Step By Step Guide :

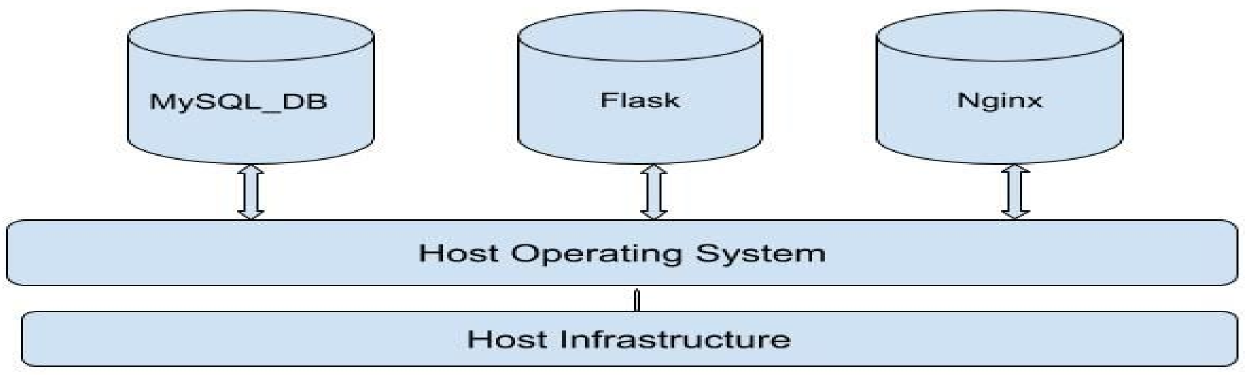
1. Read the Datasets in Python,

2. Pre process the data using multiple methods.

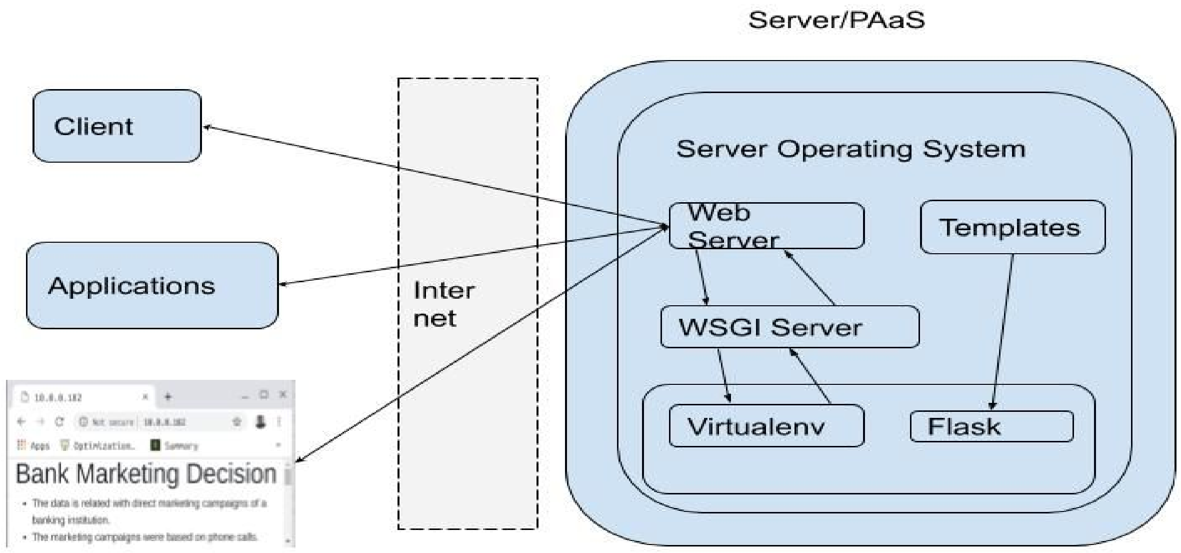
3. Do hyper parameter tuning to get the best model.

4. Once the best model with the parameters and hyper parameters is ready start then head start for the deployment.

The deployment phase involves bringing up the Application in to a web Application Which involves the creation of the server where the code will run and render the html pages on the WebApp. A web application, we might be developing on your local engine. Minimizing the differences between your local environment and production help us to avoid last minute changes because of configuration differences. Therefore we need to work with Docker containers on local environment which is a true productivity booster and saves you some last minute hassle . The Docker containers architecture required for the application is shown below.



The deployment architecture that resides inside a Docker container is given below:



From the architecture we can select the components like webserver and WSGI for the application.

WebServer : Flask has a webserver inbuilt to work for standalone basis but for production environment we need to create a webserver. From the industry standards the popular webserver for python based applications is Nginx which we are going to use for the Application now.

WSGI Server : A traditional web server does not understand or have any way to run Python applications. Therefore the python community came with WSGI as a standard interface that modules and containers could implement. WSGI is now the accepted approach for running Python web applications. Many WSGI are available we choose Gunicorn because it is a stable, commonly-used part of web app deployments

We FrameWorks : We have chosen Flask because it is a lightweight micro web framework. It is more pythonic and less boilerplate code needs to be written.

Database : We have chosen MySQL database where the data for the designing the Machine Learning Application resides.

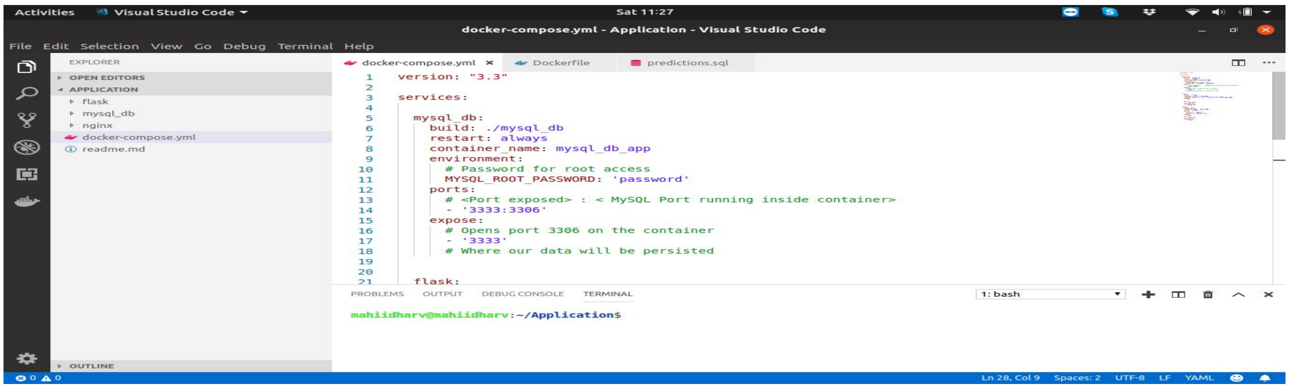
Editor/IDE : To perform the entire application code and help the connectivity for various components we require a suitable editor/IDE . We have chosen the Visual Studio Code which is a open source editor that helps in connecting the Docker, Python Code for Application,GIT for Source Control and Kubernetes for Container Orchestration and Deployment.

Since we have to perform every thing in the IDE let us start with understanding the VS Code and start to build the application required code in it.

Introduction to Code :

- Visual Studio Code combines the simplicity of a source code editor with powerful developer tooling, like IntelliSense code completion and debugging.
- Visual Studio Code supports macOS, Linux, and Windows - so you can hit the ground running, no matter the platform.

At its heart, Visual Studio Code is a code editor. Like many other code editors, VS Code adopts a common user interface and layout of an explorer on the left, showing all of the files and folders you have access to, and an editor on the right, showing the content of the files you have opened as shown below:



```
version: "3.3"

services:

  mysql_db:
    build: ./mysql_db
    restart: always
    container_name: mysql_db_app
    environment:
      # Password for root access
      MYSQL_ROOT_PASSWORD: 'password'
    ports:
      # <Port exposed> : < MySQL Port running inside container>
      - '3333:3306'
    expose:
      # Opens port 3306 on the container
      - '3333'
      # Where our data will be persisted


  flask:
```

```
mahiidharv@mahiidharv:~/Application$
```

Basic Layout:

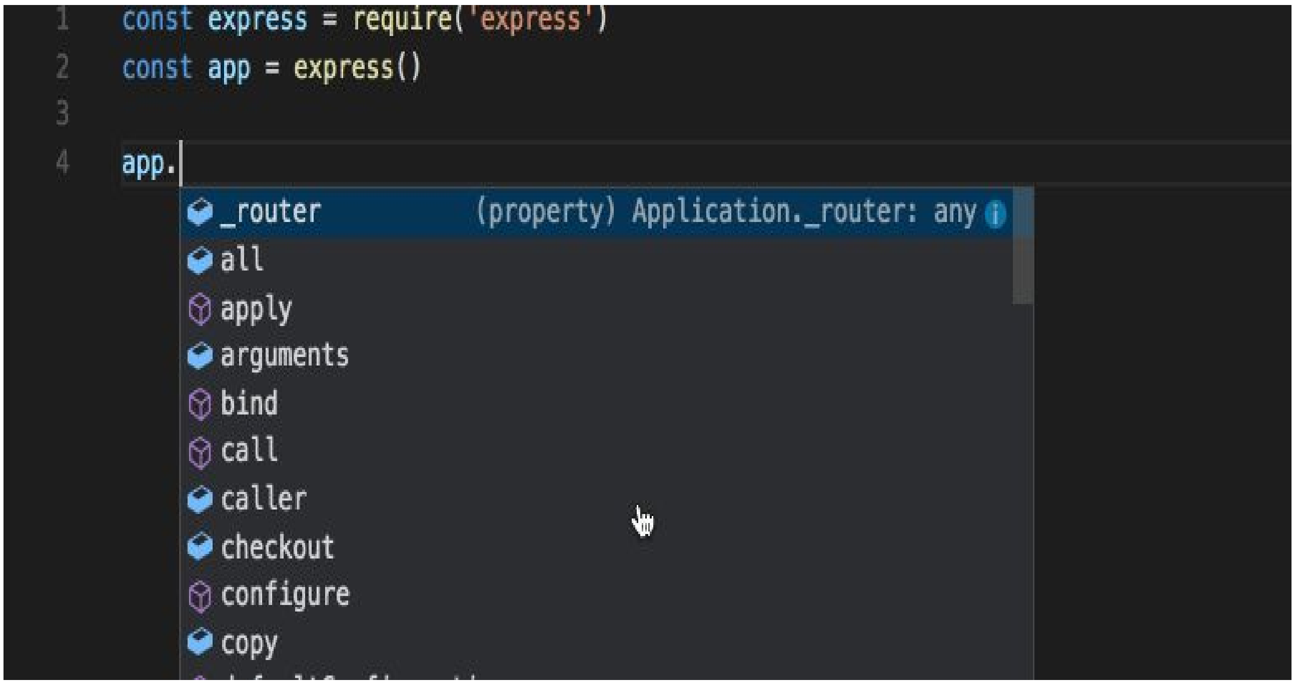
VS Code comes with a simple and intuitive layout that maximizes the space provided for the editor while leaving ample room to browse and access the full context of your folder or project. The UI is divided into five areas:

- Editor - The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- Side Bar - Contains different views like the Explorer to assist you while working on your project.
- Status Bar - Information about the opened project and the files you edit.
- Activity Bar - Located on the far left-hand side, this lets you switch between views and gives you additional context-specific indicators, like the number of outgoing changes when Git is enabled.
- Panels - You can display different panels below the editor region for output or debug information, errors and warnings, or an integrated terminal. Panel can also be moved to the right for more vertical space.

Each time you start VS Code, it opens up in the same state it was in when you last closed it. The folder, layout, and opened files are preserved.

Open files in each editor are displayed with tabbed headers (Tabs) at the top of the editor region.

IntelliSense:

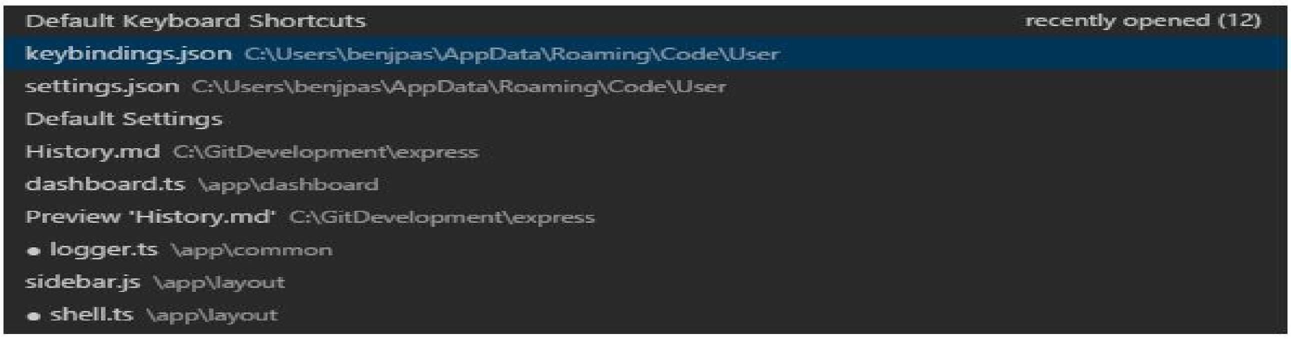


```
const express = require('express')
const app = express()

app.
```

IntelliSense is a general term for a variety of code editing features including: code completion, parameter info, quick info, and member lists. IntelliSense features are sometimes called by other names such as "code completion", "content assist", and "code hinting."

Code Navigation:

Visual Studio Code has a high productivity code editor which, when combined with programming language services, gives you the power of an IDE and the speed of a text editor.


```
Default Keyboard Shortcuts                                    recently opened (12)
keybindings.json  C:\Users\benjpas\AppData\Roaming\Code\User
settings.json  C:\Users\benjpas\AppData\Roaming\Code\User
Default Settings
History.md  C:\GitDevelopment\express
dashboard.ts  \app\dashboard
Preview 'History.md'  C:\GitDevelopment\express
● logger.ts  \app\common
sidebar.js  \app\layout
● shell.ts  \app\layout
```

Debugging:

VS Code has built-in debugging support for the Node.js runtime and can debug JavaScript, TypeScript, and any other language that gets transpiled to JavaScript.

For debugging other languages and runtimes (including PHP, Ruby, Go, C#, Python, C++, Powershell and many others), look for Debuggers extensions in our VS Code Marketplace or select Install Additional Debuggers in the top-level Debug menu.

Version Control:

Visual Studio Code has integrated source control and includes Git support in-the-box. Many other source control providers are available through extensions on the VS Code Marketplace.

Azure Extensions:

You can use Azure directly from Visual Studio Code through extensions. Using either the Azure CLI or the Azure App Service extension, you can have your application running in Azure in minutes

Docker:

Docker is a very popular container platform that lets you easily package, deploy, and consume applications and services. Whether you are a seasoned Docker developer or just getting started, Visual Studio Code makes it easy to author Dockerfile and docker-compose.yml files in your workspace. Visual Studio Code even supports generating and adding the appropriate Docker files based on your project type.

Let's start using the Code for our Application.

ENVIRONMENT CREATION:

Step1:

Creating the required environment.

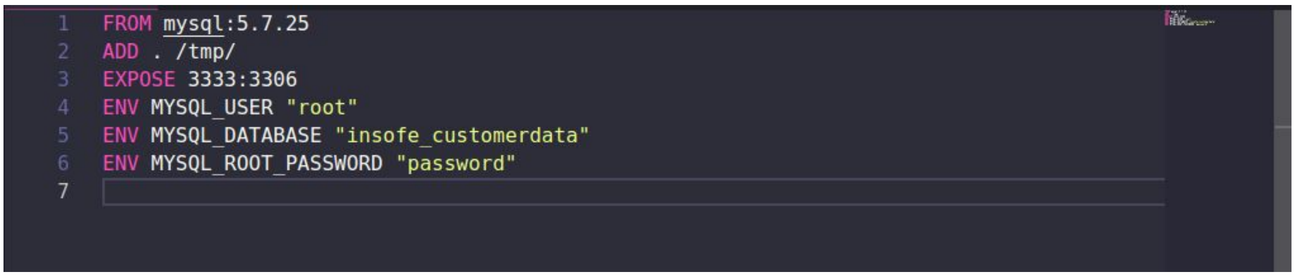
I. Create a folder called app which contains the remaining folders for the Application
II. Inside the app create the folders flask,nginx and mysql_db for creating the respective containers in the Docker

Step 2:

Create the Dockerfile for the MySQL inside the mysql_db folder

Step 3:

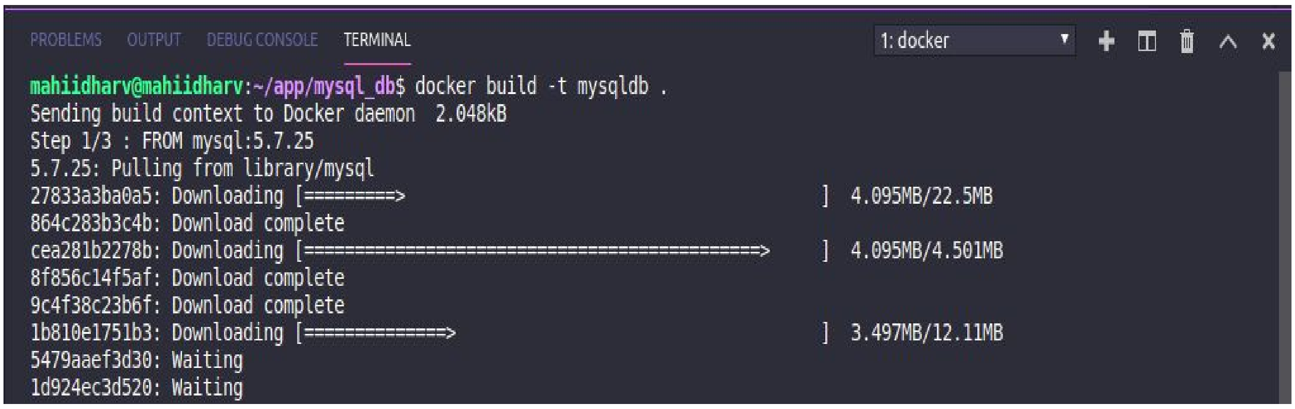
Copy the following code in the Dockerfile created.



```
FROM mysql:5.7.25
ADD . /tmp/
EXPOSE 3333:3306
ENV MYSQL_USER "root"
ENV MYSQL_DATABASE "insofe_customerdata"
ENV MYSQL_ROOT_PASSWORD "password"

```

Step 4:

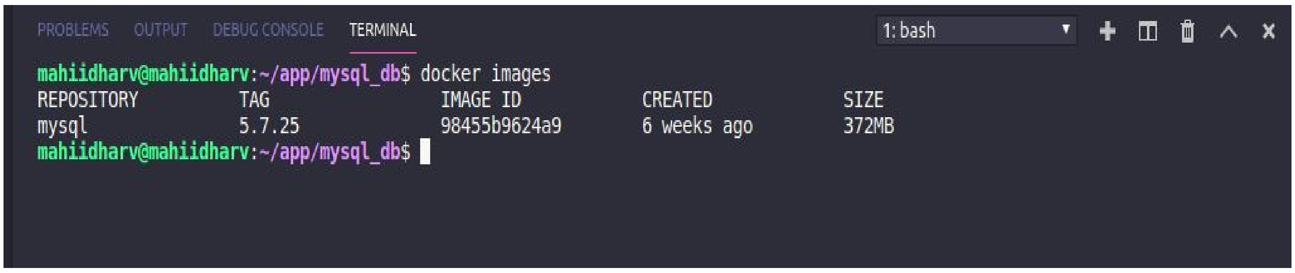
To run the docker file enter the command in the terminal of the code as



```
mahiidharv@mahiidharv:~/app/mysql_db$ docker build -t mysqldb .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM mysql:5.7.25
5.7.25: Pulling from library/mysql
27833a3ba0a5: Downloading [=========>                                         ]  4.095MB/22.5MB
864c283b3c4b: Download complete
cea281b2278b: Downloading [=============================================>     ]  4.095MB/4.501MB
8f856c14f5af: Download complete
9c4f38c23b6f: Download complete
1b810e1751b3: Downloading [==============>                                    ]  3.497MB/12.11MB
5479aaef3d30: Waiting
1d924ec3d520: Waiting
```

Step 5:

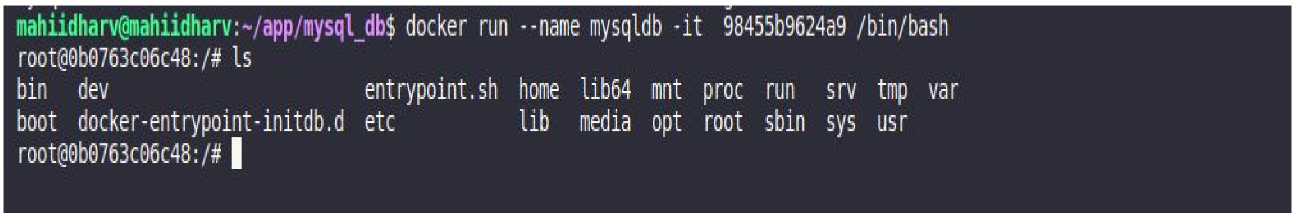
To know whether the docker image has been created run the command
Syntax : docker images



```
mahiidharv@mahiidharv:~/app/mysql_db$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mysql               5.7.25              98455b9624a9        6 weeks ago         372MB
mahiidharv@mahiidharv:~/app/mysql_db$ 
```

Step 6:

To create the container of the image we need to run the following command
Syntax : docker run --name <container name> -it <docker image name/ID> /bin/bash

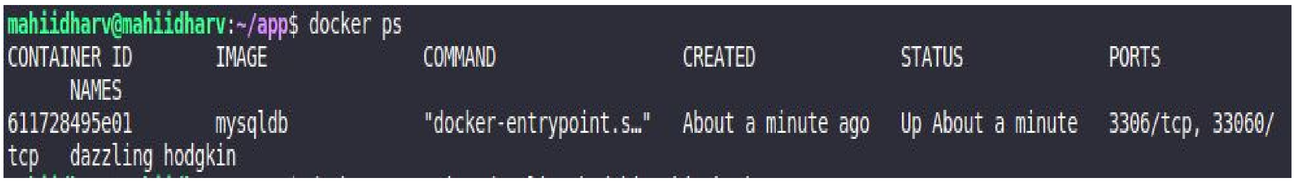


```
mahiidharv@mahiidharv:~/app/mysql_db$ docker run --name mysqldb -it 98455b9624a9 /bin/bash
root@0b0763c06c48:/# ls
bin   dev                         entrypoint.sh  home  lib64  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint-initdb.d  etc            lib   media  opt  root  sbin  sys  usr
root@0b0763c06c48:/# 
```

Step 7:

To run the container check for container id.

Syntax : docker ps

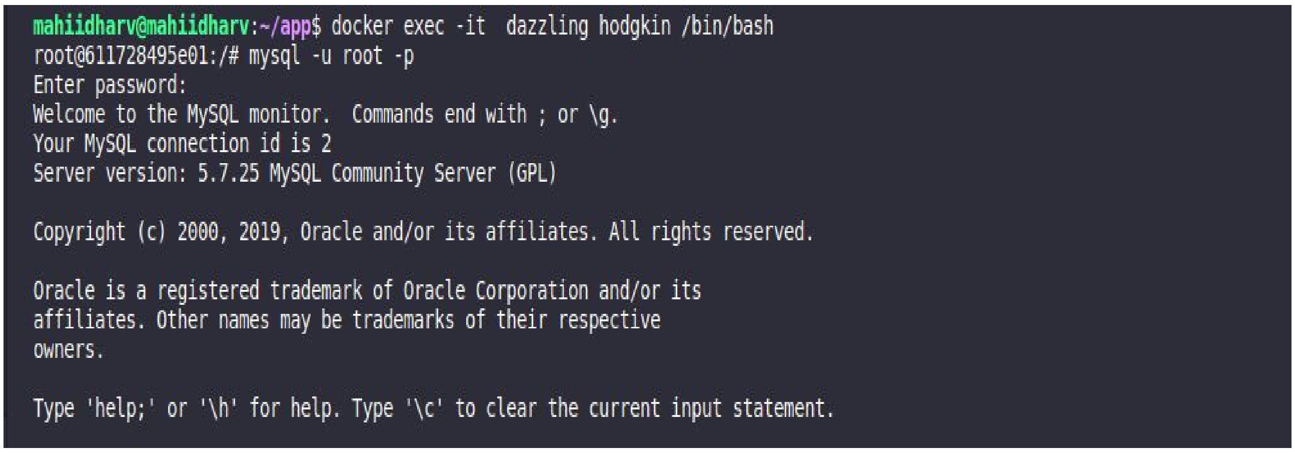


```
mahiidharv@mahiidharv:~/app$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
     NAMES
611728495e01        mysqldb             "docker-entrypoint.s…"   About a minute ago   Up About a minute   3306/tcp, 33060/
tcp   dazzling_hodgkin
```

Step 8:

To execute the commands inside the container run the command

Syntax : docker exec -it <container_id> /bin/bash



```
mahiidharv@mahiidharv:~/app$ docker exec -it  dazzling_hodgkin /bin/bash
root@611728495e01:/# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.25 MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

MYSQL - WORKBENCH :

Step 9:

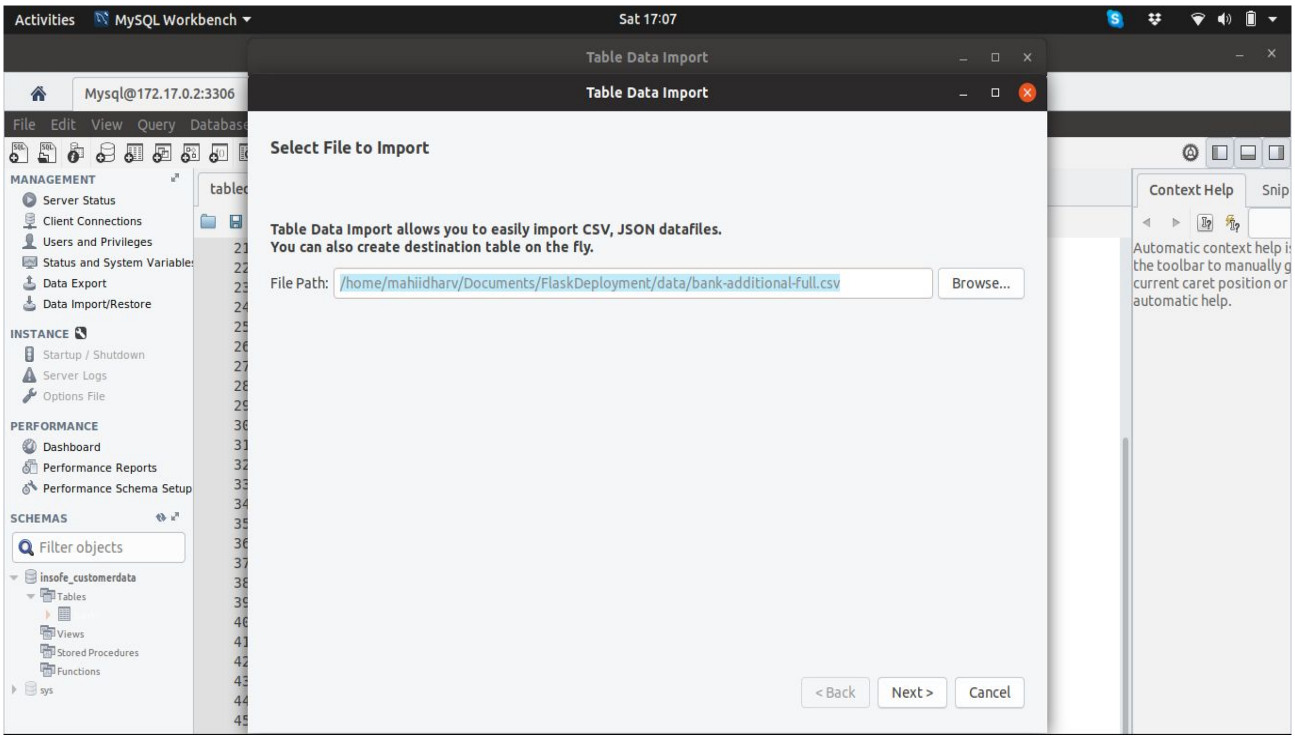
Run mysql queries from either command prompt or using the mysql-workbench.

Step 10:

Connecting the mysql-workbench use the
host : <docker ip>
port : 3306
user : root
password : password

Step 11:

Run the queries in the table_definitions.sql and import the data using the csv file and table import wizard in the mysql-workbench.



Select File to Import

Table Data Import allows you to easily import CSV, JSON datafiles.
You can also create destination table on the fly.

File Path: /home/mahiidharv/Documents/FlaskDeployment/data/bank-additional-full.csv

Now the mysql container is setup,we need to use the data in the mysql to train and build the machine learning model. The saved model should also be inside the docker environment. Therefore we need to create a container flask which consists of the web application code that uses the data from the database and lets out the application as a web page.

Flask:

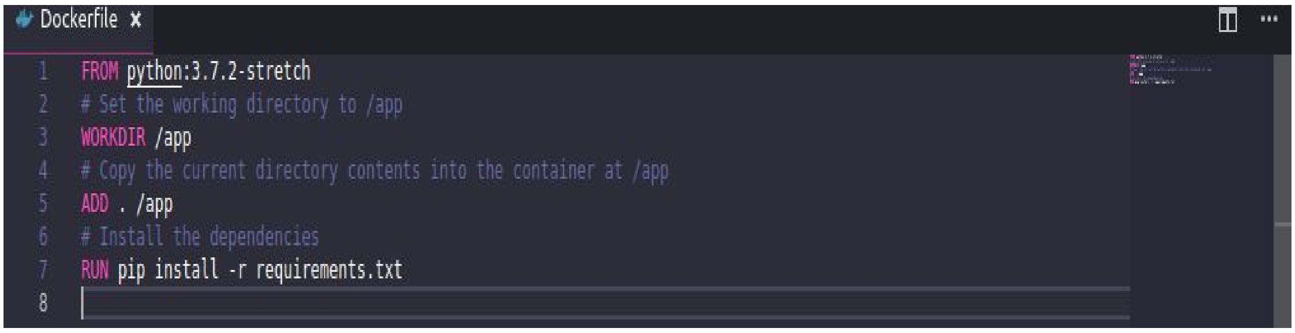
The container flask contains both the code for the model development and the code for the web app creation. The code for the model creation requires connecting to the database. The database details and the functions and the files required for coding are given as below. First of all the program is to be written in a language which is python here we need to install the python from the image. We use the Dockerfile again to create the environment for running our code inside the container.

Step 1:

Create a folder called flask inside the app folder.

Step 2:

Create the Dockerfile using the following commands

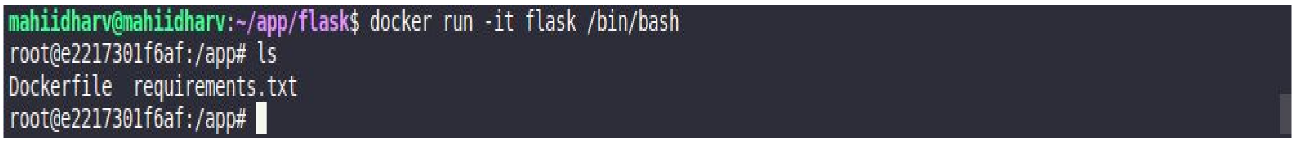


```
FROM python:3.7.2-stretch
# Set the working directory to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
ADD . /app
# Install the dependencies
RUN pip install -r requirements.txt
```

Step 3:

Similar to the above procedure build the image using the command

Syntax : docker build --tag <image_name> .



```
mahiidharv@mahiidharv:~/app/flask$ docker run -it flask /bin/bash
root@e2217301f6af:/app# ls
Dockerfile  requirements.txt
root@e2217301f6af:/app#
```

Step 4:

Create a directory apppkg inside the flask container. This apppkg consists of the code required to create the model and Flask app code as a package and can be accessed from a file outside.

Step 5:

Change the directory to apppkg and start to create the files required for creating the model that has been optimized and studied at the starting of the problem.

- Create a file called config.ini
- Create files called __init__.py,core.py , helpers.py ,model.py and views.py
- Create two folders data and pickle to hold the model related info and the data required for the test.
- Create a file wsgi.py to

Step 6:

Keeping the mysql container running we need to run the python codes to build the model and save it to the pickle folder.

Step 7:

We need to run the above codes to build the model as package and can be called from outside wsgi.py file

Let us see some basics of Flask

<u>Web Application Creation:</u>

We are using Flask on the Virtual environment to create the application. Therefore we need to understand the functionality of Flask to create a basic micro web application.

A minimal Flask application looks something like this:

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

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

1. First we imported the Flask class. An instance of this class will be our WSGI application.

2. Next we create an instance of this class. The first argument is the name of the application's module or package. If you are using a single module (as in this example), you should use __name__ because depending on if it's started as application or imported as module the name will be different ('__main__' versus the actual import name). This is needed so that Flask knows where to look for templates, static files, and so on.
3. We then use the route() decorator to tell Flask what URL should trigger our function.

4. The function is given a name which is also used to generate URLs for that particular function, and returns the message we want to display in the user's browser.

Just save it as hello.py or something similar. Make sure to not call your application flask.py because this would conflict with Flask itself. To run the application you can either use the flask command or python's -m switch with Flask.

Before you can do that you need to tell your terminal the application to work with by exporting the FLASK_APP environment variable:

```
$ export FLASK_APP=hello.py
$ flask run
 * Running on http://127.0.0.1:5000/
```

If you are on Windows, the environment variable syntax depends on command line interpreter.

On Command Prompt:

C:\path\to\app>set FLASK_APP=hello.py

And on PowerShell: PS

C:\path\to\app> $env:FLASK_APP = "hello.py"

Alternatively you can use python -m flask:

$ export FLASK_APP=hello.py

$ python -m flask run * Running on <u>http://127.0.0.1:5000/</u>

This launches a very simple builtin server, which is good enough for testing but probably not what you want to use in production. And also for the data we required some front end for the user to interact with the application. This will requires the templates to be rendered on the webpages and based on the user interaction we need to have the server code responding back with the apt messages. Also in the afore mentioned code there is a route function which will redirect us to different web pages. Let us understand the code further.

Routing :

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page. Use the route() decorator to bind a function to a URL.

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

Variable Rules:

You can add variable sections to a URL by marking sections with variable names.Your function then receives the variable name as a keyword argument. Optionally, you can use a converter to specify the type of the argument like <converter:variablename>

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return 'Subpath %s' % subpath
```

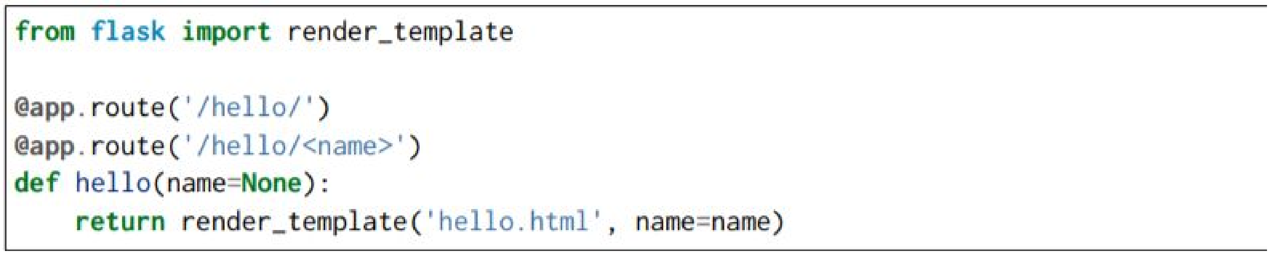
HTTP Methods:

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to GET requests. You can use the methods argument of the route() decorator to handle different HTTP methods.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

Rendering Templates:

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the Jinja2 template engine for you automatically. To render a template you can use the render_template() method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

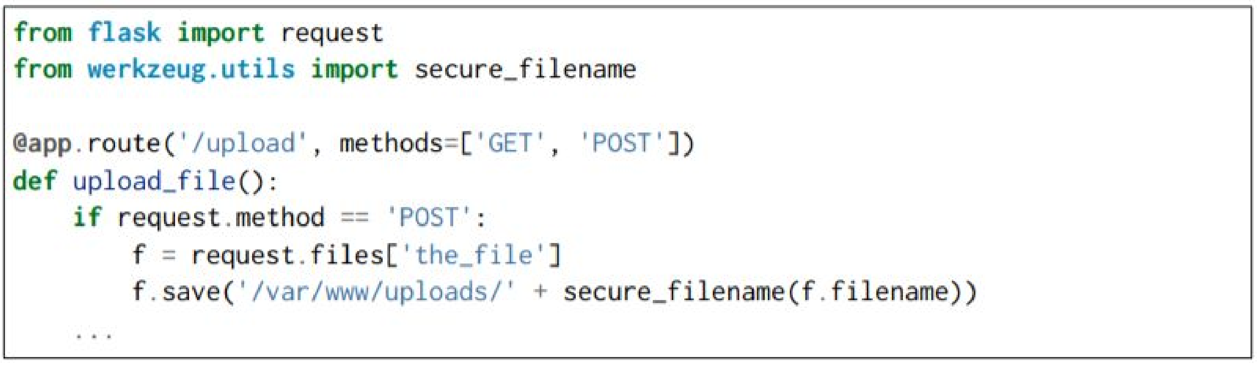


```python
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

File Uploads :

We can handle uploaded files with Flask easily. Just make sure not to forget to set the enctype="multipart/form-data" attribute on your HTML form, otherwise the browser will not transmit your files at all.

Uploaded files are stored in memory or at a temporary location on the file system. You can access those files by looking at the files attribute on the request object. Each uploaded file is stored in that dictionary. It behaves just like a standard Python file object, but it also has a save() method that allows you to store that file on the file system of the server. Here is a simple example showing how that works:



```python
from flask import request
from werkzeug.utils import secure_filename

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/' + secure_filename(f.filename))
    ...
```

Before going into the steps that are required for the deployment let us see the folder structure inside the app

Steps for Creating the Views Code :

Step 1:

Import the required libraries. Since we want to check the folder structure we can use the editor's instead of the Jupiter notebook like spyder or vscode. Let us start with the __init__.py. Here we import the Flask library and create the instance of the wsgi instance app. Also the actual code where the requirements for the application are generated we are calling it as views which are stored in the app package.

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

Step2: Importing the required libraries and creating the required functions.

i) Define the Libraries required

```
from app import app
import os
import pandas as pd
from flask import Flask, request, Response,render_template,flash
from flask_paginate import Pagination,get_page_args
from werkzeug import secure_filename
from joblib import load
```

ii) For running the model load the pickle file from the pickle folder created before and adjust the paths.

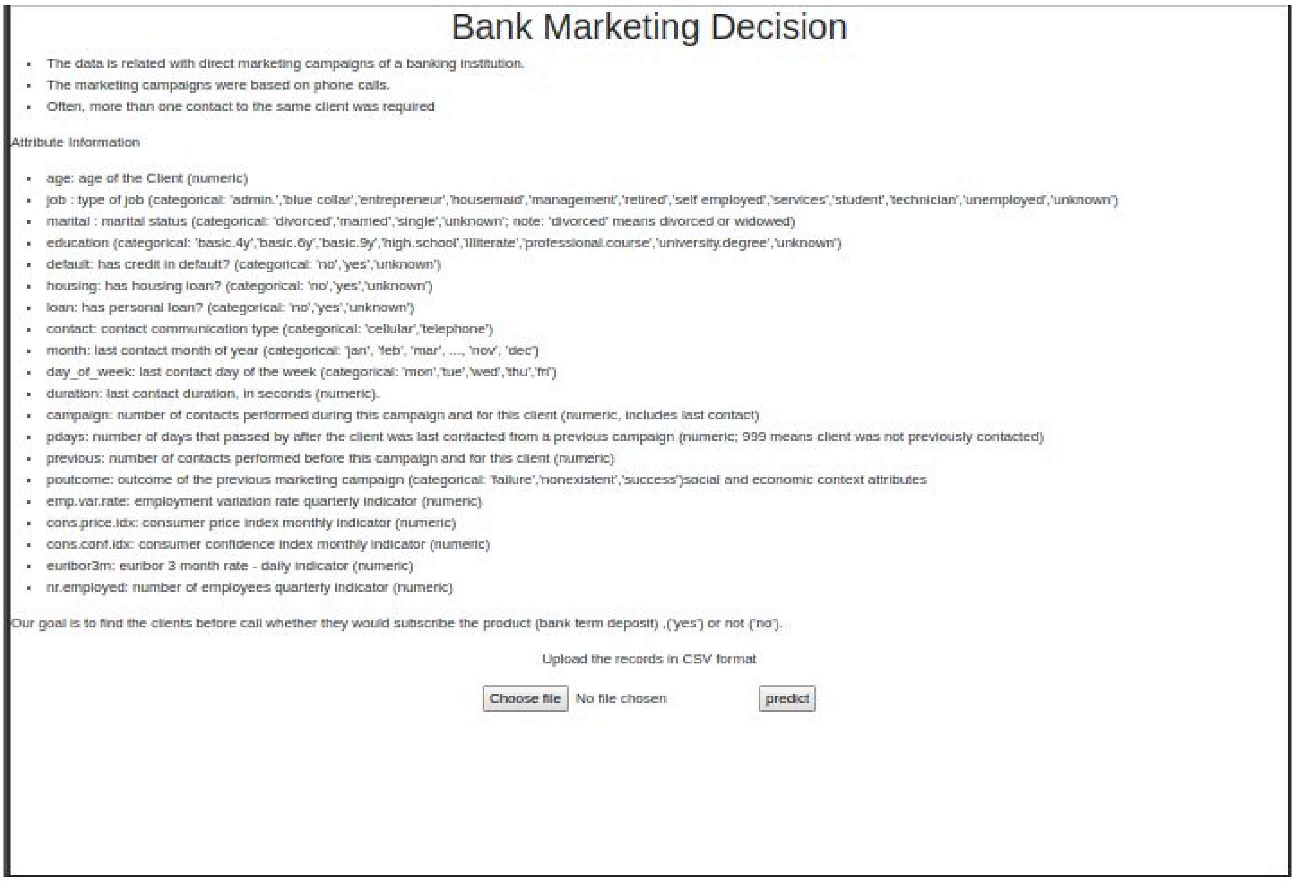
```
#Load the model
APP_ROOT = os.path.dirname(os.path.abspath(__file__))
print(APP_ROOT)
model = load(APP_ROOT+'/pickle/model.pkl_2019-04-10 10:34:05.282838')
```

iii) For calculating the predictions of the model we need to create multiple functions to get there . Define the functions as required.

```

def get_pages(data,per_page=10,offset=0):
    return data[offset:offset+per_page]


def result(APP_ROOT,model):
    data = pd.read_csv(APP_ROOT+"/data/test_cases.csv")
    try:
        data = data.drop('y',axis=1)
        data = data.drop('Unnamed: 0',axis=1)
    except:
        pass
    data = data.reset_index(drop=True)

    prediction = model.predict(data)
    output = prediction
    output = output.tolist()

    data['prediction'] = output
    cols = list(data.columns)
    col=[cols[0]]+[cols[len(cols)-1]]+cols[1:len(cols)-1]
    data = data[col]
    outputcsv = data
    return outputcsv
```

Step 3: Routing has to be created on the web pages based on the requirements. The plan of the webpages used some html templates. Please load the templates to the templates folder. The following are the templates. There are three templates as part of the application used upload.html,layout.html,submit.html.

Upload.html:



Bank Marketing Decision

- The data is related with direct marketing campaigns of a banking institution.
- The marketing campaigns were based on phone calls.
- Often, more than one contact to the same client was required

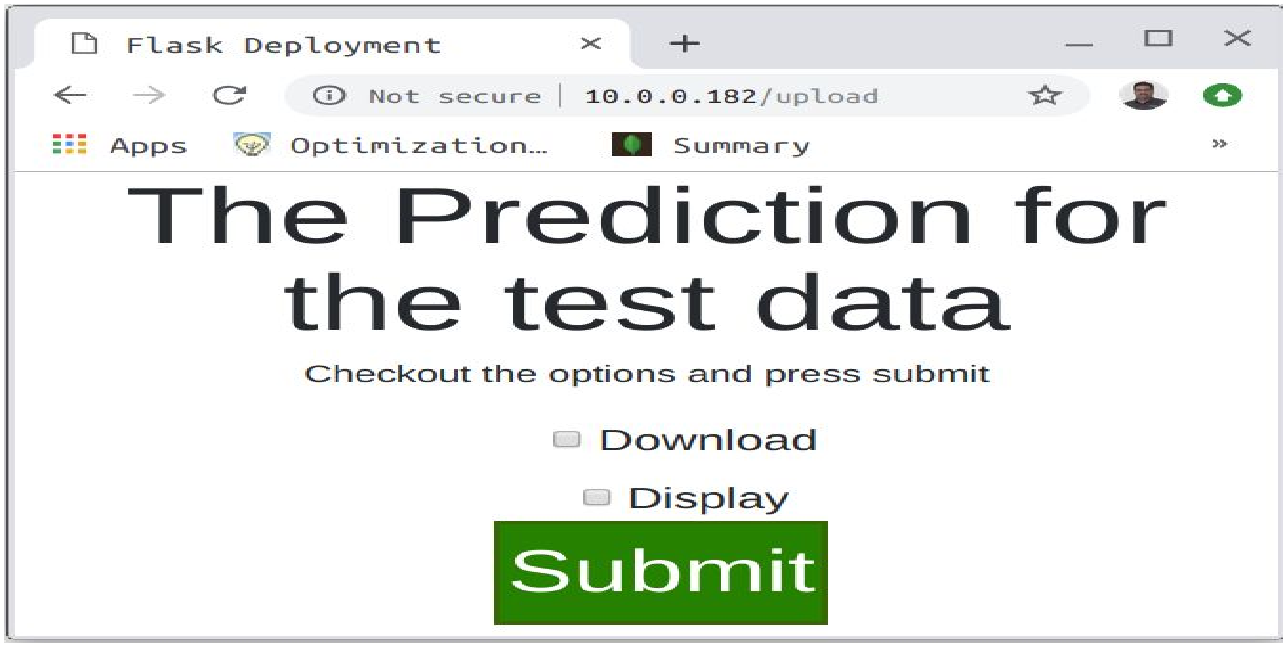
Attribute Information

- age: age of the Client (numeric)
- job : type of job (categorical: 'admin.','blue collar','entrepreneur','housemaid','management','retired','self employed','services','student','technician','unemployed','unknown')
- marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
- education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
- default: has credit in default? (categorical: 'no','yes','unknown')
- housing: has housing loan? (categorical: 'no','yes','unknown')
- loan: has personal loan? (categorical: 'no','yes','unknown')
- contact: contact communication type (categorical: 'cellular','telephone')
- month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
- day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
- duration: last contact duration, in seconds (numeric).
- campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
- pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
- previous: number of contacts performed before this campaign and for this client (numeric)
- poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')social and economic context attributes
- emp.var.rate: employment variation rate quarterly indicator (numeric)
- cons.price.idx: consumer price index monthly indicator (numeric)
- cons.conf.idx: consumer confidence index monthly indicator (numeric)
- euribor3m: euribor 3 month rate - daily indicator (numeric)
- nr.employed: number of employees quarterly indicator (numeric)

Our goal is to find the clients before call whether they would subscribe the product (bank term deposit) ,('yes') or not ('no').

Upload the records in CSV format

Layout.html:



The Prediction for the test data

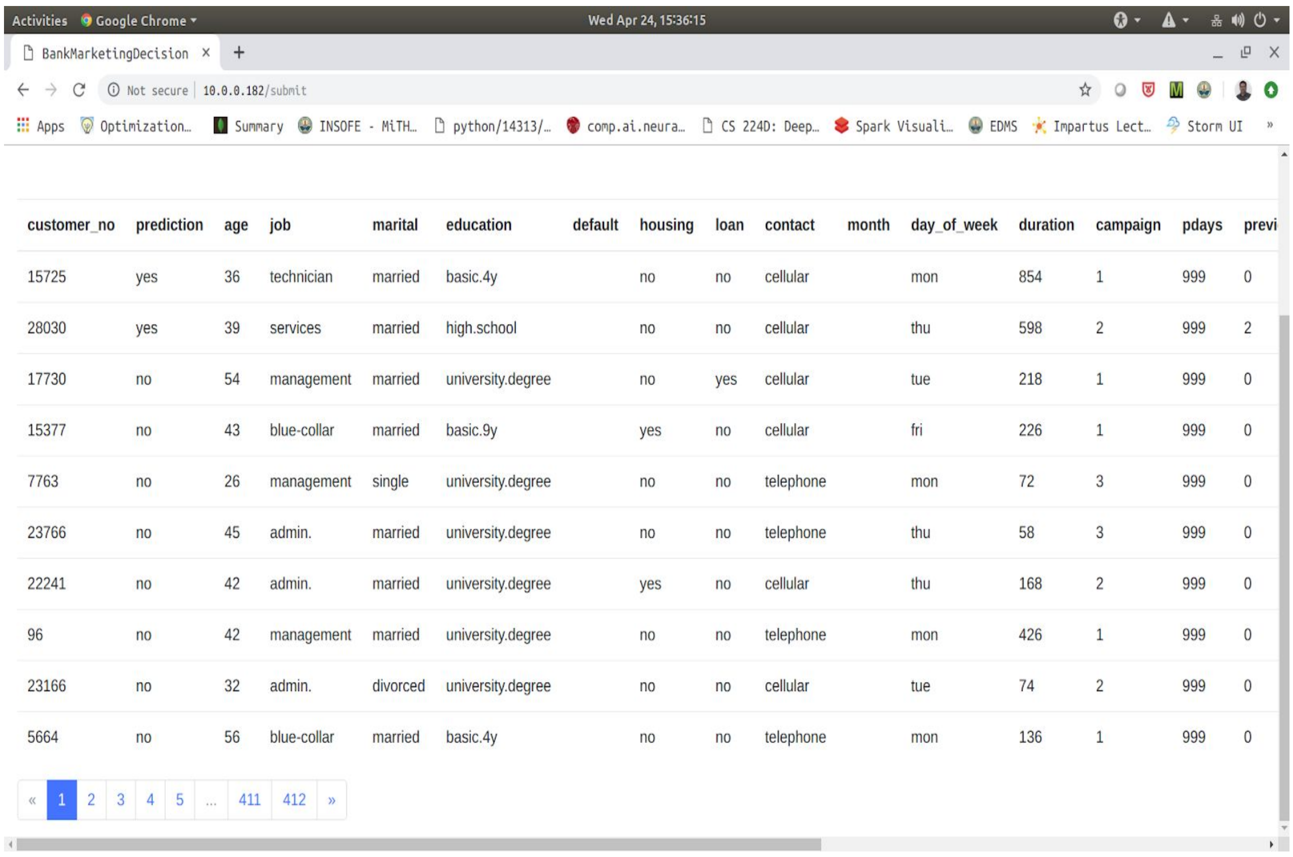
Checkout the options and press submit

Download

Display

Submit

Submit.html:



customer_no	prediction	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previ
15725	yes	36	technician	married	basic.4y		no	no	cellular		mon	854	1	999	0
28030	yes	39	services	married	high.school		no	no	cellular		thu	598	2	999	2
17730	no	54	management	married	university.degree		no	yes	cellular		tue	218	1	999	0
15377	no	43	blue-collar	married	basic.9y		yes	no	cellular		fri	226	1	999	0
7763	no	26	management	single	university.degree		no	no	telephone		mon	72	3	999	0
23766	no	45	admin.	married	university.degree		no	no	telephone		thu	58	3	999	0
22241	no	42	admin.	married	university.degree		yes	no	cellular		thu	168	2	999	0
96	no	42	management	married	university.degree		no	no	telephone		mon	426	1	999	0
23166	no	32	admin.	divorced	university.degree		no	no	cellular		tue	74	2	999	0
5664	no	56	blue-collar	married	basic.4y		no	no	telephone		mon	136	1	999	0

« 1 2 3 4 5 ... 411 412 »

i) Loading the first page (i.e) upload

```
@app.route('/')
def index():
   return render_template('upload.html')
```

ii) Loading the layout page based on the upload page inputs and store the data to the data folder which is uploaded in the first page

```
@app.route('/upload',methods=['POST'])
def upload():

  target = os.path.join(APP_ROOT,'data/')
  #print(target)
  if not os.path.isdir(target):
    os.mkdir(target)

  file= request.files['file']

  if file:
    filename = secure_filename(file.filename)
    destination = '/'.join([target,filename])
    data = pd.read_csv(file)

    data.to_csv(destination)



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

iii) Loading the submit page and get the requests from the user whether they want to disppla the results or save the csv file

```
@app.route('/submit',methods=['GET','POST'])
def submit():


  if request.form.getlist("download")==['on']:

      print(type(request.form.getlist("download")))

      outputcsv = result(APP_ROOT,model)
      #outputcsv.to_csv(destination)
      csv = outputcsv.to_csv()
      return Response(
          csv,
      mimetype="text/csv",
      headers={"Content-disposition":
           "attachment; filename=result.csv"})

  else:

      print(type(request.form.getlist("display")))

      outputcsv = result(APP_ROOT,model)

      dat = outputcsv.to_dict(orient="records")
      page, per_page, offset = get_page_args(page_parameter='page',
                         per_page_parameter='per_page')
      total = len(dat)

      pagination_pages = get_pages(data = dat,offset=offset, per_page=per_page)

      pagination = Pagination(page=page, per_page=per_page, total=total,
                 css_framework='bootstrap4')
          return render_template('submit.html',data = pagination_pages,page = page,per_page =
per_page,pagination=pagination)
```

Step 4 :
Create a run file outside the app folder calling it from outside with the following code call it as wsgi.py. For development purpose the debug=True and for the production we have only app.run()

```
from app import app

if __name__=="__main__":
  app.run(host="0.0.0.0",port=80,debug=True)
```

<u>WSGI server:</u>

As mentioned in the requirements.txt we are going to use the gunicorn as webserver gateway interface.(WSGI). It is just a python package and a command line execution required for the gunicorn.

The common commands used for running the application involve:

-c CONFIG, --config=CONFIG - Specify a config file in the form $(PATH), file:$(PATH), or python:$(MODULE_NAME).
-b BIND, --bind=BIND - Specify a server socket to bind. Server sockets can be any of $(HOST), $(HOST):$(PORT), or unix:$(PATH). An IP is a valid $(HOST).
-w WORKERS, --workers=WORKERS - The number of worker processes. This number should generally be between 2-4 workers per core in the server. Check the FAQ for ideas on tuning this parameter.

-k WORKERCLASS, --worker-class=WORKERCLASS - The type of worker process to run. You'll definitely want to read the production page for the implications of this parameter. You can set this to $(NAME) where $(NAME) is one of sync, eventlet, gevent, tornado, gthread, gaiohttp(deprecated). sync is the default. See the worker_class documentation for more information.
-n APP_NAME, --name=APP_NAME - If setproctitle is installed you can adjust the name of Gunicorn process as they appear in the process system table (which affects tools like ps and top).

Ex: gunicorn --workers=2 –b 0.0.0.0:80 wsgi:app

This will run two worker process which can bind to the ip address ported at 80. The module name can be a full dotted path. The variable name refers to a WSGI callable that should be found in the specified module.

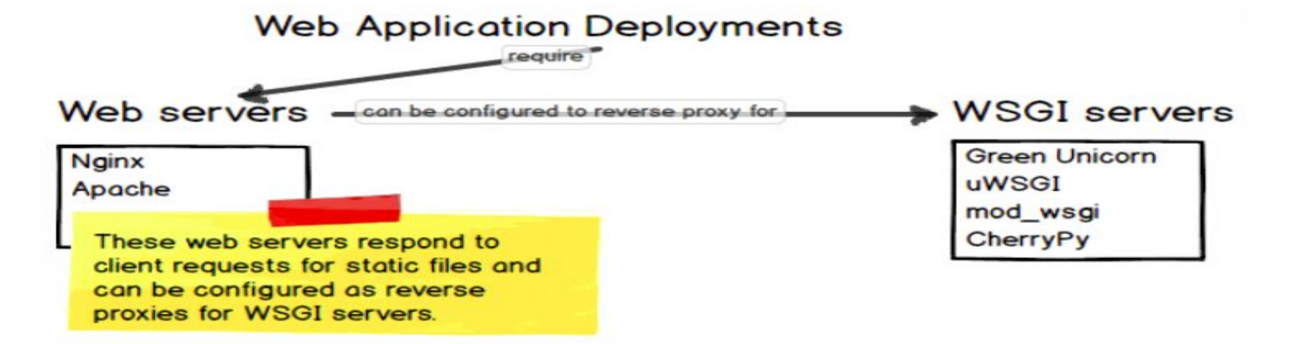
WebServer:

We have chosen Apache Nginx as the web server. If required we can choose the http server from apache or lihghtppd also . The server and client speak the standardized language of the World Wide Web. This standard language is why an old Mozilla Netscape browser can still talk to a modern Apache or Nginx web server, even if it cannot properly render the page design like a modern web browser can.

The basic language of the Web with the request and response cycle from client to server then server back to client remains the same as it was when the Web was invented by Tim Berners-Lee at CERN in 1989. Modern browsers and web servers have simply extended the language of the Web to incorporate new standards.

Nginx is commonly used as a web server to serve static assets such as images, CSS and JavaScript to web browser clients.

Nginx is also typically configured as a reverse proxy, which passes appropriate incoming HTTP requests to a WSGI server. The WSGI server produces dynamic content by running Python code. When the WSGI server passes its response, which is often in the HTML, JSON or XML format, the reverse proxy then responds to the client with that result.


Web Application Deployments
require
Web servers
can be configured to reverse proxy for
WSGI servers
Nginx
Apache
These web servers respond to client requests for static files and can be configured as reverse proxies for WSGI servers.
Green Unicorn
uWSGI
mod_wsgi
CherryPy

Setting up Nginx :

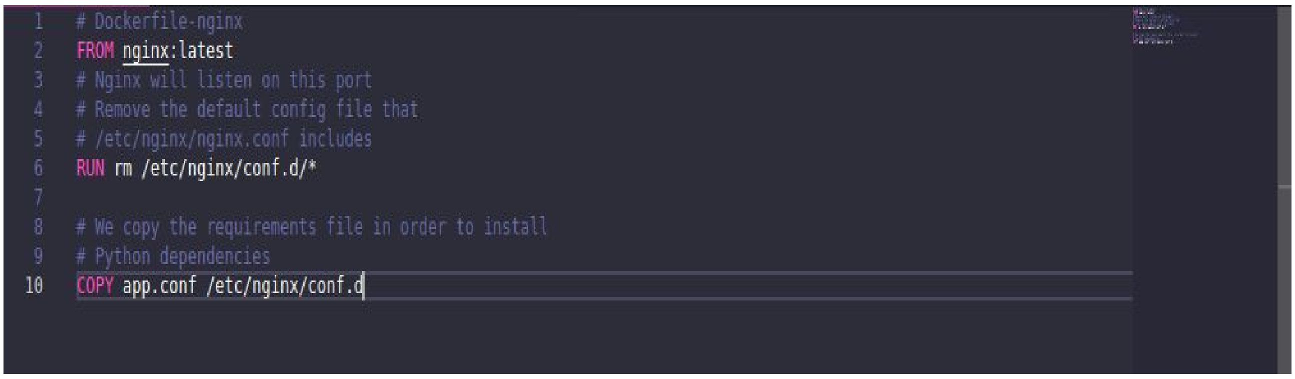
When nginx is installed (through **apt**), it provides a solid basic structure for how to set up your config files. We are using Nginx as a reverse proxy server . The following are the configurations that are required for the Nginx container

Step 1:

Create the nginx folder in the app folder

Step 2:

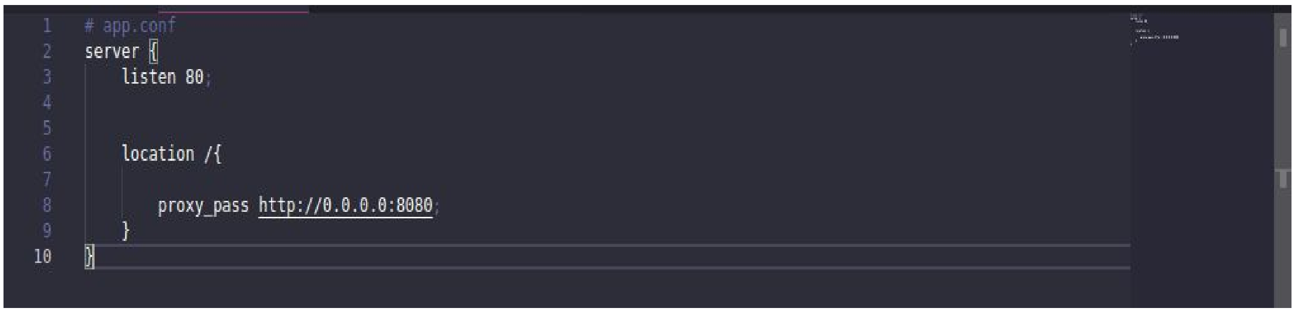
Create a Dockerfile for Nginx which has the following codes



```
# Dockerfile-nginx
FROM nginx:latest
# Nginx will listen on this port
# Remove the default config file that
# /etc/nginx/nginx.conf includes
RUN rm /etc/nginx/conf.d/*

# We copy the requirements file in order to install
# Python dependencies
COPY app.conf /etc/nginx/conf.d
```

Step 3:

Create a app.conf file



```
# app.conf
server {
    listen 80;


    location /{

        proxy_pass http://0.0.0.0:8080;
    }
}
```

To run the Nginx as a reverse proxy server we require every container to communicate each other further. To create that scenario we need to create a Docker Compose yaml file that can create the ports that can help the containers to get created and communicate each other.

Docker Compose Yml:

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. Compose works in all environments: production, staging, development, testing.

Using Compose is basically a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.

3. Run docker-compose up and Compose starts and runs your entire app.

   The docker-compose yml for the application is as follows:

```
version: "3.3"

services:

 mysql_db:
   build : ./mysql_db
   container_name: mysql_db_app
   environment:

     # So you don't have to use root, but you can if you like
     MYSQL_ROOT_PASSWORD: 'password'
     MYSQL_ROOT_HOST: '%'
   ports:
     # <Port exposed> : < MySQL Port running inside container>
     - '3333:3306'
   expose:
     # Opens port 3306 on the container
     - '3333'
     # Where our data will be persisted


 flask:
   build: ./flask
   container_name: flask_app
   command: gunicorn -w 4 --bind 0.0.0.0:8080 wsgi:app
   expose:
     - 80

   ports:
     - "80:8080"
   depends_on:
     - mysql_db
  nginx:
   image: nginx
   restart: always
   build:
     context: ./nginx/
     dockerfile: Dockerfile
   expose:
     - 8000
   ports:
     - "8000:80"
   depends_on:
     - flask
```