

Creating an HTML/Javascript UI for a Data Science Module

Introduction

This document is intended for software developers who will assist Data Scientists in building a complete Data Science web application. This document demonstrates how an HTML/Javascript User Interface (UI) can be used as a front end for a Data Science module. The UI will allow the user to interact with the Data Science module from building the model through using the model for predictions. The project associated with this document is a proof of concept that once a Data Scientist writes a model with the appropriate APIs (As explained in the document called *DataScienceModules.pdf*), a software developer can create a UI that allows the user access to the model, and this model together with the UI form a complete web application.

Why a Web Application?

If we wish to make an application available to users, we have three basic choices:

1. Install the application on the users' machines (Desktop app).
2. Install a UI on the users' machines and make REST calls to services hosted on a server.
3. Make the application available as a web app.

Choice (1) involves configuration of every user's machine. Experience has shown that software developers would like to avoid configuration whenever possible. The more users, the more configuration. Any changes to the UI or business logic requires reinstallation of the application on the user's machine. This choice is just not acceptable when you have a large number of apps to deploy. Most importantly, this choice is not able to make sufficient use of the accepted standard of a CI/CD pipeline for development and deployment.

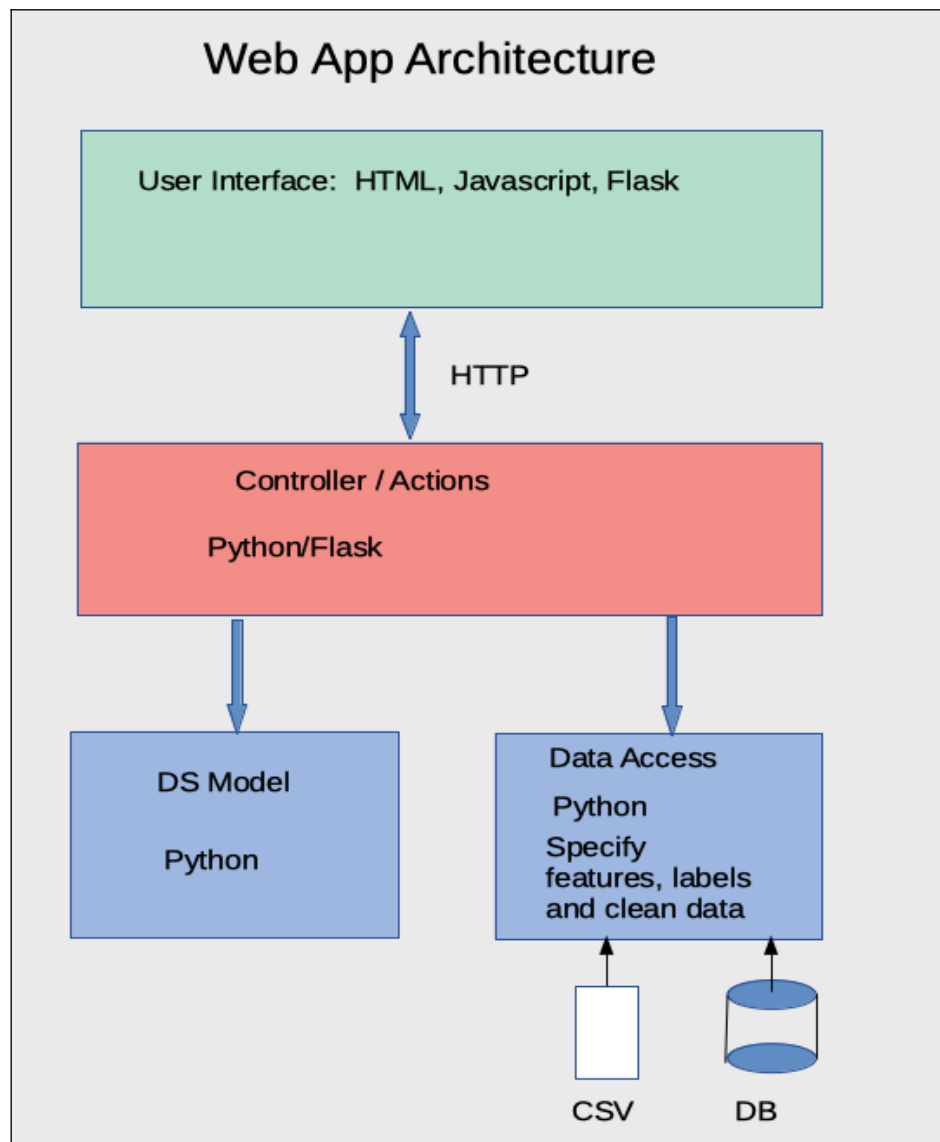
Choice (2) still brings configuration issues, but to a much lesser extent since the dependencies to support a UI are less numerous than dependencies to support a whole application. The backend REST services can make use of a CI/CD pipeline, but the front end UI can not make use of the CD part of the pipeline since it is installed on the user's machine.

Choice (3) is the most attractive approach because it allows user access to the application simply through a web browser. There is no client side configuration because all browsers contain support for HTML and Javascript. The Data Science and Data Access modules can be written as either REST services, or as API's that can be called directly by object instantiation. Of course, development and deployment of both the UI and backend modules can make use of the CI/CD pipeline.

The Application Architecture

The architecture that we use was discussed in the document *DataScienceModules.pdf*. For completeness we will reproduce the diagram of the architecture to the right.

There are four components or modules that make up the application. The two blue modules represent the code that the Data Scientist would write. The blue modules are **completely independent from the context in which they are used**. That is they have no dependence upon the web framework in which they are used, and they have no dependence upon the Controller/Actions as well as the UI. Simply stated, the blue modules can be used in any application context. They could be used locally in a desktop application, they could be used in a cloud application, they could be used in any web framework, they could be used in combination with any UI.



Web Application Frameworks

All web applications must make use of a web framework that offers support for the following:

- Handling HTTP requests and responses
- Handling of URL/HTML page mappings
- Handling of URL/Controller Action mappings

In all web applications, an action is triggered by the URL that appears in the HTTP request. The web application framework maps URLs to functions or class methods that carry out the actions.

For Python web applications there are many web frameworks that are available. To name a few, we have: Django, Pyramid, TurboGears, Web2Py and Flask. In this document we will use Flask. Flask is a lightweight web framework that is simple to use for development, but when put into an OpenShift container, can also be used in production.

The Flask Web Application Framework

Since Flask is a simple framework, we can put to work most of the support mentioned above into one .py file, usually named application.py, or sometimes just app.py. The most minimal Flask application looks like:

```
1  from flask import Flask
2
3  my_app = Flask(__name__)
4
5  |
6  @my_app.route('/')
7  def hello_world():
8      return 'Hello World!'
9
10
11  if __name__ == '__main__':
12      my_app.run(port=5002, debug=True)
```

Assuming that we have installed the **flask** library, we can import the Flask class and instantiate a Flask object that we call **my_app** on line 3. The global variable, **__name__** is the usual Python variable that holds the name of the Python application when it is run. When we run app.py from the command line with: **python app.py**, the variable **__name__** gets assigned the value **'__main__'**. This is a Python characteristic, not a Flask one.

Line 11 checks to see if the Python file is being run with the Python interpreter. If so (ie. `__name__ == '__main__'`), then we call on Flask's method, `run()`. The two parameters, **port** and **debug** are set to 5002 and True respectively. The newly started web server will listen on port 5002 for HTTP requests, and `debug = True` allows us to use the runtime debugger on the application.

Line 6 uses a Python decorator on the function **hello_world()**. Recall that a Python decorator is itself a function that calls upon the function that it decorates. In addition, our decorator called **route** receives one parameter whose value is `'/'`. The value of this parameter, `'/'` is called an **endpoint**. The decorator maps the endpoint to the code (**hello_world()** function) that will be executed when the endpoint is recognized. A web application will have many endpoints, each one representing some action that will be carried out by the application.

In our simple hello app, the decorator `@my_app.route('/')` is equivalent to the code:

```
my_app.add_url_rule('/', 'hello_world', hello_world)
```

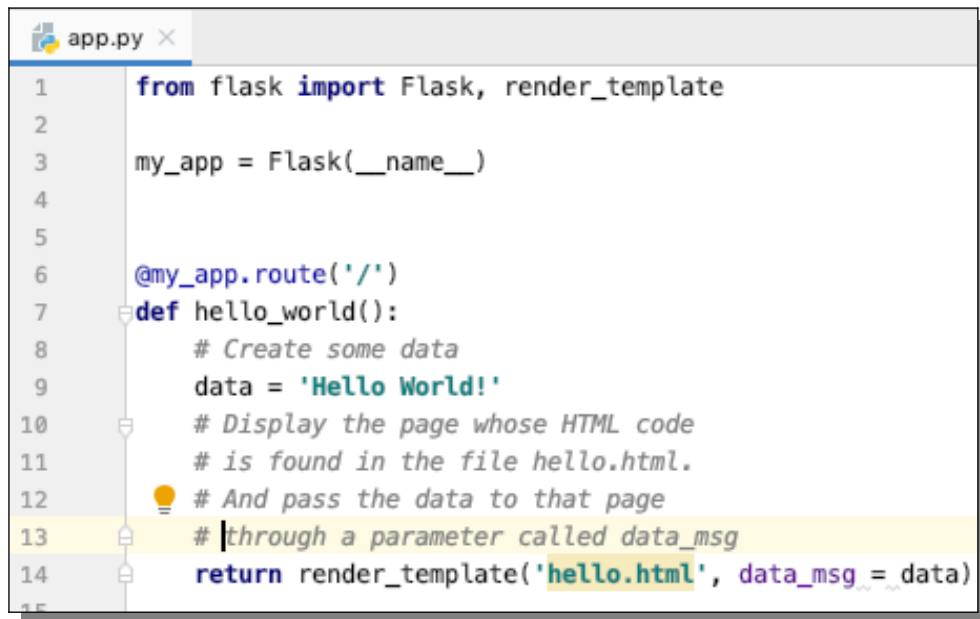
So the decorator just adds a mapping(Python dictionary) to the Flask object. The key used in the mapping is the string name of the function that gets decorated while the value in the mapping is the function name. For a deeper explanation of the decorators used in Flask, please use this [link](#).

If we were to run the above code from the command line with: **python app.py**, the web server starts and listens on port 5002. We can view the simple results of our app by opening a browser and typing: **localhost:5002**

You will see the output text: "Hello World!" in the browser. Note that the browser only received the string, "Hello World!" which is not proper HTML, so the browser created the proper HTML tags and placed the string into the body of the newly created HTML document.

Before moving on to look at the full web application, let's do a small modification to the "Hello" web app that will give us some additional insight. Instead of relying on the browser to build some HTML that wraps our message, let's create our own HTML page and pass the output of the function **hello_world()** to it. In doing so, we will see how to render application generated data into a specified HTML page.

Below is our new app.py with the modifications that will render our own HTML page with data that we have generated:



```

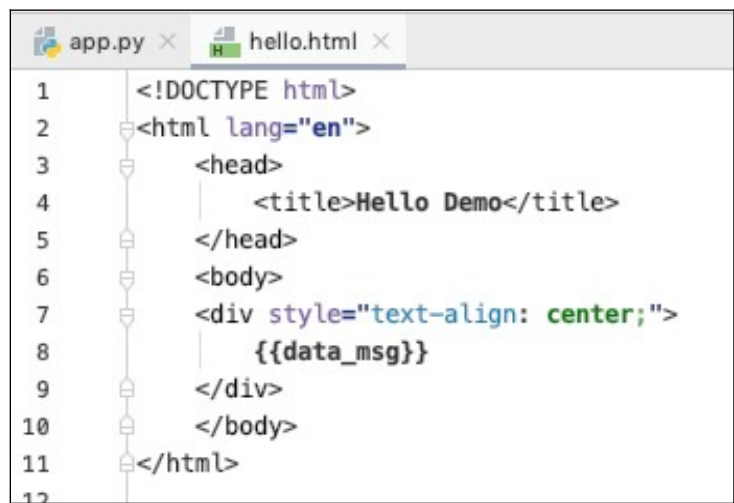
1  from flask import Flask, render_template
2
3  my_app = Flask(__name__)
4
5
6  @my_app.route('/')
7  def hello_world():
8      # Create some data
9      data = 'Hello World!'
10     # Display the page whose HTML code
11     # is found in the file hello.html.
12     # And pass the data to that page
13     # through a parameter called data_msg
14     return render_template('hello.html', data_msg = data)

```

On line 1 we must import the **flask** function **render_template**. On line 9 we create some data. In a real Data Science application we would call upon a module that returned some prediction data from a DS model. On line 14 we make a call to the flask method, **render_template**. We pass two parameters to the **render_template** function. The first parameter is the name of our new HTML page. In the second parameter, we define a parameter that we call **data_msg**, and pass it the value of our newly created **data**. The function **render_template** will cause the page whose HTML is in **hello.html** to be displayed, and at the same time the value of the parameter **data_msg** will be passed to the page.

All we have left to do is to write the HTML for **hello.html**:

Note that on line 8 we insert the value of the variable, **data_msg** that was passed as a parameter in the **render_template()** function call. The mechanism that allows us to pass parameters to an HTML page where we use the symbols `{{ variable name }}` to display the variable value is called Jinja2. [Jinja2](#) ([Another useful link](#)) is a template engine that works with Flask. Other web frameworks have similar template engines.



```

1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <title>Hello Demo</title>
5      </head>
6      <body>
7          <div style="text-align: center;">
8              {{data_msg}}
9          </div>
10     </body>
11 </html>

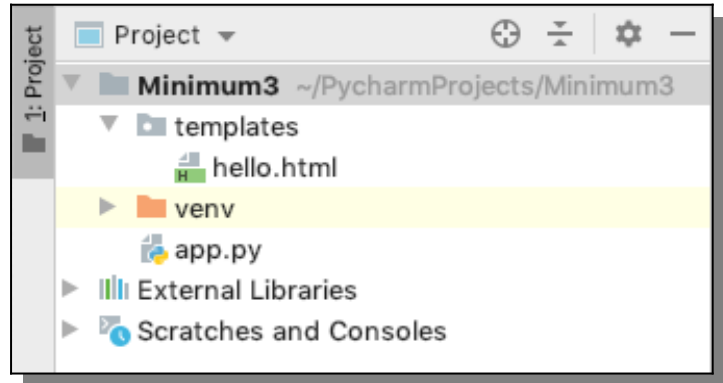
```

When the flask method, **render_template()** gets called, the template engine, Jinja2 first parses the html code and inserts the value of the parameter **data_msg** into the html code. Then the html page is rendered. This mechanism is similar in theory to how Java JSPs are rendered. That is, the JSP is translated into HTML on the server before it is rendered.

Note that Flask by default looks for HTML files in a project directory called **templates**. In the case of our minimal hello app, the directory tree would look like:

For now all we care about are the two files, `hello.html` and `app.py`.

Before we leave the introductory “hello” applications, let’s summarize the mechanism that enabled us to type in a URL into a browser, and the web app rendered a page that displayed some data.



1. We first entered a URL in a browser: **localhost:5002**
2. The browser creates an HTTP request that contains the URL **localhost:5002/**
3. The Flask web server looks up the endpoint, '/', finds that the endpoint is mapped to the function **hello_world()** and executes that function.
4. The function **hello_world()** creates some data and stores the data in the variable, **data**.
5. The function then executes flask's **render_template()**, passing the name of the web page, 'hello.html' and passes the data as a parameter known as **data_msg**.
6. The browser renders 'hello.html', with the value of the parameter, **data_msg** inserted into the document where specified.

So all of the URLs that a web application responds to are mapped to decorated functions in the web application's main script, `app.py`. This implies that the main script (The Python file that starts the server) reads like an action menu and a reader of `app.py` would have an excellent idea of the actions that the web application actually carries out.

The Demo Application

The only differences between our “hello” web applications and a Data Science app is that there are more action mappings, and the Data Science modules process and generate prediction data. The demo app that we will use is called **Titanic2** and can be downloaded from Git.

We will assume that the Data Science modules have been written and the appropriate APIs have been exposed (See *DataScienceModules.pdf*).

The demo app allows the user to do the following actions:

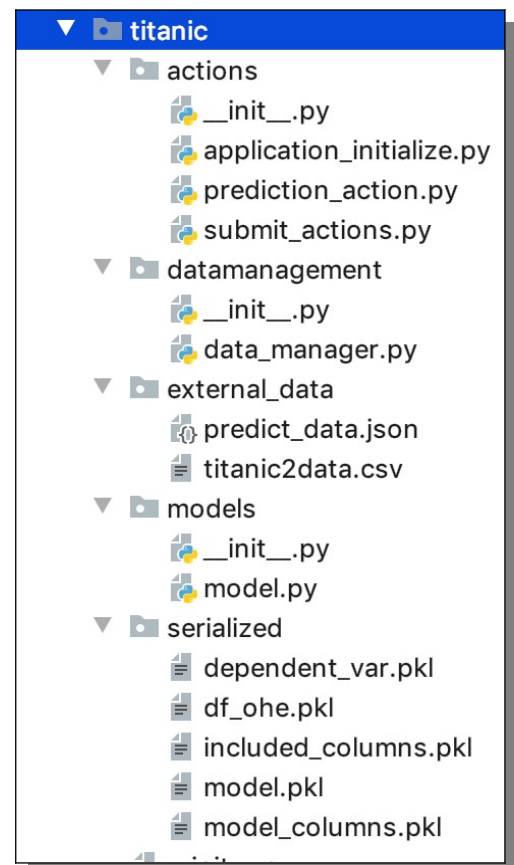
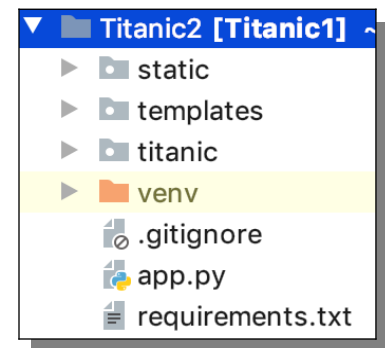
1. Specify features and labels for the model
2. Clean and prepare the data for training and testing
3. Set into motion model training
4. Set into motion model testing
5. Use the trained model for prediction on user defined data

The project to support these actions is organized into the project tree on the right. There are three main folders in the project: **static**, **templates**, and **titanic**. The **static** folder contains Javascript and CSS files. The **templates** folder contains HTML files (Flask calls them templates). The **titanic** folder contains all Python code that makes up the Titanic2 project. (Note that the main controller, app.py is in the root of the project so is not contained in any project folder.

The venv folder will be [discussed later](#) as will the requirements.txt file.

If we expand the titanic folder, we see the components of the application as described in the [Web Application Architecture](#) figure at the beginning of this article. The **models** folder is the DS Model module (In blue), which in this application is the LogisticRegression model. The **datamanagement** folder is the Data Access module (In blue), which contains the code that retrieves the data from the source, cleans the data, and does an OHE on the data columns.

The **actions** folder is the Controller/Actions module (In red) which contains code that controls execution of application initialization, and button actions. When you look at the code in app.py, you will see that there has been an attempt to minimize the amount of code in order to make the controller easier to read. So code that could normally go in app.py has been offloaded and placed in the **actions** folder.

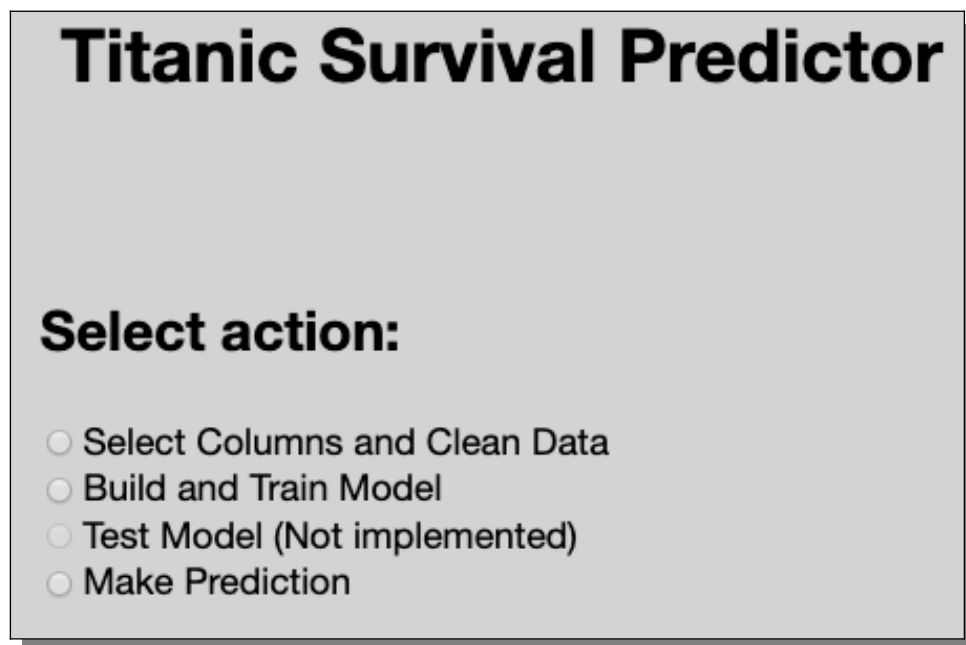


The folder named **external_data** contains both the source training data in csv form and the prediction data in json form. The folder named **serialized** contains the serialization of user state data.

The demo app has been set to run on port 5001.

The UI is a simple one page design where radio buttons are used to allow the user select the type of action desired. A click on a radio button will configure the display to accommodate the desired action. The appropriate action buttons will be displayed that allow the user execute the desired actions.

The UI in the demo initially looks like:



Titanic Survival Predictor

Select action:

- ☐ Select Columns and Clean Data
- ☐ Build and Train Model
- ☐ Test Model (Not implemented)
- ☐ Make Prediction

Notice that the UI design is a wizard, but sort of with random access. The steps from feature selection, cleaning the data, training the model, testing the model, and prediction follow the sequence from top to bottom. As each step is completed, the relevant data is serialized and stored so that the user can partially finish the whole process and then return where she/he left off without having to start over again at step 1.

We will use Javascript to display the UI components that are needed for each action. Each radio button will have an “onclick” event handler that will trigger the appropriate UI configuration. The html for the radio buttons follows:

```
35 <p style="...">
36   <input type="radio" name="actionradio" id="radioaction1" value="1" onclick="selectClean();">Select Columns and Clean Data <br>
37   <input type="radio" name="actionradio" id="radioaction2" value="2" onclick="buildTrain();">Build and Train Model<br>
38   <input type="radio" name="actionradio" id="radioaction2a" value="2" disabled>Test Model (Not implemented)<br>
39   <input type="radio" name="actionradio" id="radioaction3" value="3" onclick="predict();">Make Prediction</p><br>
40
41 </p>
```


Action of Select Columns and Clean Data

When the user clicks the first radio button, the javascript function, selectClean() is executed:

```
198 // Display and hide widgets for "Select Columns and Clean Data" action configuration
199 function selectClean(){
200     tagReferences.actionBtnObj.style.display = "block";
201     tagReferences.divContainerObj.style.display = "block";
202     tagReferences.divBtnsSelColsObj.style.display = "block";
203     tagReferences.divSelectedColsObj.style.display = "block";
204     tagReferences.divBtnDepVarObj.style.display = "none";
205     tagReferences.divDepVarObj.style.display = "none";
206     tagReferences.predictDivObj.style.display = "none";
207     tagReferences.outputdivObj.style.display = "none";
208     tagReferences.predictionBtnObj.style.display = "none";
209     tagReferences.msgDivObj.innerHTML = "";
210 }
```

This script is purely for cosmetic effect. The appropriate div, button, and select elements are shown or hidden by setting their **style.display** properties to “**block**” or “**none**”. The two **select** elements are displayed, pre-populated with values that may have been entered in a previous session. If no previous data is found, they will be empty. Also, the button labeled “Do Selected Action” is displayed.

In this page, the user is given the necessary controls to select which of the available data columns to include in the model. Javascript functions allow the user to select columns from the left select list and place them in the right select list as selected columns for the model. When the user is finished selecting columns, the button “Do Selected Action” is pressed, and the selected columns list is serialized and stored for later use. the interaction between the

The screenshot shows a web application titled "Titanic Survival Predictor". Below the title, there is a section labeled "Select action:" with four radio button options: "Select Columns and Clean Data" (which is selected), "Build and Train Model", "Test Model (Not implemented)", and "Make Prediction". Below these options is a button labeled "Do Selected Action". At the bottom, there are two columns of data. The left column, titled "Available Columns", contains a list of features: PassengerId, Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, and Fare. The right column, titled "Selected Columns", contains a list of features: Age, Sex, Embarked, and Survived. Between the two columns are two buttons: "-->" and "<--", used for moving items between the lists.

The Javascript that supports the select elements is found in */static/two_selects.js*.

We have described in words what happens when the user specifies and executes the first action (Select Columns and Clean Data). Now let's look at the code that accomplishes that task.

When the user enters into the browser: localhost: 5001, the URL '/' is sent as an HTTP request to the web server. In the controller app, namely app.py, the endpoint '/' is found and its associated function, **main()** is executed. Here is a snippet of that code:

```

11 @app.route('/')
12 def main():
13     """Entry point; the view for the main page"""
14     # Get stored values to load into ui
15     controller_manager = InitApplicationManager()
16     all_columns = controller_manager.get_all_column_names()
17     selected_columns = controller_manager.get_selected_columns()
18     dependent_variable_list = controller_manager.get_dependent_variable_list()
19
20     return render_template('main.html', all_columns_option_list=all_columns,
21                             selected_columns_option_list=selected_columns, dependent_var_option = dependent_variable_list)
22

```

The first job of **main()** is to check and retrieve any user data that may have been stored from a previous user session. The class, **InitApplicationManager**, which is found in the **actions** folder contains methods that do the following:

1. Retrieves all of the column names available in the original data source.
2. Retrieves if found all the columns that the user has selected as model features.
3. Retrieves if found the column that will serve as the dependent variable (label).

If you take a look at the **InitApplicationManager** class, you will see the strategy that is used to retrieve stored state data. Here is a snippet showing that strategy for retrieving selected columns:

```

26     # Check if selected columns are in session and retrieve. If not in session, retrieve from pickle storage
27     if 'session_selected_columns' in session:
28         self.selected_columns = session['session_selected_columns']
29     else:
30         if os.path.isfile('titanic/serialized/included_columns.pkl'):
31             self.selected_columns = joblib.load('titanic/serialized/included_columns.pkl')
32             session['selected_columns'] = self.selected_columns

```

This code snippet shows that we can use the Flask variable, **session**, which is available to all code in the application's context. The application context is any code that is executed within or called by code in the main application, **app.py**. The variable, **session** represents the session object, which can be used as a dictionary to store session data between HTTP requests. The code first looks in the session for the key "session_selected_columns". If found, the associated value is retrieved. If not, the serialized version stored in the file "included_columns.pkl" is searched for, and if found the columns are deserialized and put into the session.

So if there was any user data from previous application usages, that data is made available to the main app, and the **render_template()** call in **app.py** sends the data via parameters specified in lines 20 and 21. Notice that there are three parameters named in the call: **all_columns_option**, **selected_columns_option**, and **selected_columns**. Remember that the **render_template()** function uses the specified html file (In this case, **main.html**) and inserts the values of the specified parameters with their values into the given spots in the html code.

For example, let's look at the html code that populates the Available Columns select list in the UI.

```
54 <div id="div_available_columns" style="...">
55   <label for="all_columns" style="...">Available Columns</label>
56   <select name="all_columns" id="all_columns" size="10" style="...">
57     {% for col in all_columns_option_list %}
58       <option name="{{col}}" >{{col}}</option>
59     {% endfor %}
60   </select>
61 </div>
```

Line 56 starts a `<select>` tag whose name parameter is “all_columns”. We must take the list of all the column names passed through the parameter, **all_columns_option_list** and make `<option>` tags out of each column name. This means we must iterate over the list. The template engine, Jinja2 allows us to insert repetition code that is enclosed by the delimiters `{% %}`. Lines 57-59 show how to construct a **for** loop that iterates over the list. The code in the three specified lines cause the line 58 to be executed as many times as there are column names in the list. In line 59 you see the use of the Jinja2 symbols `{{ }}` to insert each column name into the option list. Therefore the result is that there will be as many `<option>` tags as there are column names. This completes the population of the Available Columns select list.

If you look at the rest of the html code in main.html, you will see similar code for populating other select lists.

So far we have traced the code support that gets us to the [UI configuration](#) that allows us to select model features from a list of available columns. We will now look at how the user selections get processed on the server side. The action is initiated when the user clicks on the “Do Selected Action” button. That simple click puts into action a number of events. Let's trace those events starting with the button tag in the html page.

```
42 <p>
43   <input type="button" value="Do Selected Action" id="actionBtn" name="actionBtn" style="..."
44     onclick="doSelectedAction();" />
45 </p>
```

Notice that we use a **button** input tag instead of a **submit** tag. By using the **button** input tag, we are able to use an **onclick** event where we can attach a custom javascript function **doSelectedAction()** as an event handler. The strategy that we will use is to take advantage of the fact that the **select** element that represents the selected features, whose id is “**selected_columns**” is contained inside a form tag:

```
27 <form action="{{url_for('action_controller')}}" id="mainForm" method="post">
```

Notice that the **action** property of the **form** tag uses the Jinja2 function, **url_for()**. This function finds the URL for the decorated function **action_controller()**. We could have replaced the Jinja2 use with:

action="/action_controller"

There is no difference between the two uses.

The javascript that handles the event associated with the “Do Selected Action” button click is listed below.

```
163  /*
164     Handles submits generated by the "Do Selected Action" button. The status of the radio buttons, radio1 and radio2
165     determine the particular action.
166  */
167  function doSelectedAction() {
168
169      var actionNumber;
170      if(tagReferences.radio1Obj.checked) {
171          if (tagReferences.selectedColumnsObj.length == 0) {
172              alert("No columns have been selected");
173              return;
174          }
175          // Select all options in selected_columns so they will all be posted
176          for (let i = 0; i < tagReferences.selectedColumnsObj.length; i++){
177              tagReferences.selectedColumnsObj.options[i].selected = true;
178          }
179          tagReferences.formObj.submit();
180      }
181      if(tagReferences.radio2Obj.checked) {
182          // ...
183      }
184  }
```

In the code snippet above, the identifier, **tagReferences** is a dictionary that has been set up to include all the elements in the html page that we need a reference to. The reference, **selectedColumnsObj** represents the **select** element that contains the features that the user selected. The for loop simply selects all the options in the select element. The loop was necessary because when we finally do a submit() in line 179, only the selected options will be included in the submit. The submit in line 179 creates an HTTP Post request that includes the URL:

localhost:5001/action_controller

Of course also included in the request are the values of all the elements in the form. When we eventually process this request, we will only be interested in the list of values from selectedColumnsObj in the html page.

Once the HTTP Post request is made, our Controller application, app.py will recognize the URL, **/action_controller** in its dictionary of URLs and execute the associated function. Here is the code from app.py that handles this URL:

```
24  @app.route('/action_controller', methods=['GET', 'POST'])
25  def action_controller():
26      """This url responds to the 'Do Selected Action' button.
27          There are two kinds of action that this button initiates depending on which radio button is active when the
28          action button is pressed. The SubmitController() class determines which radio button is active and activates
29          the appropriate methods to carry out the action.
30
31          Notice that this function returns a redirect() to the URL that is associated with the function main().
32          This redirect has the effect that the main page is refreshed with the appropriate page components displayed.
33      """
34      controller = SubmitController()
35      controller.submit_action()
36      return redirect(url_for('main')) # go to main page
```

Notice that we try to minimize the amount of code in app.py for ease of reading. We outsource the code to the class **SubmitController()**, and simply call on its **submit_action()** method.

```
8
9 class SubmitController():
10     """
11     Handles the submits generated by the "Do Selected Actions" button in the UI. In particular it handles:
12     (1) Submit when radio button 1 is active. Action is to collect and store column names that user has selected.
13         Then clean the data.
14     (2) Submit when radio button 2 is active. Action is to retrieve dependent variable name (label) and to
15         then to build and train the model.
16     """
17     def submit_action(self):
18         radio_selected = request.form['actionradio']
19         # Select columns and clean data
20         if radio_selected == "1":
21             self.select_columns()
22             flash("Columns successfully selected")
23             return
24         else:
25             # Build and train model
26             if radio_selected == "2":
27                 dependent_variable = request.form.get('dependent_var')
28                 session['session_dependent_variable'] = dependent_variable
29                 joblib.dump(dependent_variable, 'titanic/serialized/dependent_var.pkl')
30                 retval, model = self.build_train_model(dependent_variable)
31                 if retval:
32                     flash("Model was successfully built and trained")
33                 else:
34                     flash("Must do a 'Select Columns and Clean Data' first")
35             return
```

The **submit_action()** method handles two separate actions, depending on which of the first two radio buttons are selected. Line 18 uses the Flask global variable, **request** to retrieve the value of the “**actionradio**” radio button group. The request object has a property called **form** which is a dictionary of all input tags in the html form. So if the first radio button is selected while we click on the “Do Selected Action” button, the value returned in line 18 will be “1”.

On line 21, The **SubmitController.select_columns()** is then executed:

```
43 def select_columns(self):
44     """
45     Get a list of the user selected columns from the UI and put them into the
46     session as a list.
47     Also store the OHE version of the DataFrame.
48     """
49     included_columns = request.form.getlist('selected_columns') # list of str
50     data_manager = DataManager()
51     data_manager.set_included_columns(included_columns)
52     session['selected_columns'] = included_columns
53     dataframe_ohe = data_manager.get_df_ohe()
54     # Note: we do not put a dataframe into the session. It is much too large. Store
55     joblib.dump(dataframe_ohe, 'titanic/serialized/df_ohe.pkl')
56     return
```

The purpose of this method is to generate a One Hot Encoded version of the data using the columns that were selected in the UI. In line 49 we retrieve the list of selected columns from the **request** object. In line 50 we create a **DataManager** object, and on the next line set the **included_columns** list to that object so that it will be able to create an OHE version of the data. In line 52 we add to the session the selected columns list. In line 53 we call the method **get_df_ohe()** of the **DataManager** object. Finally in line 55 we serialize the OHE data and store it in a file called **df_ohe.pkl**. This data will be used when the user is ready to train the model.

The **return** on line 56 finishes the call that was made on [line 21](#). On line 22 we see a call to **flash()**:

```
21         self.select_columns()
22         flash("Columns successfully selected")
23         return
```

The call on line 22 to **flash()** stores a string message in the session so that it may be retrieved on the next page that is set up for such a message. More on this message on the next page.

Once all of the **returns** have completed, we end up back in the main app.py:

```
34         controller = SubmitController()
35         controller.submit_action()
36         return redirect(url_for('main')) # go to main page
```

The final line that finishes the action, caused by clicking on “Do Selected Action” button, is line 36. Our desire is to re-render our page, **main.html** that allows the user to make another radio button selection. But we want to retain the data that populated the two select elements. It would be convenient if we had a way to just execute the function, **main()** in app.py. There is a convenient way to do so with the flask function, **redirect()**. We pass the flask function, **url_for()**, passing the name of the function that was decorated. In this case the name of the function is **main**. Fortunately, we have written the class, **InitApplicationManager** so that first tries data retrieval from the session, and then from the serialized files if not found in the session. So the retrieval design of user generated data is just to use a cache system.

One final detail needs attention. Remember in the **SubmitController** class on line 22, we put a message into the session with the function **flash()**. When we redirected control back to the **main()** function and then re-rendered **main.html**, that message waiting in the session can be rendered in the page. The html to do so in **main.html** is:

```
17         <div id="msgDiv" name="msgDiv" class="msgDiv">
18             {% with messages = get_flashed_messages() %}
19                 {% if messages %}
20                     {% for message in messages %}
21                         {{ message }}
22                     {% endfor %}
23                 {% endif %}
24             {% endwhile %}
25         </div>
```

We create a message **div** that contains Jinja2 code to render the message that is currently stored in the session. Line 17 calls the Jinja2 function, `get_flashed_messages()` to retrieve the messages. In this case there is only one message in the session. We use the Jinja2 symbols `{{ }}` to actually render the message in blue text in the html page.

Titanic Survival Predictor

Columns successfully selected

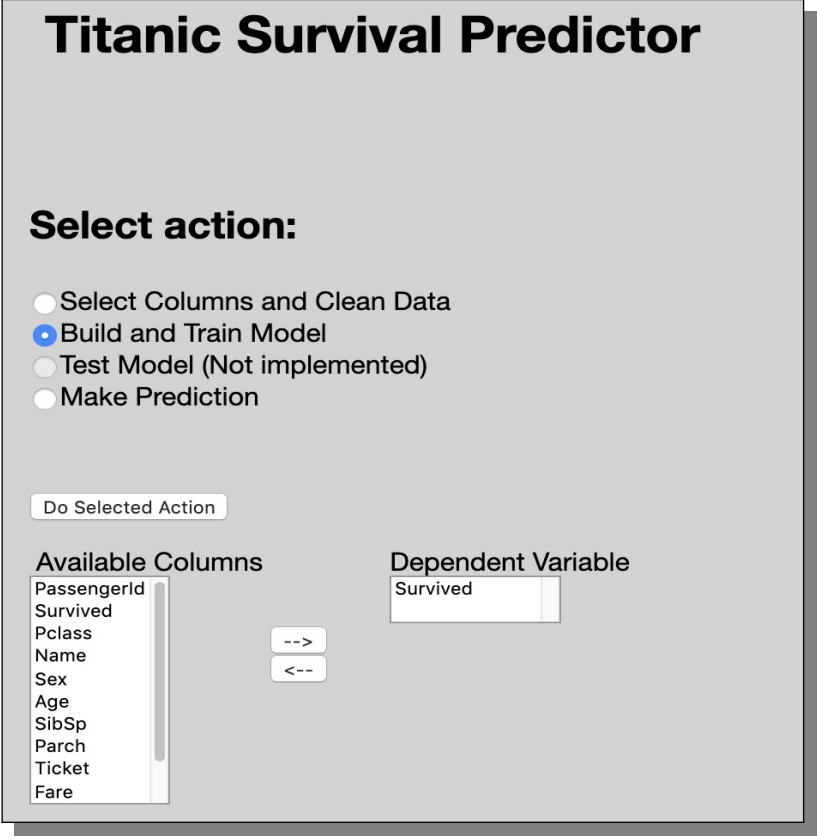
Select action:

- ☐ Select Columns and Clean Data
- ☐ Build and Train Model
- ☐ Test Model (Not implemented)
- ☐ Make Prediction

Note that when a **flash()** message gets rendered, **flask** automatically removes that message from the session.

Action of Build and Train Model

The second radio button with the label “Build and Train Model” presents the following UI:



Titanic Survival Predictor

Select action:

- ☐ Select Columns and Clean Data
- ☒ Build and Train Model
- ☐ Test Model (Not implemented)
- ☐ Make Prediction

Available Columns

PassengerId
Survived
Pclass
Name
Sex
Age
SibSp
Parch
Ticket
Fare

-->

<--

Dependent Variable

Survived

The user is allowed to select one column as the dependent variable (label).

When the user clicks on the “Do Selected Action” button, the code for building and training the model is executed. The pattern of that code is exactly the same as that of “Select Columns and Clean Data” that we did in the last section.

Make Prediction

In order to make predictions using the trained model, the user must first be able to upload a file that contains the prediction data. This will require some javascript to launch a file choose dialog. The strategy for our UI will be to launch the file choose dialog when the user clicks on the [“Make Prediction” radio button](#). Notice in the html code, the prediction radio button has an id of “radioaction3”. The **onclick** event is handled by the javascript function, **predict()**, which just prepares the UI by hiding all the irrelevant elements such as the selects and the “Do Selected Action” button.

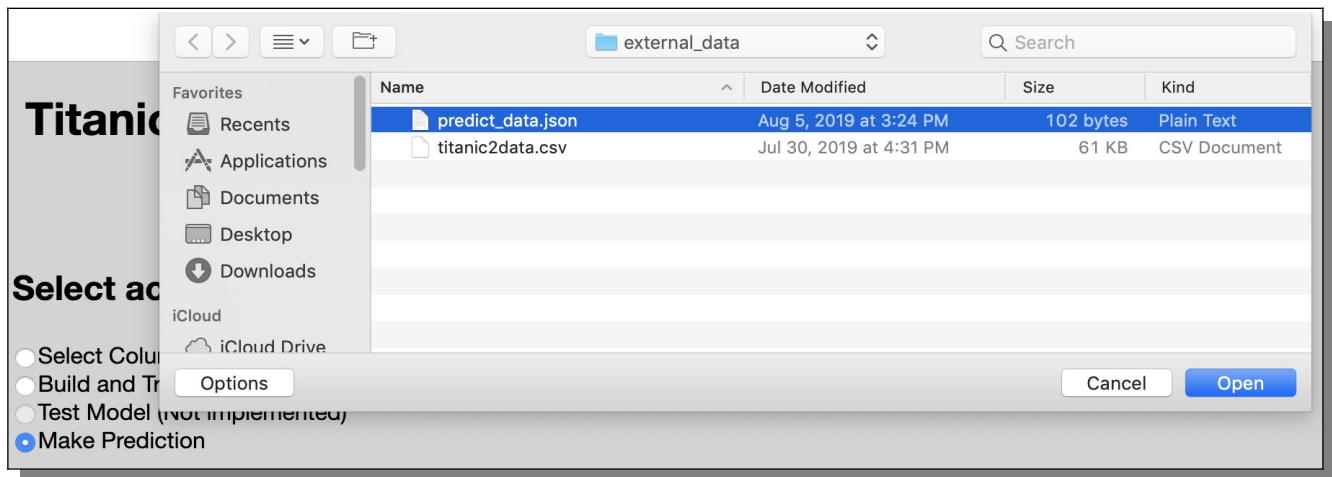
So how do we launch a file chooser dialog by clicking on a radio button? We start by creating an element within the **form** element in main.html:

```
27 <form action="{{url_for('action_controller')}}" id="mainForm" method="post">
28 <input type="file" style="display: none" id="hidden_file_input"/>
```

The “file” input element comes with a button that normally opens the file chooser dialog. Instead, we want to launch the dialog when the last radio button is clicked. So we hide (style=”display: none”) the “file” input element and add an event listener to the radio button that captures a “click” event.

```
137 tagReferences.radio30bj.addEventListener("click", function () {
138 tagReferences.hiddenFileInputObj.click();
139 tagReferences.radio10bj.checked = false;
140 tagReferences.radio20bj.checked = false;
141 tagReferences.radio30bj.checked = true;
142 });
```

In the event listener handler we simulate a “click” action on the “hidden_file_input” element seen on line 138. So the effect we create in the above code is that a click on the last radio button simulates a click on the “file” input element button and the result is that a file chooser dialog will be displayed.



In the file chooser dialog we navigate to the `external_data` folder in our project and find the file, `predict_data.json`.

A click on the “Open” button creates a “change” event on the “file” input element. So we will add an event listener to the “file” input element that will specify the name of the javascript function that will read the contents of the file. The function name is **handleFileSelect**.

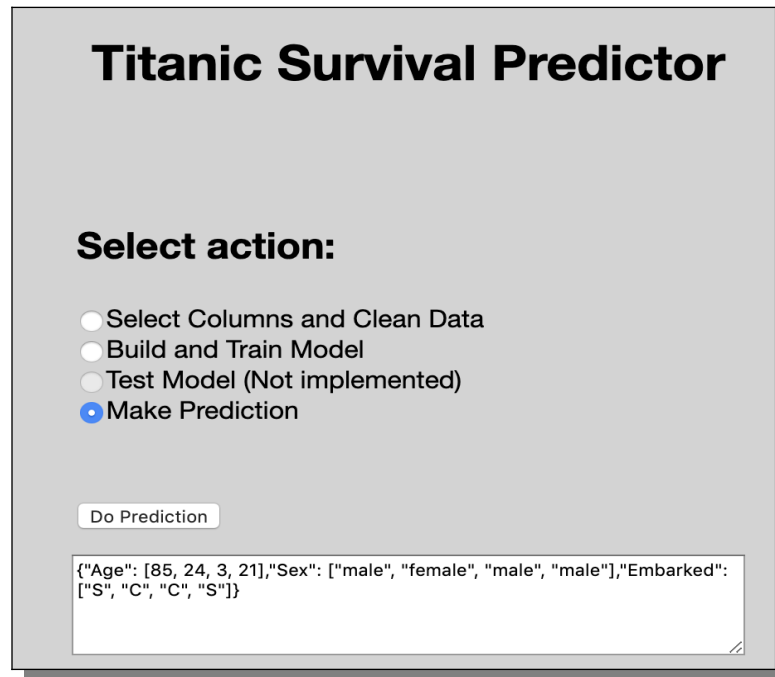
```
156 tagReferences.hiddenFileInputObj.addEventListener('change', handleFileSelect, false);
157
```

The javascript function, `handleFileSelect()` reads the selected file and places its contents in a text area.

```
143 function handleFileSelect(evt) {
144     var content;
145     var file = evt.target.files[0];
146     var reader = new FileReader();
147     reader.onload = () => {
148         content = reader.result;
149         tagReferences.predictionTextareaObj.innerText = content;
150         tagReferences.predictDivObj.style.display = "block";
151         tagReferences.predictionBtnObj.style.display = "block";
152     }
153     reader.readAsText(file, 'UTF-8');
154 }
155
```

Line 145 retrieves the selected file as an object. Line 154 reads the contents of the file as text. Lines 147-151 defines an anonymous **onload** handler that executes when the file read has been completed.

Notice that in line 149 the contents of the file are inserted into the text area so the user can read the json test data.



The screenshot shows a web application titled "Titanic Survival Predictor". Under the heading "Select action:", there are four radio button options: "Select Columns and Clean Data", "Build and Train Model", "Test Model (Not implemented)", and "Make Prediction". The "Make Prediction" option is selected. Below these options is a button labeled "Do Prediction". Underneath the button is a text area containing a JSON string: {"Age": [85, 24, 3, 21], "Sex": ["male", "female", "male", "male"], "Embarked": ["S", "C", "C", "S"]}. The text area has a small icon in the bottom right corner, likely for clearing the text.

Once our test file has been chosen and read, a new action button “Do Prediction” is displayed. And we will now follow the action of this button click.

Python Code for Prediction

Before we look at the javascript needed to make the prediction request and then put the results into an html table, let's first see how the prediction will be made on the server side. That is, how we get the prediction from the Data Science model, put it into an easy to read form, so that later we can then display that table in the UI. Let's assume that somehow we have made a request on the client side that contains the URL: “/prediction” as well as the test data in json form. In the main controller, app.py we have a function called prediction() that is decorated with the URL, “/prediction”.

```
39 @app.route('/prediction', methods=['GET', 'POST'])
40 def prediction():
41     """
42     Note there is no call to render_template() or redirect() because /prediction is an asynchronous call to the server
43     where client side javascript captures this return value and builds HTML that is rendered on the client's web page.
44     :return: # json serialized object. For example:  b' '{"result": [0, 1, 0, 0]}' '
45     """
46     return PredictionController.predict()
```

There is only one executable line in **prediction()**, and we call the static method **predict()** of the class **PredictionController**.

```

10 class PredictionController:
11     @staticmethod
12     def predict():
13         json_test_data = request.get_data()
14         # Convert json serialized object to a Python dictionary object
15         json_dict_obj = json.loads(json_test_data) # dict object
16         # Get the serialized model
17         if os.path.isfile('titanic/serialized/model.pkl'):
18             lr_model = joblib.load("titanic/serialized/model.pkl")
19         else:
20             flash("Model not found")
21             return redirect(url_for('main')) # go to main page
22         if os.path.isfile('titanic/serialized/model_columns.pkl'):
23             model_columns = joblib.load('titanic/serialized/model_columns.pkl')
24         else:
25             flash("Model columns not found")
26             return redirect(url_for('main')) # go to main page
27         lr_model_obj = LogisticRegressionModel()
28         lr_model_obj.setModel(lr_model)
29
30         list_results = lr_model_obj.predict_with_data(json_dict_obj,
31             model_columns) # ndarray [0, 1, 0, 0] this is the prediction made by the model
32         # Convert ndarray to list:
33         list_results_aslist = list_results.tolist()
34
35         # MUST PREPARE THE RESULTS FROM THE MODEL TO BE PUT IN JSON FORM. The model returns a
36         # list: [0, 1, 0, 0]. That list must be put into a json string as a dictionary.
37         # Now since results is a list, we must make a dictionary with the label "result"
38         result_dictionary = {"result": list_results_aslist} # dictionary {"result": [0, 1, 0, 0]}
39         # make the dictionary into a string
40         return_value = json.dumps(result_dictionary) # String form of dictionary: '{"result": [0, 1, 0, 0]}'
41         # Return the jsonify of the string which is a byte string (serialized json)
42         return jsonify(return_value) # json object b' '{"result": [0, 1, 0, 0]}'

```

The **predict()** method does four things:

1. Take the json data from the request and convert it to a Python dictionary
2. Retrieve the serialized trained model from storage (model.pkl)
3. Call on the trained model to make a prediction using the Python dictionary of test data
4. Convert the prediction results into a serialized json string.

In line 13 we simply get the test data from the **request** and then convert it to a Python dictionary in line 15. In line 18 we retrieve and deserialize the **LogisticRegression** object. In line 23 we retrieve and deserialize the model columns (labels). In line 27 we create a **LogisticRegressionModel** object and then set the newly deserialized **LogisticRegression** object to it in line 28.

(NOTE: The **LogisticRegressionModel** class contains an **sklearn.linear_model.LogisticRegression** object. Back when we trained the model, we serialized and stored only the **LogisticRegression** object. So when we want to rebuild the **LogisticRegressionModel**, we first deserialize the **LogisticRegression** object, create a **LogisticRegressionModel** object and set the **LogisticRegression** object to it.)

Then we make the prediction calculation in line 30. Notice that the prediction value is a Python **ndarray** with a value of [0, 1, 0, 0]. Our goal is to convert this **ndarray** into a serialized json string. Notice that there are 4 results with values of 0 or 1. The value of 1

stands for “survival”. The model returns the prediction in the form of a Python ndarray, and we need to convert that ndarray to a json serialized string.

Line 33 converts the **ndarray** to a Python **list**. Line 38 creates a Python dictionary out of the list with the key of “result”. Line 40 creates a string form of the dictionary, and finally line 42 returns the json serialized form of the string.

Javascript Code for Request and Response

We will start with the html specification for the prediction button:

```
46 <p>
47 <input type="button" value="Do Prediction" id="predictionBtn" name="predictionBtn" style="..."
48 <input type="button" value="Do Prediction" id="predictionBtn" name="predictionBtn" style="..."
49 </p>
```

The two previous actions, selecting columns and training the model were done with synchronous Post requests. That is, they were both initiated with the javascript **submit()** action, which blocks the web page until all the resources on the web page are completely loaded. In this section we illustrate how to make an asynchronous request in order to display a table of prediction results. An asynchronous request is made with the javascript function, **fetch()**.

The javascript for handling of the “Do Prediction” button click is stored in a separate file, **/static/ajax_predict.js**. Rather than doing a synchronous Post, it would be informative to do an asynchronous Post. An asynchronous request to the server frees up the UI to still respond to user actions while the request is being processed. This type of request is handy if the web page has several large resources displayed and we wish to update only a single resource without having to update all the others. A typical scenario is if we have several graphs on a web page that respond to real time data. We can update each graph separately with asynchronous requests.

In this demo we will take prediction results, combine them with the test data and asynchronously display the resulting table in the UI.

The “Do Prediction” button is given an **onclick** event that executes the javascript function, **doPredictionAction1()**:

```
1
2 function doPredictionAction1(){
3   let predictionData = tagReferences.textareaObj.value; // Text area where test data is displayed
4   postRequest( url: '/prediction', predictionData)
5     .then(responseData => formatAndDisplayHTML(responseData, predictionData))
6     .catch(error => console.error(error))
7
8 }
```

Line 3 uses a reference to the text area element (where we have already displayed the json test data) to retrieve the test data. Line 4 executes a user defined **postRequest()** that contains the endpoint URL for the action and the json test data:

```
14      /*
15         Posts a json request to the given url. The requestData param is a json string
16         The response value is a json string
17      */
18      function postRequest(url, requestData) {
19          return fetch(url, init: {
20              method: 'POST',
21              body: requestData,
22              headers: new Headers( init: {
23                  'Content-Type': 'application/json'
24              }),
25          })
26          .then((response) => response.json())
27          );
28      }
```

The javascript function, **postRequest()** first builds an HTTPRequest, and then returns a response in json form. All the work is done by the javascript function, **fetch()**. We pass two parameters to **fetch()**: the target URL and a dictionary of values that are used to build the HTTPRequest. We specify the values for the keys: **method**, **body**, and **headers**. The function builds the request and sends it to the specified URL. When **fetch()** returns with a **response**, the **then()** on line 26 receives the returned **response**, and returns the results of the anonymous function, which is the json form of the **response**. The **fetch()** function is asynchronous and therefore does not block execution while it waits for the response.

Note that this **postRequest()** function is generic enough that it could be reused for any asynchronous post that uses json data.

Back at the **doPredictionAction1()** function, when the **postRequest()** function returns with the json response, the **then()** on line 5 executes an anonymous function, passing the json response to an anonymous function which calls upon the function **formatAndDisplayHTML()**. This function takes the response data in json form as well as the reference to the text area where the test data is displayed. Remember that we want to combine both the prediction results with the test data in a table. We do so with the function **formatAndDisplayHTML()**:

```

29  /*
30      Format both the responseData and the predictData into an HTML table.
31  */
32  function formatAndDisplayHTML(responseData, predictData){
33      //NOTE: Returned data will be a json dict with a root label of "result"
34      console.log("returned data: " + responseData);
35      // Extract data from responseData and predictData for display in HTML table
36      let json_dict = JSON.parse(responseData); // dictionary
37      let listResults = json_dict.result; // list object of results just returned from model
38      let predictDict = JSON.parse(predictData); // dict object of prediction data used in model
39      let age = predictDict.Age;
40      let sex = predictDict.Sex;
41      let embarked = predictDict.Embarked;
42      // At this point, the lists listResult, age, sex, embarked are parallel lists"
43      let returnHTML = "";
44      returnHTML += "<table>";
45      returnHTML += "<tr><td>Survived</td>" + "<td>Age</td>" + "<td>Sex</td>" + "<td>Embarked</td>"
46      for(let i = 0; i < listResults.length; i++){
47          returnHTML += "<tr>";
48          returnHTML += "<td>" + listResults[i] + "</td>";
49          returnHTML += "<td>" + age[i] + "</td>";
50          returnHTML += "<td>" + sex[i] + "</td>";
51          returnHTML += "<td>" + embarked[i] + "</td>";
52          returnHTML += "</tr>";
53      }
54      returnHTML += "</table>";
55      // Put HTML into the output div
56      tagReferences.outputdivObj.innerHTML = returnHTML;
57      tagReferences.outputdivObj.style.display = "block";
58      return ;
59  }

```

The function **formatAndDisplayHTML()** does two jobs: (1) It creates a large string that contains the html we want to display, (2) It places the html string in the output div.

The html string is simply a table that contains all the test data with an additional column prepended to the table that shows the predictions. The displayed predictions have values of either 1 for survived or 0 for not survived.

The final result looks like:

Titanic Survival Predictor

Select action:

☐ Select Columns and Clean Data

☐ Build and Train Model

☐ Test Model (Not implemented)

☒ Make Prediction

Do Prediction

```
{ "Age": [85, 24, 3, 21], "Sex": ["male", "female", "male", "male"], "Embarked": ["S", "C", "C", "S"] }
```

Survived	Age	Sex	Embarked
0	85	male	S
1	24	female	C
0	3	male	C
0	21	male	S

Virtual environments

In a [previous section](#) we made a reference to a project folder called **venv**. Some IDEs such as PyCharm will gather all project dependencies and put them into a folder called a virtual environment. When the project is executed, it will use the virtual environment for all used libraries. This convenience allows us to deploy an application that is self contained with respect to dependencies. Even if the project survives for many years, it will still make use of the library versions that existed when the project was created.

The **venv** can be large in size and awkward to store in a Git repository. The PyCharm IDE creates a file named `dependencies.txt` that contains all the version numbers of the Python modules that the project uses. This text file can be placed in the Git repository and then downloaded to allow the IDE to recreate the **venv**.

This virtual environment concept manages dependencies similar to **maven**.