Seedlot Selection Tool Documentation

Release 1.0

Conservation Biology Institute

Oct 24, 2017

Contents

The seedlot selection tool (SST) is a GIS mapping program designed to help forest managers match seedlots with planting sites based on climatic information. The tool can be used to map current climates, or future climates based on selected climate change scenarios. Although it is tailored for matching seedlots and planting sites, it can be used by anyone interested in mapping present or future climates defined by temperature and precipitation.

Note: This documentation is intended for developers. For user documentation, please see the tool's website: https://seedlotselectiontool.org/sst/. To report an problem, please create an issue on the issue tracker.

CHAPTER 1

Getting Started

Install & Project Setup

Requirements

Python Requirements

- Python 3.5+ (https://www.python.org/)
- Django 1.8 (https://www.djangoproject.com/)
- `ncdjango` 0.4.0 (http://ncdjango.readthedocs.io/)
- `clover` 0.2.0 (https://github.com/consbio/clover)
- `amqp`
- `celery`
- `django-celery`
- `djangorestframework`
- `Fiona`
- `kombu`
- `mercantile`
- `ncdjango==0.4.0`
- `netCDF4`
- `numpy`
- `Pillow`
- `psycopg2`
- `pyproj`

- pytest
- pytest-django
- rasterio
- Shapely
- django-filter
- aiohttp
- geopy
- raven
- WeasyPrint
- gunicorn
- django-celery-results
- social-auth-app-django
- python-pptx
- django-webpack

Other Requirements

- PostgreSQL (https://www.postgresql.org/)
- PostGIS (http://postgis.net/)
- NodeJS (https://nodejs.org/en/)
- Nginx (https://nginx.org/en/)
- Supervisor (http://supervisord.org/)
- RabbitMQ (https://www.rabbitmq.com/)
- UglifyJS (https://github.com/mishoo/UglifyJS)
- Babel (https://babeljs.io/)

Installation

This document covers installing the application stack on Linux using Nginx for a webserver proxy and Gunicorn for a WSGI server. You can certainly use other software to fill these needs.

It's a good idea to be familiar with deploying Django in a production environment: https://docs.djangoproject.com/en/1.8/howto/deployment/wsgi/

Note: These instructions cover deployment to a production environment. Setup for a develop environment will be similar but have have some differences depending on platform, and may not require certain dependencies such as Nginx, Gunicorn, and Supervisor.

Install Dependencies

Install the Python 3.5 and the non-Python dependencies first. Then install `numpy` followed by `GDAL`, `rasterio` and then `clover` (see https://github.com/consbio/clover). After that, the remainder of the Python requirements should be install fairly easily, though you may need to install a few development libraries first (e.g., for psycopg).

Note: It's recommended that you not use the root account to run your application or web server. Instead, use one account for nginx (if you install nginx through a package managet, this should be done automatically) and another account for the application itself (e.g., `seedsource`). It's also recommended to create a Python virtual environment as the application user and use it to install all Python dependencies and run all Python commands.

Note: `GDAL` will probably be the most challenging dependency to get installed. You may need to have the GDAL development libraries installed first, and if you get stuck, search for installing GDAL Python for your platform. Once GDAL is installed, everything else should be eaiser.

Note: There is a `clover` package on PyPI. This is *not* the clover you need to install. Make sure to install clover from the repository: https://github.com/consbio/clover

---

Install Project

Choose a location for the project directory (e.g., `/home/seedsource/apps/`). Navigate to the directory, and clone the repository:

```
$ git checkout https://github.com/consbio/seedsource.git
```

Setup & Configuration

Make sure PostgreSQL (+PostGIS), Nginx, Supervisor, and RabbitMQ are installed, running, and configured to start at launch. Create a spatially-enabled database, and a database user for the application (e.g., `seedsource`).

Configure Django

Create a file in `seedsource/source` directory called `config.json`. Add the following to this file, and fill out the values:

```
{
  "amqp_username": "",
  "amqp_password": "",
  "django_secret_key": "",
  "db_password": ""
}
```

You can also add the following optional keys to your `config.json`:

```
{
    "raven_dsn": "",
    "logfile_path": "",
    "db_name": "",
```

```
    "db_user": "",
    "db_host": ""
}
```

These keys are needed for social authentication:

```
{
    "google_oauth2_key": "",
    "google_oauth2_secret": "",
    "facebook_key": "",
    "facebook_secret": "",
    "twitter_key": "",
    "twitter_secret": ""
}
```

Make sure access to user email is activated by the OAuth provider.

Create a new Python module in `seedsource/source/seedsource_project/settings` called `custom.py`. Add the following to this new file:

```
from .production import *  # For development, import from .local instead

ALLOWED_HOSTS = []  # Add your host name or names here. E.g., 'seedlotselectiontool.
↪org'
```

Note: You can also add additional settings to `custom.py` or override settings specified in `production.py` and `base.py` as needed.

Run the database migrations:

```
$ python manage.py migrate
```

Configure Supervisor

If you don't have a supervisor configuration file already, create one with:

```
$ echo_supervisord_conf > /etc/supervisord.conf
```

Edit `/etc/supervisord.conf` and add programs for gunicorn, celery, and celery beat, filling in the paths as needed:

```
[program:gunicorn]
user=seedsource
directory=/path/to/seedsource/source
command=/path/to/bin/gunicorn --bind=127.0.0.1:8000 --pid=/path/to/gunicorn.pid --
↪error-logfile=/path/to/error.log --timeout=180 --graceful-timeout=180 --workers=4␣
↪seedsource_project.wsgi:application
autorestart=true

[program:django-celery-worker]
user=seedsource
directory=/path/to/seedsource/source
command=/path/to/bin/celery -A seedsource_project worker --loglevel=info --
↪concurrency=1
```

```
[program:django-celerybeat-worker]
user=seedsource
directory=/path/to/seedsource/source
command=/path/to/bin/celery -A seedsource_project beat --loglevel=info
```

Restart the supervisord process.

Configure Nginx

Edit your nginx configuration and add a location directive for the seedsource application, and another location directive for your static files:

```
location / {
    proxy_set_header Host $http_host;
    proxy_pass http://app_server;
}

location /static/ {
    alias /var/www/static/;
}
```

Note: If you want to store the static files in another location, you will also need to override the `STATIC_ROOT` setting in `custom.py`.

Restart or reload nginx.

Build & Deploy Static Content

Navigate to the `seedsource` root directory, install the npm dependencies, and run the build script:

```
$ npm install
$ npm run-script webpack_production
$ npm run-script merge-regions
```

One this completes, navigate to the `source` folder and run the `collectstatic` manage command:

```
$ python manage.py collectstatic
```

You should now be able to access the tool at `http://<your-server>/sst/`. Of course, for it to be useful, you will need data. This is covered in the *Adding Data* document.

Adding Data

The Seedlot Seletion Tool depends on three types of data: climate data, as NetCDF files; region boundary data, as shapefiles; and seed zone data, also as shapefiles.

Climate Data

Climate data is represented as ncdjango services. To import the data, first place your data in your `NC_SERVICE_DATA_ROOT` directory (see *Install & Project Setup*) under a directory named `regions`. The data

should be in a directory matching the region the data are for. The DEM should be placed in the directory for the region and named `<region>_dem.nc`. Sub-directories should be created for each year / climate scenario. For example, the directory structure for the `west2` region should be:

```
<NC_SERVICE_DATA_ROOT>
+-- regions
|   +-- west2
|   |   +-- 1961_1990Y
|   |   +-- 1991_2010Y
|   |   +-- rcp45_2025Y
|   |   +-- rcp45_2055Y
|   |   +-- rcp45_2085Y
|   |   +-- rcp85_2025Y
|   |   +-- rcp85_2055Y
|   |   +-- rcp85_2085Y
|   |   +-- west2_dem.nc
```

Inside each directory for a year/scenario, each climate variable dataset should be named according to region, RCP, year, and variable name in the following format: `<region>_<rcp45/rcp85>_<year>Y_<variable>.nc`. The current (1961_1990) and historic (1981_2010) years should not include an RCP. For example, the contents of the `1961_1990Y` and `rcp45_2025Y` directories for the `west2` region should be:

```
.
+-- 1961_1990Y
|   +-- west2_1961_1990Y_AHM.nc
|   +-- west2_1961_1990Y_bFFP.nc
|   +-- west2_1961_1990Y_CMD.nc
|   +-- <...>
+-- rcp45_2025Y
|   +-- west2_rcp45_2025Y_AHM.nc
|   +-- west2_rcp45_2025Y_bFFP.nc
|   +-- west2_rcp45_2025Y_CMD.nc
```

Once all the data is in place, you can run the following command to create services for the region elevation and all climate variables:

```
$ python manage.py populate_services <region>
```

The command will assume the variables: `'MAT', 'MWMT', 'MCMT', 'TD', 'MAP', 'MSP', 'AHM', 'SHM', 'DD_0', 'DD5', 'FFP', 'PAS', 'EMT', 'EXT', 'Eref', 'CMD'` and the years: `'1961_1990', '1981_2010', 'rcp45_2025', 'rcp45_2055', 'rcp45_2085', 'rcp85_2025', 'rcp85_2055', 'rcp85_2085'`. If you are using difference variables and/or years, you will need to edit the script, which is located at `source/seedsource/management/commands/populate_services.py`.

Region Boundary Data

You should simplify your boundary data before importing it into the tool. Next, import the region into the tool:

```
$ python manage.py add_region <region> <path to shapefile>
```

You should also convert the region boundary to GeoJSON and, it to the directory `source/seedsource/static/sst/geometry/<region>_boundary.json`, and re-run:

```
$ npm run-script merge-regions
```

Seed Zone Data

In order to import a set of seed zones into the tool, create a ZIP archive with a `config.json` file, and a folder containing the shapefile and related files. For example:

```
wa_seed_zones.zip
+-- config.json
+-- WA_NEW_ZONES
|   +-- TSHE.shp
|   +-- TSHE.dbf
|   +-- TSHE.shx
|   +-- TSHE.shp.xml
```

The `config.json` file contains information about the seed zones, and how to use and display them in the tool. For example:

```
{
  "label": "Washington",
  "dir": "WA_NEW_ZONES",
  "species": {
    "psme": {
      "file": "PSME.shp",
      "label": "Washington (2002) Douglas-fir Zone {zone_id}",
      "name": "wa_psme_{zone_id}",
      "column": "ZONE_NO",
      "bands_fn": "wa_psme"
    },
    "pico": {
      "file": "PICO.shp",
      "label": "Washington (2002) lodgepole pine Zone {zone_id}",
      "name": "wa_pico_{zone_id}",
      "column": "ZONE_NO",
      "bands_fn": "wa_pico"
    },
    "pipo": {
      "file": "PIPO.shp",
      "label": "Washington (2002) ponderosa pine Zone {zone_id}",
      "name": "wa_pipo_{zone_id}",
      "column": "ZONE_NO",
      "bands_fn": "wa_pipo"
    },
    "thpl": {
      "file": "THPL.shp",
      "label": "Washington (2002) western redcedar Zone {zone_id}",
      "name": "wa_thpl_{zone_id}",
      "column": "ZONE_NO",
      "bands_fn": "wa_thpl"
    },
    "pimo": {
      "file": "PIMO.shp",
      "label": "Washington (2002) western white pine Zone {zone_id}",
      "name": "wa_pimo_{zone_id}",
      "column": "ZONE_NO",
      "bands_fn": "wa_pimo"
    }
  }
}
```

The `label` and `name` properties both have substitutions for the zone id. The `label` will be shown to users with the zone id substituted. The `name` is used to uniquely identify the zone in the database.

The `column` property specifies the column in the shapefile table which contains the ID for each zone.

The `bans_fn` property specifies an elevations bands function to use in generating elevation bands. The following band functions are also available:

- `historical` Generates 500-ft elevation bands
- `no_bands` Generates a single elvation band for the entire elevation range of the zone

Generic (not species-specific seed zones) can use the "generic" key in the `config.json` file:

```
{
  "label": "Historic",
  "dir": "historic_seed_zones",
  "species": {
    "generic": {
      "file": "historic_seed_zones.shp",
      "label": "",
      "name": "wa_or_historic_{zone_id}",
      "column": "SUBJ_FSZ",
      "bands_fn": "historical"
    }
  }
}
```

Once you have created the ZIP archive, you can import it with the following command:

```
$ python manage.py import_seed_zones <path_to_zones_file>.zip
```

After importing the zones, you should run the `calculate_zone_transfers` command to generate transfer limits for each zone and elevation band (you will need to have service data for the appropriate region loaded first). Running the command with no arguments will process all zone sets:

```
$ python manage.py calculate_zone_transfers
```

Running the command with a `source` argument (<directory>/<shapefile>.shp) will process only zones for a single set:

```
$ python manage.py calculate_zone_transfers WA_NEW_ZONES/TSHE.shp
```

CHAPTER 2

Code Structure

The SST is web application comprised of both client (frontend) and server (backend) code. The frontend is a single page application built using React and Leaflet, and mostly written in JavaScript ES6. The backend is a Django project written Python 3.5.

Backend Structure

The server-side application is a Django project split into a few main apps, using Django Rest Framework to implement a REST API, and using ncdjango to provide the map service backend.

Frontend Structure

The frontend is a single page application built using React and Leaflet, and mostly written in JavaScript ES6.

Most of the application logic is contained within the `assets` directory. This code is all written in ES6. Some additional supporting JavaScript (non-ES6) code, as well as CSS documents are located in `source/seedsource/static/sst`.

All code in `assetts` is built using webpack with the output going to `source/seedsource/static/sst/build/main.js`.

During production deployment, the `collectstatic` Django command will copy all static content to a separate location to be served to the client by Nginx. During the `collectstatic` process, all unminified JS code is minified, and all uncompressed files are gzipped.

React Application

Most of the user interface is driven by a React application, the code for which is located in `assets`.

Note: This document assumes familiarity with basic React and Redux concepts. If you are not familiar with these frameworks, some good places to start are Thinking in React and Redux Basics.

Overview

The application is structured as follows:

```
assets/
|-- actions/ (Redux actions)
|-- async/ (async calls based on state changes)
|-- components/ (React components (rendering logic only)
|-- containers/ (state logic and event handling for components)
|-- reducers/ (Redux reducers)
|-- config.js (variables and species configurations, unit conversions, etc.)
|-- index.jsx (entry point)
|-- io.js (async utils)
|-- resync.js (utility to manage async calls based on state changes)
|-- utils.js (general utility functions)
```

Application State

The application state is composed of the following attributes:

- `isLoggedIn` (single value indicating whether the current user is logged in)
- `activeTab` (single value indicating the currently selected navigation tab)
- `activeVariable` (single value with the name of the variable currently displayed in the map)
- `activeStep` (single value indicating the currently selected configuration step)
- `runConfiguration` (object representing the current state of configuration options)
- `lastRun` (object representing the configuration options from the last successful run)
- `map` (object representing the state of various map componenets)
- `job` (object representing the state of an in-progress job)
- `saves` (object containing saved runs)
- `legends` (object containing variable and result legend information)
- `pdfIsFetching` (single value indicating whether a request for a PDF report is pending)
- `error` (object containing information about the most recent error)
- `popup` (object representing the state of the map popup and data shown within)

React Components

React components are split into two parts: the component itself, which only handles logic necessary to rendering the component (there are a couple of exceptions, where a component may handle local component state); and a container, which maps application state and event handlers to component properties. This follows the recommended practice outlined in the Redux documentation.

Connection /w Leaflet Map

A special-purpose component, `MapConnector` manages the interface between the Redux application state and the Leaflet map, so that user interaction with the map will update the state, and changes to the state will update the map.

`MapConnector` initializes the Leaflet map and does one-time setup during the mount phase in `componentWillMount()`. The `render()` method is used to compare the application state with the actual map state and update the map as necessary. `render()` returns null and the connector itself does not have any presence on page.

The Redux connect function is used to map relevant application state to the `MapConnector`, and dispatch actions based on map-related events.

Async Calls

There are two workflows for making asynchronous requests to the backend server. The first is in response to user action. This workflow is fairly straight-forward, a Redux action is dispatched, and the request is handled using Redux Thunk. Actions such as "Run Tool" and "Create PDF" are handled in this way.

The second workflow is a request triggered by a change in state. This is managed with the `resync` utility.

resync(*store*, *select*, *fetchData*)

Arguments

- **store** (*object*) – The Redux store
- **select** (*function*) – A function that will be called with the current application state and should return the state to watch
- **fetchData** (*function*) – The function called when the state has changed. The function signature is: `fetchData(currentState, io, dispatch, previousState)`

The utility will watch some subset of the application state, determined by the user-provided `select` function. When the watched state changes, it will call the user-defined `fetchData` function, providing references to the old and new state, an `io` utility object and the store's dispatch function. The `fetchData` function is expected to make async requests using the `io` object, and dispatch events as needed. The `io` utility automatically dismisses responses if newer requests have been made in the meantime. For example, if a watched state changes, and a request is made, then the state changes again and another request is made before the first returns, the response from the original request will be ignored.

Saving and Loading Configurations

Saving configurations is fairly straight-forward. Only the last completed run may be saved. Upon save, the `lastRun` state is serialized to JSON and sent to the server for storage. Since zone geometry does not to be represented in the saved configuration, before serialization, the geometry is set to null.

When the application is loaded, all saved configurations are fetched form the server and stored in the `saves` state. When the user loads a save, the configuration is loaded from JSON and used to update the `runConfiguration` state. The change in state then triggers an async call to load zone geometry.

CSS

All style information for the react app, and the rest of the tool UI is stored in `source/seedsource/static/sst/css/tool.css`.

Index

R