INTRODUCTION

SEVIR: Storm Event Imagery Dataset

The SEVIR dataset consists of –

- 107 files in HDF5 format containing imagery
- Files separated by image type and date range for easier access
- File sizes ranging between 1GB and 20 GB
- A catalog containing meta data of each image, including
- Unique ID assigned to each event
- Times of each image • Lat / Lon bounds of slice
- Map projection and image extent in projection coordinates for exact georeferencing
- NCEI Storm Event ID and Episode ID (for non-random cases)

The SEVIR catalog contains relevant information about each event in SEVIR. Table 1 includes a list of the catalog columns, with a short description of each. When extracting data from SEVIR, it is helpful to first group the catalog by the id column. After doing so, the size of each group will represent the number of image types that are available for each event. This is useful for building training datasets that utilize multiple image types (such as the synthetic radar problem). The catalog also allows for efficient filtering of SEVIR by time, geographic location, or by statistic for more focused training and testing sets.

SEVIR data is available as a set of HDF large files. A commonly used approach in model training is to organize data (especially images) into directories that correspond to class labels. Other approaches include combining data into TFRecord files (compatible with TensorFlow) or HDF5 files. We chose HDF5 because of the availability of open-source libraries for reading this data format in a variety of languages such as python, MATLAB, C/C++, JAVA, Fortran, etc. While it's possible to stream data from SEVIR for model fitting, randomized reads from HDF5 files can be slow. In addition, HDF5-specific choices (such as the chuck size), made during file creation can also affect the speed of file reads. A comprehensive analysis of HDF5 File I/O is beyond the scope of this work, but based on our experiments, it is recommended that when using SEVIR, the data first be read into one or more interim files where the data is shuffled a-priori and written to file sequentially.

Training effective deep models requires the tuning of hyperparameters which includes learning rates, batch sizes, number of encoder and decoder layers, number of filters per layer, filter sizes per layer, and many other configurable model parameters. A comprehensive analysis of all

possible combinations of these parameters is not possible in any reasonable amount of time and we used best practices from prior work to inform our choices. However, the choice of the batch size was dictated by the data sizes used in our study and the loss function used to implement the model.

SEVIR is a collection of temporally and spatially aligned image sequences depicting weather events captured over the contiguous US (CONUS) by GOES-16 satellite and the mosaic of NEXRAD radars. Figure 1 shows a set of frames taken from a SEVIR event. SEVIR contains five image types: GOES-16 0.6 μm visible satellite channel (vis), 6.9 μm and 10.7 μm infrared channels (ir069, ir107), a radar mosaic of vertically integrated liquid (vil), and total lightning flashes collected by the GOES-16 geostationary lightning mapper (GLM) (lght).

Events in SEVIR were selected in one of two ways - Random selection and Storm event-based selection. To select random events, a set of random times was uniformly selected through the years 2017-2019. For each time, each image type was retrieved if it was available. Event centers were sampled randomly based on the VIL data, with higher probability given to pixels with higher VIL intensity. This sampling method ensured that the dataset did not oversample the "no precipitation" case. Random events in SEVIR have an id starting with R.

FORECASTING THE IMAGES

RADAR NOWCASTING

Nowcasts are high resolution, short-term (e.g., up to 2 hours) weather forecasts of radar echoes, precipitation, cloud coverage or other meteorological quantities widely used in public safety, air traffic control, and many other areas that require high fidelity and rapidly updating forecasts. Previous work on deep learning for Nowcasting includes convolutional Long Short-Term Memory (ConvLSTM) models [30], recurrent architectures and fully convolutional networks [20, 6] for precipitation nowcasting. We frame nowcasting as a future prediction task where the model input consists of 13 VIL images sampled at 5-minute intervals. The model is trained to produce the next 12 images in the sequence, corresponding to the next hour of weather. Data from SEVIR was first extracted and processed into 44,760 sequences for training and an independent set of 12,144 sequences for testing the fit model. This was done by splitting each SEVIR event into 3 input and output sequences. The model input was of size Nx384x384x13, and the output sequence was Nx384x384x12 pixels, where N is the batch size.

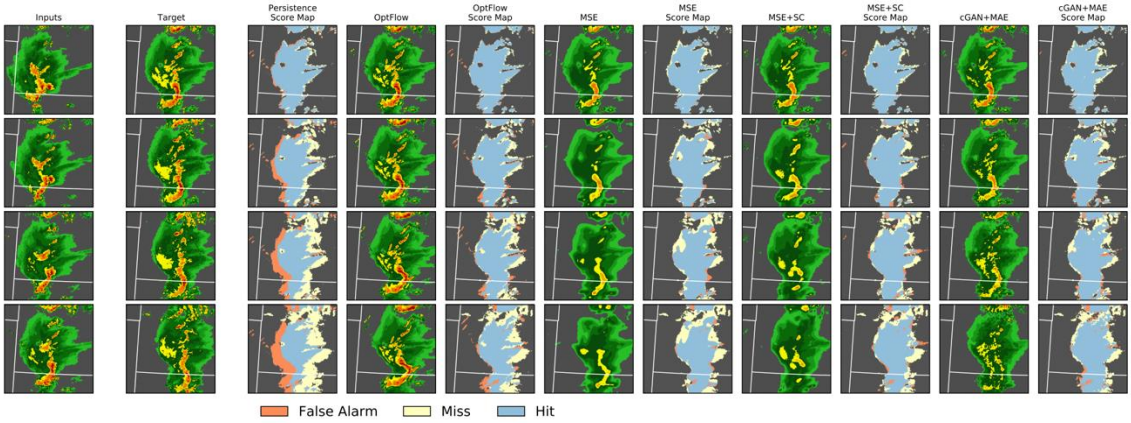


Figure above shows Nowcast output from U-Net model with different loss functions. Abbreviations used are as follows: MSE - Mean Squared Error, SC - Style and content loss, cGAN - Conditional GAN.

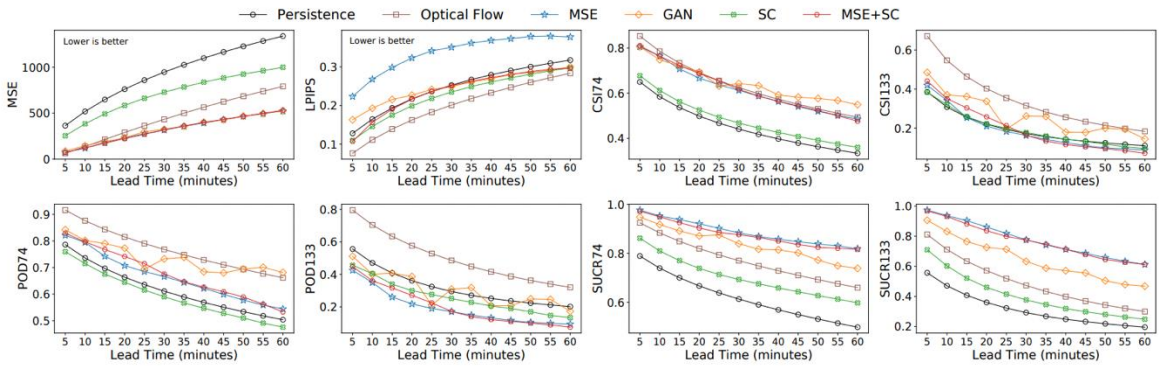
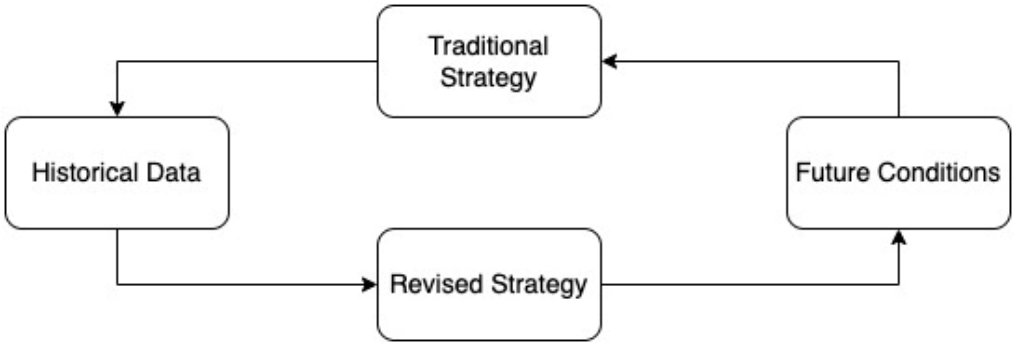


Figure above depicts Trends in evaluation metrics over lead times: All approaches show improved performance over the persistence model. However, the metrics show degraded performance as the lead time increases. Numbers 74, 133 on the y-axis correspond to threshold levels described.

BACKTEST FOR VARIOUS USE CASES

Backtesting is the process of assessing how well a trading strategy or analytical method could perform, based on historical data. It is a key component in developing an effective trading strategy. There are infinite possibilities for strategies, and any slight alteration will change the results. Therefore, backtesting is important, as it shows whether certain parameters will work better than others.

We will generate several test datasets to test the models designed in order to check their performance on the said test data. We will generate test data for a set of distinct event ids present in the dataset.



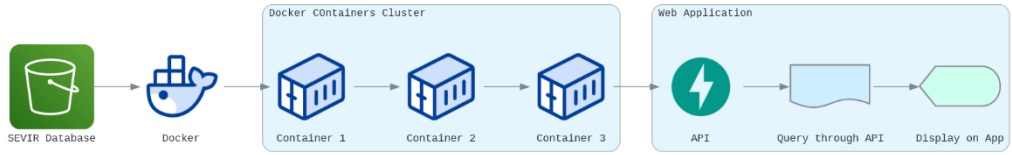
IMPLEMENTATION

A backtest is usually coded by a programmer running a simulation on the strategy. The simulation is run using historical data from weather forecast, storms, and other random events. The person facilitating the backtest will assess the returns on the model across several different generated test datasets.

It is also essential that the model is tested across many different conditions to assess performance objectively. Variables within the model are then tweaked for optimization against several different backtesting measures.

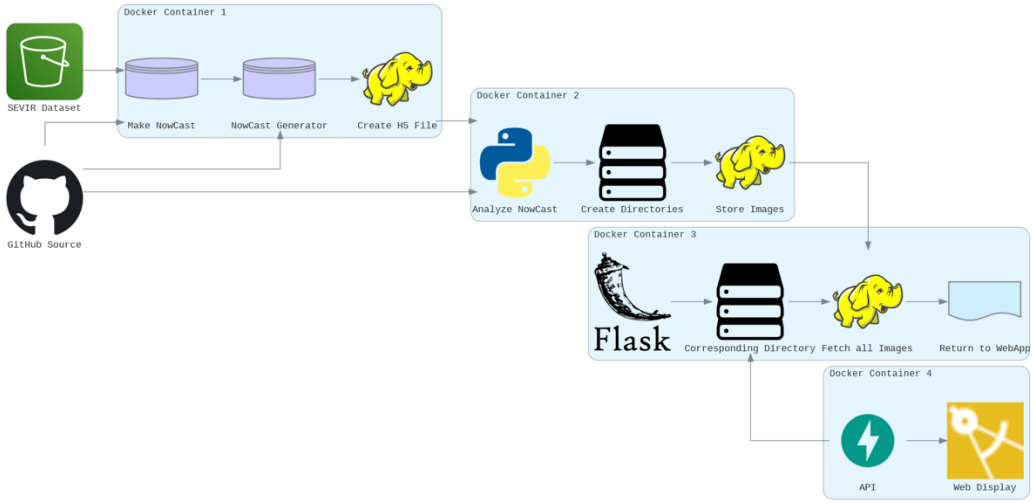
ARCHITECTURE FOR DESIGNING A NOWCASTING INTELLEGENT SYSTEM

ABSTRACT PIPELINE ARCHITECTURE



Pipeline Architecture

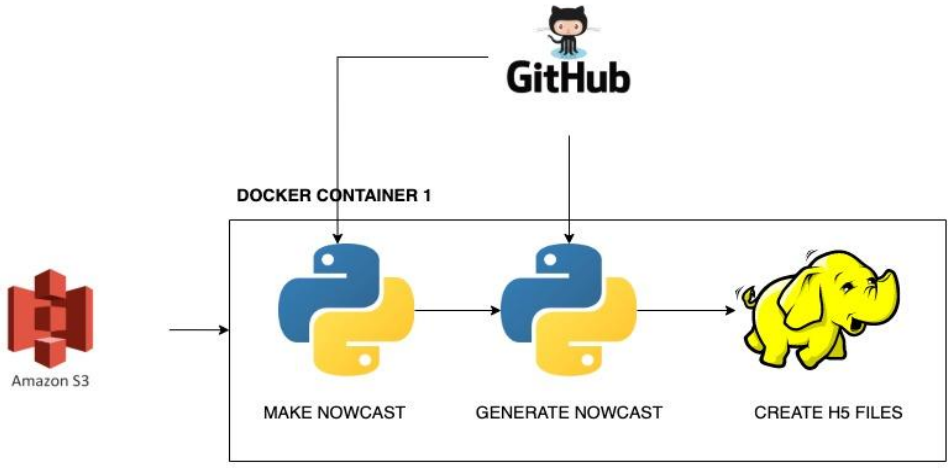
DETAILED ARCHITECTURE



Pipeline Architecture

We will be making use of Docker containers as the concept of containerization is to help run pipelines and structured environment across different systems without having any issues as it packs the whole structure as one.
We will be making use of multiple docker images to run various parts of this pipeline as to remove the stress or load on just one image.

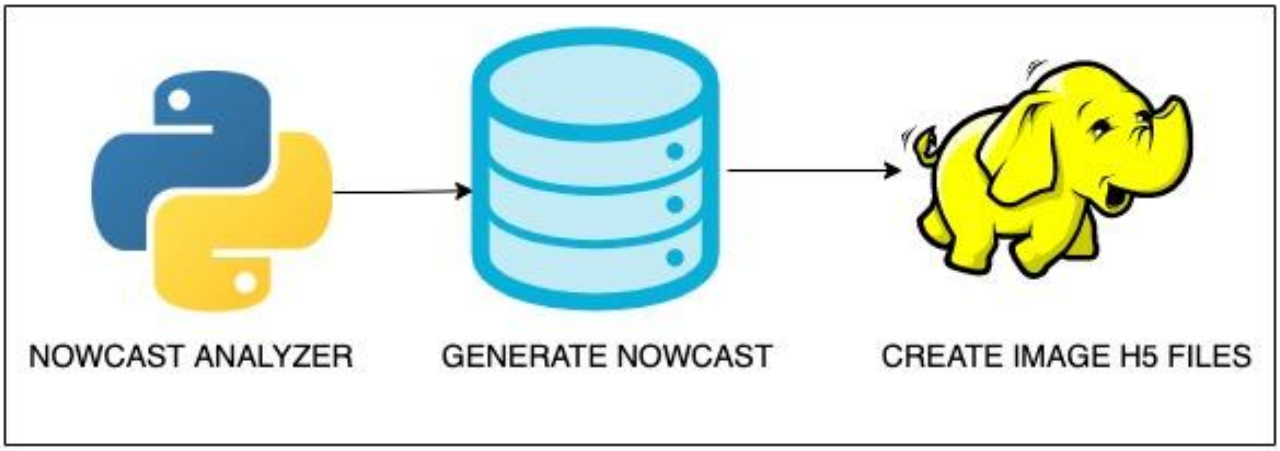
CONTAINER – 1



- This docker image will have the complete docker dataset mounted on it that will be fed from the amazon S3.
- This S3 contains the entire SEVIR dataset including the Catalog file.
- The image will be running based on the GitHub link which contains the dataset generator scripts for Nowcast.
- Mount /src from git.
- We will install all dependencies that are needed inside the docker image.
- Once the docker image starts running, it can be considered as a fully spun docker container that is going to generate the .h5 dataset.
- This will be persisted locally in the OS so it can later be used by other containers in the pipeline.

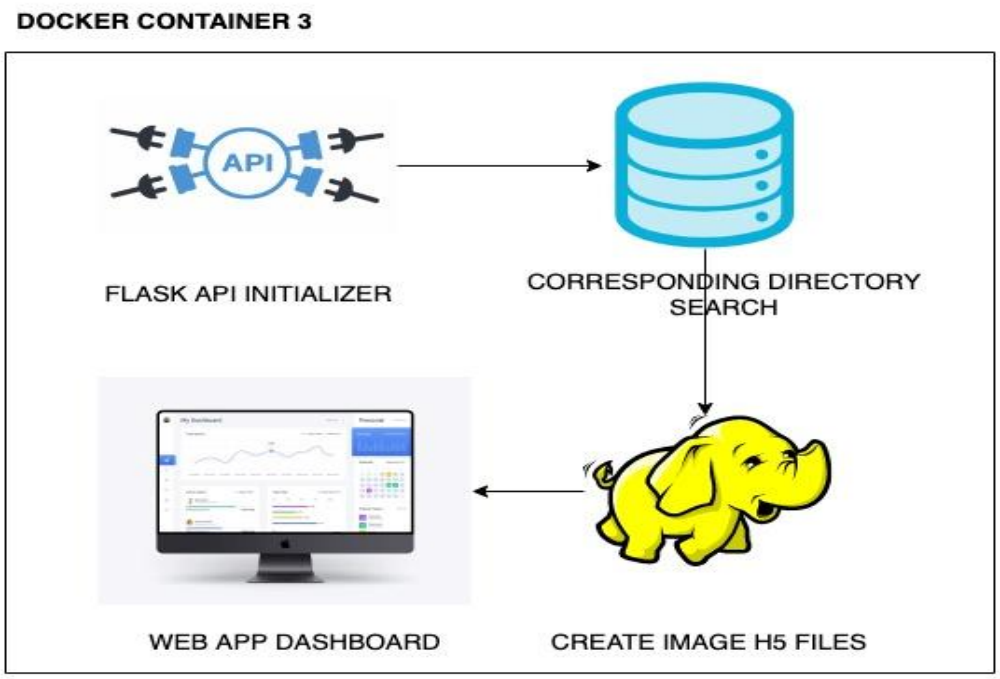
CONTAINER – 2

DOCKER CONTAINER 2



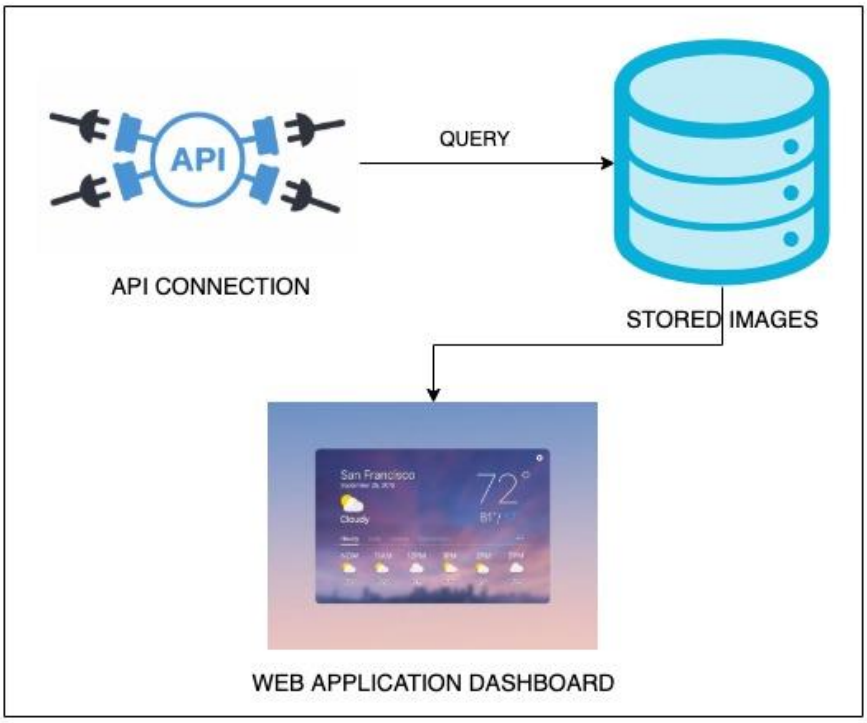
- This Docker image is going to run the Analyze Nowcast script.
- This image is also based on the GitHub that contains the script.
- We will first install and run all dependencies to resolve future issues.
- Mount all the required files/paths as needed for the script to run, including the .h5 that has been generated in container 1.
- Run and analyze the script.
- The images that are generated specific to the event ID and location, this will also be stored locally for future use on the API and Web App.

CONTAINER - 3 [API]



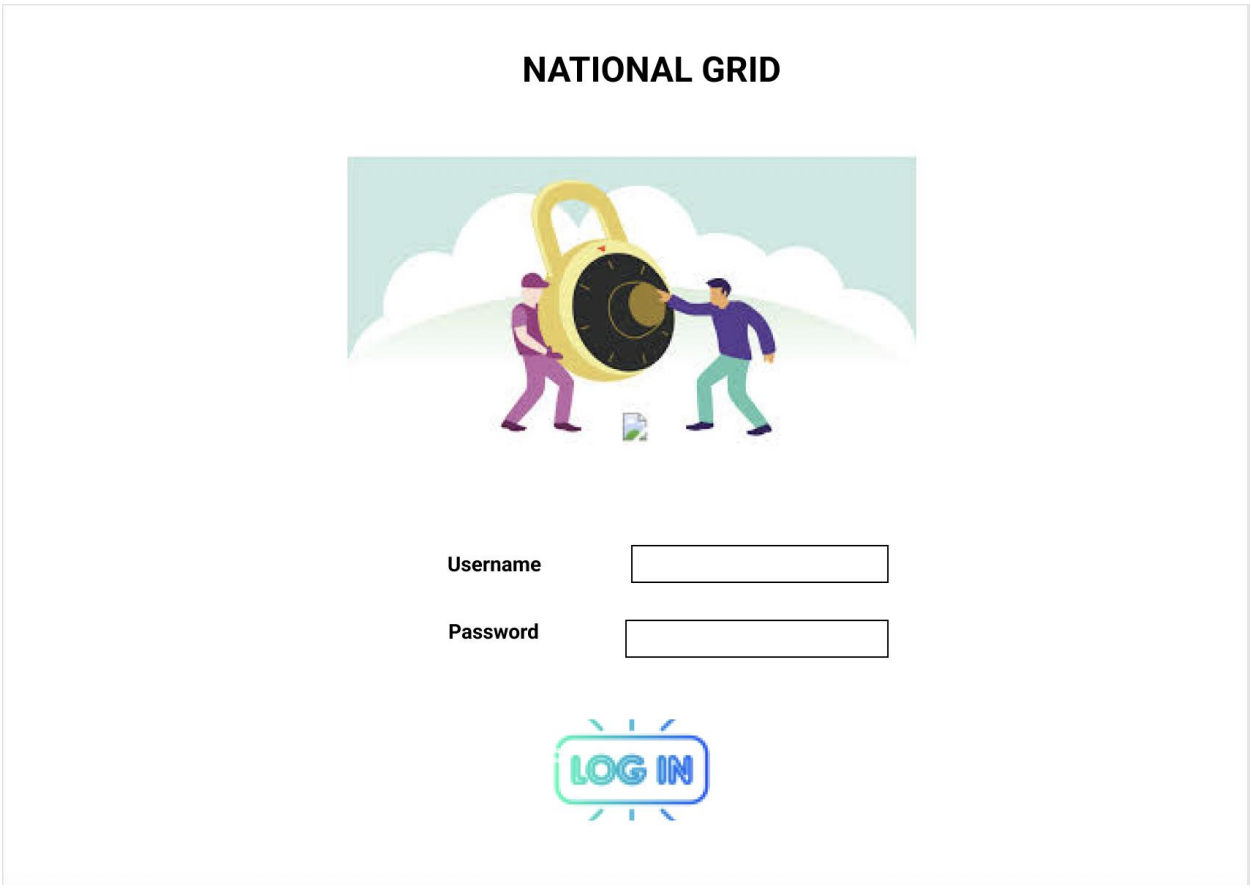
- The API that we are designing is a Python Flask Server.
- Flask is a web framework.
- A Python module that lets you develop web applications easily.
- It has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.
- This will be used as a 3rd container in our pipeline.
- We will collect all the dependencies for flask server and install the needed.
- Mount the images from the container 2 on this API server and run so that the web app gets the images when asked by user.

CONTAINER – 4

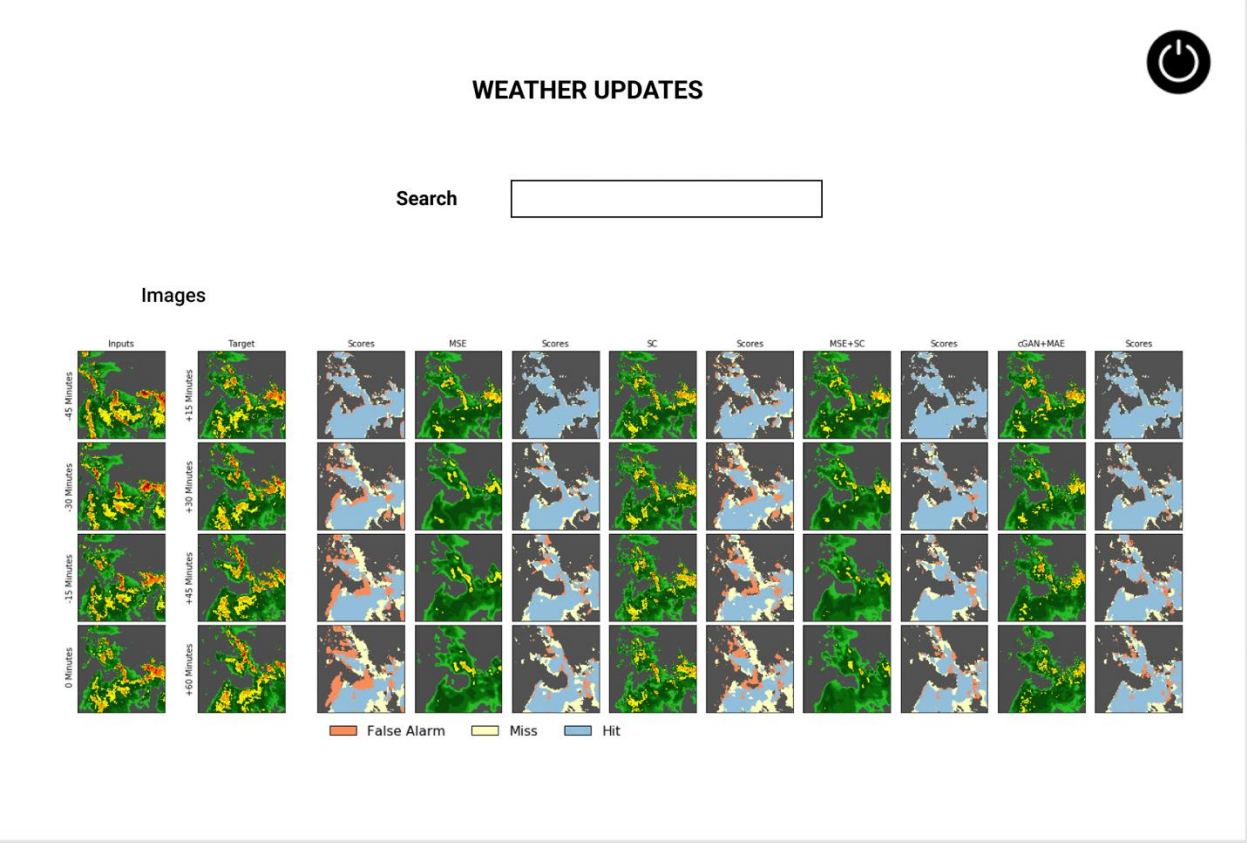


- The Web App itself will be running as a 4th container post API call to image 3.
- It has a login option for the Electricity department to gain access into.
- We will then be showing the list of locations as a dropdown for them to choose the region they want information on.
- Upon selection of the same and submit, it will pull up the image with that particular location corresponding to that event ID from the catalog file.
- The Catalog.csv file will be mounted on this image for access.

WEB APPLICATION



When the user first opens the link to the web application, they will see a login page where they can sign in and check the weather conditions in a particular place. The admins and the users log in through the same web page to investigate the application.



In the next page of the application, the user will be able to query the dataset using the location such as Boston, Canton etc. to get the visual representation of the weather conditions in the next 24 hours.

LOCATION API

- In the first container, a directory will be created for each event ID (as it corresponds to a single location).
- Upon coming across a new event ID, a new directory will be created.
- These directories hold the details for that particular location.
- Upon using the google maps API and feeding in the latitude and longitude values for location, it will parse through the catalog file for lat and lon values, gets the exact location and it is connected to the drop down in container 4.
- This helps in pulling up the details for that particular region/event ID.

API INSTALLATION AND USES

Like most widely used Python libraries, the Flask package is installable from the Python Package Index (PPI). First create a directory to work in (something like flask_todo is a fine directory name) then install the flask package. You'll also want to install flask-sqlalchemy so your Flask application has a simple way to talk to a SQL database.

Steps to get started on a terminal:

```
$ mkdir flask_todo
$ cd flask_todo
$ pipenv install --python 3.6
$ pipenv shell
(Flask-someHash) $ pipenv install flask flask-sqlalchemy
```

very explicit, which increases readability. To create the “Hello World” app, you only need a few lines of code.

This is a boilerplate code example.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

A good way to get moving is to turn the codebase into an installable Python distribution. At the project's root, create setup.py and a directory called todo to hold the source code.

The setup.py should look something like this:

```
from setuptools import setup, find_packages
requires = [
    'flask',
```

```
    'flask-sqlalchemy',
    'psycopg2',
]

setup(
    name='flask_todo',
    version='0.0',
    description='A To-Do List built with Flask',
    author='<Your actual name here>',
    author_email='<Your actual e-mail address here>',
    keywords='web flask',
    packages=find_packages(),
    include_package_data=True,
    install_requires=requires
)
```

Within the todo directory containing your source code, create an app.py file and a blank __init__.py file. The __init__.py file allows you to import from todo as if it were an installed package. The app.py file will be the application's root. This is where all the Flask application goodness will go, and you'll create an environment variable that points to that file. If you're using pipenv (like I am), you can locate your virtual environment with pipenv --venv and set up that environment variable in your environment's activate script.

```
# In your activate script, probably at the bottom (but anywhere will do)

export FLASK_APP=$VIRTUAL_ENV/../todo/app.py
export DEBUG='True'
```

CONNECTING THE DATABASE IN FLASK

While the code example above represents a complete Flask application, it doesn't do anything interesting. One interesting thing a web application can do is persist user data, but it needs the help of and connection to a database.

Flask is very much a "do it yourself" web framework. This means there's no built-in database interaction, but the flask-sqlalchemy package will connect a SQL database to a Flask application. The flask-sqlalchemy package needs just one thing to connect to a SQL database: The database URL.

```
# Top of app.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgres://localhost:5432/flask_todo'
db = SQLAlchemy(app)
```

In the same place you declared FLASK_APP, declare a DATABASE_URL pointing to the location of your Postgres database. Development tends to work locally, so just point to your local database.

```
# Also, in your activate script
export DATABASE_URL='postgres://localhost:5432/flask_todo'
```

Now in app.py, include the database URL in your app configuration.

```
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL', '')
db = SQLAlchemy(app)
```

PACKAGING A PYTHON PROJECT

This tutorial walks you through how to package a simple Python project. It will show you how to add the necessary files and structure to create the package, how to build the package, and how to upload it to the Python Package Index.

Some of the commands require a newer version of pip, so start by making sure you have the latest version installed:

Unix/macOS

```
python3 -m pip install --upgrade pip
```

Windows

```
py -m pip install --upgrade pip
```

A SIMPLE PROJECT

This tutorial uses a simple project named example_package. We recommend following this tutorial as-is using this project, before packaging your own project.

Create the following file structure locally:

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- loadDatasetBigQuery.ipnyb
│                      <- sevir_analysis_final.ipnyb
│
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- SEVIR
│   └── figures        <- SEVIR-Visualizations
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- makes project pip installable (pip install -e .) so src can be imported
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.readthedocs.io
```

__init__.py is required to import the directory as a package and should be empty.
predict_model.py is an example of a module within the package that could contain the logic (functions, classes, constants, etc.) of your package.

Once you create this structure, you'll want to run all the commands in this tutorial within the packaging_tutorial directory.

CREATING THE PACKAGING FILES

You will now add files that are used to prepare the project for distribution. When you're done, the project structure will look like this:

packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg

```
├── src/
|   └── example_package/
|       ├── __init__.py
|       └── example.py
└── tests/
```

CREATING A TEST DIRECTORY

tests/ is a placeholder for test files. Leave it empty for now.

CREATING myproject.toml

pyproject.toml tells build tools (like pip and build) what is required to build your project. This tutorial uses setup tools, so open pyproject.toml and enter the following content:

```
[build-system]
requires = [
   "setuptools>=42",
   "wheel"
]
build-backend = "setuptools.build_meta"
```

build-system.requires gives a list of packages that are needed to build your package. Listing something here will only make it available during the build, not after it is installed.

build-system.build-backend is the name of Python object that will be used to perform the build. If you were to use a different build system, such as flit or poetry, those would go here, and the configuration details would be completely different than the setuptools configuration described below.

CONFIGURING METADATA

There are two types of metadata: static and dynamic.

Static metadata (setup.cfg): guaranteed to be the same every time. This is simpler, easier to read, and avoids many common errors, like encoding errors.

Dynamic metadata (setup.py): possibly non-deterministic. Any items that are dynamic or determined at install-time, as well as extension modules or extensions to setuptools, need to go into setup.py.
Static metadata (setup.cfg) should be preferred. Dynamic metadata (setup.py) should be used only as an escape hatch when necessary. setup.py used to be required but can be omitted with newer versions of setup tools and pip.

setup.cfg (static)
setup.cfg is the configuration file for setup tools. It tells setup tools about your package (such as the name and version) as well as which code files to include. Eventually much of this configuration may be able to move to pyproject.toml.

setup.py (dynamic)
setup.py is the build script for setup tools. It tells setup tools about your package (such as the name and version) as well as which code files to include.

setup() takes several arguments. This example package uses a relatively minimal set:

- name is the distribution name of your package. This can be any name as long as it only contains letters, numbers, and -. It also must not already be taken on pypi.org. Be sure to update this with your username, as this ensures you won't try to upload a package with the same name as one which already exists.

- version is the package version.

- author and author_email is used to identify the author of the package.

- description is a short, one-sentence summary of the package.

- long_description is a detailed description of the package. This is shown on the package detail page on the Python Package Index. In this case, the long description is loaded from README.md, which is a common pattern.

- long_description_content_type tells the index what type of markup is used for the long description. In this case, it's Markdown.

- url is the URL for the homepage of the project. For many projects, this will just be a link to GitHub, GitLab, Bitbucket, or similar code hosting service.

- project_urls lets you list any number of extra links to show on PyPI. Generally, this could be to documentation, issue trackers, etc.
- classifiers give the index and pip some additional metadata about your package. In this case, the package is only compatible with Python 3, is licensed under the MIT license, and is OS-independent. You should always include at least which version(s) of Python your package works on, which license your package is available under, and which operating systems your package will work on. For a complete list of classifiers, see https://pypi.org/classifiers/.

- package_dir is a dictionary with package names for keys and directories for values. An empty package name represents the "root package" — the directory in the project that contains all Python source files for the package — so in this case the src directory is designated the root package.

- packages are a list of all Python import packages that should be included in the distribution package. Instead of listing each package manually, we can use find_packages() to automatically discover all packages and subpackages under package_dir. In this case, the list of packages will be example_package as that's the only package present.

- python_requires gives the versions of Python supported by your project. Installers like pip will look back though older versions of packages until it finds one that has a matching Python version.

GENERATING DISTRIBUTION ARCHIVES

The next step is to generate distribution packages for the package. These are archives that are uploaded to the Python Package Index and can be installed by pip.

Make sure you have the latest version of PyPA's build installed:
Unix/MacOS
python3 -m pip install --upgrade build

Windows
py -m pip install --upgrade build

Now run this command from the same directory where pyproject.toml is located:

Unix/macOS
python3 -m build

Windows
py -m build

This command should output a lot of text and once completed should generate two files in the dist directory:

Unix/macOS
python3 -m twine upload --repository testpypi dist/*

Windows
py -m twine upload --repository testpypi dist/*

The tar.gz file is a source archive whereas the .whl file is a built distribution. Newer pip versions preferentially install built distributions but will fall back to source archives if needed. You should always upload a source archive and provide built archives for the platforms your project is compatible with. In this case, our example package is compatible with Python on any platform so only one built distribution is needed.

INSTALLING NEWLY UPLOADED PACKAGE

You can use pip to install your package and verify that it works. Create a virtual environment and install your package from TestPyPI:

Unix/macOS
python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-package-YOUR-USERNAME-HERE

Windows
py -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-package-YOUR-USERNAME-HERE

Make sure to specify your username in the package name!
pip should install the package from TestPyPI

You can test that it was installed correctly by importing the package. Make sure you're still in your virtual environment, then run Python:
Unix/macOS - python3
Windows – py
and import the package:
>>from example_package import example
>>example.add_one(2)

Note that the import package is example_package regardless of what name you gave your distribution package in setup.cfg or setup.py (in this case, example-package-YOUR-USERNAME-HERE).

NEXT STEPS

Now, you've packaged and distributed a Python project!

Keep in mind that this tutorial showed you how to upload your package to Test PyPI, which isn't a permanent storage. The Test system occasionally deletes packages and accounts. It is best to use TestPyPI for testing and experiments like this tutorial.

When you are ready to upload a real package to the Python Package Index you can do much the same as you did in this tutorial, but with these important differences:

- Choose a memorable and unique name for your package. You don't have to append your username as you did in the tutorial.

- Register an account on https://pypi.org - note that these are two separate servers and the login details from the test server are not shared with the main server.

- Use twine upload dist/* to upload your package and enter your credentials for the account you registered on the real PyPI. Now that you're uploading the package in production, you don't need to specify --repository; the package will upload to https://pypi.org/ by default.

- Install your package from the real PyPI using python3 -m pip install [your-package].