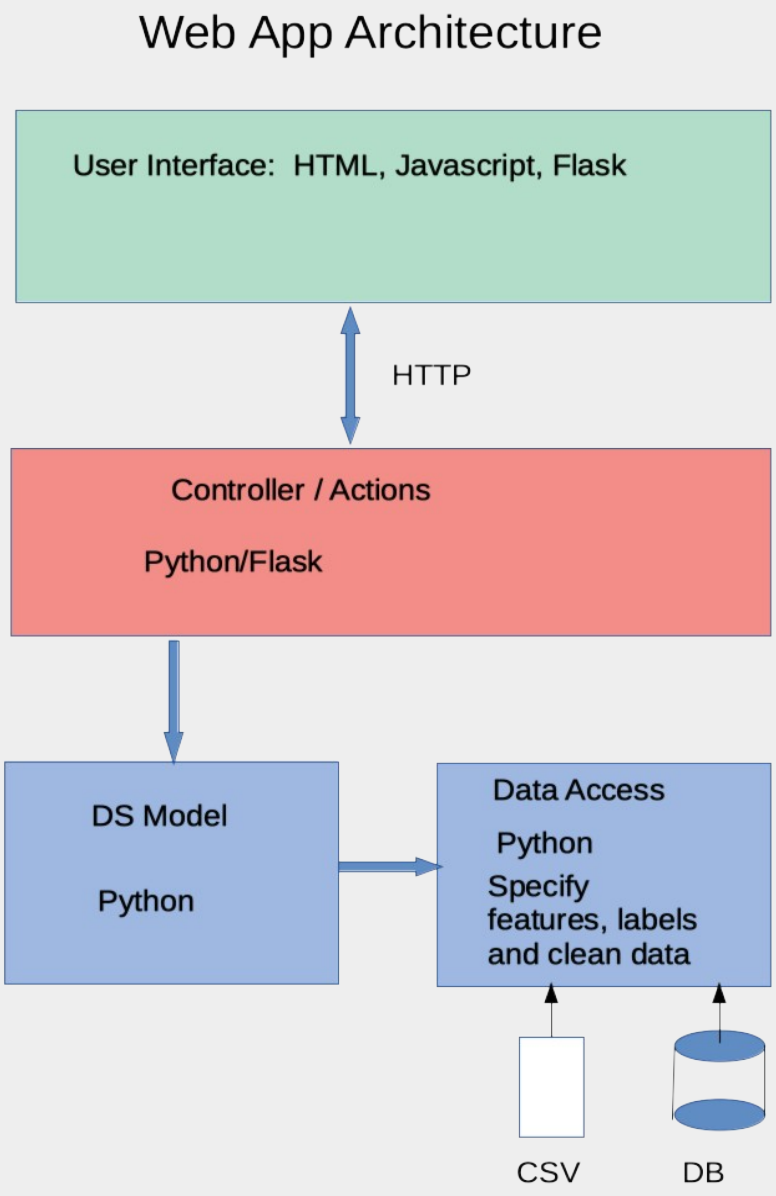
Web Application Using Flask Framework

The application architecture

This project demonstrates how a Data Science module written in Python can be placed into a web application context with a usable User Interface (UI). The main point of the project is a proof of concept that a Data Scientist can build a model (In this case a machine learning model) using Python without having to worry about the web application context in which it is used. Furthermore it is shown that when written properly, a model can be used within any web application context, that is, the model is completely independent from the web framework in which it is used. This means that the Python model can function not just in the Flask framework, but can also be used directly in Django, or front ends built with AngularJS, Dash, etc.

Before getting into details of the web application, let's look at the overall architecture that we will be using.



The figure on the right depicts how the demo web application is built. 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 red and green modules would be written by a specialist in UI design with a knowledge of the web framework that is being used.

It should be apparent that the blue modules are only a part of the whole web application. In addition to providing his/her DS knowledge, the Data Scientist also acts as a software developer.

In fact, the Data Scientist is actually a member of a software development team that is tasked to build the web application. This realization means that the code the Data Scientist writes will be read and used by other members of the team.

Application Program Interface

Let's look at the blue modules first. The module with the label "**Data Access**" represents the Python code that would allow for the following:

- Read the data from either a CSV or Database source
- Specify the features and labels in the training data
- Clean the data by handling missing values and type mismatches
- Use One Hot Encoding to handle columns that contain non-numeric values

The module labeled "DS Model" allows for the following:

- Building the model
- Training the model with the training data.
- Predict label values based on user supplied data

If we were to inspect the code of these two modules, we would find that there are no dependencies other than Python libraries. In fact, here is a code snippet of the DS Model:

```
from sklearn.linear_model import LogisticRegression
from sklearn.externals import joblib
import pandas as pd


class LogisticRegressionModel():
    """
    This class encapsulates a sklearn.linear_model.LogisticRegression object. The class contains a LogisticRegression
    model together with a pandas DataFrame that is passed to it through its constructor.  In addition, the
    constructor also receives the name of the dependent variable column (label) as well as the data columns (features)
    used in the model.
    """
```

Notice that the imports at the top of the snippet only include Python libraries. There is no inclusion of Flask (The web framework library), nor are there inclusions of any other libraries. In software engineering terms, we say that the DS Model module is completely ***decoupled*** from the other modules in the application. That is, we can say that the DS Model has no other dependencies other than Python libraries. The decoupled property of the DS Model module means that this module can be used in any context without worry that it would not have available its supporting libraries.

Let's look at a code snippet from the Data Access module:

```
import pandas as pd
import csv
from sklearn.externals import joblib


class DataManager():
    """
    This class manages the data.  It reads the original data specified in the data_url parameter of the constructor.
    It keeps track of the column names that will be included as model features.  The method get_df_ohe() uses
    the cols_to_include and produces a pandas DataFrame that has been one hot encoded.
    """
```

Again, no dependencies other than Python libraries. Then we can also declare that the Data Access module is ***decoupled*** from the other modules in the application.

The concept of decoupling is great for software design, but more importantly, it is also important for the Data Scientist.

Decoupling means that the Data Scientist can write the DS model and Access modules without having to worry about the context in which they are used. That is, the Data Scientist can just continue to write her/his code as before with only minor adjustments.

A Few Minor Adjustments

The last sentence of the previous section stated: "*the Data Scientist can just continue to write her/his code as before with only **minor adjustments**.*". In this page we will discuss those "minor adjustments".

First we need to understand the simple protocol that software developers use to facilitate communication between the web application and its components (eg. DS Model code). Since the DS Model module is decoupled, it has to have some way to receive from the web application the features and labels that it will use in the model. We do not want to hard-code in those values since that would make the module inflexible. Also, the module needs some way of returning the trained model as well as a prediction based on prediction data that the user provides.

So we will make use of a protocol that software developers use to solve the problem of communication with a decoupled module. The protocol is called "Application Program Interface", or simply API. *Implementation of API's does not involve new technology.* The implementation of API's is simply arranging your code so that it conforms to a simple standard. In fact, for our demo purposes we can say that our API's will just be functions or methods of a class that can be called by API consumers. (In this discussion we will treat the terms "function" and "class method" as interchangeable) These API's will have a documented list of parameters that are passed to the function and a documented list of values that the function will return.

A consumer of API's only needs to know three things:

1) The name of the function to call
2) The parameters of the function and their data types
3) The return values of the function and their data types

It is helpful for the both the creator and the consumer of the API to put in writing the above 3 specifications. That way the consumer's expectations will match the intentions of the API's creator.

The snippet on the right is a simple description of two APIs. The first API will be used to build and train the model, while the second is used to make a prediction. We can see that this form of the API specification is like a contract. The contract says: "Give the API values it needs, and it will return values that the consumer needs. Notice that at this stage of design, we do not have to specify the name of the API (name of the function).

Action Build and Train Model

Needs:

- A OHE pandas DataFrame that has been cleaned.
- Name of data column that will be the model's label (string).

Creates:

- Trained model: sklearn.linear_model.LogisticRegression Object
- List (strings) of model features

Action Make Prediction

Needs:

- Trained model: sklearn.linear_model.LogisticRegression Object
- Test data in dictionary form

Creates:

Prediction as an ndarray

The creator of this API can expect to be given:

1) The test data in the form of a pandas DataFrame that has been OHE
2) The name of the column that will serve as the model's label

Also, the creator of this API will be expected to return two values:

1) The trained value, which is a LogisticRegression object
2) A list of the model's features (as strings)

It is just a matter then of wrapping your code that builds and trains your model within a function that satisfies the above requirements.

In the demo project, the function that builds and trains the model is actually a method of a class called **LogisticRegressionModel**. This class has a constructor (the method with the name **__init__()** that accepts as parameters the data needed for its APIs. In the code below you can see how the training data (**data_frame**) and model label (**dependent_variable**) are passed through the class's constructor.

```
6   class LogisticRegressionModel():
7
8       def __init__(self, data_frame = None, dependent_variable = None):
9           """
10
11          :param data_frame: pandas DataFrame that contains the training data
12          :param dependent_variable: data column name of the label
13          :param model_columns: data column names of the features
14          """
15          self.lr_model = LogisticRegression()
16          self.data_frame = data_frame
17          self.dependent_variable = dependent_variable
18
19      def build_model(self):
20          """
21              Builds a LogisticRegression model using values that are data members of this class.
22              The data members used are data_frame which is a training pandas DataFrame that has been OHE,
23              dependent_variable which is a string representing the model's label.
24          :return: sklearn.linear_model.LogisticRegression object, list of string which is a list of
25                  the feature columns
26          """
27          # Get all the columns that are not the dependent_variable
28          x = self.data_frame[self.data_frame.columns.difference([self.dependent_variable])]
29          y = self.data_frame[self.dependent_variable]
30          self.lr_model.fit(x, y)
31          model_columns = list(x.columns)
32          return self.lr_model, model_columns
```

In the code above you can see the definition of the class method called **build_model().** It uses the data members, s**elf.data_frame** and **self.dependent_variable** to get the data for the feature values (**x**) and the label values (**y**).

In line 31 the actual columns that the model uses (including the OHE columns) are extracted. In line 32 both the **LogisticRegression** object and the model columns are returned.

The specification for the prediction indicates that we need to give the API the trained model as a **LogisticRegression** object together with the prediction data (feature values) in a dictionary form. The class method should return the prediction results as an **ndarray** object:

```
33
34      def predict_with_data(self, json_test_data, model_columns):
35          """
36          :param model_columns: list of str
37          :param json_test_data: dictionary
38          :return: ndarray of prediction values ( 0 or 1 )
39          :rtype: ndarray
40          """
41
42          v1 = pd.DataFrame(json_test_data)
43          query = pd.get_dummies(v1)
44          query = query.reindex(columns=model_columns, fill_value=0)
45          prediction = self.lr_model.predict(query)  # ndarray, not list
46
47          return prediction
48
```

On line 34 you can see that the method parameters **json_test_data** and **model_columns** represent the values sent to the method. Those values are then used in the method **predict_with_data()** to calculate the prediction.

The code displayed in the above snippets should be familiar to any Data Scientist. All we have done in this demo is to package the appropriate code into class methods that can then be called later by a consumer.

One final observation. The above two code snippets show the definitions of three methods of our **LogisticRegressionModel** class. Just after the method header you will see some documentation that defines the “signature” of the method and what the method does. A method (or function) signature should always contain roles and the data types of the method parameters together with a description and data types of the return values.

Here is a repetition of the documentation for the **predict_with_data()** method:

```
33
34      def predict_with_data(self, json_test_data, model_columns):
35          """
36          :param model_columns: list of str
37          :param json_test_data: dictionary
38          :return: prediction values ( 0 or 1 )
39          :rtype: ndarray
40          """
```

First, notice that we give the method a name that gives the reader an idea of what the method does. In this case we use **predict_with_data.**

The Python convention for documenting a function is to place the information inside triple quotes. We use documentation keywords "**:param and :rtype**" to specify the data types of the parameters and return values. We use the keyword "**:return:** to specify what value(s) the method returns.

In addition to documenting the method signature, we also add comments within the code in order to indicate to the reader the intention of the code:

```
# Convert the json data (dictionary) into a pandas DataFrame
test_dataframe = pd.DataFrame(json_test_data)  # DataFrame
# OHE the category columns in the test data
ohe_dataframe = pd.get_dummies(test_dataframe) # DataFrame
# Make sure that all the columns in the model are included in the test data.
query = ohe_dataframe.reindex(columns=model_columns, fill_value=0)
prediction = self.lr_model.predict(query)  # ndarray, not list

return prediction
```

Notice in the code snippet to the right that we have added comments above lines of code and have also added comments at the end of the appropriate lines to indicate the data types involved in the statements.

The author of Python, Guido van Rossum once stated: "Code is more often read than written". As you execute the role of software developer please keep in mind that many other people will be reading your code. If another person is given the task of using your APIs or adding a feature or correcting an error in your code, you want to help them navigate through that code by providing helpful documentation. A good practice then is to write code with the intention that others will read it (Including perhaps yourself at some later date).