Final Project Phase 2 Step-by-step Guide: list datafiles and run analytics

ECE 157A TAs

Fall 2020

1 Django Setup

- Verify python installation with Python 3.6 or later.

```
python --version
```

- Change your terminal to your project directory

```
cd path/to/YOUR_PROJECT_DIRECTORY
```

- OPTIONAL: Activate your virtual environment, if used for phase1

```
source env/bin/activate
# On Windows use `env\Scripts\activate`
```

  - You can also activate virtual environment by selecting python interpreter on Vscode.
  - Possible error:
    `https://stackoverflow.com/questions/4037939/powershell-says-execution-of-scripts-is-disabled-on-this-system`

- Make sure `manage.py` is present in your terminal's current working directory

```
ls manage.py
```

  If it gives you "no such file or directory" as an error, you may need to change directories again.

- Run the following command and go to `http://127.0.0.1:8000/` on your browser. You should see your page from phase1.

```
python manage.py runserver
```

2 Implement file listing

- First, we're going to add some helper functions that retrieve useful lists of things from your database. In `views.py`, we'll define some new methods above the `YourViewName` class. First, let's make a function to get every file in our database:

```
def get_file_list():
    files=FileModel.objects.values_list('file_name',flat=True)
    return files.distinct()
```

- Now that we have a list of files, we can do a similar thing for our list of algorithms. If you setup a database model for algorithms in the previous phase, you may make a helper method similar to the above. Otherwise, we'll use a stub like:

```
def get_algorithm_list():
     return ['PimaDiabetesClassifier','WineQualityRegressor','NBAOutlier']
```

- To make these lists show up, we need to provide them to Django REST framework's TemplateHTMLRenderer – to our HTML templates. We do this by providing a python dictionary, which the template can use to lookup variable names. You'll have one already from the phase 1 `post` method; note how the successful path has the argument {`'status': 'Upload successful!'`} in its call to `Response`? That's a dictionary, with `'status'` as its only key and the string `'Upload successful!'` as its only value.

  Code-wise, we can add this to the `get` method by replacing its return line like so:

```
return Response({'files': get_file_list(),
                 'algorithms': get_algorithm_list()},
                 status=status.HTTP_200_OK)
```

  It's up to you to add this dictionary to the other `Response` calls!

- As we noted a number of people struggling with HTML and the Django templating language in the last phase, we've again provided the requisite templates. Place `analytic_template.html` next to your `index.html` template, then add the line

```
{% include 'YOUR_APP_NAME/analytic_template.html' %}
```

  to your `index.html`, after the `<hr/>` tag. (Remember to correct the apostrophes if you copy/paste!) We'll cover a small piece of HTML and the template language later on, for showing the results of your analyses.

3 Test file listing

- Start the server and go to local host at `http://127.0.0.1:8000/`. You should be able to see the file uploading page, now with an extra interface giving you a list of uploaded files and processes to choose from.

```
python manage.py runserver
```

- If none of your CSVs are uploaded, try uploading one!

4 Formatting algorithms

4.1 Undergrads

- To render our algorithms' results, we'll be using the MPLD3 library. Lets' install it now:

```
python -m pip install mpld3
```

- Now, we'll create some extra python files for our models. You can just drop them in the folder `YOUR_APP_NAME` with your other python scripts. Make one now, into which we'll put our model from homework 1.
- In this new file, copy in whatever code you needed in homework 1 to create your model. Also `import mpld3` at the top of the file. Then, in place of your code for handling the `unknowns.csv`, define this new function:

```
def run_algo1(file_path):
    # Load file_path instead of unknowns.csv

    # Do the same processing you did to
    # get your unknowns.csv results from your model.
```

```
    # Get the figure object Matplotlib will be working with:
    fig = plot.figure()

    # Scatterplot your results, as per
    #   any of your scatterplots from homework 1.
    # Make sure to use plt.xlabel(), plt.ylabel()
    # and plt.title() to give clear labeling!

    return mpld3.fig_to_html(fig)
```

The line using `mpld3` converts the Matplotlib figure to HTML, which we can stick into the browser page during the template rendering!

- You can test this function by copying `unknowns.csv` into your `YOUR_APP_NAME` folder and adding the lines

```
run_algo1('unknowns.csv')
plt.show()
```

to the end of the file with no indentation.

- Repeat this subsection for homework two and homework three. For homework three, you'll just be running the outlier model on whatever data file is given – not loading the old file to generate the model.

- NOTE: **MPLD3 does not work for plots with** `projection='3d'`. For these, *only if you're an undergrad*, get around this by using the snippet:

```
with io.StringIO() as stringbuffer:
    fig.savefig(stringbuffer,format='svg')
    svgstring = stringbuffer.getvalue()
return svgstring
```

Grads, you'll be using another plotting library that has interactive 3D HTML outputs.

4.2 Grads

- You'll also need to upload and manage analytic algorithms on the administrator site. First we'll create a new model called AlgorithmModel to store our algorithms:

```
class AlgorithmModel(models.Model):
    algorithm_name = models.CharField(max_length=50)
    inference_script = models.FileField(upload_to='algorithms/scripts/')
    saved_model = models.FileField(upload_to='algorithms/saved_models/')

def __str__(self):
    return self.algorithm_name

def delete(self, *args, **kwargs):
    self.inference_script.delete()
    self.saved_model.delete()
    super().delete(*args, **kwargs)
```

Our AlgorithmModel contains three fields. A character field for the algorithm name, one file field for the inference script, and one file field for the trained model. Note: inference scripts are the code used to predict the test data. The trained model is a pickle dump of the trained classifier (see details here.)

- Now we'll create an user who can login to the admin site:

```
python manage.py createsuperuser
```

A prompt will show up asking us for Username, Email address, and Password. The Username and password are your login credential, make sure to remember it.

- Now, we can run the server and visit the admin site at http://127.0.0.1:8000/admin/. You will need to login in with the credential you just created.
- By default, you are only able to modify two objects: Groups and Users, but we are interested in modifying our AlgorithmModel objects. To get access to the algorithm objects, we need to tell the admin that algorithm objects have an admin interface. To do this, open the YOUR_APP_NAME/admin.py file, and edit it to look like this:

```
from django.contrib import admin
from .models import AlgorithmModel

admin.site.register(AlgorithmModel)
```

- Now that we've registered Algorithm, Django knows that it should be displayed on the admin index page. Click "AlgorithmModel". Now you're at the "change list" page for algorithms. This page displays all the algorithms in the database and lets you choose one to change it.

- Let's create our first algorithm, by clicking "ADD AlgorithmModel" on the top-right side. You will be redirected to a form page. The form is automatically generated from the AlgorithmModel. Here, we can give our algorithm a name, upload its inference script and saved model. Click "SAVE" after you have done so.
- Now we have uploaded our first algorithm, we can follow the previous step to create our next two algorithms. Then, we can display the algorithms to the client side and access the upload algorithm files using the same procedure as we did for the FileModel objects.

5 Run algorithms

- Now that we're able to display this second form, asking the user to select a file and algorithm, we need to be able to receive that form's result!
- Go back to `views.py`. There, in your `post` method, add

```
if 'upload' in request.data:
```

before your existing code in the method, and indent your existing code by one tab. Then, after your existing code, we can add the analytic form handling:

```
elif 'analytic' in request.data:
    # Run analytics on dataset as specified by file_name and
    # analytic_id received in the post request
    query_file_name = request.data['file_name']
    query_algorithm = request.data['algorithm']

    # Find file path to local folder
    file_obj = FileModel.objects.get(file_name=query_file_name)
    file_path = file_obj.file_content

    # Find algorithm
    algo_obj = get_algorithm(query_algorithm)

    analyticresult = run_analytic(file_path, algo_obj)

    return Response({'files':get_file_list(),
                     'algorithms':get_algorithm_list(),
                     'result_plot':analyticresult},
                     status=status.HTTP_200_OK)
```

- In this code, we've left two functions undefined: `get_algorithm` and `run_analytic`.

  For non-graduate students, `get_algorithm` can just be an if-elif ladder, returning the name of the right function to call from your old assignments (which you can simply copy into the same folder and import.) That is,

```
def get_algorithm(name):
    if name == 'MyHW1Algo':
        from .myhwone import run_algo1
        return run_algo1
    elif name == 'MyHW2Algo':
        from .myhwtwo import run_algo2
        return run_algo2
    elif name == 'MyHW3Algo':
        from .myhwthree import run_algo3
        return run_algo3
    else:
        # Http404 is imported from django.http
        raise Http404('<h1>Target algorithm not found</h1>')
```

  For graduate students, `get_algorithm` will likely look like the line to get the `file_obj`.

  `run_analytic` for undergrads then will look like

```
def run_analytic(file_path, algo):
    file_abs_path = os.path.join(settings.MEDIA_ROOT, str(file_path))
    return algo(file_abs_path)
```

  and that's it! Assuming your `get_algorithm` is returning a function from one of your old assignment files, you're set. We covered exactly how to format these functions in the previous section.

- **Graduate student hint:** Your `run_analytic` will look like:

```
def run_analytic(file_path, algo_object):
    algo_model = algo_object.saved_model
    algo_script = algo_object.algorithm_script
    file_abs_path = os.path.join(settings.MEDIA_ROOT, str(file_path))
    algo_model_path = os.path.join(settings.MEDIA_ROOT, str(algo_model))

    algo_model_import_path = str(algo_script).\
        replace('/','.').replace('\\', '.')[:-3]
    algo_model_import_path = 'media.'+algo_model_import_path
    print('Running algorithm from: ',
```

```
            algo_model_import_path)

        algo_model = __import__(algo_model_import_path,
                                fromlist=[algo_model_import_path])

        return algo_model.run(file_abs_path, algo_model_path)
```

- **Graduate students:** you'll need to save the `analyticresult` into a list of history entries. This saving process, just before your return in `post`, will look like

```
to_save = {'analytic_name': query_file_name+'_'+query_algorithm,
           'result_plot': analyticresult}
analytic_serializer = AnalyticSerializer(data=to_save)

if analytic_serializer.is_valid():
    analytic_serializer.save()
    # The return statement as before
else:
    # HttpResponse is imported from django.http
    return HttpResponse('The server encountered an internal error'
                        +'while processing'+to_save['analytic_name'],
                        status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

  Note how a serializer helps you generate the model object, here!

6 Rendering the resulting plot

- In debugging, you may have seen the large blobs of HTML that pop out of `mpld3.fig_to_html()`. Now we'll cover how to get those onto your page!
- We covered passing this string of html to our template in the definition of `run_analytic` and `post`, back in section 5. Now, go to `index.html`, so we can add the template code we need to display it.
- Head to the bottom of `index.html`, just before the {% endblock %}. Add:

```
{% if result_plot %}
<div class="text-center">
    <center>
    <h5>result</h5>
    {{ result_plot | safe }}
    </center>
</div>
```

```
{% endif %}
```

- {% if ... %} blocks in the Django template language let you check if a variable is defined. Technically, it checks the "truthiness" of the value. Empty lists will act like `false` or `None` for this check – the HTML between the `if` and `endif` (or between the `if` and `else`, if a `else` is present) will cease existing.

- `<div>` tags just define a region of the HTML page – an invisible box around elements. The `class="text-center"` bit tells "bootstrap" to make all text in child elements of the tag centered.

- `<center>` makes a tag centered inside of the surrounding tag. This tag exists because we give you the "bootstrap" styling package; it isn't a part of base HTML.

- `<h5>` forms a small-ish header, here saying just "result"

- {{ `result_plot | safe` }} accesses and prints into the HTML the variable `result_plot`, if it was provided to the template renderer. The `| safe` bit forces Django to include it into the HTML as raw text. Normally, Django will put in extra metadata to make sure the text renders as text, rather than as HTML. This bypasses that behaviour.

- {% `endif` %} closes the prevous {% `if ...` %} statement.

- With all of this included, you should now be able to run and see results from analytics.