A dataframe composed by features name and Importance value
        The resulted object with importance mean, importance std, and overall importance
    Raises:
        InvalidDatasetException: When the dataset is None.
    """

    if x_test is None or y_test is None:
        raise InvalidDatasetException

    result = permutation_importance(
        classifier, x_test, y_test.ravel(), n_repeats=number_of_repetition, random_state=random_state
    )

    feature_importance = pd.DataFrame({"Feature": feature_names, "Importance": result.importances_mean})

    feature_importance["Importance"] = feature_importance["Importance"] * 100
    feature_importance = feature_importance.sort_values(by="Importance", ascending=False)

    return feature_importance, result
```

4.4 K-means clustering

`k_means_clustering(data, number_of_clusters=None, random_state=None)`

Perform k-means clustering on the input data.

Parameters:

Name	Type	Description	Default
`data`	`GeoDataFrame`	A GeoDataFrame containing the input data.	required
`number_of_clusters`	`Optional[int]`	The number of clusters (>= 1) to form. Optional parameter. If not provided, optimal number of clusters is computed using the elbow method.	`None`
`random_state`	`Optional[int]`	A random number generation for centroid initialization to make the randomness deterministic. Optional parameter.	`None`

Returns:

Type	Description
`GeoDataFrame`	GeoDataFrame containing assigned cluster labels.

Raises:

Type	Description
`EmptyDataFrameException`	The input GeoDataFrame is empty.
`InvalidParameterException`	The number of clusters is less than one.

Source code in `eis_toolkit/exploratory_analyses/k_means_cluster.py`

```python
@beartype
def k_means_clustering(
    data: gdp.GeoDataFrame, number_of_clusters: Optional[int] = None, random_state: Optional[int] = None
) -> gdp.GeoDataFrame:
    """
    Perform k-means clustering on the input data.

    Args:
        data: A GeoDataFrame containing the input data.
        number_of_clusters: The number of clusters (>= 1) to form. Optional parameter. If not provided,
            optimal number of clusters is computed using the elbow method.
        random_state: A random number generation for centroid initialization to make
            the randomness deterministic. Optional parameter.

    Returns:
        GeoDataFrame containing assigned cluster labels.

    Raises:
        EmptyDataFrameException: The input GeoDataFrame is empty.
        InvalidParameterException: The number of clusters is less than one.
    """

    if data.empty:
        raise EmptyDataFrameException("The input GeoDataFrame is empty.")

    if number_of_clusters is not None and number_of_clusters < 1:
        raise InvalidParameterValueException("The input value for number of clusters must be at least one.")

    k_means_gdf = _k_means_clustering(data, number_of_clusters, random_state)

    return k_means_gdf
```

4.5 Plot parallel coordinates

```
plot_parallel_coordinates(df, color_column_name, plot_title=None,
palette_name=None, curved_lines=True)
```

Plot a parallel coordinates plot.

Automatically removes all rows containing null/nan values. Tries to convert columns to numeric to be able to plot them. If more than 8 columns are present (after numeric filtering), keeps only the first 8 to plot.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The DataFrame to plot.	required
`color_column_name`	`str`	The name of the column in df to use for color encoding.	required
`plot_title`	`Optional[str]`	The title for the plot. Default is None.	`None`
`palette_name`	`Optional[str]`	The name of the color palette to use. Default is None.	`None`
`curved_lines`	`bool`	If True, the plot will have curved instead of straight lines. Default is True.	`True`

Returns:

Type	Description
`Figure`	A matplotlib figure containing the parallel coordinates plot.

Raises:

Type	Description
`EmptyDataFrameException`	Raised when the DataFrame is empty.
`InvalidColumnException`	Raised when the color column is not found in the DataFrame.
`InconsistentDataTypesException`	Raised when the color column has multiple data types.

Source code in `eis_toolkit/exploratory_analyses/parallel_coordinates.py`

```python
@beartype
def plot_parallel_coordinates(
    df: pd.DataFrame,
    color_column_name: str,
    plot_title: Optional[str] = None,
    palette_name: Optional[str] = None,
    curved_lines: bool = True,
) -> matplotlib.figure.Figure:
    """Plot a parallel coordinates plot.

    Automatically removes all rows containing null/nan values. Tries to convert columns to numeric
    to be able to plot them. If more than 8 columns are present (after numeric filtering), keeps only
    the first 8 to plot.

    Args:
        df: The DataFrame to plot.
        color_column_name: The name of the column in df to use for color encoding.
        plot_title: The title for the plot. Default is None.
        palette_name: The name of the color palette to use. Default is None.
        curved_lines: If True, the plot will have curved instead of straight lines. Default is True.

    Returns:
        A matplotlib figure containing the parallel coordinates plot.

    Raises:
        EmptyDataFrameException: Raised when the DataFrame is empty.
        InvalidColumnException: Raised when the color column is not found in the DataFrame.
        InconsistentDataTypesException: Raised when the color column has multiple data types.
    """

    if df.empty:
        raise exceptions.EmptyDataFrameException("The input DataFrame is empty.")

    if color_column_name not in df.columns:
        raise exceptions.InvalidColumnException(
            f"The provided color column {color_column_name} is not found in the DataFrame."
        )

    df = df.convert_dtypes()
    df = df.apply(pd.to_numeric, errors="ignore")

    color_data = df[color_column_name].to_numpy()
    if len(set([type(elem) for elem in color_data])) != 1:
        raise exceptions.InconsistentDataTypesException(
            "The color column should have a consistent datatype. Multiple data types detected in the color column."
        )

    df = df.select_dtypes(include=np.number)

    # Drop non-numeric columns and the column used for coloring
    columns_to_drop = [color_column_name]
    for column in df.columns.values:
        if df[column].isnull().all():
            columns_to_drop.append(column)
    df = df.loc[:, ~df.columns.isin(columns_to_drop)]

    # Keep only first 8 columns if more are still present
    if len(df.columns.values) > 8:
        df = df.iloc[:, :8]

    data_labels = df.columns.values
    data = df.to_numpy()

    fig = _plot_parallel_coordinates(
        data=data,
        data_labels=data_labels,
        color_data=color_data,
        color_column_name=color_column_name,
        plot_title=plot_title,
        palette_name=palette_name,
        curved_lines=curved_lines,
    )
    return fig
```

4.6 PCA

```
compute_pca(data, number_of_components, scaler_type='standard', nodata=None,
color_column_name=None)
```

Compute given number of principal components for numeric input data.

Various input data formats are accepted and the output format depends on the input format. If input is (Geo)DataFrame, a pairplot is produced additionally. A column name used for coloring can be specified in this case.

Parameters:

Name	Type	Description	Default
`data`	`Union[ndarray, DataFrame, GeoDataFrame, DatasetReader]`	Input data for PCA.	required
`number_of_components`	`int`	The number of principal components to compute Should be >= 1 and at most the number of numeric columns if input is (Geo)DataFrame or number of bands if input is raster.	required
`scaler_type`	`Literal['standard', 'min_max', 'robust']`	Transform data according to a specified Sklearn scaler. Options are "standard", "min_max" and "robust". Defaults to "standard".	`'standard'`
`nodata`	`Optional[Number]`	Define nodata value to be masked out. Optional parameter. If None and input is raster, looks for nodata value from raster metadata. Defaults to None.	`None`
`color_column_name`	`Optional[str]`	If input data is a DataFrame or a GeoDataFrame, column name used for coloring data points in the produced pairplot can be defined. Defaults to None.	`None`

Returns:

Type	Description
`Union[ndarray, Tuple[DataFrame, PairGrid], Tuple[GeoDataFrame, PairGrid], Tuple[ndarray, Profile]]`	The computed principal components in corresponding format as the input data (for raster, output is
`ndarray`	Numpy array containing the data and raster profile) and the explained variance ratios for each component.

Raises:

Type	Description
`EmptyDataException`	The input is empty.
`InvalidNumberOfPrincipalComponents`	The number of principal components is less than 1 or more than number of columns if input was (Geo)DataFrame.

Source code in `eis_toolkit/exploratory_analyses/pca.py`

```python
@beartype
def compute_pca(
    data: Union[np.ndarray, pd.DataFrame, gpd.GeoDataFrame, rasterio.io.DatasetReader],
    number_of_components: int,
    scaler_type: Literal["standard", "min_max", "robust"] = "standard",
    nodata: Optional[Number] = None,
    color_column_name: Optional[str] = None,
) -> Tuple[
    Union[
        np.ndarray,
        Tuple[pd.DataFrame, sns.PairGrid],
        Tuple[gpd.GeoDataFrame, sns.PairGrid],
        Tuple[np.ndarray, rasterio.profiles.Profile],
    ],
    np.ndarray,
]:
    """
    Compute given number of principal components for numeric input data.

    Various input data formats are accepted and the output format depends on the input format. If
    input is (Geo)DataFrame, a pairplot is produced additionally. A column name used for coloring can
    be specified in this case.

    Args:
        data: Input data for PCA.
        number_of_components: The number of principal components to compute Should be >= 1 and at most
            the number of numeric columns if input is (Geo)DataFrame or number of bands if input is raster.
        scaler_type: Transform data according to a specified Sklearn scaler.
            Options are "standard", "min_max" and "robust". Defaults to "standard".
        nodata: Define nodata value to be masked out. Optional parameter. If None and input is raster, looks
            for nodata value from raster metadata. Defaults to None.
        color_column_name: If input data is a DataFrame or a GeoDataFrame, column name used for
            coloring data points in the produced pairplot can be defined. Defaults to None.

    Returns:
        The computed principal components in corresponding format as the input data (for raster, output is
        Numpy array containing the data and raster profile) and the explained variance ratios for each component.

    Raises:
        EmptyDataException: The input is empty.
        InvalidNumberOfPrincipalComponents: The number of principal components is less than 1 or more than
            number of columns if input was (Geo)DataFrame.
    """
    if scaler_type not in SCALERS:
        raise exceptions.InvalidParameterValueException(f"Invalid scaler. Choose from: {list(SCALERS.keys())}")

    if number_of_components < 1:
        raise exceptions.InvalidParameterValueException("The number of principal components should be >= 1.")

    # Get feature matrix (Numpy array) from various input types
    if isinstance(data, np.ndarray):
        feature_matrix = data
        if feature_matrix.ndim == 2:  # Table-like data (assumme it is a DataFrame transformed to Numpy array)
            feature_matrix, nan_mask = _prepare_array_data(feature_matrix, nodata_value=nodata, reshape=False)
        elif feature_matrix.ndim == 3:  # Assume data represents multiband raster data
            rows, cols = feature_matrix.shape[1], feature_matrix.shape[2]
            feature_matrix, nan_mask = _prepare_array_data(feature_matrix, nodata_value=nodata, reshape=True)
        else:
            raise exceptions.InvalidParameterValueException(
                f"Unsupported input data format. {feature_matrix.ndim} dimensions detected."
            )
        if feature_matrix.size == 0:
            raise exceptions.EmptyDataException("Input array is empty.")

    elif isinstance(data, rasterio.io.DatasetReader):
        feature_matrix = data.read()
        if feature_matrix.ndim < 3:
            raise exceptions.InvalidParameterValueException("Input raster should have multiple bands.")
        rows, cols = feature_matrix.shape[1], feature_matrix.shape[2]
        if nodata is None:
            nodata = data.nodata
        feature_matrix, nan_mask = _prepare_array_data(feature_matrix, nodata_value=nodata, reshape=True)

    elif isinstance(data, pd.DataFrame):
        df = data.copy()
        if df.empty:
            raise exceptions.EmptyDataException("Input DataFrame is empty.")
        if number_of_components > len(df.columns):
            raise exceptions.InvalidParameterValueException(
                "The number of principal should be at most the number of numeric columns in the input DataFrame."
            )
        if color_column_name is not None:
            color_column_data = df[color_column_name]

        if isinstance(data, gpd.GeoDataFrame):
            geometries = data.geometry
            crs = data.crs
            df = df.drop(columns=["geometry"])

        df = df.convert_dtypes()
        df = df.apply(pd.to_numeric, errors="ignore")
        df = df.select_dtypes(include=np.number)
        df = df.astype(dtype=np.number)
        feature_matrix = df.to_numpy()
        feature_matrix = feature_matrix.astype(float)
        feature_matrix, nan_mask = _handle_missing_values(feature_matrix, nodata)

    # Core PCA computation
    principal_components, explained_variances = _compute_pca(feature_matrix, number_of_components, scaler_type)

    # Put nodata back in and consider new dimension of data
    if nodata is not None:
        principal_components[nan_mask[:, number_of_components]] = nodata
    else:
        principal_components[nan_mask[:, :number_of_components]] = np.nan

    # Convert PCA output to proper format
    if isinstance(data, np.ndarray):
```

```
        if data.ndim == 3:
            result_data = principal_components.reshape(rows, cols, -1).transpose(2, 0, 1)
        else:
            result_data = principal_components

    elif isinstance(data, rasterio.io.DatasetReader):
        principal_components = principal_components.reshape(rows, cols, -1).transpose(2, 0, 1)
        out_profile = data.profile.copy()
        out_profile["count"] = number_of_components
        out_profile["dtype"] = "float32"
        result_data = (principal_components, out_profile)

    elif isinstance(data, pd.DataFrame):
        component_names = [f"principal_component_{i+1}" for i in range(number_of_components)]
        pca_df = pd.DataFrame(data=principal_components, columns=component_names)
        if color_column_name is not None:
            pca_df[color_column_name] = color_column_data
        sns_pair_grid = plot_pca(pca_df, explained_variances, color_column_name)
        if isinstance(data, gpd.GeoDataFrame):
            pca_df = gpd.GeoDataFrame(pca_df, geometry=geometries, crs=crs)
        result_data = (pca_df, sns_pair_grid)

    return result_data, explained_variances
```

`plot_pca(pca_df, explained_variances=None, color_column_name=None, save_path=None)`

Plot a scatter matrix of different principal component combinations.

Parameters:

Name	Type	Description	Default
`pca_df`	`DataFrame`	A DataFrame containing computed principal components.	required
`explained_variances`	`Optional[ndarray]`	The explained variance ratios for each principal component. Used for labeling axes in the plot. Optional parameter. Defaults to None.	`None`
`color_column_name`	`Optional[str]`	Name of the column that will be used for color-coding data points. Typically a categorical variable in the original data. Optional parameter, no colors if not provided. Defaults to None.	`None`
`save_path`	`Optional[str]`	The save path for the plot. Optional parameter, no saving if not provided. Defaults to None.	`None`

Returns:

Type	Description
`PairGrid`	A Seaborn pairgrid containing the PCA scatter matrix.

Raises:

Type	Description
`InvalidColumnException`	DataFrame does not contain the given color column.

Source code in `eis_toolkit/exploratory_analyses/pca.py`

```python
@beartype
def plot_pca(
    pca_df: pd.DataFrame,
    explained_variances: Optional[np.ndarray] = None,
    color_column_name: Optional[str] = None,
    save_path: Optional[str] = None,
) -> sns.PairGrid:
    """Plot a scatter matrix of different principal component combinations.

    Args:
        pca_df: A DataFrame containing computed principal components.
        explained_variances: The explained variance ratios for each principal component. Used for labeling
            axes in the plot. Optional parameter. Defaults to None.
        color_column_name: Name of the column that will be used for color-coding data points. Typically a
            categorical variable in the original data. Optional parameter, no colors if not provided.
            Defaults to None.
        save_path: The save path for the plot. Optional parameter, no saving if not provided. Defaults to None.

    Returns:
        A Seaborn pairgrid containing the PCA scatter matrix.

    Raises:
        InvalidColumnException: DataFrame does not contain the given color column.
    """

    if color_column_name and color_column_name not in pca_df.columns:
        raise exceptions.InvalidColumnException("DataFrame does not contain the given color column.")

    pair_grid = sns.pairplot(pca_df, hue=color_column_name)

    # Add explained variances to axis labels if provided
    if explained_variances is not None:
        labels = [f"PC {i+1} ({var:.1f}%)" for i, var in enumerate(explained_variances * 100)]
    else:
        labels = [f"PC {i+1}" for i in range(len(pair_grid.axes))]

    # Iterate over axes objects and set the labels
    for i, ax_row in enumerate(pair_grid.axes):
        for j, ax in enumerate(ax_row):
            if j == 0:  # Only the first column
                ax.set_ylabel(labels[i], fontsize="large")
            if i == len(ax_row) - 1:  # Only the last row
                ax.set_xlabel(labels[j], fontsize="large")

    if save_path is not None:
        plt.savefig(save_path)

    return pair_grid
```

4.7 Statistical (hypothesis) testing

`chi_square_test(data, target_column, columns=None)`

Compute Chi-square test for independence on the input data.

It is assumed that the variables in the input data are independent and that they are categorical, i.e. strings, booleans or integers, but not floats.

Parameters:

Name	Type	Description	Default
`data`	`DataFrame`	Dataframe containing the input data	required
`target_column`	`str`	Variable against which independence of other variables is tested.	required
`columns`	`Sequence[str]`	Variables that are tested against the variable in target_column. If None, every column is used.	`None`

Raises:

Type	Description
`EmptyDataFrameException`	The input Dataframe is empty.
`InvalidParameterValueException`	The target_column is not in input Dataframe or invalid column is provided.

Returns:

Type	Description
`dict`	Test statistics for each variable (except target_column).

Source code in `eis_toolkit/exploratory_analyses/statistical_tests.py`

```python
@beartype
def chi_square_test(data: pd.DataFrame, target_column: str, columns: Sequence[str] = None) -> dict:
    """Compute Chi-square test for independence on the input data.

    It is assumed that the variables in the input data are independent and that they are categorical, i.e. strings,
    booleans or integers, but not floats.

    Args:
        data: Dataframe containing the input data
        target_column: Variable against which independence of other variables is tested.
        columns: Variables that are tested against the variable in target_column. If None, every column is used.

    Raises:
        EmptyDataFrameException: The input Dataframe is empty.
        InvalidParameterValueException: The target_column is not in input Dataframe or invalid column is provided.

    Returns:
        Test statistics for each variable (except target_column).
    """
    if check_empty_dataframe(data):
        raise exceptions.EmptyDataFrameException("The input Dataframe is empty.")

    if not check_columns_valid(data, target_column):
        raise exceptions.InvalidParameterValueException("Target column not found in the Dataframe.")

    if columns is not None:
        invalid_columns = [column for column in columns if column not in data.columns]
        if any(invalid_columns):
            raise exceptions.InvalidParameterValueException(
                f"The following variables are not in the dataframe: {invalid_columns}"
            )
    else:
        columns = data.columns

    statistics = {}
    for column in columns:
        if column != target_column:
            contingency_table = pd.crosstab(data[target_column], data[column])
            chi_square, p_value, degrees_of_freedom, _ = chi2_contingency(contingency_table)
            statistics[column] = (chi_square, p_value, degrees_of_freedom)

    return statistics
```

`correlation_matrix(data, correlation_method='pearson', min_periods=None)`

Compute correlation matrix on the input data.

It is assumed that the data is numeric, i.e. integers or floats.

Parameters:

Name	Type	Description	Default
`data`	`DataFrame`	Dataframe containing the input data.	required
`correlation_method`	`Literal[pearson, kendall, spearman]`	'pearson', 'kendall', or 'spearman'. Defaults to 'pearson'.	`'pearson'`
`min_periods`	`Optional[int]`	Minimum number of observations required per pair of columns to have valid result. Optional.	`None`

Raises:

Type	Description
`EmptyDataFrameException`	The input Dataframe is empty.
`InvalidParameterValueException`	min_periods argument is used with method 'kendall'.

Returns:

Type	Description
`DataFrame`	Dataframe containing the correlation matrix

Source code in `eis_toolkit/exploratory_analyses/statistical_tests.py`

```
@beartype
def correlation_matrix(
    data: pd.DataFrame,
    correlation_method: Literal["pearson", "kendall", "spearman"] = "pearson",
    min_periods: Optional[int] = None,
) -> pd.DataFrame:
    """Compute correlation matrix on the input data.

    It is assumed that the data is numeric, i.e. integers or floats.

    Args:
        data: Dataframe containing the input data.
        correlation_method: 'pearson', 'kendall', or 'spearman'. Defaults to 'pearson'.
        min_periods: Minimum number of observations required per pair of columns to have valid result. Optional.

    Raises:
        EmptyDataFrameException: The input Dataframe is empty.
        InvalidParameterValueException: min_periods argument is used with method 'kendall'.

    Returns:
        Dataframe containing the correlation matrix
    """
    if check_empty_dataframe(data):
        raise exceptions.EmptyDataFrameException("The input Dataframe is empty.")

    if correlation_method == "kendall" and min_periods is not None:
        raise exceptions.InvalidParameterValueException(
            "The argument min_periods is available only with correlation methods 'pearson' and 'spearman'."
        )

    matrix = data.corr(method=correlation_method, min_periods=min_periods)

    return matrix
```

`covariance_matrix(data, min_periods=None, delta_degrees_of_freedom=1)`

Compute covariance matrix on the input data.

It is assumed that the data is numeric, i.e. integers or floats.

Parameters:

Name	Type	Description	Default
`data`	`DataFrame`	Dataframe containing the input data.	required
`min_periods`	`Optional[int]`	Minimum number of observations required per pair of columns to have valid result. Optional.	`None`
`delta_degrees_of_freedom`	`int`	Delta degrees of freedom used for computing covariance matrix. Defaults to 1.	`1`

Raises:

Type	Description
`EmptyDataFrameException`	The input Dataframe is empty.
`InvalidParameterValueException`	Provided value for delta_degrees_of_freedom is negative.

Returns:

Type	Description
`DataFrame`	Dataframe containing the covariance matrix

Source code in `eis_toolkit/exploratory_analyses/statistical_tests.py`

```python
@beartype
def covariance_matrix(
    data: pd.DataFrame, min_periods: Optional[int] = None, delta_degrees_of_freedom: int = 1
) -> pd.DataFrame:
    """Compute covariance matrix on the input data.

    It is assumed that the data is numeric, i.e. integers or floats.

    Args:
        data: Dataframe containing the input data.
        min_periods: Minimum number of observations required per pair of columns to have valid result. Optional.
        delta_degrees_of_freedom: Delta degrees of freedom used for computing covariance matrix. Defaults to 1.

    Raises:
        EmptyDataFrameException: The input Dataframe is empty.
        InvalidParameterValueException: Provided value for delta_degrees_of_freedom is negative.

    Returns:
        Dataframe containing the covariance matrix
    """
    if check_empty_dataframe(data):
        raise exceptions.EmptyDataFrameException("The input Dataframe is empty.")

    if delta_degrees_of_freedom < 0:
        raise exceptions.InvalidParameterValueException("Delta degrees of freedom must be non-negative.")

    matrix = data.cov(min_periods=min_periods, ddof=delta_degrees_of_freedom)

    return matrix
```

`normality_test(data)`

Compute Shapiro-Wilk test for normality on the input data.

It is assumed that the input data is normally distributed and numeric, i.e. integers or floats.

Parameters:

Name	Type	Description	Default
`data`	`DataFrame`	Dataframe containing the input data.	required

Returns:

Type	Description
`dict`	Test statistics for each variable.

Raises:

Type	Description
`EmptyDataFrameException`	The input Dataframe is empty.

Source code in `eis_toolkit/exploratory_analyses/statistical_tests.py`

```python
@beartype
def normality_test(data: pd.DataFrame) -> dict:
    """Compute Shapiro-Wilk test for normality on the input data.

    It is assumed that the input data is normally distributed and numeric, i.e. integers or floats.

    Args:
        data: Dataframe containing the input data.

    Returns:
        Test statistics for each variable.

    Raises:
        EmptyDataFrameException: The input Dataframe is empty.
    """
    if check_empty_dataframe(data):
        raise exceptions.EmptyDataFrameException("The input Dataframe is empty.")

    statistics = {}
    for column in data.columns:
        statistic, p_value = shapiro(data[column])
        statistics[column] = (statistic, p_value)

    return statistics
```

5. Prediction

5.1 Fuzzy overlay

`and_overlay(data)`

Compute an 'and' overlay operation with fuzzy logic.

Parameters:

Name	Type	Description	Default
`data`	`ndarray`	The input data as a 3D Numpy array. Each 2D array represents a raster band. Data points should be in the range [0, 1].	required

Returns:

Type	Description
`ndarray`	2D Numpy array with the result of the 'and' overlay operation. Values are in range [0, 1].

Raises:

Type	Description
`InvalidParameterValueException`	If data values are not in range [0, 1].

Source code in `eis_toolkit/prediction/fuzzy_overlay.py`

```
14  @beartype
15  def and_overlay(data: np.ndarray) -> np.ndarray:
16      """Compute an 'and' overlay operation with fuzzy logic.
17
18      Args:
19          data: The input data as a 3D Numpy array. Each 2D array represents a raster band.
20              Data points should be in the range [0, 1].
21
22      Returns:
23          2D Numpy array with the result of the 'and' overlay operation. Values are in range [0, 1].
24
25      Raises:
26          InvalidParameterValueException: If data values are not in range [0, 1].
27      """
28      _check_input_data(data=data)
29
30      return data.min(axis=0)
```

`gamma_overlay(data, gamma)`

Compute a 'gamma' overlay operation with fuzzy logic.

Parameters:

Name	Type	Description	Default
`data`	`ndarray`	The input data as a 3D Numpy array. Each 2D array represents a raster band. Data points should be in the range [0, 1].	required
`gamma`	`float`	The gamma parameter. With gamma value 0, result will be same as 'product'overlay. When gamma is closer to 1, the weight of 'sum' overlay is increased. Value must be in the range [0, 1].	required

Returns:

Type	Description
`ndarray`	2D Numpy array with the result of the 'gamma' overlay operation. Values are in range [0, 1].

Raises:

Type	Description
`InvalidParameterValueException`	If data values or gamma are not in range [0, 1].

Source code in `eis_toolkit/prediction/fuzzy_overlay.py`

```
90  @beartype
91  def gamma_overlay(data: np.ndarray, gamma: float) -> np.ndarray:
92      """Compute a 'gamma' overlay operation with fuzzy logic.
93
94      Args:
95          data: The input data as a 3D Numpy array. Each 2D array represents a raster band.
96              Data points should be in the range [0, 1].
97          gamma: The gamma parameter. With gamma value 0, result will be same as 'product'overlay.
98              When gamma is closer to 1, the weight of 'sum' overlay is increased.
99              Value must be in the range [0, 1].
100
101     Returns:
102         2D Numpy array with the result of the 'gamma' overlay operation. Values are in range [0, 1].
103
104     Raises:
105         InvalidParameterValueException: If data values or gamma are not in range [0, 1].
106     """
107     if gamma < 0 or gamma > 1:
108         raise exceptions.InvalidParameterValueException("The gamma parameter must be in range [0, 1]")
109
110     sum = sum_overlay(data=data)
111     product = product_overlay(data=data)
112     return product ** (1 - gamma) * sum**gamma
```

`or_overlay(data)`

Compute an 'or' overlay operation with fuzzy logic.

Parameters:

Name	Type	Description	Default
`data`	`ndarray`	The input data as a 3D Numpy array. Each 2D array represents a raster band. Data points should be in the range [0, 1].	required

Returns:

Type	Description
`ndarray`	2D Numpy array with the result of the 'or' overlay operation. Values are in range [0, 1].

Raises:

Type	Description
`InvalidParameterValueException`	If data values are not in range [0, 1].

Source code in `eis_toolkit/prediction/fuzzy_overlay.py`

```
@beartype
def or_overlay(data: np.ndarray) -> np.ndarray:
    """Compute an 'or' overlay operation with fuzzy logic.

    Args:
        data: The input data as a 3D Numpy array. Each 2D array represents a raster band.
            Data points should be in the range [0, 1].

    Returns:
        2D Numpy array with the result of the 'or' overlay operation. Values are in range [0, 1].

    Raises:
        InvalidParameterValueException: If data values are not in range [0, 1].
    """
    _check_input_data(data=data)

    return data.max(axis=0)
```

`product_overlay(data)`

Compute a 'product' overlay operation with fuzzy logic.

Parameters:

Name	Type	Description	Default
`data`	`ndarray`	The input data as a 3D Numpy array. Each 2D array represents a raster band. Data points should be in the range [0, 1].	required

Returns:

Type	Description
`ndarray`	2D Numpy array with the result of the 'product' overlay operation. Values are in range [0, 1].

Raises:

Type	Description
`InvalidParameterValueException`	If data values are not in range [0, 1].

Source code in `eis_toolkit/prediction/fuzzy_overlay.py`

```
@beartype
def product_overlay(data: np.ndarray) -> np.ndarray:
    """Compute a 'product' overlay operation with fuzzy logic.

    Args:
        data: The input data as a 3D Numpy array. Each 2D array represents a raster band.
            Data points should be in the range [0, 1].

    Returns:
        2D Numpy array with the result of the 'product' overlay operation. Values are in range [0, 1].

    Raises:
        InvalidParameterValueException: If data values are not in range [0, 1].
    """
    _check_input_data(data=data)

    return np.prod(data, axis=0)
```

`sum_overlay(data)`

Compute a 'sum' overlay operation with fuzzy logic.

Parameters:

Name	Type	Description	Default
`data`	`ndarray`	The input data as a 3D Numpy array. Each 2D array represents a raster band. Data points should be in the range [0, 1].	required

Returns:

Type	Description
`ndarray`	2D Numpy array with the result of the 'sum' overlay operation. Values are in range [0, 1].

Raises:

Type	Description
`InvalidParameterValueException`	If data values are not in range [0, 1].

Source code in `eis_toolkit/prediction/fuzzy_overlay.py`

```
71  @beartype
72  def sum_overlay(data: np.ndarray) -> np.ndarray:
73      """Compute a 'sum' overlay operation with fuzzy logic.
74
75      Args:
76          data: The input data as a 3D Numpy array. Each 2D array represents a raster band.
77              Data points should be in the range [0, 1].
78
79      Returns:
80          2D Numpy array with the result of the 'sum' overlay operation. Values are in range [0, 1].
81
82      Raises:
83          InvalidParameterValueException: If data values are not in range [0, 1].
84      """
85      _check_input_data(data=data)
86
87      return data.sum(axis=0) - np.prod(data, axis=0)
```

5.2 Weights of evidence

`weights_of_evidence_calculate_responses(output_arrays, nr_of_deposits, nr_of_pixels)`

Calculate the posterior probabilities for the given generalized weight arrays.

Parameters:

Name	Type	Description	Default
`output_arrays`	`Sequence[Dict[str, ndarray]]`	List of output array dictionaries returned by weights of evidence calculations. For each dictionary, generalized weight and generalized standard deviation arrays are used and summed together pixel-wise to calculate the posterior probabilities. If generalized arrays are not found, the W+ and S_W+ arrays are used (so if outputs from unique weight calculations are used for this function).	required
`nr_of_deposits`	`int`	Number of deposit pixels in the input data for weights of evidence calculations.	required
`nr_of_pixels`	`int`	Number of evidence pixels in the input data for weights of evidence calculations.	required

Returns:

Type	Description
`ndarray`	Array of posterior probabilites.
`ndarray`	Array of standard deviations in the posterior probability calculations.
`ndarray`	Array of confidence of the prospectivity values obtained in the posterior probability array.

Source code in `eis_toolkit/prediction/weights_of_evidence.py`

```python
@beartype
def weights_of_evidence_calculate_responses(
    output_arrays: Sequence[Dict[str, np.ndarray]], nr_of_deposits: int, nr_of_pixels: int
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Calculate the posterior probabilities for the given generalized weight arrays.

    Args:
        output_arrays: List of output array dictionaries returned by weights of evidence calculations.
            For each dictionary, generalized weight and generalized standard deviation arrays are used and summed
            together pixel-wise to calculate the posterior probabilities. If generalized arrays are not found,
            the W+ and S_W+ arrays are used (so if outputs from unique weight calculations are used for this function).
        nr_of_deposits: Number of deposit pixels in the input data for weights of evidence calculations.
        nr_of_pixels: Number of evidence pixels in the input data for weights of evidence calculations.

    Returns:
        Array of posterior probabilites.
        Array of standard deviations in the posterior probability calculations.
        Array of confidence of the prospectivity values obtained in the posterior probability array.
    """
    gen_weights_sum = sum(
        [
            item[GENERALIZED_WEIGHT_PLUS_COLUMN]
            if GENERALIZED_WEIGHT_PLUS_COLUMN in item.keys()
            else item[WEIGHT_PLUS_COLUMN]
            for item in output_arrays
        ]
    )
    gen_weights_variance_sum = sum(
        [
            np.square(item[GENERALIZED_S_WEIGHT_PLUS_COLUMN])
            if GENERALIZED_S_WEIGHT_PLUS_COLUMN in item.keys()
            else np.square(item[WEIGHT_S_PLUS_COLUMN])
            for item in output_arrays
        ]
    )

    prior_probabilities = nr_of_deposits / nr_of_pixels
    prior_odds = np.log(prior_probabilities / (1 - prior_probabilities))
    posterior_probabilities = np.exp(gen_weights_sum + prior_odds) / (1 + np.exp(gen_weights_sum + prior_odds))

    posterior_probabilities_squared = np.square(posterior_probabilities)
    posterior_probabilities_std = np.sqrt(
        (1 / nr_of_deposits + gen_weights_variance_sum) * posterior_probabilities_squared
    )

    confidence_array = posterior_probabilities / posterior_probabilities_std
    return posterior_probabilities, posterior_probabilities_std, confidence_array
```

`weights_of_evidence_calculate_weights(evidential_raster, deposits, raster_nodata=None, weights_type='unique', studentized_contrast_threshold=1, arrays_to_generate=None)`

Calculate weights of spatial associations.

Parameters:

Name	Type	Description	Default
`evidential_raster`	`DatasetReader`	The evidential raster.	required
`deposits`	`GeoDataFrame`	Vector data representing the mineral deposits or occurences point data.	required
`raster_nodata`	`Optional[Number]`	If nodata value of raster is wanted to specify manually. Optional parameter, defaults to None (nodata from raster metadata is used).	`None`
`weights_type`	`Literal[unique, categorical, ascending, descending]`	Accepted values are 'unique', 'categorical', 'ascending' and 'descending'. Unique weights does not create generalized classes and does not use a studentized contrast threshold value while categorical, cumulative ascending and cumulative descending do. Categorical weights are calculated so that all classes with studentized contrast below the defined threshold are grouped into one generalized class. Cumulative	`'unique'`

Name	Type	Description	Default
		ascending and descending weights find the class with max contrast and group classes above/ below into generalized classes. Generalized weights are also calculated for generalized classes.	
`studentized_contrast_threshold`	`Number`	Studentized contrast threshold value used with 'categorical', 'ascending' and 'descending' weight types. Used either as reclassification threshold directly (categorical) or to check that class with max contrast has studentized contrast value at least the defined value (cumulative). Defaults to 1.	`1`
`arrays_to_generate`	`Optional[Sequence[str]]`	Arrays to generate from the computed weight metrics. All column names in the produced weights_df are valid choices. Defaults to ["Class", "W+", "S_W+] for "unique" weights_type and ["Class", "W+", "S_W+",	`None`

Name	Type	Description	Default
		"Generalized W+", "Generalized S_W+"] for the cumulative weight types.	

Returns:

Type	Description
`DataFrame`	Dataframe with weights of spatial association between the input data.
`dict`	Dictionary of arrays for specified metrics.
`dict`	Raster metadata.
`int`	Number of deposit pixels.
`int`	Number of all evidence pixels.

Source code in `eis_toolkit/prediction/weights_of_evidence.py`

```
@beartype
def weights_of_evidence_calculate_weights(
    evidential_raster: rasterio.io.DatasetReader,
    deposits: gpd.GeoDataFrame,
    raster_nodata: Optional[Number] = None,
    weights_type: Literal["unique", "categorical", "ascending", "descending"] = "unique",
    studentized_contrast_threshold: Number = 1,
    arrays_to_generate: Optional[Sequence[str]] = None,
) -> Tuple[pd.DataFrame, dict, dict, int, int]:
    """
    Calculate weights of spatial associations.

    Args:
        evidential_raster: The evidential raster.
        deposits: Vector data representing the mineral deposits or occurences point data.
        raster_nodata: If nodata value of raster is wanted to specify manually. Optional parameter, defaults to None
            (nodata from raster metadata is used).
        weights_type: Accepted values are 'unique', 'categorical', 'ascending' and 'descending'.
            Unique weights does not create generalized classes and does not use a studentized contrast threshold value
            while categorical, cumulative ascending and cumulative descending do. Categorical weights are calculated so
            that all classes with studentized contrast below the defined threshold are grouped into one generalized
            class. Cumulative ascending and descending weights find the class with max contrast and group classes
            above/below into generalized classes. Generalized weights are also calculated for generalized classes.
        studentized_contrast_threshold: Studentized contrast threshold value used with 'categorical', 'ascending' and
            'descending' weight types. Used either as reclassification threshold directly (categorical) or to check
            that class with max contrast has studentized contrast value at least the defined value (cumulative).
            Defaults to 1.
        arrays_to_generate: Arrays to generate from the computed weight metrics. All column names
            in the produced weights_df are valid choices. Defaults to ["Class", "W+", "S_W+]
            for "unique" weights_type and ["Class", "W+", "S_W+", "Generalized W+", "Generalized S_W+"]
            for the cumulative weight types.

    Returns:
        Dataframe with weights of spatial association between the input data.
        Dictionary of arrays for specified metrics.
        Raster metadata.
        Number of deposit pixels.
        Number of all evidence pixels.
    """

    if arrays_to_generate is None:
        if weights_type == "unique":
            metrics_to_arrays = DEFAULT_METRICS_UNIQUE
        else:
            metrics_to_arrays = DEFAULT_METRICS_CUMULATIVE
    else:
        for col_name in arrays_to_generate:
            if col_name not in VALID_DF_COLUMNS:
                raise exceptions.InvalidColumnException(
                    f"Arrays to generate contains invalid metric / column name: {col_name}."
                )
        metrics_to_arrays = arrays_to_generate.copy()

    # 1. Preprocess data
    evidence_array = _read_and_preprocess_evidence(evidential_raster, raster_nodata)
    raster_meta = evidential_raster.meta

    # Rasterize deposits
    deposit_array, _ = rasterize_vector(
        geodataframe=deposits, default_value=1.0, base_raster_profile=raster_meta, fill_value=0.0
    )

    # Mask NaN out of the array
    nodata_mask = np.isnan(evidence_array)
    masked_evidence_array = evidence_array[~nodata_mask]
    masked_deposit_array = deposit_array[~nodata_mask]

    # 2. WofE calculations
    if weights_type == "unique" or weights_type == "categorical":
        wofe_weights = _unique_weights(masked_deposit_array, masked_evidence_array)
    elif weights_type == "ascending":
        wofe_weights = _cumulative_weights(masked_deposit_array, masked_evidence_array, ascending=True)
    elif weights_type == "descending":
        wofe_weights = _cumulative_weights(masked_deposit_array, masked_evidence_array, ascending=False)
    else:
        raise exceptions.InvalidParameterValueException(
            "Expected weights_type to be one of unique, categorical, ascending or descending."
        )

    # 3. Create DataFrame based on calculated metrics
    df_entries = []
    for cls, metrics in wofe_weights.items():
        metrics = [round(metric, 4) if isinstance(metric, np.floating) else metric for metric in metrics]
        A, _, C, _, w_plus, s_w_plus, w_minus, s_w_minus, contrast, s_contrast, studentized_contrast = metrics
        df_entries.append(
            {
                CLASS_COLUMN: cls,
                PIXEL_COUNT_COLUMN: A + C,
                DEPOSIT_COUNT_COLUMN: A,
                WEIGHT_PLUS_COLUMN: w_plus,
                WEIGHT_S_PLUS_COLUMN: s_w_plus,
                WEIGHT_MINUS_COLUMN: w_minus,
                WEIGHT_S_MINUS_COLUMN: s_w_minus,
                CONTRAST_COLUMN: contrast,
                S_CONTRAST_COLUMN: s_contrast,
                STUDENTIZED_CONTRAST_COLUMN: studentized_contrast,
            }
        )
    weights_df = pd.DataFrame(df_entries)

    # 4. If we use cumulative weights type, calculate generalized classes and weights
    if weights_type == "categorical":
        weights_df = _generalized_classes_categorical(weights_df, studentized_contrast_threshold)
        weights_df = _generalized_weights_categorical(weights_df, masked_deposit_array)
    elif weights_type == "ascending" or weights_type == "descending":
        weights_df = _generalized_classes_cumulative(weights_df, studentized_contrast_threshold)
        weights_df = _generalized_weights_cumulative(weights_df, masked_deposit_array)
```

```
    # 5. Generate arrays for desired metrics
    arrays_dict = _generate_arrays_from_metrics(evidence_array, weights_df, metrics_to_arrays)

    # Return nr. of deposit pixels  and nr. of all evidence pixels for to be used in calculate responses
    nr_of_deposits = int(np.sum(masked_deposit_array == 1))
    nr_of_pixels = int(np.size(masked_evidence_array))

    return weights_df, arrays_dict, raster_meta, nr_of_deposits, nr_of_pixels
```

6. Raster processing

6.1 Check raster grids

`check_raster_grids(rasters, same_extent=False)`

Check the set of input rasters for matching gridding and optionally matching bounds.

Parameters:

Name	Type	Description	Default
`rasters`	`List[DatasetReader]`	List of rasters to test for matching gridding.	required
`same_extent`	`bool`	optional boolean argument that determines if rasters are tested for matching bounds. Default set to False.	`False`

Returns:

Type	Description
`bool`	True if gridding and optionally bounds matches, False if not.

Source code in `eis_toolkit/raster_processing/check_raster_grids.py`

```
23  def check_raster_grids(  # type: ignore[no-any-unimported]
24      rasters: List[rasterio.io.DatasetReader], same_extent: bool = False
25  ) -> bool:
26      """
27      Check the set of input rasters for matching gridding and optionally matching bounds.
28
29      Args:
30          rasters: List of rasters to test for matching gridding.
31          same_extent: optional boolean argument that determines if rasters are tested for matching bounds.
32              Default set to False.
33
34      Returns:
35          True if gridding and optionally bounds matches, False if not.
36      """
37      check = _check_raster_grids(rasters=rasters, same_extent=same_extent)
38      return check
```

6.2 Clipping

`clip_raster(raster, geodataframe)`

Clips a raster with polygon geometries.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	The raster to be clipped.	required
`geodataframe`	`GeoDataFrame`	A geodataframe containing the geometries to do the clipping with. Should contain only polygon features.	required

Returns:

Type	Description
`ndarray`	The clipped raster data.
`dict`	The updated metadata.

Raises:

Type	Description
`NonMatchingCrsException`	The raster and geodataframe are not in the same CRS.
`NotApplicableGeometryTypeException`	The input geometries contain non-polygon features.

Source code in `eis_toolkit/raster_processing/clipping.py`

```
@beartype
def clip_raster(raster: rasterio.io.DatasetReader, geodataframe: geopandas.GeoDataFrame) -> Tuple[np.ndarray, dict]:
    """Clips a raster with polygon geometries.

    Args:
        raster: The raster to be clipped.
        geodataframe: A geodataframe containing the geometries to do the clipping with.
            Should contain only polygon features.

    Returns:
        The clipped raster data.
        The updated metadata.

    Raises:
        NonMatchingCrsException: The raster and geodataframe are not in the same CRS.
        NotApplicableGeometryTypeException: The input geometries contain non-polygon features.
    """
    geometries = geodataframe["geometry"]

    if not check_matching_crs(
        objects=[raster, geometries],
    ):
        raise NonMatchingCrsException("The raster and geodataframe are not in the same CRS.")

    if not check_geometry_types(
        geometries=geometries,
        allowed_types=["Polygon", "MultiPolygon"],
    ):
        raise NotApplicableGeometryTypeException("The input geometries contain non-polygon features.")

    out_image, out_meta = _clip_raster(
        raster=raster,
        geometries=geometries,
    )

    return out_image, out_meta
```

6.3 Create constant raster

```
create_constant_raster(constant_value, template_raster=None, coord_west=None,
coord_north=None, coord_east=None, coord_south=None, target_epsg=None,
target_pixel_size=None, raster_width=None, raster_height=None,
nodata_value=None)
```

Create a constant raster based on a user-defined value.

Provide 3 methods for raster creation: 1. Set extent and coordinate system based on a template raster. 2. Set extent from origin, based on the western and northern coordinates and the pixel size. 3. Set extent from bounds, based on western, northern, eastern and southern points.

Always provide values for height and width for the last two options, which correspond to the desired number of pixels for rows and columns.

Parameters:

Name	Type	Description	Default
`constant_value`	`Number`	The constant value to use in the raster.	required
`template_raster`	`Optional[DatasetReader]`	An optional raster to use as a template for the output.	`None`
`coord_west`	`Optional[Number]`	The western coordinate of the output raster in [m].	`None`
`coord_east`	`Optional[Number]`	The eastern coordinate of the output raster in [m].	`None`
`coord_south`	`Optional[Number]`	The southern coordinate of the output raster in [m].	`None`
`coord_north`	`Optional[Number]`	The northern coordinate of the output raster in [m].	`None`
`target_epsg`	`Optional[int]`	The EPSG code for the output raster.	`None`
`target_pixel_size`	`Optional[int]`	The pixel size of the output raster.	`None`
`raster_width`	`Optional[int]`	The width of the output raster.	`None`
`raster_height`	`Optional[int]`	The height of the output raster.	`None`
`nodata_value`	`Optional[Number]`	The nodata value of the output raster.	`None`

Returns:

Type	Description
`Tuple[ndarray, dict]`	A tuple containing the output raster as a NumPy array and updated metadata.

Raises:

Type	Description
`InvalidParameterValueException`	Provide invalid input parameter.

Source code in `eis_toolkit/raster_processing/create_constant_raster.py`

```python
@beartype
def create_constant_raster(  # type: ignore[no-any-unimported]
    constant_value: Number,
    template_raster: Optional[rasterio.io.DatasetReader] = None,
    coord_west: Optional[Number] = None,
    coord_north: Optional[Number] = None,
    coord_east: Optional[Number] = None,
    coord_south: Optional[Number] = None,
    target_epsg: Optional[int] = None,
    target_pixel_size: Optional[int] = None,
    raster_width: Optional[int] = None,
    raster_height: Optional[int] = None,
    nodata_value: Optional[Number] = None,
) -> Tuple[np.ndarray, dict]:
    """Create a constant raster based on a user-defined value.

    Provide 3 methods for raster creation:
    1. Set extent and coordinate system based on a template raster.
    2. Set extent from origin, based on the western and northern coordinates and the pixel size.
    3. Set extent from bounds, based on western, northern, eastern and southern points.

    Always provide values for height and width for the last two options, which correspond to
    the desired number of pixels for rows and columns.

    Args:
        constant_value: The constant value to use in the raster.
        template_raster: An optional raster to use as a template for the output.
        coord_west: The western coordinate of the output raster in [m].
        coord_east: The eastern coordinate of the output raster in [m].
        coord_south: The southern coordinate of the output raster in [m].
        coord_north: The northern coordinate of the output raster in [m].
        target_epsg: The EPSG code for the output raster.
        target_pixel_size: The pixel size of the output raster.
        raster_width: The width of the output raster.
        raster_height: The height of the output raster.
        nodata_value: The nodata value of the output raster.

    Returns:
        A tuple containing the output raster as a NumPy array and updated metadata.

    Raises:
        InvalidParameterValueException: Provide invalid input parameter.
    """

    if template_raster is not None:
        out_array, out_meta = _create_constant_raster_from_template(constant_value, template_raster, nodata_value)

    elif all(coords is not None for coords in [coord_west, coord_east, coord_south, coord_north]):
        if raster_height <= 0 or raster_width <= 0:
            raise InvalidParameterValueException("Invalid raster extent provided.")
        if not check_minmax_position((coord_west, coord_east) or not check_minmax_position((coord_south, coord_north))):
            raise InvalidParameterValueException("Invalid coordinate values provided.")

        out_array, out_meta = _create_constant_raster_from_bounds(
            constant_value,
            coord_west,
            coord_north,
            coord_east,
            coord_south,
            target_epsg,
            raster_width,
            raster_height,
            nodata_value,
        )

    elif all(coords is not None for coords in [coord_west, coord_north]) and all(
        coords is None for coords in [coord_east, coord_south]
    ):
        if raster_height <= 0 or raster_width <= 0:
            raise InvalidParameterValueException("Invalid raster extent provided.")
        if target_pixel_size <= 0:
            raise InvalidParameterValueException("Invalid pixel size.")

        out_array, out_meta = _create_constant_raster_from_origin(
            constant_value,
            coord_west,
            coord_north,
            target_epsg,
            target_pixel_size,
            raster_width,
            raster_height,
            nodata_value,
        )

    else:
        raise InvalidParameterValueException("Suitable parameter values were not provided for any of the 3 methods.")

    constant_value = cast_scalar_to_int(constant_value)
    nodata_value = cast_scalar_to_int(out_meta["nodata"])

    if isinstance(constant_value, int) and isinstance(nodata_value, int):
        target_dtype = np.result_type(get_min_int_type(constant_value), get_min_int_type(nodata_value))
        out_array = out_array.astype(target_dtype)
        out_meta["dtype"] = out_array.dtype
    elif isinstance(constant_value, int) and isinstance(nodata_value, float):
        out_array = out_array.astype(get_min_int_type(constant_value))
        out_meta["dtype"] = np.float64.__name__
    elif isinstance(constant_value, float):
        out_array = out_array.astype(np.float64)
        out_meta["dtype"] = out_array.dtype

    return out_array, out_meta
```

6.4 Extract values from raster

`extract_values_from_raster(raster_list, geodataframe, raster_column_names=None)`

Extract raster values using point data to a DataFrame.

If custom column names are not given, column names are file_name for singleband files and file_name_bandnumber for multiband files. If custom column names are given, there should be column names for each raster provided in the raster list.

Parameters:

Name	Type	Description	Default
`raster_list`	`Sequence[DatasetReader]`	List to extract values from.	required
`geodataframe`	`GeoDataFrame`	Object to extract values with.	required
`raster_column_names`	`Optional[Sequence[str]]`	List of optional column names for bands.	`None`

Returns:

Type	Description
`DataFrame`	Dataframe with x & y coordinates and the values from the raster file(s) as columns.

Raises:

Type	Description
`NonMatchingParameterLengthsException`	raster_list and raster_columns_names have different lengths.

Source code in `eis_toolkit/raster_processing/extract_values_from_raster.py`

```
@beartype
def extract_values_from_raster(
    raster_list: Sequence[rasterio.io.DatasetReader],
    geodataframe: gpd.GeoDataFrame,
    raster_column_names: Optional[Sequence[str]] = None,
) -> pd.DataFrame:
    """Extract raster values using point data to a DataFrame.

       If custom column names are not given, column names are file_name for singleband files
       and file_name_bandnumber for multiband files. If custom column names are given, there
       should be column names for each raster provided in the raster list.

    Args:
        raster_list: List to extract values from.
        geodataframe: Object to extract values with.
        raster_column_names: List of optional column names for bands.

    Returns:
        Dataframe with x & y coordinates and the values from the raster file(s) as columns.

    Raises:
        NonMatchingParameterLengthsException: raster_list and raster_columns_names have different lengths.
    """
    if raster_column_names == []:
        raster_column_names = None

    if raster_column_names is not None and len(raster_list) != len(raster_column_names):
        raise NonMatchingParameterLengthsException("Raster list and raster columns names have different lengths.")

    data_frame = _extract_values_from_raster(
        raster_list=raster_list, geodataframe=geodataframe, raster_column_names=raster_column_names
    )

    return data_frame
```

6.5 Reprojecting

`reproject_raster(raster, target_crs, resampling_method=warp.Resampling.nearest)`

Reprojects raster to match given coordinate reference system (EPSG).

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	The raster to be reprojected.	required
`target_crs`	`int`	Target CRS as EPSG code.	required
`resampling_method`	`Resampling`	Resampling method. Most suitable method depends on the dataset and context. Nearest, bilinear and cubic are some common choices. This parameter defaults to nearest.	`nearest`

Returns:

Type	Description
`ndarray`	The reprojected raster data.
`dict`	The updated metadata.

Raises:

Type	Description
`NonMatchinCrsException`	Raster is already in the target CRS.

Source code in `eis_toolkit/raster_processing/reprojecting.py`

```
55  @beartype
56  def reproject_raster(
57      raster: rasterio.io.DatasetReader, target_crs: int, resampling_method: warp.Resampling = warp.Resampling.nearest
58  ) -> Tuple[np.ndarray, dict]:
59      """Reprojects raster to match given coordinate reference system (EPSG).
60
61      Args:
62          raster: The raster to be reprojected.
63          target_crs: Target CRS as EPSG code.
64          resampling_method: Resampling method. Most suitable method depends on the dataset and context.
65              Nearest, bilinear and cubic are some common choices. This parameter defaults to nearest.
66
67      Returns:
68          The reprojected raster data.
69          The updated metadata.
70
71      Raises:
72          NonMatchinCrsException: Raster is already in the target CRS.
73      """
74      if target_crs == int(raster.crs.to_string()[5:]):
75          raise MatchingCrsException("Raster is already in the target CRS.")
76
77      out_image, out_meta = _reproject_raster(raster, target_crs, resampling_method)
78
79      return out_image, out_meta
```

6.6 Resampling

`resample(raster, resolution, resampling_method=Resampling.bilinear)`

Resamples raster according to given resolution.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	The raster to be resampled.	required
`resolution`	`Number`	Target resolution i.e. cell size of the output raster.	required
`resampling_method`	`Resampling`	Resampling method. Most suitable method depends on the dataset and context. Nearest, bilinear and cubic are some common choices. This parameter defaults to bilinear.	`bilinear`

Returns:

Type	Description
`ndarray`	The resampled raster data.
`dict`	The updated metadata.

Raises:

Type	Description
`NumericValueSignException`	Resolution is not a positive value.

Source code in `eis_toolkit/raster_processing/resampling.py`

```
52  @beartype
53  def resample(
54      raster: rasterio.io.DatasetReader,
55      resolution: Number,
56      resampling_method: Resampling = Resampling.bilinear,
57  ) -> Tuple[np.ndarray, dict]:
58      """Resamples raster according to given resolution.
59
60      Args:
61          raster: The raster to be resampled.
62          resolution: Target resolution i.e. cell size of the output raster.
63          resampling_method: Resampling method. Most suitable
64              method depends on the dataset and context. Nearest, bilinear and cubic are some
65              common choices. This parameter defaults to bilinear.
66
67      Returns:
68          The resampled raster data.
69          The updated metadata.
70
71      Raises:
72          NumericValueSignException: Resolution is not a positive value.
73      """
74      if resolution <= 0:
75          raise exceptions.NumericValueSignException(f"Expected a positive value for resolution: {resolution})")
76
77      out_image, out_meta = _resample(raster, resolution, resampling_method)
78      return out_image, out_meta
```

6.7 Snapping

`snap_with_raster(raster, snap_raster)`

Snaps/aligns raster to given snap raster.

Raster is snapped from its left-bottom corner to nearest snap raster grid corner in left-bottom direction. If rasters are aligned, simply returns input raster data and metadata.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	The raster to be clipped.	required
`snap_raster`	`DatasetReader`	The snap raster i.e. reference grid raster.	required

Returns:

Type	Description
`ndarray`	The snapped raster data.
`dict`	The updated metadata.

Raises:

Type	Description
`NonMatchingCrsException`	Raster and and snap raster are not in the same CRS.
`MatchingRasterGridException`	Raster grids are already aligned.

Source code in `eis_toolkit/raster_processing/snapping.py`

```
@beartype
def snap_with_raster(raster: rasterio.DatasetReader, snap_raster: rasterio.DatasetReader) -> Tuple[np.ndarray, dict]:
    """Snaps/aligns raster to given snap raster.

    Raster is snapped from its left-bottom corner to nearest snap raster grid corner in left-bottom direction.
    If rasters are aligned, simply returns input raster data and metadata.

    Args:
        raster: The raster to be clipped.
        snap_raster: The snap raster i.e. reference grid raster.

    Returns:
        The snapped raster data.
        The updated metadata.

    Raises:
        NonMatchingCrsException: Raster and and snap raster are not in the same CRS.
        MatchingRasterGridException: Raster grids are already aligned.
    """

    if not check_matching_crs(
        objects=[raster, snap_raster],
    ):
        raise NonMatchingCrsException("Raster and and snap raster have different CRS.")

    if snap_raster.bounds.bottom == raster.bounds.bottom and snap_raster.bounds.left == raster.bounds.left:
        raise MatchingRasterGridException("Raster grids are already aligned.")

    out_image, out_meta = _snap(raster, snap_raster)
    return out_image, out_meta
```

6.8 Unifying

```
unify_raster_grids(base_raster, rasters_to_unify,
resampling_method=Resampling.nearest, same_extent=False)
```

Unifies (reprojects, resamples, aligns and optionally clips) given rasters relative to base raster.

Parameters:

Name	Type	Description	Default
`base_raster`	`DatasetReader`	The base raster to determine target raster grid properties.	required
`rasters_to_unify`	`Sequence[DatasetReader]`	Rasters to be unified with the base raster.	required
`resampling_method`	`Resampling`	Resampling method. Most suitable method depends on the dataset and context. Nearest, bilinear and cubic are some common choices. This parameter defaults to nearest.	`nearest`
`same_extent`	`bool`	If the unified rasters will be forced to have the same extent/bounds as the base raster. Expands smaller rasters with nodata cells. Defaults to False.	`False`

Returns:

Type	Description
`List[Tuple[ndarray, dict]]`	List of unified rasters' data and metadata. First element is the base raster.

Raises:

Type	Description
`InvalidParameterValueException`	Rasters to unify is empty.

Source code in `eis_toolkit/raster_processing/unifying.py`

```python
@beartype
def unify_raster_grids(
    base_raster: rasterio.io.DatasetReader,
    rasters_to_unify: Sequence[rasterio.io.DatasetReader],
    resampling_method: Resampling = Resampling.nearest,
    same_extent: bool = False,
) -> List[Tuple[np.ndarray, dict]]:
    """Unifies (reprojects, resamples, aligns and optionally clips) given rasters relative to base raster.

    Args:
        base_raster: The base raster to determine target raster grid properties.
        rasters_to_unify: Rasters to be unified with the base raster.
        resampling_method: Resampling method. Most suitable
            method depends on the dataset and context. Nearest, bilinear and cubic are some
            common choices. This parameter defaults to nearest.
        same_extent: If the unified rasters will be forced to have the same extent/bounds
            as the base raster. Expands smaller rasters with nodata cells. Defaults to False.

    Returns:
        List of unified rasters' data and metadata. First element is the base raster.

    Raises:
        InvalidParameterValueException: Rasters to unify is empty.
    """
    if len(rasters_to_unify) == 0:
        raise InvalidParameterValueException("Rasters to unify is empty.")

    out_rasters = _unify_raster_grids(base_raster, rasters_to_unify, resampling_method, same_extent)
    return out_rasters
```

6.9 Windowing

`extract_window(raster, center_coords, height, width)`

Extract window from raster.

Center coordinate must be inside the raster but window can extent outside the raster in which case padding with raster nodata value is used. Args: raster: Source raster. center_coords: Center coordinates for window in form (x, y). The coordinates should be in the raster's CRS. height: Window height in pixels. width: Window width in pixels.

Returns:

Type	Description
`ndarray`	The extracted raster window.
`dict`	The updated metadata.

Raises:

Type	Description
`InvalidParameterValueException`	Window size is too small.
`CoordinatesOutOfBoundException`	Window center coordinates are out of raster bounds.

Source code in `eis_toolkit/raster_processing/windowing.py`

```python
@beartype
def extract_window(
    raster: rasterio.io.DatasetReader,
    center_coords: Tuple[Number, Number],
    height: int,
    width: int,
) -> Tuple[np.ndarray, dict]:
    """Extract window from raster.

       Center coordinate must be inside the raster but window can extent outside the raster in which case padding with
       raster nodata value is used.
    Args:
        raster: Source raster.
        center_coords: Center coordinates for window in form (x, y). The coordinates should be in the raster's CRS.
        height: Window height in pixels.
        width: Window width in pixels.

    Returns:
        The extracted raster window.
        The updated metadata.

    Raises:
        InvalidParameterValueException: Window size is too small.
        CoordinatesOutOfBoundException: Window center coordinates are out of raster bounds.
    """

    if height < 1 or width < 1:
        raise InvalidParameterValueException(f"Window size is too small: {height}, {width}.")

    center_x = center_coords[0]
    center_y = center_coords[1]

    if (
        center_x < raster.bounds.left
        or center_x > raster.bounds.right
        or center_y < raster.bounds.bottom
        or center_y > raster.bounds.top
    ):
        raise CoordinatesOutOfBoundsException("Window center coordinates are out of raster bounds.")

    out_image, out_meta = _extract_window(raster, center_coords, height, width)

    return out_image, out_meta
```

7. Training data tools

7.1 Class balancing

```
balance_SMOTETomek(X, y, sampling_strategy='auto', random_state=None)
```

Balances the classes of input dataset using SMOTETomek resampling method.

Parameters:

Name	Type	Description	Default
`X`	`Union[DataFrame, ndarray]`	The feature matrix (input data as a DataFrame).	required
`y`	`Union[Series, ndarray]`	The target labels corresponding to the feature matrix.	required
`sampling_strategy`	`Union[float, str, dict]`	Parameter controlling how to perform the resampling. If float, specifies the ratio of samples in minority class to samples of majority class, if str, specifies classes to be resampled ("minority", "not minority", "not majority", "all", "auto"), if dict, the keys should be targeted classes and values the desired number of samples for the class. Defaults to "auto", which will resample all classes except the majority class.	`'auto'`
`random_state`	`Optional[int]`	Parameter controlling randomization of the algorithm. Can be given a seed (number). Defaults to None, which randomizes the seed.	`None`

Returns:

Type	Description
`tuple[Union[DataFrame, ndarray], Union[Series, ndarray]]`	Resampled feature matrix and target labels.

Raises:

Type	Description
`NonMatchingParameterLengthsException`	If X and y have different length.

Source code in `eis_toolkit/training_data_tools/class_balancing.py`

```python
@beartype
def balance_SMOTETomek(
    X: Union[pd.DataFrame, np.ndarray],
    y: Union[pd.Series, np.ndarray],
    sampling_strategy: Union[float, str, dict] = "auto",
    random_state: Optional[int] = None,
) -> tuple[Union[pd.DataFrame, np.ndarray], Union[pd.Series, np.ndarray]]:
    """Balances the classes of input dataset using SMOTETomek resampling method.

    Args:
        X: The feature matrix (input data as a DataFrame).
        y: The target labels corresponding to the feature matrix.
        sampling_strategy: Parameter controlling how to perform the resampling.
            If float, specifies the ratio of samples in minority class to samples of majority class,
            if str, specifies classes to be resampled ("minority", "not minority", "not majority", "all", "auto"),
            if dict, the keys should be targeted classes and values the desired number of samples for the class.
            Defaults to "auto", which will resample all classes except the majority class.
        random_state: Parameter controlling randomization of the algorithm. Can be given a seed (number).
            Defaults to None, which randomizes the seed.

    Returns:
        Resampled feature matrix and target labels.

    Raises:
        NonMatchingParameterLengthsException: If X and y have different length.
    """

    if len(X) != len(y):
        raise exceptions.NonMatchingParameterLengthsException(
            "Feature matrix X and target labels y must have the same length."
        )

    X_res, y_res = SMOTETomek(sampling_strategy=sampling_strategy, random_state=random_state).fit_resample(X, y)
    return X_res, y_res
```

8. Transformations

8.1 Binarize

`binarize(raster, bands=None, thresholds=[Number], nodata=None)`

Binarize data based on a given threshold.

Replaces values less or equal threshold with 0. Replaces values greater than the threshold with 1.

Takes one nodata value which will be re-written after transformation.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands. The threshold can be set for each band individually.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`thresholds`	`Sequence[Number]`	Threshold values for transformation.	`[Number]`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.

Source code in `eis_toolkit/transformations/binarize.py`

```python
@beartype
def binarize(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    bands: Optional[Sequence[int]] = None,
    thresholds: Sequence[Number] = [Number],
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Binarize data based on a given threshold.

    Replaces values less or equal threshold with 0.
    Replaces values greater than the threshold with 1.

    Takes one nodata value which will be re-written after transformation.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.
    The threshold can be set for each band individually.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        thresholds: Threshold values for transformation.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = cast_scalar_to_int(raster.nodata if nodata is None else nodata)

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection.")

    if check_parameter_length(bands, thresholds) is False:
        raise NonMatchingParameterLengthsException("Invalid threshold length.")

    expanded_args = expand_and_zip(bands, thresholds)
    thresholds = [element[1] for element in expanded_args]

    out_settings = {}

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        inital_dtype = band_array.dtype

        band_mask = np.isin(band_array, nodata)
        band_array = _binarize(band_array, threshold=thresholds[i])
        band_array = np.where(band_mask, nodata, band_array)

        if not check_dtype_for_int(nodata):
            band_array = band_array.astype(inital_dtype)
        else:
            band_array = band_array.astype(np.min_scalar_type(nodata))

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "threshold": thresholds[i],
            "nodata": nodata,
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

8.2 Clip

`clip_transform(raster, limits, bands=None, nodata=None)`

Clips data based on specified upper and lower limits.

Takes one nodata value that will be ignored in calculations. Replaces values below the lower limit and above the upper limit with provided values, respecively. Works both one-sided and two-sided but raises error if no limits provided.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands. The limits can be set for each band individually.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`limits`	`Sequence[Tuple[Optional[Number], Optional[Number]]]`	Lower and upper limits (lower, upper) as real values.	required
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.
`InvalidParameterValueException`	The input does not match the requirements (values, order of values).

Source code in `eis_toolkit/transformations/clip.py`

```python
@beartype
def clip_transform(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    limits: Sequence[Tuple[Optional[Number], Optional[Number]]],
    bands: Optional[Sequence[int]] = None,
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Clips data based on specified upper and lower limits.

    Takes one nodata value that will be ignored in calculations.
    Replaces values below the lower limit and above the upper limit with provided values, respecively.
    Works both one-sided and two-sided but raises error if no limits provided.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.
    The limits can be set for each band individually.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        limits: Lower and upper limits (lower, upper) as real values.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
        InvalidParameterValueException: The input does not match the requirements (values, order of values).
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection")

    if check_parameter_length(bands, limits) is False:
        raise NonMatchingParameterLengthsException("Invalid limit length.")

    for item in limits:
        if item.count(None) == len(item):
            raise InvalidParameterValueException(f"Limit values all None: {item}.")

        if not check_minmax_position(item):
            raise InvalidParameterValueException(f"Invalid min-max values provided: {item}.")

    expanded_args = expand_and_zip(bands, limits)
    limits = [element[1] for element in expanded_args]

    out_settings = {}

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        inital_dtype = band_array.dtype

        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = nodata_to_nan(band_array, nodata_value=nodata)

        band_array = _clip_transform(band_array, limits=limits[i])

        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_int(band_array, scalar=nodata, initial_dtype=inital_dtype)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "limit_lower": cast_scalar_to_int(limits[i][0]),
            "limit_upper": cast_scalar_to_int(limits[i][1]),
            "nodata": cast_scalar_to_int(nodata),
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

8.3 Linear

```
min_max_scaling(raster, bands=None, new_range=[(0, 1)], nodata=None)
```

Normalize data based on a specified new range.

Uses the provided new minimum and maximum to transform data into the new interval. Takes one nodata value that will be ignored in calculations.

If no band/column selection specified, all bands/columns will be used. The new_range can be set for each band individually. If a parameter contains only 1 entry, it will be applied for all bands.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`new_range`	`Sequence[Tuple[Number, Number]]`	The new interval data will be transformed into. First value corresponds to min, second to max.	`[(0, 1)]`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.
`InvalidParameterValueException`	The input does not match the requirements (values, order of values).

Source code in `eis_toolkit/transformations/linear.py`

```python
@beartype
def min_max_scaling(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    bands: Optional[Sequence[int]] = None,
    new_range: Sequence[Tuple[Number, Number]] = [(0, 1)],
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Normalize data based on a specified new range.

    Uses the provided new minimum and maximum to transform data into the new interval.
    Takes one nodata value that will be ignored in calculations.

    If no band/column selection specified, all bands/columns will be used.
    The new_range can be set for each band individually.
    If a parameter contains only 1 entry, it will be applied for all bands.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        new_range: The new interval data will be transformed into. First value corresponds to min, second to max.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
        InvalidParameterValueException: The input does not match the requirements (values, order of values).
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection")

    if check_parameter_length(bands, new_range) is False:
        raise NonMatchingParameterLengthsException("Invalid new_range length")

    for item in new_range:
        if not check_minmax_position(item):
            raise InvalidParameterValueException(f"Invalid min-max values provided: {item}")

    expanded_args = expand_and_zip(bands, new_range)
    new_range = [element[1] for element in expanded_args]

    out_settings = {}
    out_decimals = set_max_precision()

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = replace_values(band_array, values_to_replace=[nodata, np.inf], replace_value=np.nan)

        band_array = _min_max_scaling(band_array.astype(np.float64), new_range=new_range[i])

        band_array = truncate_decimal_places(band_array, decimal_places=out_decimals)
        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_float(band_array, scalar=nodata, cast_float=True)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "scaled_min": new_range[i][0],
            "scaled_max": new_range[i][1],
            "nodata": nodata,
            "decimal_places": out_decimals,
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

`z_score_normalization(raster, bands=None, nodata=None)`

Normalize data based on mean and standard deviation.

Results will have a mean = 0 and standard deviation = 1. Takes one nodata value that will be ignored in calculations.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.

Source code in `eis_toolkit/transformations/linear.py`

```python
@beartype
def z_score_normalization(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    bands: Optional[Sequence[int]] = None,
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Normalize data based on mean and standard deviation.

    Results will have a mean = 0 and standard deviation = 1.
    Takes one nodata value that will be ignored in calculations.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection.")

    out_settings = {}
    out_decimals = set_max_precision()

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = replace_values(band_array, values_to_replace=[nodata, np.inf], replace_value=np.nan)

        band_array, mean_array, sd_array = _z_score_normalization(band_array.astype(np.float64))

        band_array = truncate_decimal_places(band_array, decimal_places=out_decimals)
        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_float(band_array, scalar=nodata, cast_float=True)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "original_mean": truncate_decimal_places(mean_array, decimal_places=out_decimals),
            "original_sd": truncate_decimal_places(sd_array, decimal_places=out_decimals),
            "nodata": nodata,
            "decimal_places": out_decimals,
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

8.4 Logarithmic

```
log_transform(raster, bands=None, log_transform=['log2'], nodata=None)
```

Perform a logarithmic transformation on the provided data.

Takes one nodata value that will be ignored in calculations. Negative values will not be considered for transformation and replaced by the specific nodata value.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands. The log_transform can be set for each band individually.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`log_transform`	`Sequence[str]`	The base for logarithmic transformation. Valid values 'ln', 'log2' and 'log10'.	`['log2']`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands
`InvalidParameterValueException`	The input does not match the requirements (values, order of values)

Source code in `eis_toolkit/transformations/logarithmic.py`

```python
@beartype
def log_transform(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    bands: Optional[Sequence[int]] = None,
    log_transform: Sequence[str] = ["log2"],
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Perform a logarithmic transformation on the provided data.

    Takes one nodata value that will be ignored in calculations.
    Negative values will not be considered for transformation and replaced by the specific nodata value.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.
    The log_transform can be set for each band individually.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        log_transform: The base for logarithmic transformation. Valid values 'ln', 'log2' and 'log10'.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands
        InvalidParameterValueException: The input does not match the requirements (values, order of values)
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection")

    if check_parameter_length(bands, log_transform) is False:
        raise NonMatchingParameterLengthsException("Invalid length for log-base values.")

    for item in log_transform:
        if not (item == "ln" or item == "log2" or item == "log10"):
            raise InvalidParameterValueException(f"Invalid method: {item}.")

    expanded_args = expand_and_zip(bands, log_transform)
    log_transform = [element[1] for element in expanded_args]

    out_settings = {}
    out_decimals = set_max_precision()

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = replace_values(band_array, values_to_replace=[nodata, np.inf], replace_value=np.nan)
        band_array[band_array <= 0] = np.nan

        if log_transform[i] == "ln":
            band_array = _log_transform_ln(band_array.astype(np.float64))
        elif log_transform[i] == "log2":
            band_array = _log_transform_log2(band_array.astype(np.float64))
        elif log_transform[i] == "log10":
            band_array = _log_transform_log10(band_array.astype(np.float64))

        band_array = truncate_decimal_places(band_array, decimal_places=out_decimals)
        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_float(band_array, scalar=nodata, cast_float=True)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "log_transform": log_transform[i],
            "nodata": nodata,
            "decimal_places": out_decimals,
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

8.5 Sigmoid

`sigmoid_transform(raster, bands=None, bounds=[(0, 1)], slope=[1], center=True, nodata=None)`

Transform data into a sigmoid-shape based on a specified new range.

Uses the provided new minimum and maximum, shift and slope parameters to transform the data. Takes one nodata value that will be ignored in calculations.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands. The bounds and slope values can be set for each band individually.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`bounds`	`Sequence[Tuple[Number, Number]]`	Boundaries for the calculation of the sigmoid function (lower, upper).	`[(0, 1)]`
`slope`	`Sequence[Number]`	Value which modifies the slope of the resulting sigmoid-curve.	`[1]`
`center`	`bool`	Center array values around mean = 0 before sigmoid transformation.	`True`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.
`InvalidParameterValueException`	The input does not match the requirements (values, order of values)

Source code in `eis_toolkit/transformations/sigmoid.py`

```python
@beartype
def sigmoid_transform(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    bands: Optional[Sequence[int]] = None,
    bounds: Sequence[Tuple[Number, Number]] = [(0, 1)],
    slope: Sequence[Number] = [1],
    center: bool = True,
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Transform data into a sigmoid-shape based on a specified new range.

    Uses the provided new minimum and maximum, shift and slope parameters to transform the data.
    Takes one nodata value that will be ignored in calculations.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.
    The bounds and slope values can be set for each band individually.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        bounds: Boundaries for the calculation of the sigmoid function (lower, upper).
        slope: Value which modifies the slope of the resulting sigmoid-curve.
        center: Center array values around mean = 0 before sigmoid transformation.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
        InvalidParameterValueException: The input does not match the requirements (values, order of values)
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection")

    for parameter_name, parameter in [("bounds", bounds), ("slope", slope)]:
        if check_parameter_length(bands, parameter) is False:
            raise NonMatchingParameterLengthsException(f"Invalid length for {parameter_name}.")

    for item in bounds:
        if check_minmax_position(item) is False:
            raise InvalidParameterValueException(f"Invalid min-max values provided: {item}.")

    expanded_args = expand_and_zip(bands, bounds, slope)
    bounds = [element[1] for element in expanded_args]
    slope = [element[2] for element in expanded_args]

    out_settings = {}
    out_decimals = set_max_precision()

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = replace_values(band_array, values_to_replace=[nodata, np.inf], replace_value=np.nan)

        band_array = _sigmoid_transform(band_array.astype(np.float64), bounds=bounds[i], slope=slope[i], center=center)

        band_array = truncate_decimal_places(band_array, decimal_places=out_decimals)
        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_float(band_array, scalar=nodata, cast_float=True)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "bound_lower": truncate_decimal_places(bounds[i][0], decimal_places=out_decimals),
            "bound_upper": truncate_decimal_places(bounds[i][1], decimal_places=out_decimals),
            "slope": slope[i],
            "center": center,
            "nodata": nodata,
            "decimal_places": out_decimals,
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})

    return out_array, out_meta, out_settings
```

8.6 Winsorize

`winsorize(raster, percentiles, bands=None, inside=False, nodata=None)`

Winsorize data based on specified percentile values.

Takes one nodata value that will be ignored in calculations. Replaces values between [minimum, lower percentile] and [upper percentile, maximum] if provided. Works both one-sided and two-sided but raises error if no percentile values provided.

Percentiles are symmetrical, i.e. percentile_lower = 10 corresponds to the interval [min, 10%]. And percentile_upper = 10 corresponds to the intervall [90%, max]. I.e. percentile_lower = 0 refers to the minimum and percentile_upper = 0 to the data maximum.

Calculation of percentiles is ambiguous. Users can choose whether to use the value for replacement from inside or outside of the respective interval. Example: Given the np.array[5 10 12 15 20 24 27 30 35] and percentiles(10, 10), the calculated percentiles are (5, 35) for inside and (10, 30) for outside. This results in [5 10 12 15 20 24 27 30 35] and [10 10 12 15 20 24 27 30 30], respectively.

If no band/column selection specified, all bands/columns will be used. If a parameter contains only 1 entry, it will be applied for all bands. The percentiles can be set for each band individually, but inside parameter is same for all bands.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Data object to be transformed.	required
`bands`	`Optional[Sequence[int]]`	Selection of bands to be transformed.	`None`
`percentiles`	`Sequence[Tuple[Optional[Number], Optional[Number]]]`	Lower and upper percentile values (lower, upper) between [0, 100].	required
`inside`	`bool`	Whether to use the value for replacement from the left or right of the calculated percentile.	`False`
`nodata`	`Optional[Number]`	Nodata value to be considered.	`None`

Returns:

Name	Type	Description
`out_array`	`ndarray`	The transformed data.
`out_meta`	`dict`	Updated metadata.
`out_settings`	`dict`	Log of input settings and calculated statistics if available.

Raises:

Type	Description
`InvalidRasterBandException`	The input contains invalid band numbers.
`NonMatchingParameterLengthsException`	The input does not match the number of selected bands.
`InvalidParameterValueException`	The input does not match the requirements (values, order of values)

Source code in `eis_toolkit/transformations/winsorize.py`

```
@beartype
def winsorize(  # type: ignore[no-any-unimported]
    raster: rasterio.io.DatasetReader,
    percentiles: Sequence[Tuple[Optional[Number], Optional[Number]]],
    bands: Optional[Sequence[int]] = None,
    inside: bool = False,
    nodata: Optional[Number] = None,
) -> Tuple[np.ndarray, dict, dict]:
    """
    Winsorize data based on specified percentile values.

    Takes one nodata value that will be ignored in calculations.
    Replaces values between [minimum, lower percentile] and [upper percentile, maximum] if provided.
    Works both one-sided and two-sided but raises error if no percentile values provided.

    Percentiles are symmetrical, i.e. percentile_lower = 10 corresponds to the interval [min, 10%].
    And percentile_upper = 10 corresponds to the intervall [90%, max].
    I.e. percentile_lower = 0 refers to the minimum and percentile_upper = 0 to the data maximum.

    Calculation of percentiles is ambiguous. Users can choose whether to use the value
    for replacement from inside or outside of the respective interval. Example:
    Given the np.array[5 10 12 15 20 24 27 30 35] and percentiles(10, 10), the calculated
    percentiles are (5, 35) for inside and (10, 30) for outside.
    This results in [5 10 12 15 20 24 27 30 35] and [10 10 12 15 20 24 27 30 30], respectively.

    If no band/column selection specified, all bands/columns will be used.
    If a parameter contains only 1 entry, it will be applied for all bands.
    The percentiles can be set for each band individually, but inside parameter is same for all bands.

    Args:
        raster: Data object to be transformed.
        bands: Selection of bands to be transformed.
        percentiles: Lower and upper percentile values (lower, upper) between [0, 100].
        inside: Whether to use the value for replacement from the left or right of the calculated percentile.
        nodata: Nodata value to be considered.

    Returns:
        out_array: The transformed data.
        out_meta: Updated metadata.
        out_settings: Log of input settings and calculated statistics if available.

    Raises:
        InvalidRasterBandException: The input contains invalid band numbers.
        NonMatchingParameterLengthsException: The input does not match the number of selected bands.
        InvalidParameterValueException: The input does not match the requirements (values, order of values)
    """
    bands = list(range(1, raster.count + 1)) if bands is None else bands
    nodata = raster.nodata if nodata is None else nodata

    if check_raster_bands(raster, bands) is False:
        raise InvalidRasterBandException("Invalid band selection")

    if check_parameter_length(bands, percentiles) is False:
        raise NonMatchingParameterLengthsException("Invalid length for percentiles.")

    for item in percentiles:
        if item.count(None) == len(item):
            raise InvalidParameterValueException(f"Percentile values all None: {item}.")

        if None not in item and sum(item) >= 100:
            raise InvalidParameterValueException(f"Sum >= 100: {item}.")

        if item[0] is not None and not (0 < item[0] < 100):
            raise InvalidParameterValueException(f"Invalid lower percentile value: {item}.")

        if item[1] is not None and not (0 < item[1] < 100):
            raise InvalidParameterValueException(f"Invalid upper percentile value: {item}.")

    expanded_args = expand_and_zip(bands, percentiles)
    percentiles = [element[1] for element in expanded_args]

    out_settings = {}

    for i in range(0, len(bands)):
        band_array = raster.read(bands[i])
        inital_dtype = band_array.dtype

        band_array = cast_array_to_float(band_array, cast_int=True)
        band_array = nodata_to_nan(band_array, nodata_value=nodata)

        band_array, calculated_lower, calculated_upper = _winsorize(
            band_array, percentiles=percentiles[i], inside=inside
        )

        band_array = nan_to_nodata(band_array, nodata_value=nodata)
        band_array = cast_array_to_int(band_array, scalar=nodata, initial_dtype=inital_dtype)

        band_array = np.expand_dims(band_array, axis=0)

        if i == 0:
            out_array = band_array.copy()
        else:
            out_array = np.vstack((out_array, band_array))

        current_transform = f"transformation {i + 1}"
        current_settings = {
            "band_origin": bands[i],
            "percentile_lower": cast_scalar_to_int(percentiles[i][0]),
            "percentile_upper": cast_scalar_to_int(percentiles[i][1]),
            "calculated_lower": cast_scalar_to_int(calculated_lower),
            "calculated_upper": cast_scalar_to_int(calculated_upper),
            "nodata": cast_scalar_to_int(nodata),
        }

        out_settings[current_transform] = current_settings

    out_meta = raster.meta.copy()
    out_meta.update({"count": len(bands), "nodata": nodata, "dtype": out_array.dtype.name})
```

```
    return out_array, out_meta, out_settings
```

9. Validation

9.1 Calculate AUC

`calculate_auc(x_values, y_values)`

Calculate area under curve (AUC).

Calculates AUC for curve. X-axis should be either proportion of area ore false positive rate. Y-axis should be always true positive rate. AUC is calculated with sklearn.metrics.auc which uses trapezoidal rule for calculation.

Parameters:

Name	Type	Description	Default
`x_values`	`ndarray`	Either proportion of area or false positive rate values.	required
`y_values`	`ndarray`	True positive rate values.	required

Returns:

Type	Description
`float`	The area under curve.

Raises:

Type	Description
`InvalidParameterValueException`	x_values or y_values are out of bounds.

Source code in `eis_toolkit/validation/calculate_auc.py`

```
@beartype
def calculate_auc(x_values: np.ndarray, y_values: np.ndarray) -> float:
    """Calculate area under curve (AUC).

    Calculates AUC for curve. X-axis should be either proportion of area ore false positive rate. Y-axis should be
    always true positive rate. AUC is calculated with sklearn.metrics.auc which uses trapezoidal rule for calculation.

    Args:
        x_values: Either proportion of area or false positive rate values.
        y_values: True positive rate values.

    Returns:
        The area under curve.

    Raises:
        InvalidParameterValueException: x_values or y_values are out of bounds.
    """
    if x_values.max() > 1 or x_values.min() < 0:
        raise InvalidParameterValueException("x_values should be within range 0-1")

    if y_values.max() > 1 or y_values.min() < 0:
        raise InvalidParameterValueException("y_values should be within range 0-1")

    auc_value = _calculate_auc(x_values=x_values, y_values=y_values)
    return auc_value
```

9.2 Calculate base metrics

```
calculate_base_metrics(raster, deposits, band=1, negatives=None)
```

Calculate true positive rate, proportion of area and false positive rate values for different thresholds.

Function calculates true positive rate, proportion of area and false positive rate values for different thresholds which are determined from inputted deposit locations and mineral prospectivity map. Note that calculation of false positive rate is optional and is only done if negative point locations are provided.

Parameters:

Name	Type	Description	Default
`raster`	`DatasetReader`	Mineral prospectivity map or evidence layer.	required
`deposits`	`GeoDataFrame`	Mineral deposit locations as points.	required
`band`	`int`	Band index of the mineral prospectivity map. Defaults to 1.	`1`
`negatives`	`Optional[GeoDataFrame]`	Negative locations as points.	`None`

Returns:

Type	Description
`DataFrame`	DataFrame containing true positive rate, proportion of area, threshold values and false positive rate (optional) values.

Raises:

Type	Description
`NonMatchingCrsException`	The raster and point data are not in the same CRS.
`NotApplicableGeometryTypeException`	The input geometries contain non-point features.

Source code in `eis_toolkit/validation/calculate_base_metrics.py`

```python
@beartype
def calculate_base_metrics(
    raster: rasterio.io.DatasetReader,
    deposits: geopandas.GeoDataFrame,
    band: int = 1,
    negatives: Optional[geopandas.GeoDataFrame] = None,
) -> pd.DataFrame:
    """Calculate true positive rate, proportion of area and false positive rate values for different thresholds.

    Function calculates true positive rate, proportion of area and false positive rate values for different thresholds
    which are determined from inputted deposit locations and mineral prospectivity map. Note that calculation of false
    positive rate is optional and is only done if negative point locations are provided.

    Args:
        raster: Mineral prospectivity map or evidence layer.
        deposits: Mineral deposit locations as points.
        band: Band index of the mineral prospectivity map. Defaults to 1.
        negatives: Negative locations as points.

    Returns:
        DataFrame containing true positive rate, proportion of area, threshold values and false positive
            rate (optional) values.

    Raises:
        NonMatchingCrsException: The raster and point data are not in the same CRS.
        NotApplicableGeometryTypeException: The input geometries contain non-point features.
    """
    if negatives is not None:
        geometries = pd.concat([deposits, negatives]).geometry
    else:
        geometries = deposits["geometry"]

    if not check_matching_crs(
        objects=[raster, geometries],
    ):
        raise NonMatchingCrsException("The raster and deposits are not in the same CRS.")

    if not check_geometry_types(
        geometries=geometries,
        allowed_types=["Point"],
    ):
        raise NotApplicableGeometryTypeException("The input geometries contain non-point features.")

    base_metrics = _calculate_base_metrics(raster=raster, deposits=deposits, band=band, negatives=negatives)

    return base_metrics
```

9.3 Get P-A plot intersection point

`get_pa_intersection(true_positive_rate_values, proportion_of_area_values, threshold_values)`

Calculate the intersection point for prediction rate and area curves in (P-A plot).

Threshold_values values act as x-axis for both curves. Prediction rate curve uses true positive rate for y-axis. Area curve uses inverted proportion of area as y-axis.

Parameters:

Name	Type	Description	Default
`true_positive_rate_values`	`ndarray`	True positive rate values, values should be within range 0-1.	required
`proportion_of_area_values`	`ndarray`	Proportion of area values, values should be within range 0-1.	required
`threshold_values`	`ndarray`	Threshold values that were used to calculate true positive rate and proportion of area.	required

Returns:

Type	Description
`Tuple[float, float]`	X and y coordinates of the intersection point.

Raises:

Type	Description
`InvalidParameterValueException`	true_positive_rate_values or proportion_of_area_values values are out of bounds.

Source code in `eis_toolkit/validation/get_pa_intersection.py`

```python
@beartype
def get_pa_intersection(
    true_positive_rate_values: np.ndarray, proportion_of_area_values: np.ndarray, threshold_values: np.ndarray
) -> Tuple[float, float]:
    """Calculate the intersection point for prediction rate and area curves in (P-A plot).

    Threshold_values values act as x-axis for both curves. Prediction rate curve uses true positive rate for y-axis.
    Area curve uses inverted proportion of area as y-axis.

    Args:
        true_positive_rate_values: True positive rate values, values should be within range 0-1.
        proportion_of_area_values: Proportion of area values, values should be within range 0-1.
        threshold_values: Threshold values that were used to calculate true positive rate and proportion of area.

    Returns:
        X and y coordinates of the intersection point.

    Raises:
        InvalidParameterValueException: true_positive_rate_values or proportion_of_area_values values are out of bounds.
    """
    if true_positive_rate_values.max() > 1 or true_positive_rate_values.min() < 0:
        raise InvalidParameterValueException("true_positive_rate_values values should be within range 0-1")

    if proportion_of_area_values.max() > 1 or proportion_of_area_values.min() < 0:
        raise InvalidParameterValueException("proportion_of_area_values values should be within range 0-1")

    intersection = _get_pa_intersection(
        true_positive_rate_values=true_positive_rate_values,
        proportion_of_area_values=proportion_of_area_values,
        threshold_values=threshold_values,
    )

    return intersection.x, intersection.y
```

9.4 Plot correlation matrix

`plot_correlation_matrix(matrix, annotate=True, cmap=None, plot_title=None, **kwargs)`

Create a Seaborn heatmap to visualize correlation matrix.

Parameters:

Name	Type	Description	Default
`matrix`	`DataFrame`	Correlation matrix as a DataFrame.	required
`annotate`	`bool`	If plot squares should display the correlation values. Defaults to True.	`True`
`cmap`	`Optional[ListedColormap]`	Colormap for plotting. Optional parameter. Defaults to None, in which case a default colormap is used.	`None`
`plot_title`	`Optional[str]`	Title of the plot. Optional parameter, defaults to none (no title).	`None`
`**kwargs`	`dict`	Additional parameters to pass to Seaborn and matplotlib.	`{}`

Returns:

Type	Description
`Axes`	Matplotlib axes object with the produced plot.

Raises:

Type	Description
`EmptyDataFrameException`	Input matrix is empty.

Source code in `eis_toolkit/validation/plot_correlation_matrix.py`

```python
def plot_correlation_matrix(
    matrix: pd.DataFrame,
    annotate: bool = True,
    cmap: Optional[matplotlib.colors.ListedColormap] = None,
    plot_title: Optional[str] = None,
    **kwargs: dict
) -> matplotlib.axes.Axes:
    """
    Create a Seaborn heatmap to visualize correlation matrix.

    Args:
        matrix: Correlation matrix as a DataFrame.
        annotate: If plot squares should display the correlation values. Defaults to True.
        cmap: Colormap for plotting. Optional parameter. Defaults to None, in which
            case a default colormap is used.
        plot_title: Title of the plot. Optional parameter, defaults to none (no title).
        **kwargs: Additional parameters to pass to Seaborn and matplotlib.

    Returns:
        Matplotlib axes object with the produced plot.

    Raises:
        EmptyDataFrameException: Input matrix is empty.
    """
    if matrix.empty:
        raise exceptions.EmptyDataFrameException("Input matrix DataFrame is empty.")

    # Mask for the upper triangle of the heatmap
    mask = np.triu(np.ones_like(matrix, dtype=bool))

    if cmap is None:
        # Generate a default diverging colormap
        cmap = sns.diverging_palette(230, 20, as_cmap=True)

    ax = sns.heatmap(
        matrix,
        mask=mask,
        cmap=cmap,
        vmax=0.3,
        center=0,
        square=True,
        linewidths=0.5,
        annot=annotate,
        cbar_kws={"shrink": 0.5},
        **kwargs
    )
    if plot_title is not None:
        ax.set_title(plot_title)

    return ax
```

9.5 Plot prediction-area (P-A) curves

```
plot_prediction_area_curves(true_positive_rate_values,
proportion_of_area_values, threshold_values)
```

Plot prediction-area (P-A) plot.

Plots prediction area plot that can be used to evaluate mineral prospectivity maps and evidential layers. See e.g., Yousefi and Carranza (2015).

Parameters:

Name	Type	Description	Default
`true_positive_rate_values`	`ndarray`	True positive rate values.	required
`proportion_of_area_values`	`ndarray`	Proportion of area values.	required
`threshold_values`	`ndarray`	Threshold values.	required

Returns:

Type	Description
`Figure`	P-A plot figure object.

Raises:

Type	Description
`InvalidParameterValueException`	true_positive_rate_values or proportion_of_area_values values are out of bounds.

References

Yousefi, Mahyar, and Emmanuel John M. Carranza. "Fuzzification of continuous-value spatial evidence for mineral prospectivity mapping." Computers & Geosciences 74 (2015): 97-109.

Source code in `eis_toolkit/validation/plot_prediction_area_curves.py`

```
@beartype
def plot_prediction_area_curves(
    true_positive_rate_values: np.ndarray, proportion_of_area_values: np.ndarray, threshold_values: np.ndarray
) -> matplotlib.figure.Figure:
    """Plot prediction-area (P-A) plot.

    Plots prediction area plot that can be used to evaluate mineral prospectivity maps and evidential layers. See e.g.,
    Yousefi and Carranza (2015).

    Args:
        true_positive_rate_values: True positive rate values.
        proportion_of_area_values: Proportion of area values.
        threshold_values: Threshold values.

    Returns:
        P-A plot figure object.

    Raises:
        InvalidParameterValueException: true_positive_rate_values or proportion_of_area_values values are out of bounds.

    References:
        Yousefi, Mahyar, and Emmanuel John M. Carranza. "Fuzzification of continuous-value spatial evidence for mineral
        prospectivity mapping." Computers & Geosciences 74 (2015): 97-109.
    """
    if true_positive_rate_values.max() > 1 or true_positive_rate_values.min() < 0:
        raise InvalidParameterValueException("true_positive_rate values should be within range 0-1")

    if proportion_of_area_values.max() > 1 or proportion_of_area_values.min() < 0:
        raise InvalidParameterValueException("proportion_of_area values should be within range 0-1")

    fig = _plot_prediction_area_curves(
        true_positive_rate_values=true_positive_rate_values,
        proportion_of_area_values=proportion_of_area_values,
        threshold_values=threshold_values,
    )
    return fig
```

9.6 Plot rate curve

```
plot_rate_curve(x_values, y_values, plot_type='success_rate')
```

Plot success rate, prediction rate or ROC curve.

Plot type depends on plot_type argument. Y-axis is always true positive rate, while x-axis can be either false positive rate (roc) or proportion of area (success and prediction rate) depending on plot type.

Parameters:

Name	Type	Description	Default
`x_values`	`ndarray`	False positive rate values or proportion of area values.	required
`y_values`	`ndarray`	True positive rate values.	required
`plot_type`	`str`	Plot type. Can be either: "success_rate", "prediction_rate" or "roc".	`'success_rate'`

Returns:

Type	Description
`Figure`	Success rate, prediction rate or ROC plot figure object.

Raises:

Type	Description
`InvalidParameterValueException`	Invalid plot type.
`InvalidParameterValueException`	x_values or y_values are out of bounds.

Source code in `eis_toolkit/validation/plot_rate_curve.py`

```python
@beartype
def plot_rate_curve(
    x_values: np.ndarray,
    y_values: np.ndarray,
    plot_type: str = "success_rate",
) -> matplotlib.figure.Figure:
    """Plot success rate, prediction rate or ROC curve.

    Plot type depends on plot_type argument. Y-axis is always true positive rate, while x-axis can be either false
    positive rate (roc) or proportion of area (success and prediction rate) depending on plot type.

    Args:
        x_values: False positive rate values or proportion of area values.
        y_values: True positive rate values.
        plot_type: Plot type. Can be either: "success_rate", "prediction_rate" or "roc".

    Returns:
        Success rate, prediction rate or ROC plot figure object.

    Raises:
        InvalidParameterValueException: Invalid plot type.
        InvalidParameterValueException: x_values or y_values are out of bounds.
    """
    if plot_type == "success_rate":
        label = "Success rate"
        xlab = "Proportion of area"
    elif plot_type == "prediction_rate":
        label = "Prediction rate"
        xlab = "Proportion of area"
    elif plot_type == "roc":
        label = "ROC"
        xlab = "False positive rate"
    else:
        raise InvalidParameterValueException("Invalid plot type")

    if x_values.max() > 1 or x_values.min() < 0:
        raise InvalidParameterValueException("x_values should be within range 0-1")

    if y_values.max() > 1 or y_values.min() < 0:
        raise InvalidParameterValueException("y_values should be within range 0-1")

    fig = _plot_rate_curve(x_values=x_values, y_values=y_values, label=label, xlab=xlab)

    return fig
```

10. Vector processing

10.1 Cell-Based Association

```
cell_based_association(cell_size, geodata, output_path, column=None,
subset_target_attribute_values=None, add_name=None, add_buffer=None)
```

Creation of CBA matrix.

Initializes a CBA matrix from a vector file. The mesh is calculated according to the geometries contained in this file and the size of cells. Allows to add multiple vector data to the matrix, based on targeted shapes and/or attributes.

Parameters:

Name	Type	Description	Default
`cell_size`	`int`	Size of the cells.	required
`geodata`	`List[GeoDataFrame]`	GeoDataFrame to create the CBA matrix. Additional GeoDataFrame(s) can be imputed to add to the CBA matrix.	required
`output_path`	`str`	Name of the saved .tif file.	required
`column`	`Optional[List[str]]`	Name of the column of interest. If no attribute is specified, then an artificial attribute is created representing the presence or absence of the geometries of this file for each cell of the CBA grid. A categorical attribute will generate as many columns (binary) in the CBA matrix than values considered of interest (dummification). See parameter . Additional column(s) can be imputed for each added GeoDataFrame(s).	`None`
`subset_target_attribute_values`	`Optional[List[Union[None, list, str]]]`	List of values of interest of the target attribute, in case a categorical target attribute has been specified. Allows to filter a subset of relevant values. Additional values can be imputed for each added GeoDataFrame(s).	`None`
`add_name`	`Optional[List[Union[str, None]]]`	Name of the column(s) to add to the matrix.	`None`
`add_buffer`	`Optional[List[Union[Number, bool]]]`	Allow the use of a buffer around shapes before the intersection with CBA cells for the added	`None`

Name	Type	Description	Default
		GeoDataFrame(s). Minimize border effects or allow increasing positive samples (i.e. cells with mineralization). The size of the buffer is computed using the CRS (if projected CRS in meters: value in meters).	

Returns:

Type	Description
`GeoDataFrame`	CBA matrix is created.

Source code in `eis_toolkit/vector_processing/cell_based_association.py`

```python
@beartype
def cell_based_association(
    cell_size: int,
    geodata: List[gpd.GeoDataFrame],
    output_path: str,
    column: Optional[List[str]] = None,
    subset_target_attribute_values: Optional[List[Union[None, list, str]]] = None,
    add_name: Optional[List[Union[str, None]]] = None,
    add_buffer: Optional[List[Union[Number, bool]]] = None,
) -> gpd.GeoDataFrame:
    """Creation of CBA matrix.

    Initializes a CBA matrix from a vector file. The mesh is calculated
    according to the geometries contained in this file and the size of cells.
    Allows to add multiple vector data to the matrix, based on targeted shapes
    and/or attributes.

    Args:
        cell_size: Size of the cells.
        geodata: GeoDataFrame to create the CBA matrix. Additional
            GeoDataFrame(s) can be imputed to add to the CBA matrix.
        output_path: Name of the saved .tif file.
        column: Name of the column of interest. If no attribute is specified,
            then an artificial attribute is created representing the presence
            or absence of the geometries of this file for each cell of the CBA
            grid. A categorical attribute will generate as many columns (binary)
            in the CBA matrix than values considered of interest (dummification).
            See parameter <subset_target_attribute_values>. Additional
            column(s) can be imputed for each added GeoDataFrame(s).
        subset_target_attribute_values: List of values of interest of the
            target attribute, in case a categorical target attribute has been
            specified. Allows to filter a subset of relevant values. Additional
            values can be imputed for each added GeoDataFrame(s).
        add_name: Name of the column(s) to add to the matrix.
        add_buffer: Allow the use of a buffer around shapes before the
            intersection with CBA cells for the added GeoDataFrame(s). Minimize
            border effects or allow increasing positive samples (i.e. cells
            with mineralization). The size of the buffer is computed using the
            CRS (if projected CRS in meters: value in meters).

    Returns:
        CBA matrix is created.
    """

    # Swapping None to list values
    if column is None:
        column = [""]
    if add_buffer is None:
        add_buffer = [False]

    # Consistency checks on input data
    for frame in geodata:
        if frame.empty:
            raise exceptions.EmptyDataFrameException("The input GeoDataFrame is empty.")

    if cell_size <= 0:
        raise exceptions.InvalidParameterValueException("Expected cell size to be positive and non-zero.")

    add_buffer = [False if x == 0 else x for x in add_buffer]
    if any(num < 0 for num in add_buffer):
        raise exceptions.InvalidParameterValueException("Expected buffer value to be positive, null or False.")

    for i, name in enumerate(column):
        if column[i] == "":
            if subset_target_attribute_values[i] is not None:
                raise exceptions.InvalidParameterValueException("Can't use subset of values if no column is targeted.")
        elif column[i] not in geodata[i]:
            raise exceptions.InvalidColumnException("Targeted column not found in the GeoDataFrame.")

    for i, subset in enumerate(subset_target_attribute_values):
        if subset is not None:
            for value in subset:
                if value not in geodata[i][column[i]].unique():
                    raise exceptions.InvalidParameterValueException(
                        "Subset of value(s) not found in the targeted column."
                    )

    # Computation
    for i, data in enumerate(geodata):
        if i == 0:
            # Initialization of the CBA matrix
            grid, cba = _init_from_vector_data(cell_size, geodata[0], column[0], subset_target_attribute_values[0])
        else:
            # If necessary, adding data to matrix
            cba = _add_layer(
                cba,
                grid,
                geodata[i],
                column[i],
                subset_target_attribute_values[i],
                add_name[i - 1],
                add_buffer[i - 1],
            )

    # Export
    _to_raster(cba, output_path)

    return cba
```

10.2 Distance computation

`distance_computation(raster_profile, geometries)`

Calculate distance from raster cell to nearest geometry.

Parameters:

Name	Type	Description	Default
`raster_profile`	`Union[Profile, dict]`	The raster profile of the raster in which the distances to the nearest geometry are determined.	required
`geometries`	`GeoDataFrame`	The geometries to determine distance to.	required

Returns:

Type	Description
`ndarray`	A 2D numpy array with the distances computed.

Source code in `eis_toolkit/vector_processing/distance_computation.py`

```
@beartype
def distance_computation(raster_profile: Union[profiles.Profile, dict], geometries: gpd.GeoDataFrame) -> np.ndarray:
    """Calculate distance from raster cell to nearest geometry.

    Args:
        raster_profile: The raster profile of the raster in which the distances
            to the nearest geometry are determined.
        geometries: The geometries to determine distance to.

    Returns:
        A 2D numpy array with the distances computed.

    """
    if raster_profile.get("crs") != geometries.crs:
        raise exceptions.NonMatchingCrsException("Expected coordinate systems to match between raster and geometries. ")
    if geometries.shape[0] == 0:
        raise exceptions.EmptyDataFrameException("Expected GeoDataFrame to not be empty.")

    raster_width = raster_profile.get("width")
    raster_height = raster_profile.get("height")

    if not isinstance(raster_width, int) or not isinstance(raster_height, int):
        raise exceptions.InvalidParameterValueException(
            f"Expected raster_profile to contain integer width and height. {raster_profile}"
        )

    raster_transform = raster_profile.get("transform")

    if not isinstance(raster_transform, transform.Affine):
        raise exceptions.InvalidParameterValueException(
            f"Expected raster_profile to contain an affine transformation. {raster_profile}"
        )

    return _distance_computation(
        raster_width=raster_width, raster_height=raster_height, raster_transform=raster_transform, geometries=geometries
    )
```

10.3 IDW

```
idw(geodataframe, target_column, resolution, extent=None, power=2)
```

Calculate inverse distance weighted (IDW) interpolation.

Parameters:

Name	Type	Description	Default
`geodataframe`	`GeoDataFrame`	The vector dataframe to be interpolated.	required
`target_column`	`str`	The column name with values for each geometry.	required
`resolution`	`Tuple[Number, Number]`	The resolution i.e. cell size of the output raster as (pixel_size_x, pixel_size_y).	required
`extent`	`Optional[Tuple[Number, Number, Number, Number]]`	The extent of the output raster as (x_min, x_max, y_min, y_max). If None, calculate extent from the input vector data.	`None`
`power`	`Number`	The value for determining the rate at which the weights decrease. As power increases, the weights for distant points decrease rapidly. Defaults to 2.	`2`

Returns:

Type	Description
`Tuple[ndarray, dict]`	Rasterized vector data and metadata.

Raises:

Type	Description
`EmptyDataFrameException`	The input GeoDataFrame is empty.
`InvalidParameterValueException`	Invalid resolution or target_column.

Source code in `eis_toolkit/vector_processing/idw_interpolation.py`

```python
@beartype
def idw(
    geodataframe: gpd.GeoDataFrame,
    target_column: str,
    resolution: Tuple[Number, Number],
    extent: Optional[Tuple[Number, Number, Number, Number]] = None,
    power: Number = 2,
) -> Tuple[np.ndarray, dict]:
    """Calculate inverse distance weighted (IDW) interpolation.

    Args:
        geodataframe: The vector dataframe to be interpolated.
        target_column: The column name with values for each geometry.
        resolution: The resolution i.e. cell size of the output raster as (pixel_size_x, pixel_size_y).
        extent: The extent of the output raster as (x_min, x_max, y_min, y_max).
            If None, calculate extent from the input vector data.
        power: The value for determining the rate at which the weights decrease.
            As power increases, the weights for distant points decrease rapidly.
            Defaults to 2.

    Returns:
        Rasterized vector data and metadata.

    Raises:
        EmptyDataFrameException: The input GeoDataFrame is empty.
        InvalidParameterValueException: Invalid resolution or target_column.
    """

    if geodataframe.shape[0] == 0:
        raise EmptyDataFrameException("Expected geodataframe to contain geometries.")

    if target_column not in geodataframe.columns:
        raise InvalidParameterValueException(
            f"Expected target_column ({target_column}) to be contained in geodataframe columns."
        )

    if resolution[0] <= 0 or resolution[1] <= 0:
        raise InvalidParameterValueException("Expected height and width greater than zero.")

    interpolated_values, out_meta = _idw_interpolation(geodataframe, target_column, resolution, power, extent)

    return interpolated_values, out_meta
```

10.4 Kriging interpolation

```
kriging(data, target_column, resolution, extent=None, variogram_model='linear',
coordinates_type='geographic', method='ordinary')
```

Perform Kriging interpolation on the input data.

Parameters:

Name	Type	Description	Default
`data`	`GeoDataFrame`	GeoDataFrame containing the input data.	required
`target_column`	`str`	The column name with values for each geometry.	required
`resolution`	`Tuple[Number, Number]`	The resolution i.e. cell size of the output raster as (pixel_size_x, pixel_size_y).	required
`extent`	`Optional[Tuple[Number, Number, Number, Number]]`	The extent of the output raster as (x_min, x_max, y_min, y_max). If None, calculate extent from the input vector data.	`None`
`variogram_model`	`Literal[linear, power, gaussian, spherical, exponential]`	Variogram model to be used. Either 'linear', 'power', 'gaussian', 'spherical' or 'exponential'. Defaults to 'linear'.	`'linear'`
`coordinates_type`	`Literal[euclidean, geographic]`	Determines are coordinates on a plane ('euclidean') or a sphere ('geographic'). Used only in ordinary kriging. Defaults to 'geographic'.	`'geographic'`
`method`	`Literal[ordinary, universal]`	Ordinary or universal kriging. Defaults to 'ordinary'.	`'ordinary'`

Returns:

Type	Description
`Tuple[ndarray, dict]`	Grid containing the interpolated values and metadata.

Raises:

Type	Description
`EmptyDataFrameException`	The input GeoDataFrame is empty.
`InvalidParameterValueException`	Target column name is invalid or resolution is not greater than zero.

Source code in `eis_toolkit/vector_processing/kriging_interpolation.py`

```python
@beartype
def kriging(
    data: gpd.GeoDataFrame,
    target_column: str,
    resolution: Tuple[Number, Number],
    extent: Optional[Tuple[Number, Number, Number, Number]] = None,
    variogram_model: Literal["linear", "power", "gaussian", "spherical", "exponential"] = "linear",
    coordinates_type: Literal["euclidean", "geographic"] = "geographic",
    method: Literal["ordinary", "universal"] = "ordinary",
) -> Tuple[np.ndarray, dict]:
    """
    Perform Kriging interpolation on the input data.

    Args:
        data: GeoDataFrame containing the input data.
        target_column: The column name with values for each geometry.
        resolution: The resolution i.e. cell size of the output raster as (pixel_size_x, pixel_size_y).
        extent: The extent of the output raster as (x_min, x_max, y_min, y_max).
            If None, calculate extent from the input vector data.
        variogram_model: Variogram model to be used.
            Either 'linear', 'power', 'gaussian', 'spherical' or 'exponential'. Defaults to 'linear'.
        coordinates_type: Determines are coordinates on a plane ('euclidean') or a sphere ('geographic').
            Used only in ordinary kriging. Defaults to 'geographic'.
        method: Ordinary or universal kriging. Defaults to 'ordinary'.

    Returns:
        Grid containing the interpolated values and metadata.

    Raises:
        EmptyDataFrameException: The input GeoDataFrame is empty.
        InvalidParameterValueException: Target column name is invalid or resolution is not greater than zero.
    """

    if data.empty:
        raise EmptyDataFrameException("The input GeoDataFrame is empty.")

    if target_column not in data.columns:
        raise InvalidParameterValueException(
            f"Expected target_column ({target_column}) to be contained in geodataframe columns."
        )

    if resolution[0] <= 0 or resolution[1] <= 0:
        raise InvalidParameterValueException("The resolution must be greater than zero.")

    data_interpolated, out_meta = _kriging(
        data, target_column, resolution, extent, variogram_model, coordinates_type, method
    )

    return data_interpolated, out_meta
```

10.5 Rasterize vector

```
rasterize_vector(geodataframe, resolution=None, value_column=None,
default_value=1.0, fill_value=0.0, base_raster_profile=None, buffer_value=None,
merge_strategy='replace')
```

Transform vector data into raster data.

Parameters:

Name	Type	Description	Default
`geodataframe`	`GeoDataFrame`	The vector dataframe to be rasterized.	required
`resolution`	`Optional[float]`	The resolution i.e. cell size of the output raster. Optional if base_raster_profile is given.	`None`
`value_column`	`Optional[str]`	The column name with values for each geometry. If None, then default_value is used for all geometries.	`None`
`default_value`	`float`	Default value burned into raster cells based on geometries.	`1.0`
`base_raster_profile`	`Optional[Union[Profile, dict]]`	Base raster profile to be used for determining the grid on which vectors are burned in. If None, the geometries and provided resolution value are used to compute grid.	`None`
`fill_value`	`float`	Value used outside the burned/ rasterized geometry cells.	`0.0`
`buffer_value`	`Optional[float]`	For adding a buffer around passed geometries before rasterization.	`None`
`merge_strategy`	`Literal[replace, add]`	How to handle overlapping geometries. "add" causes overlapping geometries to add together the values while "replace" does not. Adding them together is the basis for density computations where the density can be calculated by using a default value of 1.0 and the sum in each cell is the count of intersecting geometries.	`'replace'`

Returns:

Type	Description
`Tuple[ndarray, dict]`	Rasterized vector data and metadata.

Source code in `eis_toolkit/vector_processing/rasterize_vector.py`

```python
@beartype
def rasterize_vector(
    geodataframe: gpd.GeoDataFrame,
    resolution: Optional[float] = None,
    value_column: Optional[str] = None,
    default_value: float = 1.0,
    fill_value: float = 0.0,
    base_raster_profile: Optional[Union[profiles.Profile, dict]] = None,
    buffer_value: Optional[float] = None,
    merge_strategy: Literal["replace", "add"] = "replace",
) -> Tuple[np.ndarray, dict]:
    """Transform vector data into raster data.

    Args:
        geodataframe: The vector dataframe to be rasterized.
        resolution: The resolution i.e. cell size of the output raster.
            Optional if base_raster_profile is given.
        value_column: The column name with values for each geometry.
            If None, then default_value is used for all geometries.
        default_value: Default value burned into raster cells based on geometries.
        base_raster_profile: Base raster profile
            to be used for determining the grid on which vectors are
            burned in. If None, the geometries and provided resolution
            value are used to compute grid.
        fill_value: Value used outside the burned/rasterized geometry cells.
        buffer_value: For adding a buffer around passed
            geometries before rasterization.
        merge_strategy: How to handle overlapping geometries.
            "add" causes overlapping geometries to add together the
            values while "replace" does not. Adding them together is the
            basis for density computations where the density can be
            calculated by using a default value of 1.0 and the sum in
            each cell is the count of intersecting geometries.

    Returns:
        Rasterized vector data and metadata.
    """

    if geodataframe.shape[0] == 0:
        # Empty GeoDataFrame
        raise exceptions.EmptyDataFrameException("Expected geodataframe to contain geometries.")

    if resolution is None and base_raster_profile is None:
        raise exceptions.InvalidParameterValueException(
            "Expected either resolution or base_raster_profile to be given."
        )
    if resolution is not None and resolution <= 0:
        raise exceptions.NumericValueSignException(
            f"Expected a positive value resolution ({dict(resolution=resolution)})"
        )
    if value_column is not None and value_column not in geodataframe.columns:
        raise exceptions.InvalidParameterValueException(
            f"Expected value_column ({value_column}) to be contained in geodataframe columns."
        )
    if buffer_value is not None and buffer_value < 0:
        raise exceptions.NumericValueSignException(
            f"Expected a positive buffer_value ({dict(buffer_value=buffer_value)})"
        )

    if base_raster_profile is not None and not isinstance(base_raster_profile, (profiles.Profile, dict)):
        raise exceptions.InvalidParameterValueException(
            f"Expected base_raster_profile ({type(base_raster_profile)}) to be dict or rasterio.profiles.Profile."
        )

    if buffer_value is not None:
        geodataframe = geodataframe.copy()
        geodataframe["geometry"] = geodataframe["geometry"].apply(lambda geom: geom.buffer(buffer_value))

    return _rasterize_vector(
        geodataframe=geodataframe,
        value_column=value_column,
        default_value=default_value,
        fill_value=fill_value,
        base_raster_profile=base_raster_profile,
        resolution=resolution,
        merge_alg=getattr(MergeAlg, merge_strategy),
    )
```

10.6 Reproject vector

`reproject_vector(geodataframe, target_crs)`

Reprojects vector data to match given coordinate reference system (EPSG).

Parameters:

Name	Type	Description	Default
`geodataframe`	`GeoDataFrame`	The vector dataframe to be reprojected.	required
`target_crs`	`int`	Target CRS as an EPSG code.	required

Returns:

Type	Description
`GeoDataFrame`	Reprojected vector data.

Source code in `eis_toolkit/vector_processing/reproject_vector.py`

```
@beartype
def reproject_vector(geodataframe: geopandas.GeoDataFrame, target_crs: int) -> geopandas.GeoDataFrame:
    """Reprojects vector data to match given coordinate reference system (EPSG).

    Args:
        geodataframe: The vector dataframe to be reprojected.
        target_crs: Target CRS as an EPSG code.

    Returns:
        Reprojected vector data.
    """

    if geodataframe.crs.to_epsg() == target_crs:
        raise MatchingCrsException("Vector data is already in the target CRS.")

    reprojected_gdf = geodataframe.to_crs("epsg:" + str(target_crs))
    return reprojected_gdf
```

10.7 Vector density

```
vector_density(geodataframe, resolution=None, base_raster_profile=None,
buffer_value=None, statistic='density')
```

Compute density of geometries within raster.

Parameters:

Name	Type	Description	Default
`geodataframe`	`GeoDataFrame`	The dataframe with vectors of which density is computed.	required
`resolution`	`Optional[float]`	The resolution i.e. cell size of the output raster. Optional if base_raster_profile is given.	`None`
`base_raster_profile`	`Optional[Union[Profile, dict]]`	Base raster profile to be used for determining the grid on which vectors are burned in. If None, the geometries and provided resolution value are used to compute grid.	`None`
`buffer_value`	`Optional[float]`	For adding a buffer around passed geometries before computing density.	`None`

Returns:

Type	Description
`Tuple[ndarray, dict]`	Computed density of vector data and metadata.

Source code in `eis_toolkit/vector_processing/vector_density.py`

```python
@beartype
def vector_density(
    geodataframe: gpd.GeoDataFrame,
    resolution: Optional[float] = None,
    base_raster_profile: Optional[Union[profiles.Profile, dict]] = None,
    buffer_value: Optional[float] = None,
    statistic: Literal["density", "count"] = "density",
) -> Tuple[np.ndarray, dict]:
    """Compute density of geometries within raster.

    Args:
        geodataframe: The dataframe with vectors
            of which density is computed.
        resolution: The resolution i.e. cell size of the output raster.
            Optional if base_raster_profile is given.
        base_raster_profile: Base raster profile
            to be used for determining the grid on which vectors are
            burned in. If None, the geometries and provided resolution
            value are used to compute grid.
        buffer_value: For adding a buffer around passed
            geometries before computing density.

    Returns:
        Computed density of vector data and metadata.
    """
    out_raster_array, out_metadata = rasterize_vector(
        geodataframe=geodataframe,
        resolution=resolution,
        base_raster_profile=base_raster_profile,
        buffer_value=buffer_value,
        value_column=None,
        default_value=1.0,
        fill_value=0.0,
        merge_strategy="add",
    )
    max_count = np.max(out_raster_array)
    if statistic == "count" or np.isclose(max_count, 0.0):
        return out_raster_array, out_metadata
    else:
        return (out_raster_array / max_count), out_metadata
```