Web-based DAS ML System
(WebML DAS)
Software Requirement Analysis

Author	Date	Log
Rupali Roy	Aug 24, 2020	Created GUI document

DAS File Processing & Anomaly Detection

1. Problem Statement:

The aim of this summer project was to develop a web portal that is used to process DAS: Distributed Acoustic Sensing data and identify anomalies in the data for earthquake detection.

The DAS data is mostly used from PoroTomo and Forge files, for your reference you can refer the following link that has distributed acoustic sensing data acquired from University of Wisconsin Porotomo Ftp server.

ftp://roftp.ssec.wisc.edu/porotomo/PoroTomo2/DATA/DASH/20160321/

Specifically, we are interested in detecting the earthquake event located in this dataset as below

ftp://roftp.ssec.wisc.edu/porotomo/PoroTomo2/DATA/DASH/20160321/PoroTomo_iDAS16043_160321000721.sgy

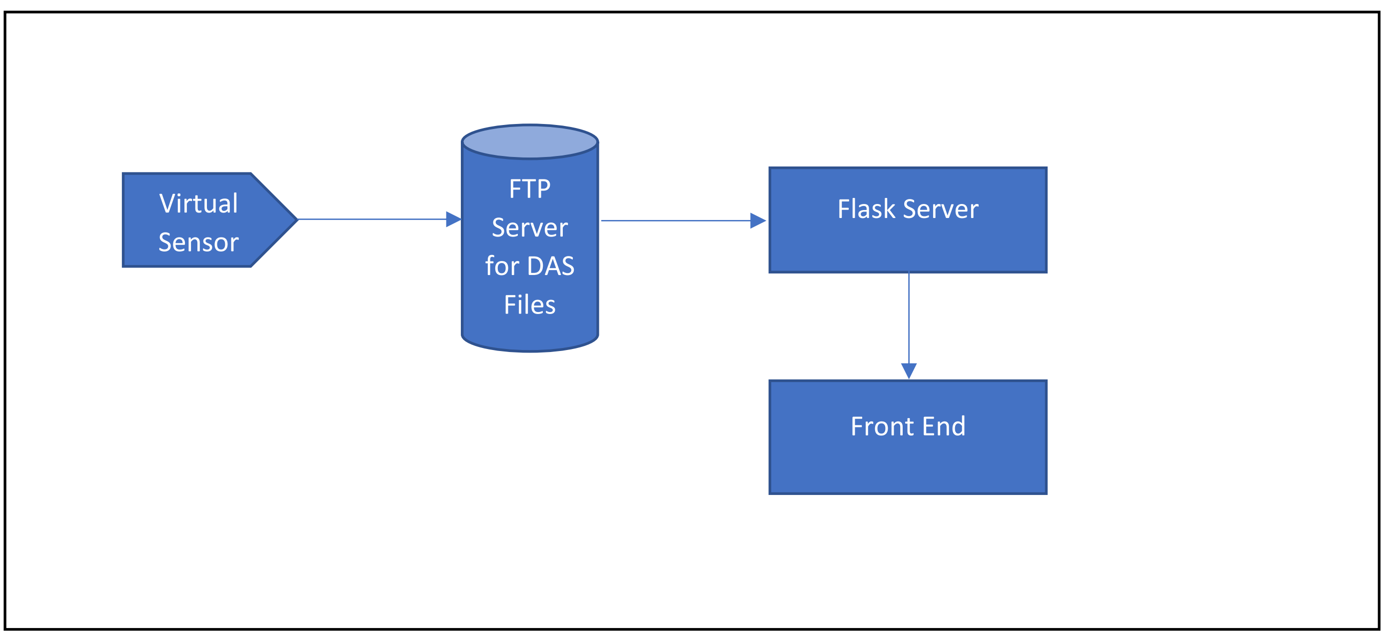
Each dataset includes data recorded in 30 sec

2. Overall System Design for DAS Data

The WebML DAS system is an unsupervised anomaly detection portal for DAS data. There are several virtual sensors that are installed at different locations (Channels) that continuously record the ground vibrations along with timestamp and channel number (this indicate the location details). The virtual sensor data is stored in seg-y files recorded data is stored in segy files. The DAS data is stored as seg-y files (Society of Exploration Geophysicists (SEG) for storing geophysical data) on different FTP servers.

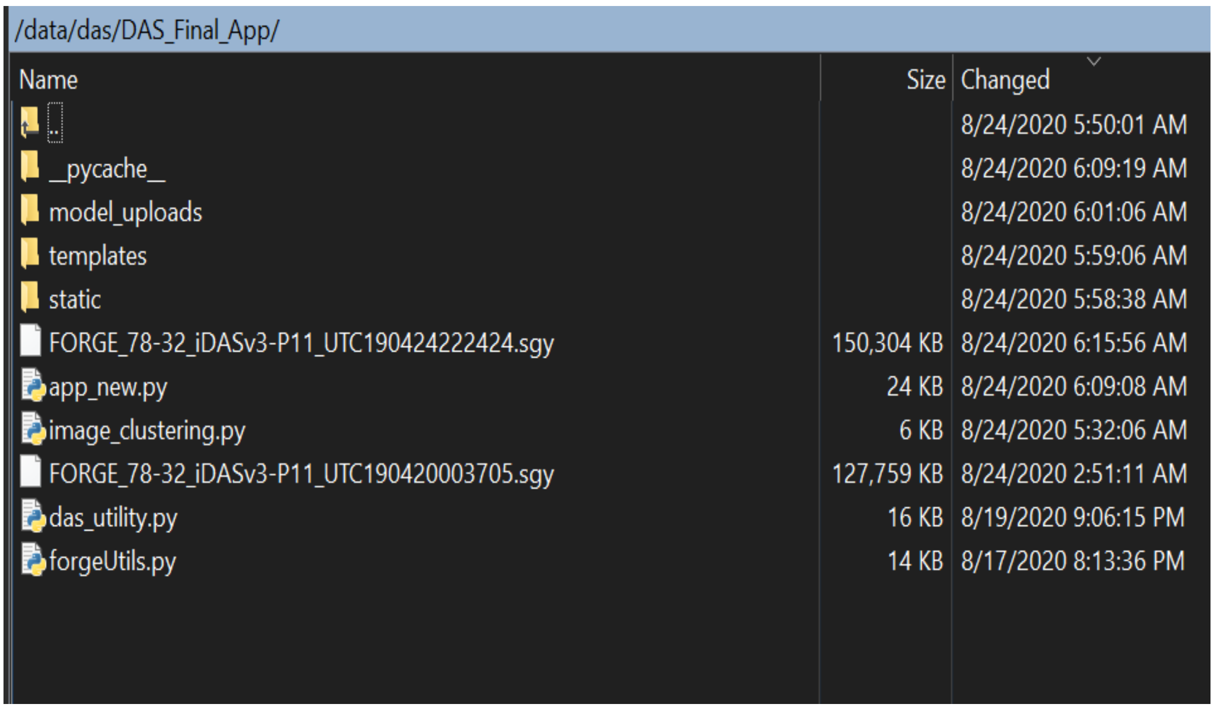
To detect anomalies in the seg-y files, we process the DAS data to generate different kinds of plots that can be used to train a deep learning model.

The current webML system uses deep unsupervised clustering to detect anomalies in the data.



3. File Structure for the project

- The Static folder includes all the CSS styling scripts, JavaScript files and the "Obspy_Plots" folder which has all the images that needs to be displayed on UI.
- All the HTML files for the front-end needs to be there in the templates folder.
- If we want to display any particular image we need to keep it in the static folder. For this particular project I have included all the images that are shown on the web page in the "static/Obspy_plots/diff_plots". This folder needs to be specified in the app_new.py file.
- All the JavaScript files that are used in the project are stored in "Static/js" folder.


/data/das/DAS_Final_App/

Name	Size	Changed
..		8/24/2020 5:50:01 AM
pycache		8/24/2020 6:09:19 AM
model_uploads		8/24/2020 6:01:06 AM
templates		8/24/2020 5:59:06 AM
static		8/24/2020 5:58:38 AM
FORGE_78-32_iDASv3-P11_UTC190424222424.sgy	150,304 KB	8/24/2020 6:15:56 AM
app_new.py	24 KB	8/24/2020 6:09:08 AM
image_clustering.py	6 KB	8/24/2020 5:32:06 AM
FORGE_78-32_iDASv3-P11_UTC190420003705.sgy	127,759 KB	8/24/2020 2:51:11 AM
das_utility.py	16 KB	8/19/2020 9:06:15 PM
forgeUtils.py	14 KB	8/17/2020 8:13:36 PM

4. Steps to run the web application on server

1. Open Command prompt and enter the command **" ssh –i .ssh/key cc@129.114.24.214"**
   Key is your public key that is used to authenticate you to login to server.
2. Login to server and change your directory to /data/das using the command **"cd /data/das"**
3. The DAS Web App is into **"cd /data/das/DAS_Final_App".**
4. We have a requirements_new.txt file in this directory, this will help you install all the dependencies for this project. Open the command prompt to install all the python libraries using "**pip install -r requirements.txt** "
5. Run the application by using the following command**: "python app_new.py". This will run the web application on port 5000.**

5. DAS Web Application Flow

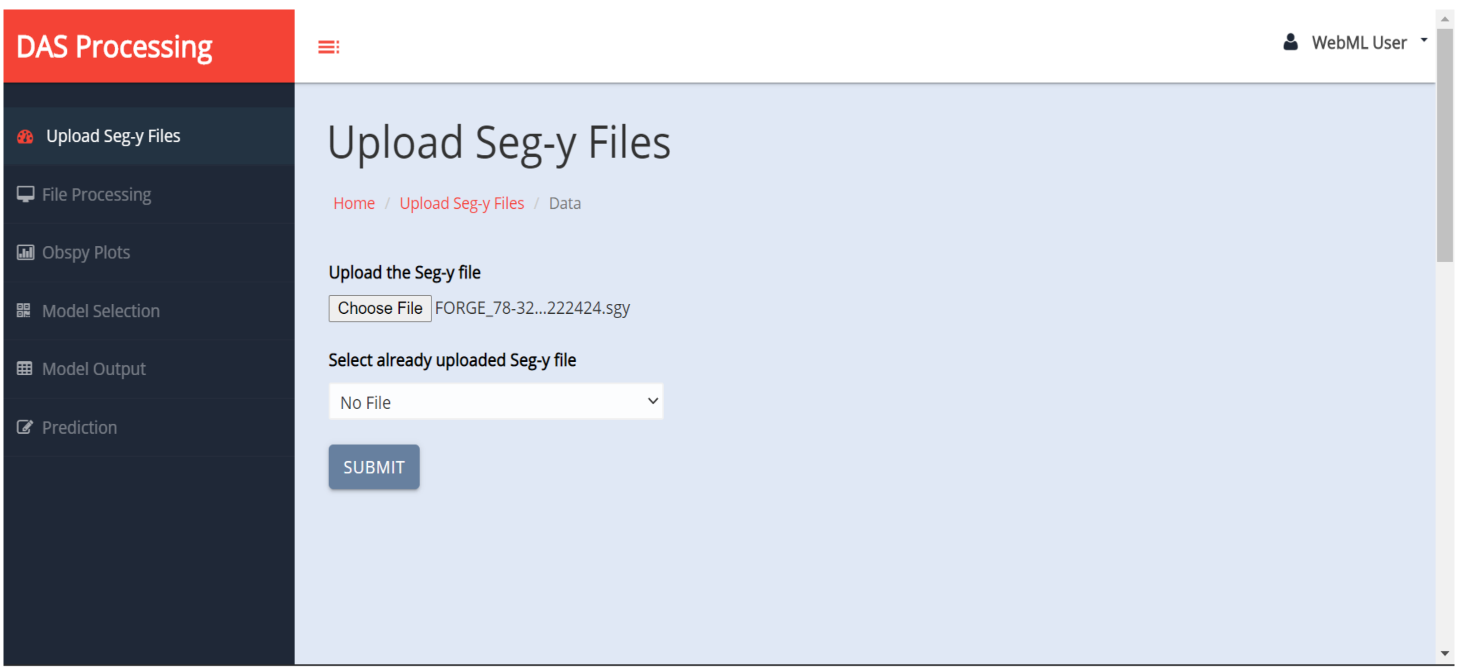
The WebML DAS front end portal is designed to serve the following two purposes:

1. Process the DAS data
2. Determine anomalies in DAS data

Process the DAS data

Step 1. Upload Seg-y files to server

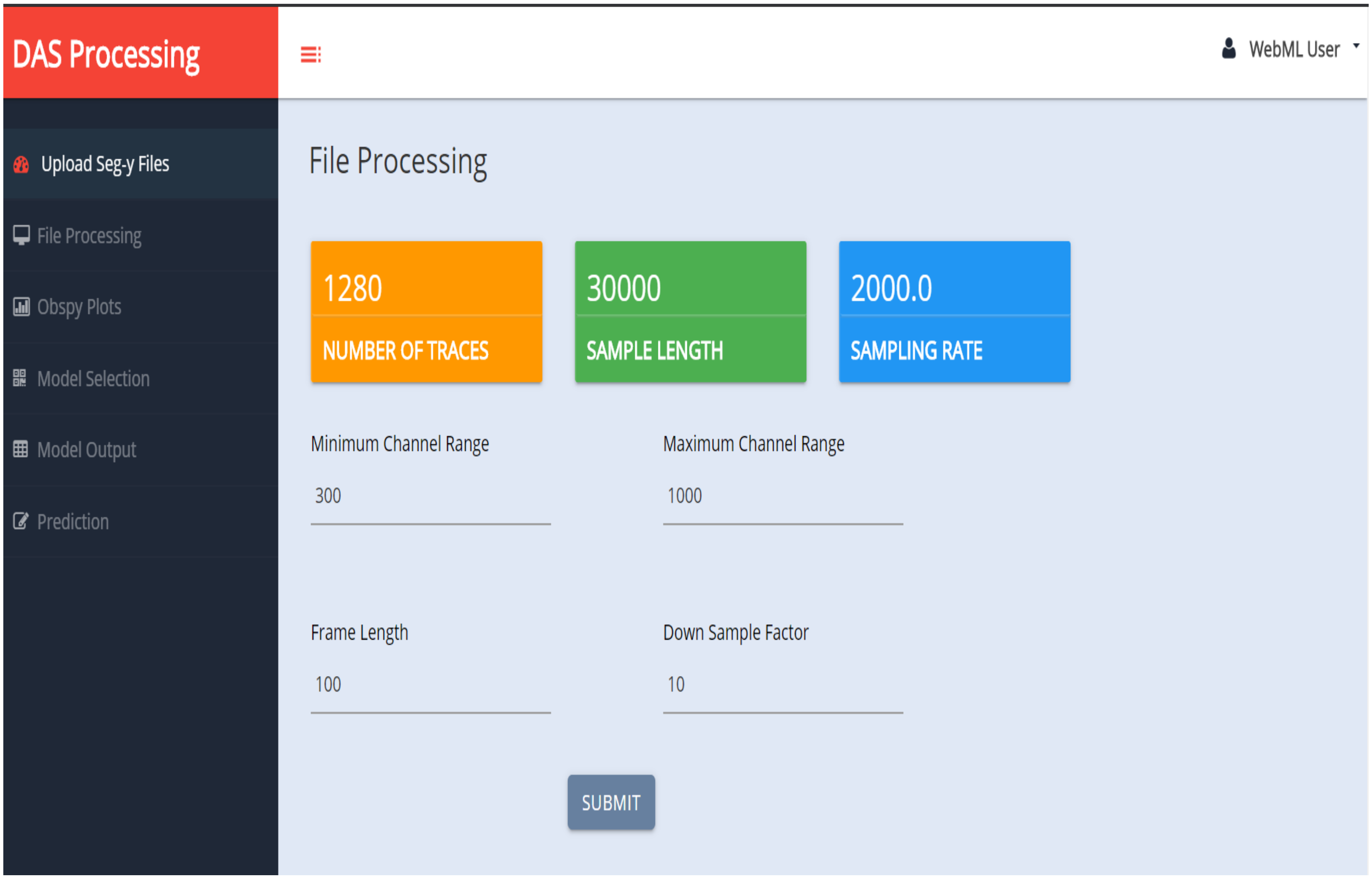
Use the “Choose File” option on the screen to upload the seg-y DAS data file that needs to be processed. Since file uploading takes time, the UI also has an option to select the already uploaded seg-y files. When we click the submit button on this page, an API endpoint “/process” is called that executes the function “processdata()”. This function reads the uploaded .sgy file and returns the number of traces, sample length of each trace and sampling rate for each trace. The below image is for “DASindex.html” from the templates folder.



Step 2. DAS File Processing for Generating different kinds of plots using Obsby Module

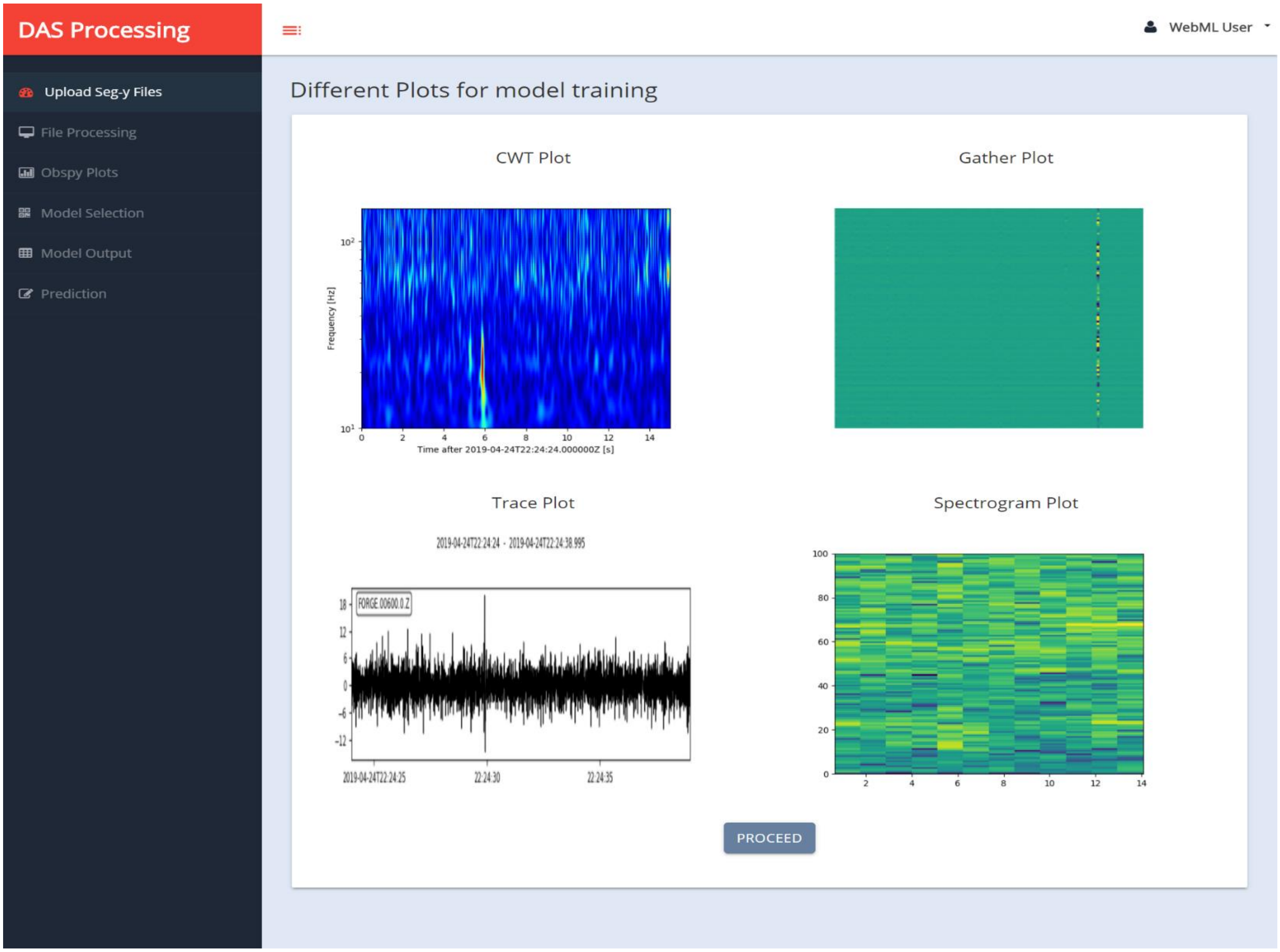
The following screen shows the details about the uploaded seg-y files. It has an input form that allows users to enter the minimum and maximum channel range, the required frame length and down sample factor. These details are used to generate different kinds of plots like Scalogram, Spectrogram, Trace and Gather plots for the input channel range. The

Obspy python module is used to generate all these plots. Once the user clicks the "Submit" button all the input details are given as input to the API endpoint "/model" which calls the function display_plots().



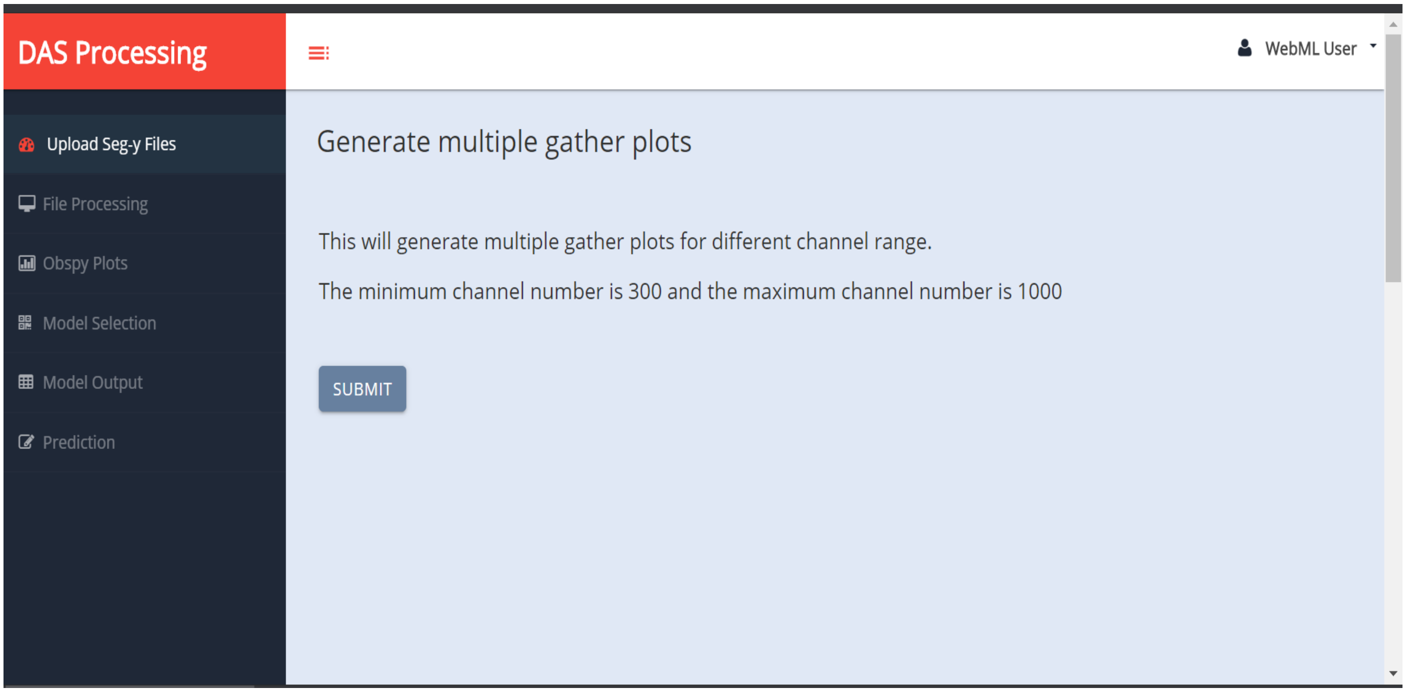
Step 3. Displaying the generated Obspy Plots

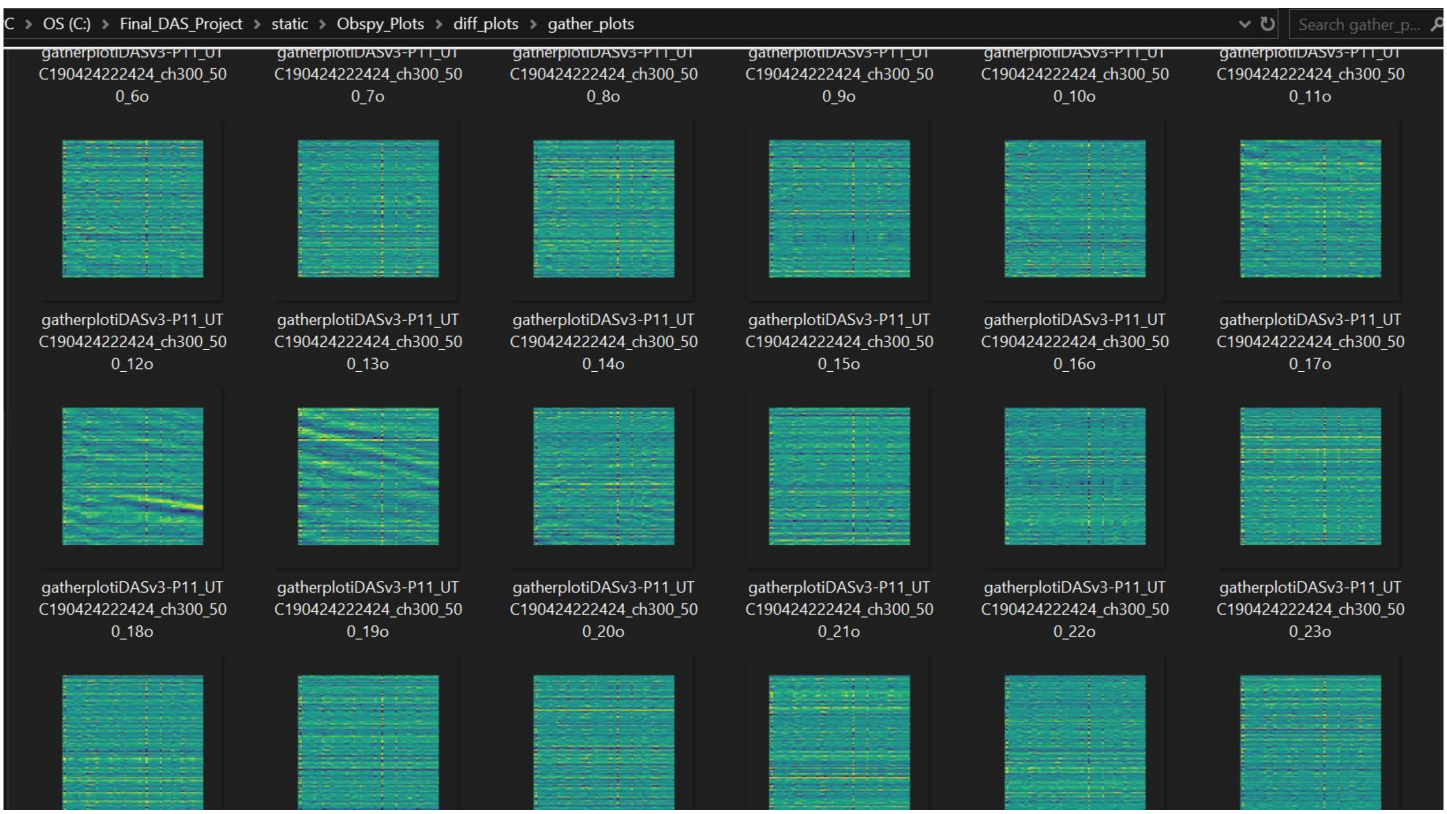
This screen shows the different types of plots that are generated using the inputs from the user related to the file processing. These plots can be used for training our model for a anomaly detection.



Step 4: Generating multiple gather plots from the uploaded seg-y files

In this step we are generating gather plots using random channel range. The channels that are used for generating these plots are in the randomly selected, which are in the range of min and max channel that was entered during the DAS Processing stage (Step 2). These gather plots are stored at the following location: *'static/Obspy_Plots/diff_plots/gather_plots'.* These plots are then used to generate clusters. This step takes a bit time to generate the plots arnd 50-65 secs.

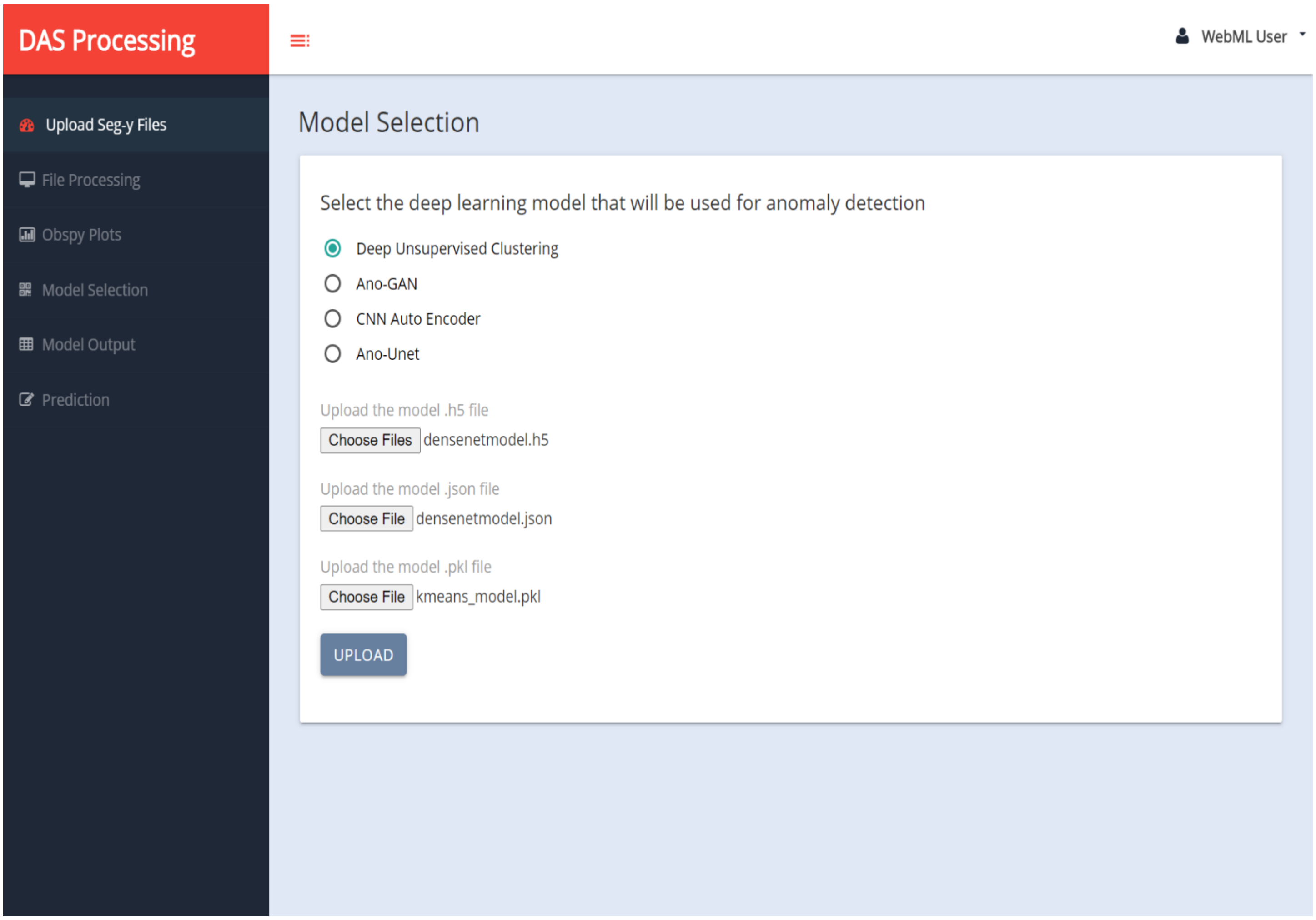

DAS Processing
WebML User
Upload Seg-y Files
File Processing
Obspy Plots
Model Selection
Model Output
Prediction
Generate multiple gather plots
This will generate multiple gather plots for different channel range.
The minimum channel number is 300 and the maximum channel number is 1000
SUBMIT


C › OS (C:) › Final_DAS_Project › static › Obspy_Plots › diff_plots › gather_plots
Search gather_p...
gatherplotiDASv3-P11_UTC190424222424_ch300_500_6o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_7o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_8o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_9o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_10o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_11o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_12o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_13o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_14o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_15o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_16o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_17o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_18o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_19o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_20o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_21o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_22o
gatherplotiDASv3-P11_UTC190424222424_ch300_500_23o

Step 5: Model Selection

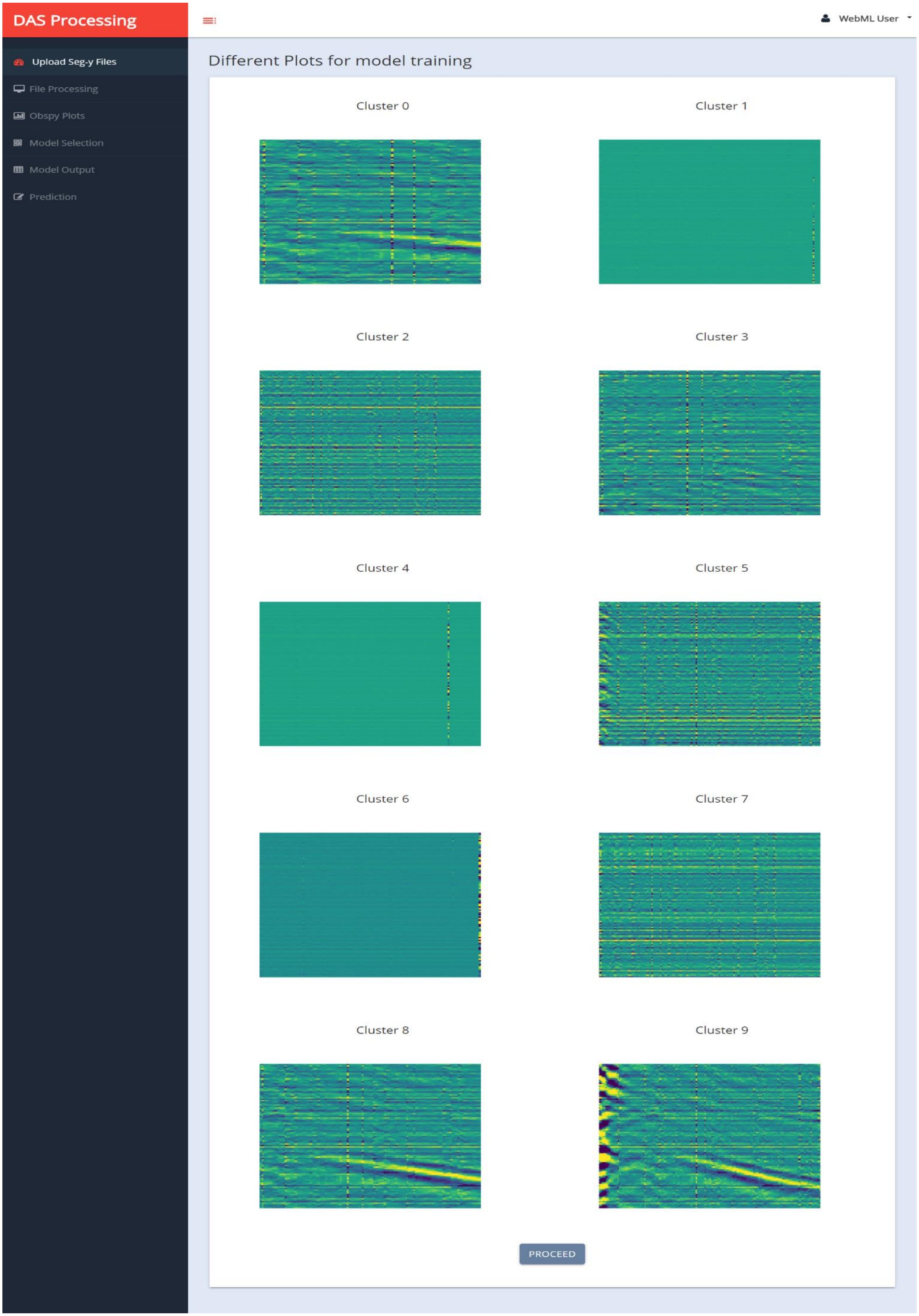
In this step, we can select which deep learning model we want to use for detecting the anomalies in the DAS data for earthquake detection. Currently we have used “Deep Unsupervised Clustering” for anomaly detection, in this we use the image vectors that are generated using the “imagenet” weights and the densenet model.

The GUI allows user to upload the trained model, for eg. In our case we can upload the .h5 and json file for the densenet model and the .pkl file for the Kmeans model.

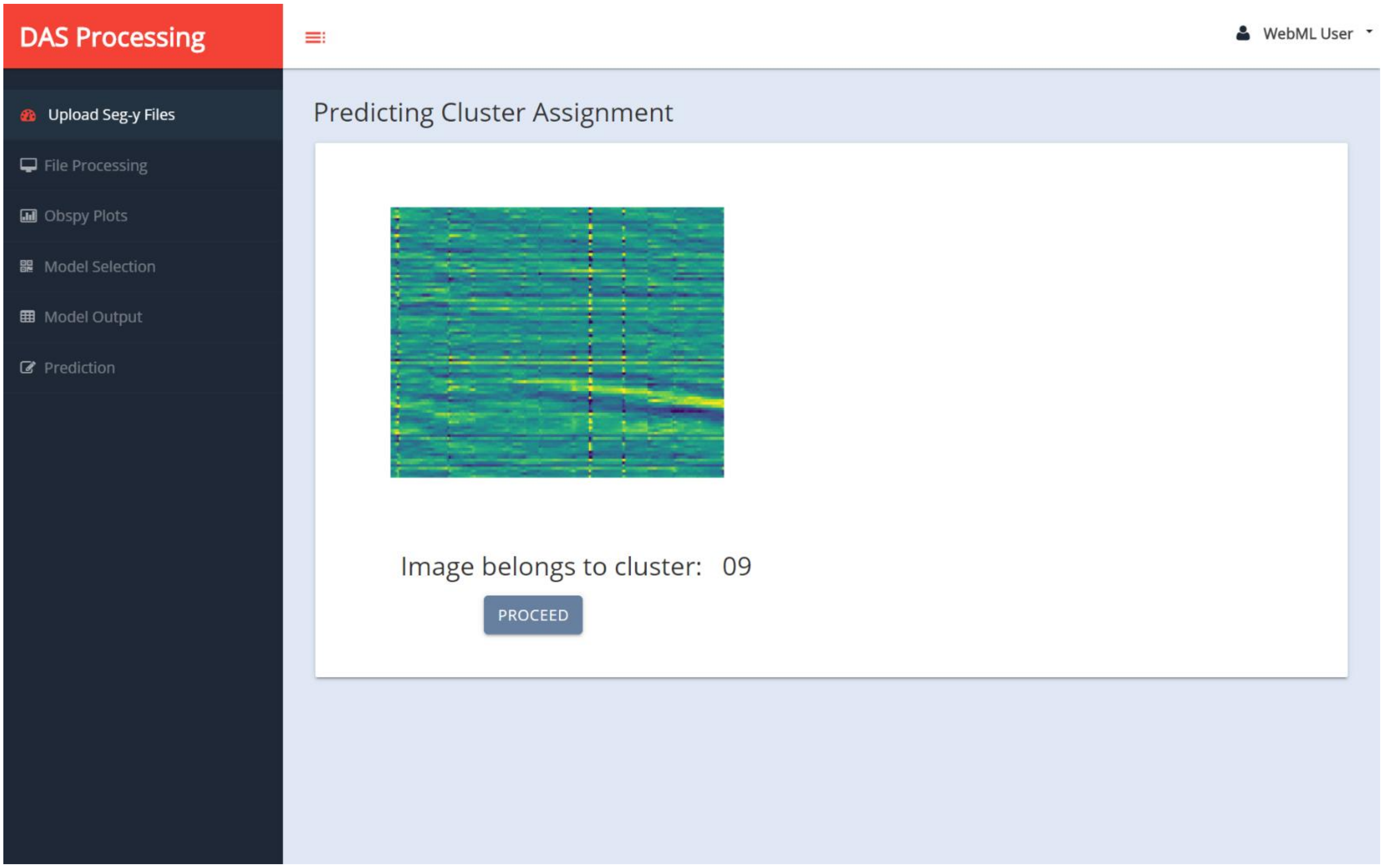


Step 6: Image clusters using gather plots

The image below shows a sample image from each cluster. After visual inspection cluster 0,3,8 and 9 are most likely to have gather plots that correspond to anomalies.


DAS Processing
WebML User
Upload Seg-y Files
File Processing
Obspy Plots
Model Selection
Model Output
Prediction
Different Plots for model training
Cluster 0
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
Cluster 7
Cluster 8
Cluster 9
PROCEED

Step 7: Predict the cluster assignment for a sample gather plot



6. Code Walkthrough for app_new.py

```
## Imports for Flask

from __future__ import division, print_function
from random import randint
import os
from time import strftime
from flask import Flask, render_template, flash, request
from wtforms import Form, TextField, TextAreaField, validators, StringField

# Flask utils
from flask import Flask, redirect, url_for, request, render_template
from werkzeug.utils import secure_filename
from gevent.pywsgi import WSGIServer
import random

################ import cluster py  ##################
import image_clustering
from image_clustering import *


############################# imports for obspy ################

import os
import matplotlib.pyplot as plt
import numpy as np
import argparse
import time, datetime
from obspy.io.segy.core import _read_segy
import obspy
```

1. At the start of the app_new.py file we include all the major python libraries that are used for DAS file processing and also those that are required for running the FLASK application. The above image shows some of the imports that are included in the file.

2. The image below shows the functions that are used for processing the DAS data and getting details for each trace List. We use the object of the class "Forge", to generate different kinds of plots.

```
######################################################################

def moving_avg(a, halfwindow, mask=None): ···


def filterSingleTrace(tr, *args): ···

def getSingleTrace(tr, dnRate, isIntegrate=False): ···

############## Main Class for file processing #####################

class Forge(): ···
```

```
class Forge():
    """
    Main class for loading and processing Forge data
    Forge has 1088 channels, sampling Rate 2000
    From Forge training,
    gaugelength     = 10.0
    dx_in_m         = 1.02
    das_units       = 'n$\epsilon$/s'
    geophone_units  = 'm/s^2'
    geophone_fac    = 2.333e-7
    fo_start_ch     = 197
    """
    def __init__(self, segyfile, ···


    def load(self, segyfile): ···

    def _getGather(self): ···
```

3. Below function is used to generate scalogram plots. We include the folder location where we need to save the image that is generated.

```python
############################## Function for Scalogram Plot ##########################################

def getChannelScalogram(traceList, channelNo, channelStart, outfile, imagefolder='static/Obspy_Plots/diff_plots'):
    tr = traceList[channelNo]
    dt = tr.stats.delta
    f_min = 10
    f_max = 150
    npts = tr.stats.npts

    t = np.linspace(0, dt * npts, npts)

    scalogram = cwt(tr.data, dt, 8, f_min, f_max)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    x, y = np.meshgrid(
        t,
        np.logspace(np.log10(f_min), np.log10(f_max), scalogram.shape[0]))
    #ax.pcolormesh(x, y, np.abs(scalogram), cmap=obspy_sequential)
    ax.pcolormesh(x, y, np.abs(scalogram), cmap='jet')
    ax.set_xlabel("Time after %s [s]" % tr.stats.starttime)
    ax.set_ylabel("Frequency [Hz]")
    ax.set_yscale('log')
    ax.set_ylim(f_min, f_max)
    image_name = '{0}_scalogram_channel{1}.png'.format(outfile, channelNo+channelStart)
    print(image_name)
    plt.savefig('{0}/{1}_scalogram_channel{2}.png'.format(imagefolder, outfile, channelNo+channelStart))
    plt.close()

    return image_name
```

4. getChannelSpecgram() is used to generate spectrogram plots that we display on the UI. It takes the input from user such as channel start and outfile

```python
################# Function for Spectrogram Plot ####################
def getChannelSpecgram(datatype, traceList, outfile, channelStart, channelStep=10):
    assert(datatype in ['mat', 'segy'])
    if datatype=='segy':
        st = obspy.Stream(traceList)
        nTraces = len(st)
    else:
        raise Exception('not implemented')
    sampleRate = traceList[0].stats.sampling_rate
    print ('in spectrogram sampleRate=', sampleRate)
    window = 256
    nfft = np.min([256, len(traceList[0].data)])
    frac_overlap = 0.1
    img_list=[]

    for itr in range(0,nTraces,channelStep):
        F,T,SXX = signal.spectrogram(st[itr].data, fs=sampleRate, window='hann')
        S1 = np.log10(np.abs(SXX/np.max(SXX)))
        if DEBUG:
            plt.figure()
            plt.pcolormesh(T, F, S1)
            print (channelStart+itr)
            image_name = 'tracespectrogram_{0}_ch{1}.png'.format(outfile, channelStart+itr)
            print(image_name)
            img_list.append(image_name)
            plt.savefig('static/Obspy_Plots/diff_plots/tracespectrogram_{0}_ch{1}.png'.format(outfile, channelStart+itr))
            plt.close()

    return img_list[0]
```

5. Function to generate gather_plots

```python
######################## Fucntion for Gather Plot ################################

def getGatherPlot(datatype, traceList, sampleRate, outfile, channelStart, channelEnd,
                  winlen=1000, clim=None, outimagefolder='static/Obspy_Plots/diff_plots/gather_plots'):
        nTraces = len(traceList)
        #data length in the das file
        nperlen = len(traceList[0].data)

        gatherArr = np.zeros((nTraces,nperlen),dtype=np.float64)

        for itr in range(nTraces):
            gatherArr[itr,:] = traceList[itr].data

        gatherArr = np.flipud(gatherArr.T)
        vmin = np.nanmin(gatherArr)
        vmax = np.nanmax(gatherArr)
        print ('vmin', vmin, 'vmax', vmax)
        #if True scale to [-1,1]
        isScale = False
        if isScale:
            gatherArr =  (gatherArr-gatherArr.min())/(gatherArr.max()-gatherArr.min())

        if winlen>=nperlen:
            nFrames=1
        else:
```

```python
        if winlen>=nperlen:
            nFrames=1
        else:
            nFrames = int(nperlen/winlen)

        img_list=[]
        for iframe in range(nFrames):
            if DEBUG:
                vmin = np.nanmin(gatherArr)
                vmax = np.nanmax(gatherArr)
                t_ = (traceList[0].stats.endtime-traceList[0].stats.starttime)/nFrames
                dx_ = traceList[1].stats.distance - traceList[0].stats.distance
                extent = [0,len(traceList)*dx_/1e3,0,t_]
                xlabel = 'Linear Fiber Length [km]'

                plt.figure(figsize=(10,10))
                if clim is None:
                    plt.imshow(gatherArr[iframe*winlen:(iframe+1)*winlen,:],
                               origin='lower', vmin=vmin/10.0, vmax=vmax/10.0,
                               extent=extent,
                               aspect='auto')
                else:
                    plt.imshow(gatherArr[iframe*winlen:(iframe+1)*winlen,:],
                               origin='lower', vmin=clim[0], vmax=clim[1],
                               extent=extent,
                               aspect='auto')

                ax = plt.gca()
```

```python
            plt.figure(figsize=(10,10))
            if clim is None:
                plt.imshow(gatherArr[iframe*winlen:(iframe+1)*winlen,:],
                           origin='lower', vmin=vmin/10.0, vmax=vmax/10.0,
                           extent=extent,
                           aspect='auto')
            else:
                plt.imshow(gatherArr[iframe*winlen:(iframe+1)*winlen,:],
                           origin='lower', vmin=clim[0], vmax=clim[1],
                           extent=extent,
                           aspect='auto')

            ax = plt.gca()
            ax.axis('off')
            img_name = 'gatherplot{0}_ch{1}_{2}_{3}o.png'.format(outfile, channelStart, channelEnd, iframe)
            img_list.append(img_name)
            filename = '{0}/gatherplot{1}_ch{2}_{3}_{4}o.png'.format(outimagefolder,
                                                                      outfile,
                                                                      channelStart,
                                                                      channelEnd, iframe)
            plt.savefig(filename, transparent=True)
            plt.close()
    return img_list[0]
```

6. App configuration settings

- Include "SECRET_KEY": it can be any random string input. It is mostly required for encryption.
- App.config['UPLOAD_FOLDER'] has path to the folder where you need to store the images that are generated. It may also contain any files that are uploaded via the web app.
- The variables like framelen, dsfactor, minchannelrange etc are the global variables that are used later on for different purposes.

```python
################################## app functionality ###############################################

DEBUG = True
app = Flask(__name__)
app.config.from_object(__name__)
app.config['SECRET_KEY'] = 'SjdnUends821Jsdlkvxh391ksdODnejdDw'
SPEC_FOLDER = os.path.join('static', 'Obspy_Plots','diff_plots')
# GATHER_FOLDER = os.path.join('static','Obspy_Plots','gather_plots')
print(SPEC_FOLDER)
app.config['UPLOAD_FOLDER'] = SPEC_FOLDER
minchannelrange=""
maxchannelrange=""
framelen = ""
dsfactor = ""
pathh5 = ""
pathjson = ""
pklpath = ""
gather_full_filename = ""
```

7. App route for index page, the segy_files variable has the list of segy files that are shown in the dropdown in UI. You can change these file names to include other segy files for processing. Currently we just have the upload functionality.

```
@app.route("/", methods=['GET', 'POST'])
def segydata():
    form1 = request.form
    #segy_files = ['PoroTomo_iDAS16043_160321000521.sgy', 'PoroTomo_iDAS16043_160321000721.sgy', 'PoroTomo_iDAS16043_
    #return render_template('DASindex.html',form=form1,files=segy_files)
    return render_template('DASindex.html',form=form1)
```

8. Once we click the submit button for uploading the seg-y file, this function is called. This retrieves the filename and saves it on the server. It processes the files and gets the trace count, sampling rate and the number of samples per trace. These details are shown on the next HTML page “DAS_Process.html”

```
@app.route("/process", methods=['GET', 'POST'])
def processdata():
    global filename
    form = request.form
    if request.method == 'POST':
        f = request.files['file']
        filename = f.filename
        path = str(filename)
        f.save(path)
        # filter_type=['Low Pass','High Pass','Band Pass']
        trace_cnt, sampling_rate,npts = get_segy_details(filename)
        file_data={'tr_cnt':trace_cnt,'samp_rate':sampling_rate,'number_sample':npts}

    return render_template('DAS_Process.html',form=form,data=file_data)
```

9. After entering the input details for processing the DAS data like minchannelrange, maxchannelrange, framelen and downsampling factor we read it on the following function. This then calls the function “get_obspy_plots” , which returns the image name for all the different kinds of plots. We then get the file location for these plots and send them to the UI via dictionary (test_data).

```python
@app.route("/model", methods=['GET', 'POST'])
def display_plots():
    form = request.form
    print('##########testgen######')
    print(request.method)
    global gather_full_filename
    global minchannelrange
    global maxchannelrange
    global framelen
    global dsfactor
    if request.method == 'POST':
        print("----------process")
        minchannelrange=request.form['minchannelrange']
        maxchannelrange=request.form['maxchannelrange']
        framelen=request.form['framelen']
        dsfactor=request.form['dsfactor']
        #### generate gather_plots
        plot_details = get_obspy_plots(int(minchannelrange),int(maxchannelrange),int(framelen),int(dsfactor),str(file
        spec_full_filename = os.path.join(app.config['UPLOAD_FOLDER'], plot_details['specgram'])
        CWT_full_filename = os.path.join(app.config['UPLOAD_FOLDER'], plot_details['scalogram'])
        TF_full_filename = os.path.join(app.config['UPLOAD_FOLDER'], plot_details['trace'])
        gather_full_filename = os.path.join(app.config['UPLOAD_FOLDER'],'gather_plots\\' + plot_details['gather'] )
        print("full_filename is " ,gather_full_filename)

        test_data={'spec':spec_full_filename,'cwt':CWT_full_filename,'tf':TF_full_filename,'gather':gather_full_filen
        #get_gather_plots(int(minchannelrange),int(maxchannelrange),int(framelen),int(dsfactor),str(filename))
    return render_template('plots.html', form=form, data = test_data)
```

10. The following app route are for generating multiple gather plots for different channel ranges within the same uploaded file. These plots may be used for deep clustering. Once the plots are generated, we render the HTML that allows user to upload different models for deep learning.

```python
@app.route("/gatherplots", methods=['GET', 'POST'])
def get_multiple_gatherplots():
    form = request.form
    form_data = {'minchannel':minchannelrange,'maxchannel':maxchannelrange}
    return render_template('multiple_gather_plots.html',form=form,data=form_data)

@app.route("/plots", methods=['GET', 'POST'])
def model_upload():
    form = request.form
    getMultipleGatherPlots(int(minchannelrange),int(maxchannelrange),int(framelen),int(dsfactor),str(filename))
    return render_template('model_upload_UI.html',form=form)
```

11. In this step we read the files that were uploaded during the model upload step and save it in the model_uploads folder on server. The variables pathh5, pathjson, pklpath are global variables which can be used later on while using the uploaded files for prediction.

```python
@app.route("/predict",methods=['GET', 'POST'])
def gen_image_clusters():
    global pathh5
    global pathjson
    global pklpath

    if request.method == 'POST':

        h5file = request.files['h5file']
        h5filename = h5file.filename
        pathh5 = 'model_uploads/' + str(h5filename)
        print(pathh5)
        h5file.save(pathh5)

        jsonfile = request.files['jsonfile']
        jsonfilename = jsonfile.filename
        pathjson = 'model_uploads/' + str(jsonfilename)
        print(pathjson)
        jsonfile.save(pathjson)

        pklfile = request.files['pklfile']
        pklfilename = pklfile.filename
        pklpath = 'model_uploads/' + str(pklfilename)
        print(pklpath)
        pklfile.save(pklpath)


    return render_template('image_cluster.html')
```

12. In the following code we read the uploaded models and pass it to a function used for predicting the cluster assignment. Currently, I have commented off this code and hardcoded the cluster assignment as I am getting unpickling error.

```python
@app.route("/getcluster", methods=['GET', 'POST'])
def predict_cluster():
    input_gather = 'static/Obspy_Plots/diff_plots/predict'
    kmeans_model = pathh5
    densenet_json = pathjson
    densenet_h5 =  pathh5

    # result = predicting_cluster(input_gather, str(kmeans_model), str(densenet_json),str(densenet_h5))
    #result = 9
    return render_template('predict_cluster.html')




if __name__ == "__main__":
    app.run(use_reloader=False)
```

HTML Code Changes

In the form tag, we enter the action that needs to be performed when we click the submit button for that particular form. In this case, we enter all the input details like min and max channelrange etc. Once we click the submit button "display_plots()" function is called from the app.py file.

```
<div class="row">
    <form action="{{ url_for('display_plots') }}" method="POST">
        {{ form.csrf }}
        <div class="form-group" >
            <div class="col-md-12">
              <div class="row">
                <div class="col-md-3">
                  <label for="minchannelrange" style="color: black;">Minimum Channel Range</label>
                  <input type="number" required  id="minchannelrange" name="minchannelrange" placeholder="Min Chann
                </div>
                <div class="col-md-1"></div>
                <div class="col-md-3">
                  <label for="maxchannelrange" style="color: black;">Maximum Channel Range</label>
                  <input type="number" required  id="maxchannelrange" name="maxchannelrange" placeholder="Max Chann
                </div>

              </div>
```