Geomancer Documentation

Release 1.1.0

Thinking Machines Data Science

Feb 23, 2021

General

Geomancer is a geospatial feature engineering library. It leverages geospatial data such as OpenStreetMap (OSM) alongside a data warehouse like BigQuery. You can use this to create, share, and iterate geospatial features for your downstream tasks (analysis, modelling, visualization, etc.).

- **Free Software**: MIT License
- **Github Repository**: https://github.com/thinkingmachines/geomancer

CHAPTER 1

Features

Geomancer is a geospatial feature engineering library. It allows you to query from a geospatial data warehouse in order to create features for downstream tasks (analysis, modelling, visualization, etc.). Its features include:

- Feature primitives for geospatial feature engineering
- Ability to switch out data warehouses
- Compilation and sharing your features

1.1 Feature Primitives

The basic building blocks in Geomancer are called Spells. These are SQL queries that were packaged in logical groups. Given a set of coordinates, you can obtain features such as the distance to the nearest point-of-interest (POIs), number of POIS within a certain range, and etc.

For example, we wish to obtain the distance to the nearest embassy given a sample of coordinates:

```
In [1]: # Load the sample_points as a dataframe
In [2]: from tests.conftest import sample_points
In [3]: df = sample_points
```

```
In  [4]: df.head()
Out [4]:

                                WKT  code
0  POINT (121.0042183 14.6749145)  2082
1  POINT (121.0052375 14.6767411)  2110
2     POINT (121.009712 14.68067)  2082
3  POINT (121.0093311 14.6799482)  2082
4  POINT (121.0073296 14.6783498)  2082
```

The geometries are encoded as a *str* inside a column named *WKT*. In addition, there is a *code* column that represents any arbitrary column in your data. What Geomancer will do is just add another column for your chosen feature while retaining the columns you originally have.

```
In [5]: from geomancer.spells import DistanceToNearest

In [6]: # Configure and cast the spell
In [7]: spell = DistanceToNearest("embassy",
                                  source_table="geospatial.ph_osm.gis_osm_pois_free_1",
                                  feature_name="dist_embassy")
In [8]: df_with_features = spell.cast(df, dburl="bigquery://geospatial")
```

It then returns a DataFrame with an added column, *dist_embassy*:

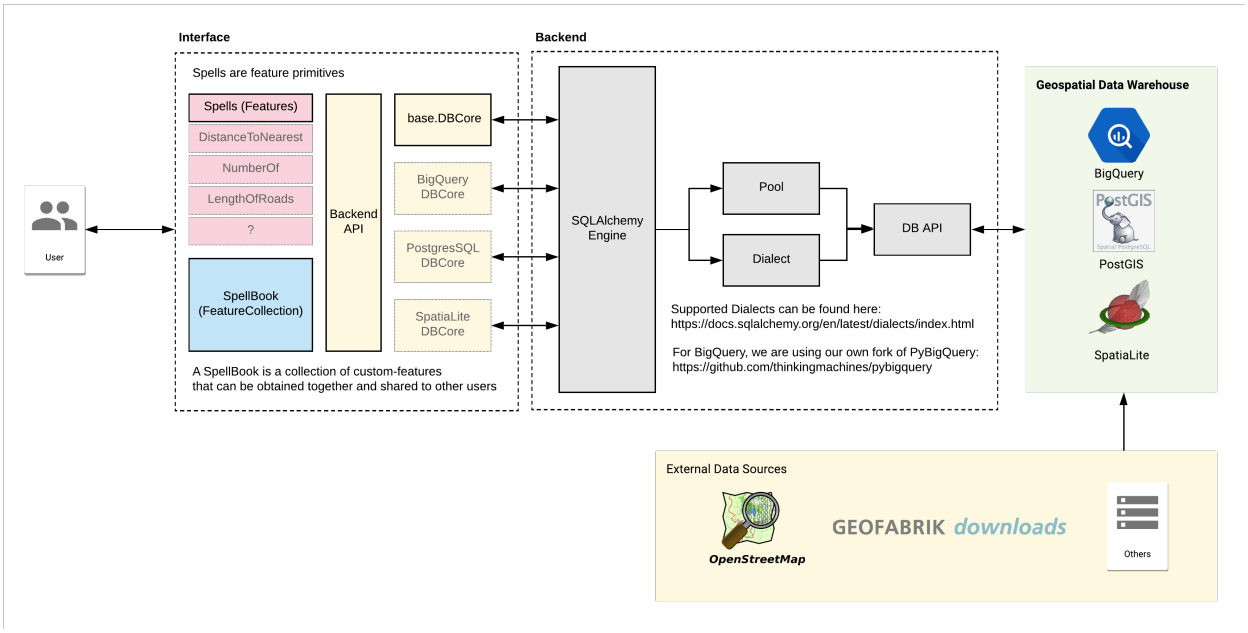
```
In  [9]: df_with_features.head()
Out [9]:
                                WKT  code  dist_embassy
0  POINT (121.0042183 14.6749145)  2082   4948.580211
1  POINT (121.0052375 14.6767411)  2110   5084.787270
2     POINT (121.009712 14.68067)  2082   5319.746371
3  POINT (121.0093311 14.6799482)  2082   5256.165257
4  POINT (121.0073296 14.6783498)  2082   5162.177598
```

1.2 Data Warehouse Flexibility

Geomancer is powered by a data warehouse backend for engineering features. It is then possible to compile features from different sources through this flexible API. So far, we've supported (and planning to support) the following database backends:

- Google BigQuery, an analytics data warehouse from the Google Cloud Platform
- PostGIS, a geospatial extension for PostGreSQL
- SpatiaLite, a geospatial extension for SQLite

Most of our examples harness the power of Open Data, particularly of Open Street Maps. For our geographical columns we depend on Geofabrik's OSM data.



Note: First you need to setup your data warehouse in order to accommodate Geomancer. For more instructions, please see the Setup instructions in this documentation.

1.3 Compile and share features

Once you've created a good set of features (or transformations), you can then compile them into a SpellBook and share it to others. For example, if I identified from my experiments that the number of supermarkets and distance to primary roads are good economic indicators, I can bind them together and share with other researchers to try on their own data.

```
from geomancer.spells import DistancetoNearest, NumberOf
from geomancer.spellbook import SpellBook

# Create a spellbook
spellbook = SpellBook(
          spells=[
              DistanceToNearest("primary",
                                dburl="bigquery://geospatial",
                                source_table="ph_osm.gis_osm_roads_free_1",
                                feature_name="dist_primary"),
              NumberOf("supermarket"
                       dburl="bigquery://geospatial",
                       source_table="geospatial.ph_osm.gis_osm_pois_free_1",
                       feature_name="num_supermarkets"),
          ])

# Export SpellBook into a file
spellbook.author = "Juan dela Cruz"
spellbook.description = "Good Features for Economic Indicators"
spellbook.to_json("features_dela_cruz.json")
```

You can then share this to other people so that they can cast it on their own datasets

```
from geomancer.spellbook import SpellBook
from tests.conftest import sample_points

spellbook = SpellBook.read_json("features_dela_cruz.json")
df = sample_points() # load your own data

# Cast someone's Spells into your own data
df_with_features = spellbook.cast(df)
```

CHAPTER 2

Setup

2.1 Installing the library

Geomancer can be installed using *pip*.

```
$ pip install geomancer
```

This will install **all** dependencies for every data warehouse we support. If you wish to do this for only a specific warehouse, then you can add an identifier.

```
$ pip install geomancer[bq] # For BigQuery
$ pip install geomancer[sqlite] # For SQLite
$ pip install geomancer[psql] # For PostgreSQL (*In Progress*)
```

Alternatively, you can also clone the repository then run *install*.

```
$ git clone https://github.com/thinkingmachines/geomancer.git
$ cd geomancer
$ python setup.py install
```

2.2 Setting-up your data warehouse

We highly-recommend using BigQuery as your data warehouse and Geofabrik's OSM catalog as your source of Points and Lines of interest.

1. First, **download a '.shp.zip' file for your Region-of-Interest (ROI)**. In this example we can choose the Philippines (.shp.zip)

Note: You can definitely choose multiple ROIs as long as you keep them in separate contexts. A BigQuery best practice is to put them in separate datasets. So if you will use Geomancer in US and in Japan, you should have two

datasets in your project (e.g., `us_osm` and `jp_osm`).

2. Then, **convert all your shape files into WKT** because BigQuery accepts WKT files. For example, let's convert `gis_osm_pois_free_1.shp` into WKT. You can use `ogr2ogr` for this kind of task:

```
$ ogr2ogr -f CSV gis_osm_pois_free_1.csv gis_osm_pois_free_1.shp -lco GEOMETRY=AS_WKT
```

You can do this for all shapefiles you currently have. At the end of this operation, you should have a CSV file (with a WKT) column for all files.

3. **Load your files into BigQuery**. Our best practice is to just use the CSV filename as the table name. Refer to the bq load instructions in the Google Cloud Platform documentation. In this example, we'll load all WKT files in a dataset called *ph_osm*

```
$ bq load --source_format=CSV    \
          --skip_leading_rows=1  \
          --autodetect           \
          ph_osm.gis_osm_pois_free_1 gis_osm_pois_free_1.csv
```

Geomancer assumes that your polygons are of the type `STRING`. Thus, we recommend passing the `--autodetect` option when loading to BigQuery.

And that's it! When casting a spell, you can then use `bigquery://project-name` for your `dburl` and *project-name.ph_osm.gis_osm_pois_free_1* as your `source_table`.

2.2.1 Using other data warehouses aside from BigQuery

You can use other data warehouses. This enables us to support SpatiaLite and PostGIS data warehouses. In order for them to interface with Geomancer, it should have the following characteristics:

- The column where geometries are stored should be of type `STRING` and named *WKT*
- It should support GIS functions (*ST_GeomFromText*, *ST_Distance*, *ST_Length*, etc.)

2.2.2 Using other source datasets aside from OSM

Do you have other geospatial data? Want to use Geomancer on them? It is possible, as long as they fulfill the following requirements:

- They should have a column of geometries
- There must be a unique identifier or a primary key for each row
- There should be a way of filtering them properly

In this way, it is then possible to query from other datasets. For example, the `DistanceToNearest` spell accepts an argument `on`. You can add a colon `:` to specify which column this spell will filter upon (default is `fclass`). For example:

```
>>> DistanceToNearest("embassy", **kwargs)    # Will filter embassy in fclass (default)
>>> DistanceToNearest("user_group:4G-users") # Will filter 4G-users in user_group
```

CHAPTER 3

Usage

All of the feature engineering functions in Geomancer are called "spells". For example, you want to get the distance to the nearest supermarket for each point.

```
from geomancer.spells import DistanceToNearest

# Load the dataset in a pandas dataframe
# df = load_dataset()

dist_spell = DistanceToNearest(
    "supermarket",
    source_table="ph_osm.gis_osm_pois_free_1",
    feature_name="dist_supermarket",
).cast(df, dburl="bigquery://project-name")
```

Compose multiple spells into a "spell book" which you can export as a JSON file.

```
from geomancer.spells import DistanceToNearest
from geomancer.spellbook import SpellBook

spellbook = SpellBook([
    DistanceToNearest(
        "supermarket",
        source_table="ph_osm.gis_osm_pois_free_1",
        feature_name="dist_supermarket",
        dburl="bigquery://project-name",
    ),
    DistanceToNearest(
        "embassy",
        source_table="ph_osm.gis_osm_pois_free_1",
        feature_name="dist_embassy",
        dburl="bigquery://project-name",
    ),
])
spellbook.to_json("dist_supermarket_and_embassy.json")
```

You can share the generated file so other people can re-use your feature extractions with their own datasets.

```
from geomancer.spellbook import SpellBook

# Load the dataset in a pandas dataframe
# df = load_dataset()

spellbook = SpellBook.read_json("dist_supermarket_and_embassy.json")
dist_supermarket_and_embassy = spellbook.cast(df)
```

CHAPTER 4

Changelog

4.1 v1.0.0 (2019-03-29)

First release on PyPI.

- **NEW**: Spells as feature-primitives (DistanceToNearest, NumberOf, LengthOf)
- **NEW**: Various database backends (BigQuery, SQLite)
- **NEW**: Ability to compile and share features using SpellBook

4.1.1 v1.0.1 (2019-04-09)

- **FIX**: Pin google-cloud dependencies to avoid import errors - #66, #69
- **DOC**: Flat-style for README badges! - #63
- **DOC**: Filter example in "on" parameter - #63

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/thinkingmachines/geomancer/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it. Those that are tagged with "first-timers-only" is suitable for those getting started in open-source software.

5.1.4 Write Documentation

Geomancer could always use more documentation, whether as part of the official Geomancer docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/thinkingmachines/geomancer/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *geomancer* for local development.

1. Fork the *geomancer* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/geomancer.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ cd geomancer/
$ make venv  # Creates a virtual environment
$ make dev  # Installs development requirements
```

   For windows users, you can do the following:

```
$ cd geomancer
$ python -m venv venv
$ venv\Scripts\activate
$ pip install pip-tools
$ pip install -r requirements.txt
$ pip install -r requirements-dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox. In addition, ensure that your code is formatted using black:

```
$ flake8 geomancer tests
$ black geomancer tests
$ pytest -m "not bqtest and not slow" -v
```

To get flake8, black, and tox, just pip install them into your virtualenv. If you wish, you can add pre-commit hooks for both flake8 and black to make all formatting easier.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, and above. Check https://cloud.drone.io/thinkingmachines/geomancer/ and make sure that the tests pass for all supported Python versions.

CHAPTER 6

Code of Conduct

6.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

6.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

6.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

6.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

6.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at hello@thinkingmachin.es. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

6.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq.

CHAPTER 7

Spells

Spells are primitives that you can use for engineering features. They are first instantiated, and then casted on a DataFrame of points.

7.1 geomancer.spells.base

Base class for all feature or spell implementations

In Geomancer, all feature transform primitives are of the class `Spell`. When defining your own feature primitive, simply create a class that inherits from `Spell`:

```
from geomancer.spells.base import Spell
class MyNewFeature(Spell):
    def __init__(self):
        super(MyNewFeature, self).__init__()
```

All methods must be implemented in order to not raise a `NotImplementedError`.

class geomancer.spells.base.**Spell**(*source_table*, *feature_name*, *source_id='osm_id'*, *dburl=None*, *options=None*)

Bases: abc.ABC

Base class for all feature/spell implementations

__init__(*source_table*, *feature_name*, *source_id='osm_id'*, *dburl=None*, *options=None*)

Spell constructor

Parameters

- **source_table** (*str*) – Table URI to run queries against.
- **feature_name** (*str*) – Column name for the output feature.
- **dburl** (*str, optional*) – Database url used to configure backend connection
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

cast(*target*, *dburl=None*, *column='WKT'*, *keep_index=False*, *features_only=False*, *pkey='__index_level_0__'*)
Apply the feature transform to an input `pandas.DataFrame`

Parameters

- **target** (`pandas.DataFrame` or str) – Object containing the points to compare upon. Can be a DataFrame or a database URL. By default, we will look into the `WKT` column. You can specify your own column by passing an argument to the `column` parameter.
- **dburl** (*str, optional*) – Database url used to configure backend connection
- **column** (*str, optional*) – Column to look the geometries from. The default is `WKT`
- **keep_index** (*boolean, optional*) – Include index in output dataframe
- **features_only** (*boolean, optional*) – Only return features as output dataframe. Automatically sets `keep_index` to `True`.
- **pkey** (*str, optional*) – The primary key column in the database. Default is __index_level_0__. This is useful if the table you're passing `cast` to is a database URL rather than a dataframe.

Returns Output dataframe with the features per given point

Return type `pandas.DataFrame`

extract_columns(*x*)
Extract column and filter from a string input

Parameters **x** (*str*) – The column and filter pair in the form `column:filter`

Returns The extracted column and filter pair

Return type (str, str)

get_core(*dburl*)
Instantiates an appropriate core based on given database url

Parameters **dburl** (*str*) – Database url used to configure backend connection

Returns **core** – DBCore instance to access DB-specific methods

Return type `geomancer.backend.cores.DBCore`

query(*source*, *target*, *core*, *column*, *pkey*)
Build the query used to extract features

Parameters

- **source** (`sqlalchemy.schema.Table`) – Source table to extract features from.
- **target** (`sqlalchemy.schema.Table`) – Target table to add features to.
- **core** (*geomancer.backend.cores.base.DBCore*) – DBCore instance to access DB-specific methods
- **column** (*string*) – Column to look the geometries from. The default is `WKT`
- **pkey** (*str, optional*) – The primary key column in the database. Default is __index_level_0__. This is useful if the table you're passing `cast` to is a database URL rather than a dataframe.

Returns The statement to query features with.

Return type `sqlalchemy.sql.expression.ClauseElement`

Raises `NotImplementedError` – This is an abstract method

7.2 geomancer.spells.distance_to_nearest

Spell DistanceToNearest obtains the distance to the nearest Point-of-Interest or geographic feature. Suppose you want to find the distance to the nearest embassy:

```
from geomancer.spells import DistanceToNearest
from tests.conftest import sample_points

# Load sample points
df = sample_points()

# Configure and cast the spell
spell = DistanceToNearest("embassy",
                          source_table="geospatial.ph_osm.gis_osm_pois_free_1",
                          feature_name="dist_embassy")

# Will create a new column, `dist_embassy` with the
# appropriate features
df_with_features = spell.cast(df, dburl="bigquery://geospatial")
```

class geomancer.spells.distance_to_nearest.**DistanceToNearest**(*on*, *within=10000*, ***kwargs*)

Bases: *geomancer.spells.base.Spell*

Obtain the distance to the nearest Point-of-Interest or geographic feature

__init__(*on*, *within=10000*, ***kwargs*)

Spell constructor

Parameters

- **on** (*str*) – Feature class to compare upon
- **within** (*float, optional*) – Look for values within a particular range. Its value is in meters, the default is `10,000` meters.
- **source_table** (*str*) – Table URI to run queries against.
- **feature_name** (*str*) – Column name for the output feature.
- **column** (*str, optional*) – Column to look the geometries from. The default is `WKT`
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

query(*source*, *target*, *core*, *column*, *pkey*)

Build the query used to extract features

Parameters

- **source** (`sqlalchemy.schema.Table`) – Source table to extract features from.
- **target** (`sqlalchemy.schema.Table`) – Target table to add features to.
- **core** (*geomancer.backend.cores.base.DBCore*) – DBCore instance to access DB-specific methods
- **column** (*string*) – Column to look the geometries from. The default is `WKT`
- **pkey** (*str, optional*) – The primary key column in the database. Default is __index_level_0__. This is useful if the table you're passing `cast` to is a database URL rather than a dataframe.

Returns The statement to query features with.

Return type `sqlalchemy.sql.expression.ClauseElement`

Raises `NotImplementedError` – This is an abstract method

7.3 geomancer.spells.number_of

Spell NumberOf obtains the number of Points-of-Interests or geographic features within a particular range. Suppose you want to find the number of supermarkets given a set of points

```
from geomancer.spells import NumberOf
from tests.conftest import sample_points

# Load sample points
df = sample_points()

# Configure and cast the spell
spell = NumberOf("supermarket",
                 source_table="geospatial.ph_osm.gis_osm_pois_free_1",
                 feature_name="num_supermarket")

# Will create a new column, `num_supermarket` with the
# appropriate features
df_with_features = spell.cast(df, dburl="bigquery://geospatial")
```

class geomancer.spells.number_of.**NumberOf**(*on*, *within=10000*, ***kwargs*)

Bases: *geomancer.spells.base.Spell*

Obtain the number of nearest Point-of-Interests or geographic features

__init__(*on*, *within=10000*, ***kwargs*)

Spell constructor

Parameters

- **on** (*str*) – Feature class to compare upon
- **within** (*float, optional*) – Look for values within a particular range. Its value is in meters, the default is `10,000` meters.
- **source_table** (*str*) – Table URI to run queries against.
- **feature_name** (*str*) – Column name for the output feature.
- **column** (*str, optional*) – Column to look the geometries from. The default is `WKT`
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

query(*source*, *target*, *core*, *column*, *pkey*)

Build the query used to extract features

Parameters

- **source** (`sqlalchemy.schema.Table`) – Source table to extract features from.
- **target** (`sqlalchemy.schema.Table`) – Target table to add features to.
- **core** (*geomancer.backend.cores.base.DBCore*) – DBCore instance to access DB-specific methods

- **column** (*string*) – Column to look the geometries from. The default is `WKT`
- **pkey** (*str, optional*) – The primary key column in the database. Default is __index_level_0__. This is useful if the table you're passing `cast` to is a database URL rather than a dataframe.

Returns The statement to query features with.

Return type `sqlalchemy.sql.expression.ClauseElement`

Raises `NotImplementedError` – This is an abstract method

7.4 geomancer.spells.length_of

Spell LengthOf obtains the length of all Lines-of-Interest with a certain radius. Suppose you want to find the length of residential roads given a set of points:

```
from geomancer.spells import LengthOf
from tests.conftest import sample_points

# Load sample points
df = sample_points()

# Configure and cast the spell
spell = LengthOf("residential",
                 source_table="geospatial.ph_osm.gis_osm_roads_free_1",
                 feature_name="len_residential")

# Will create a new column, `len_residential` with the
# appropriate features
df_with_features = spell.cast(df, dburl="bigquery://geospatial")
```

Warning: This spell currently doesn't work in BigQuery. In addition, the runtime for casting this spell is slow.

class geomancer.spells.length_of.**LengthOf**(*on*, *within=10000*, ***kwargs*)

Bases: *geomancer.spells.base.Spell*

Obtain the length of all Lines-of-Interest within a certain radius

__init__(*on*, *within=10000*, ***kwargs*)

Spell constructor

Parameters

- **on** (*str*) – Feature class to compare upon
- **within** (*float, optional*) – Look for values within a particular range. Its value is in meters, the default is `10,000` meters.
- **source_table** (*str*) – Table URI to run queries against.
- **feature_name** (*str*) – Column name for the output feature.
- **column** (*str, optional*) – Column to look the geometries from. The default is `WKT`
- **options** (*geomancer.backend.settings.Config*) – Specify configuration for interacting with the database backend. Default is a BigQuery Configuration

query(*source*, *target*, *core*, *column*, *pkey*)
Build the query used to extract features

Parameters

- **source** (sqlalchemy.schema.Table) – Source table to extract features from.
- **target** (sqlalchemy.schema.Table) – Target table to add features to.
- **core** (*geomancer.backend.cores.base.DBCore*) – DBCore instance to access DB-specific methods
- **column** (*string*) – Column to look the geometries from. The default is WKT
- **pkey** (*str, optional*) – The primary key column in the database. Default is __index_level_0__. This is useful if the table you're passing cast to is a database URL rather than a dataframe.

Returns The statement to query features with.

Return type sqlalchemy.sql.expression.ClauseElement

Raises NotImplementedError – This is an abstract method

CHAPTER 8

SpellBook

A `SpellBook` is a collection of spells that can be sequentially casted and merged in a single dataframe. This is useful if you have a feature-collection that you want to reuse or share to other people:

```
from geomancer.spells import DistanceToNearest, NumberOf
from geomancer.spellbook import SpellBook
from tests.conftest import sample_points


# Create a spellbook
spellbook = SpellBook(
             spells=[
                 DistanceToNearest("primary",
                                   source_table="geospatial.ph_osm.gis_osm_
↪roads_free_1",
                                   feature_name="dist_primary"),
                 NumberOf("supermarket"
                          source_table="geospatial.ph_osm.gis_osm_pois_free_
↪1",
                          feature_name="num_supermarkets"),
             ])
```

SpellBooks can be distributed by exporting them to JSON files.

```
# Export SpellBook into a file
spellbook.author = "Juan dela Cruz"
spellbook.description = "Good Features for Economic Indicators"
spellbook.to_json("features_dela_cruz.json")
```

Now other people can easily reuse your feature extractions in with their own datasets!

```
from geomancer.spellbook import SpellBook
from tests.conftest import sample_points

spellbook = SpellBook.read_json("features_dela_cruz.json")
```

(continues on next page)

(continued from previous page)

```
df = sample_points() # load your own data

# Cast someone's Spells into your own data
df_with_features = spellbook.cast(df)
```

8.1 geomancer.spellbook.SpellBook

class geomancer.spellbook.spellbook.**SpellBook**(*spells*, *column='WKT'*, *author=None*, *description=None*)

Bases: object

__init__(*spells*, *column='WKT'*, *author=None*, *description=None*)

SpellBook constructor

Parameters

- **spells** (list of geomancer.spells.Spell) – List of spell instances.
- **column** (*str, optional*) – Column to look the geometries from. The default is WKT
- **author** (*str, optional*) – Author of the spell book
- **description** (*str, optional*) – Description of the spell book

cast(*df*)

Runs the cast method of each spell in the spell book

Parameters **df** (pandas.DataFrame) – Dataframe containing the points to compare upon. By default, we will look into the geometry column. You can specify your own column by passing an argument to the column parameter.

Returns Output dataframe with the features from all spells

Return type pandas.DataFrame

classmethod read_json(*filename*)

Reads a JSON exported spell book

Parameters **filename** (*str*) – Filename of JSON file to read.

Returns SpellBook instance parsed from given JSON file.

Return type geomancer.spellbook.SpellBook

to_json(*filename=None*, ***kwargs*)

Exports spell book as a JSON string

Parameters **filename** (*str, optional*) – Output filename. If none is given, output is returned

Returns Export of spell book in JSON format

Return type str or None

CHAPTER 9

Backend

Backend module for interacting with various databases

9.1 Cores

The Core module contains various engines that can interact with different databases. We strive for Geomaner to be database-agnostic: a query should ideally be executable across **all** types of data warehouse.

9.1.1 geomancer.backend.cores.base

Base class for all DBCore implementations

A DBCore is simply a database backend. It can be BigQuery, PostGIS, or an SQLite database. Whenever you want to add a new DBCore, simply subclass from the base `DBCore` class and implement the required methods

class geomancer.backend.cores.base.**DBCore**(*dburl*, *options=None*)

Bases: abc.ABC

Base class for all DBCore implementations

ST_GeoFromText(*x*)

Custom-implementation of converting a string into a geographical type

As it turns out, `ST_GeogFromText` only exists in BigQuery and PostGIS. Only `ST_GeomFromText` is available for Spatialite. Thus, we need to construct our own method for type-casting

__init__(*dburl*, *options=None*)

Initialize the database core

Parameters

- **dburl** (*str*) – Database url used to configure backend connection
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

get_engine()
Get the engine from the DBCore

Returns Engine with the database dialect

Return type sqlalchemy.engine.base.Engine

get_tables(*source_uri*, *target*, *engine*)
Create tables given a sqlalchemy.engine.base.Engine

Parameters

- **source_uri** (*str*) – Source table URI to run queries against.
- **target** (pandas.DataFrame or str) – Target table to add features to. If a string, must point to a table location found in the database.
- **engine** (sqlalchemy.engine.base.Engine) – Engine with the database dialect

Returns Source and Target table

Return type (sqlalchemy.schema.Table, sqlalchemy.schema.Table)

load(*df*)
Load a pandas.Dataframe into the database

Parameters **df** (pandas.DataFrame) – Input dataframe

Raises NotImplementedError – This is an abstract method

9.1.2 geomancer.backend.cores.bq

class geomancer.backend.cores.bq.**BigQueryCore**(*dburl*, *options=None*)
Bases: *geomancer.backend.cores.base.DBCore*

BigQuery DBCore

client
google.cloud.client.Client – BigQuery client for handling BQ interactions

ST_GeoFromText(*x*)
Custom-implementation of converting a string into a geographical type

As it turns out, ST_GeogFromText only exists in BigQuery and PostGIS. Only ST_GeomFromText is available for Spatialite. Thus, we need to construct our own method for type-casting

__init__(*dburl*, *options=None*)
Initialize the database core

Parameters

- **dburl** (*str*) – Database url used to configure backend connection
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

load(*df*, *dataset_id*, *expiry=3*, *max_retries=10*, ***kwargs*)
Upload a pandas.DataFrame as a BigQuery table with a unique 32-char ID

Parameters

- **df** (pandas.DataFrame) – Input dataframe to upload to BigQuery
- **dataset_id** (*str*) – ID to name the created Dataset
- **expiry** (*int, None*) – Number of hours for a given table to expire. Default is 3.

- **max_retries** (*int*) – Number of retries for the upload job to ensure that the table exists. Default is `10`

Returns The full path for the created table

Return type str

9.1.3 geomancer.backend.cores.sqlite

class geomancer.backend.cores.sqlite.**SQLiteCore**(*dburl*, *options=None*)

Bases: *geomancer.backend.cores.base.DBCore*

SQLite Core with Spatialite Extension

ST_GeoFromText(*x*)

Custom-implementation of converting a string into a geographical type

As it turns out, `ST_GeogFromText` only exists in BigQuery and PostGIS. Only `ST_GeomFromText` is available for Spatialite. Thus, we need to construct our own method for type-casting

__init__(*dburl*, *options=None*)

Initialize the database core

Parameters

- **dburl** (*str*) – Database url used to configure backend connection
- **options** (*geomancer.backend.settings.Config*, optional) – Specify configuration for interacting with the database backend. Auto-detected if not set.

get_engine()

Get the engine from the DBCore and load spatialite

load(*df*, *index_label=None*, *index=False*, *if_exists='replace'*, ***kwargs*)

Upload a pandas.DataFrame inside SQLite with a unique 32-char ID

9.2 Settings

Contains default configurations for interacting with the database engines

If you wish to override the default configurations, simply instantiate the class and set the value you desire. For example, you wish to lengthen the expiry date of a BigQuery table from `3` hours (the default) to `12` hours:

```
from geomancer import BQConfig
myconfig = BQConfig()
assert myconfig.EXPIRY == 3 # Default value
myconfig.EXPIRY = 12 # Update config
assert myconfig.EXPIRY == 12
```

9.2.1 geomancer.backend.settings.Config

class geomancer.backend.settings.**Config**

Bases: abc.ABC

Base abstract configuration

name

9.2.2 geomancer.backend.settings.BQConfig

class geomancer.backend.settings.**BQConfig**
Bases: *geomancer.backend.settings.Config*

Configuration for interacting with BigQuery

DATASET_ID
str – Specify the BQ dataset where the input pandas.DataFrame will be loaded into. Internally, we load the dataframe into a BigQuery table before running the actual query. Default is `geomancer`.

EXPIRY
int, None – Number of hours for a given table to expire. Default is `3`

MAX_RETRIES
int – Number of retries for the upload job to ensure that the table exists. Default is `10`

DATASET_ID = 'geomancer'

EXPIRY = 3

MAX_RETRIES = 10

name

9.2.3 geomancer.backend.settings.SQLiteConfig

class geomancer.backend.settings.**SQLiteConfig**
Bases: *geomancer.backend.settings.Config*

Configuration for interacting with a SQLite Database

INDEX
bool – Write pandas.DataFrame index as column. Uses INDEX_LABEL for column name Default is `False`

INDEX_LABEL
str or None – Column label for the index columns Default is `None`

IF_EXISTS
str – How to behave if the table already exists. Default is `replace` (drop the table before inserting new values). Other options are `fail` (raise a ValueError) and `append` (insert new values to the existing table)

IF_EXISTS = 'replace'

INDEX = False

INDEX_LABEL = None

name

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

B

C

D

E

G

I

L