TagMe

REST API FOR OBJECT RECGONITION

SHAWAIZ SHAH (S76940)

Contents

Introduction

TagMe is an end-to-end pipeline for object recognition requests with a Rest API interface. It can be thought as a basic version of Google Vision Cloud API.

Following project was completed in two milestones;

- Milestone 1:

  Development of an object labelling routine. **Tensor flow's object detection API** was used for this purpose

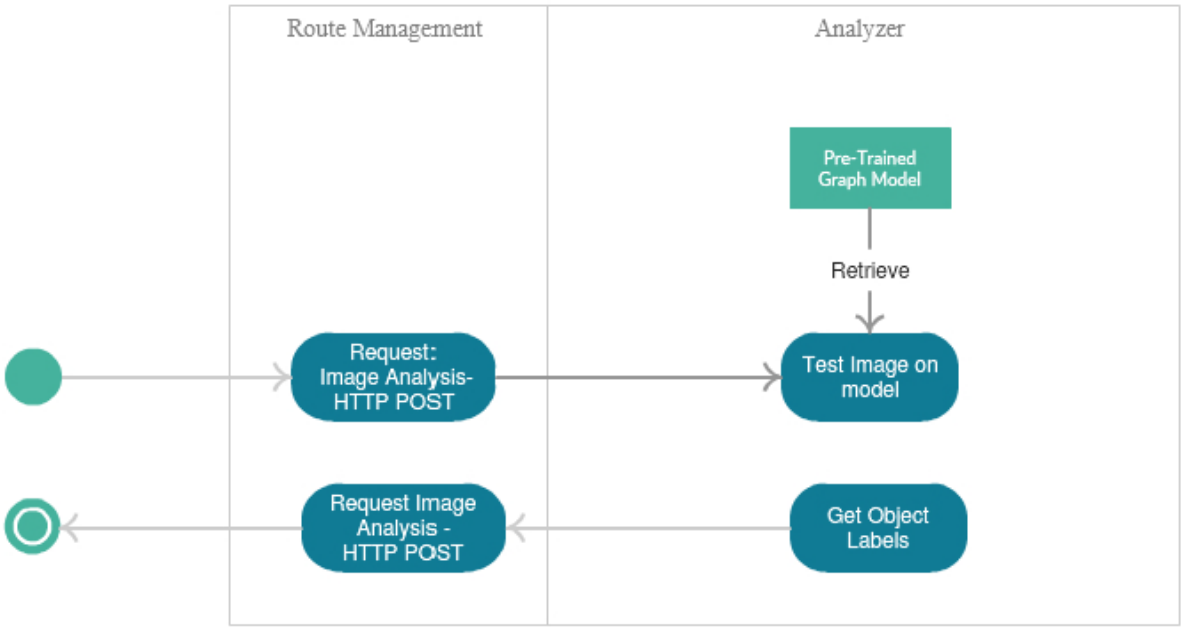
- Milestone 2:

  A Restful API interface for handling HTTP requests. API endpoints were developed using. Flask Restful (an extension of project Flask)

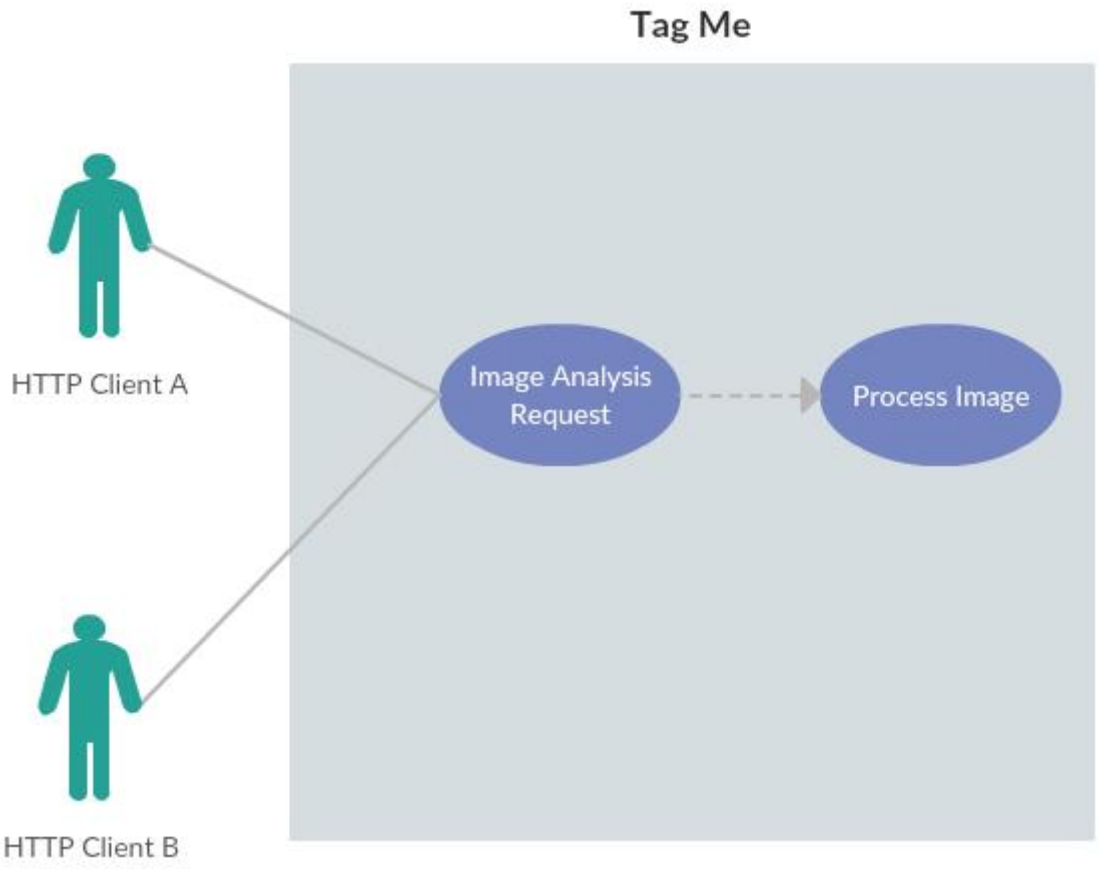
UML

Following are the UML diagrams of project TagMe.

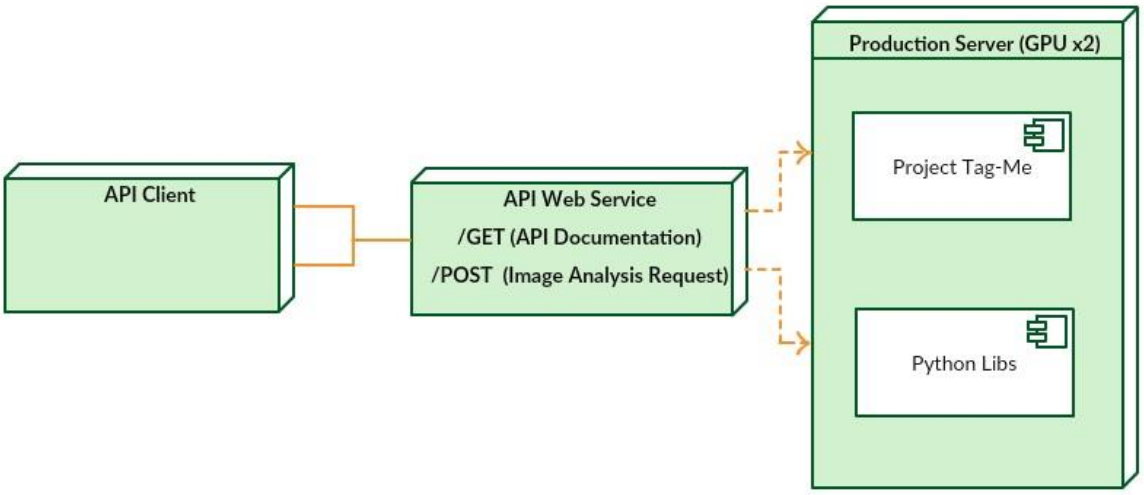
1. Activity Diagram



2. Use Case Diagram



3. Deployment Diagram



Metrics

For the code analysis and review, the industry standard tool, “SonarQube” was used. Initially code smell was significantly higher, which was gradually mitigated by applying refactoring techniques.

Please note that during code analysis, source codes from 3rd party libraries (e.g. object detection module from TensorFlow) were excluded.

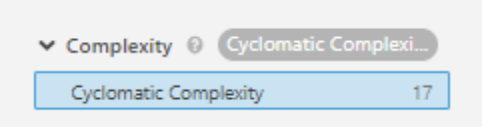
For Example;

- In one case, a large code block was broken down into smaller code blocks, hence reducing code complexity.
- Function parameters were increased/decreased for making method calls more simpler and easier.
- Logically related pieces of code were merged together under same functions.

Software metrics that were monitor during course of development were;

1. Complexity Metrics

In post-development code analysis, SonarQube indicated presence of Cyclomatic Complexity with a score of 17.

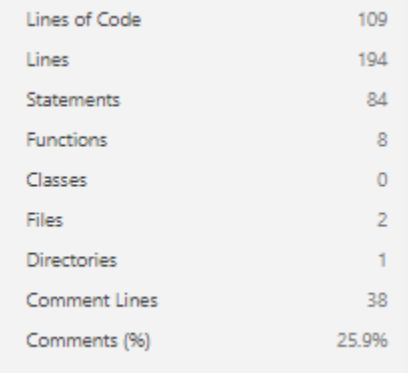


Complexity Score

Cyclomatic Complexity measures the minimum number of test cases required for full test coverage. Following measure indicates that my project requires 17 additional test cases for getting full test coverage.

2. Architectural Metrics

Following are screenshots from metric measures.



Architectural Metric Measures

Clean Code Development

1. Consistent naming convention

Throughout coding, naming scheme of camel case is used for both variables and functions. Moreover, for immutable/final variables all-upper-casing scheme is used.

2. Minimizing of Side-Effect using Context Management

Wherever necessary, side-effects have been tried to minimized by isolating its effect at local scope by making use of python's context management.

```
# get detault graph
with detection_graph.as_default():

  # get ref to graph def
  od_graph_def = tf.GraphDef()

  with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid: # read frozen graph in rb mode

    # serialized frozen graph
    serialized_graph = fid.read()

    # parse graph def
    od_graph_def.ParseFromString(serialized_graph) # copy frozen graph meta to prior initliazed tf def
    tf.import_graph_def(od_graph_def, name='') # import graph to tf session
```

Context Management Usage

3. Modularity

Source code has been divided into two modules. Module "Analyzer" is responsible for object recognition tasks, while module "Router" is responsible for routing of incoming and outgoing rest API requests.

```
from flask import Flask, jsonify, request
import analyzer

app = Flask(__name__)

@app.route('/', methods=['POST'])
def analyze():
    if not request.json or not 'url' in request.json:
        abort(400)

    url = request.json['url']

    labels = analyzer.predict_label(url)

    response = {
        "status": 200,
        "labels": labels
    }

    return jsonify(response), 201
```

Module Analyzer(analyzer.py) being imported in Module Router(app.py)

4. Exception Handling

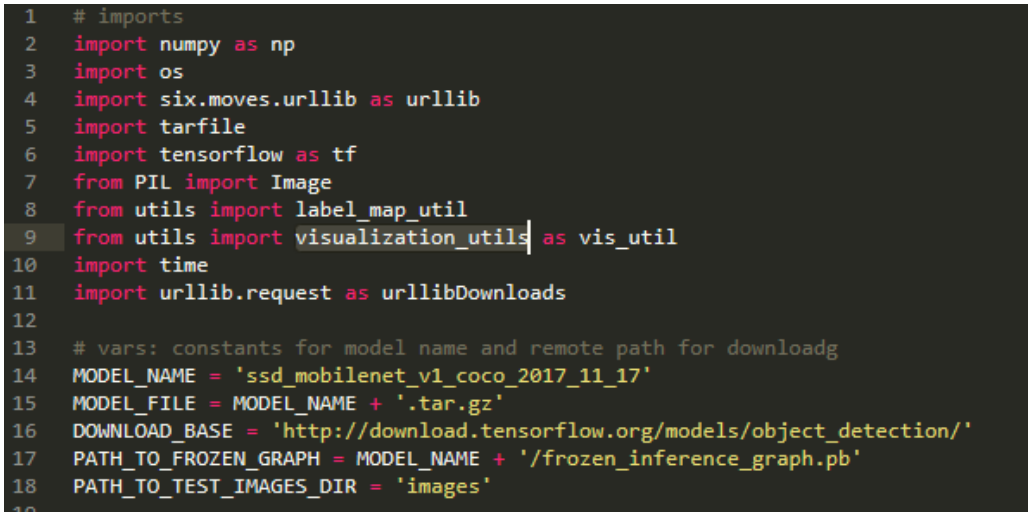
Wherever necessary exception handling blocks have been added to ensure that no runtime errors are thrown.

```
  try:
    text_format.Merge(label_map_string, label_map)
  except text_format.ParseError:
    label_map.ParseFromString(label_map_string)
```

Exception Handling Usage

5. Configurable Data at Higher Level

All configurable data (for example: Directory names, paths), have been placed at higher level, making tweaking of values easier for the purpose of debugging.



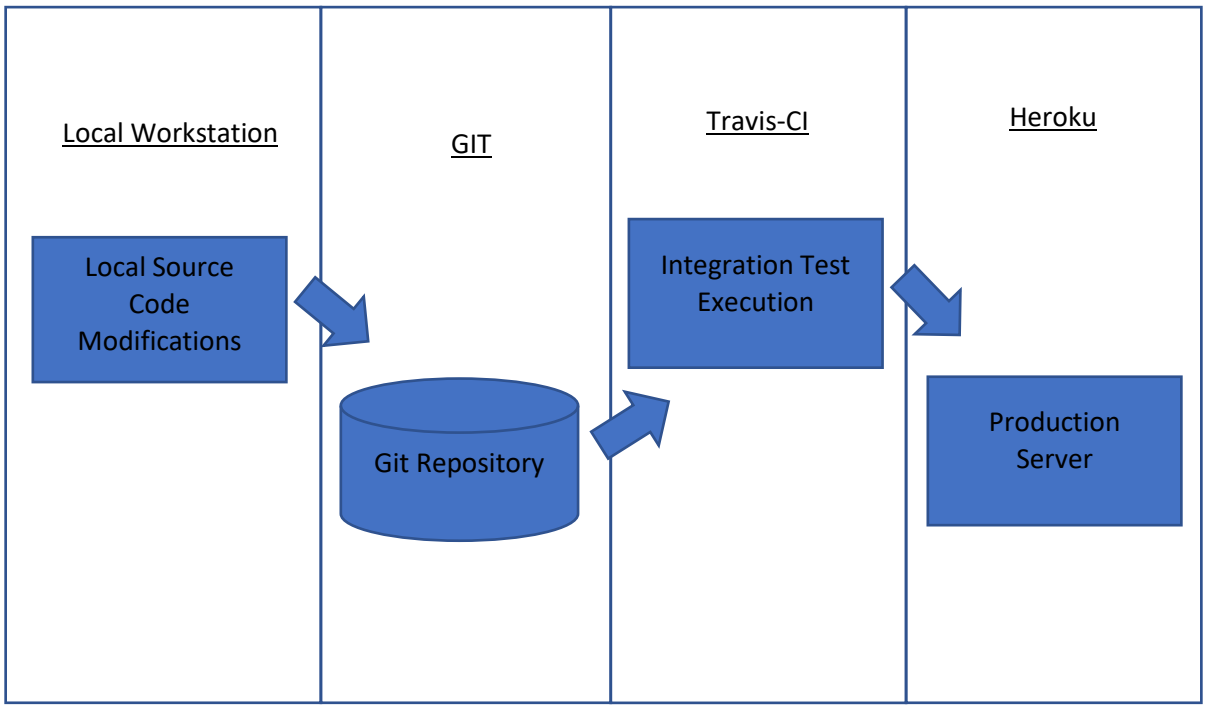
```
# imports
import numpy as np
import os
import six.moves.urllib as urllib
import tarfile
import tensorflow as tf
from PIL import Image
from utils import label_map_util
from utils import visualization_utils as vis_util
import time
import urllib.request as urllibDownloads

# vars: constants for model name and remote path for downloadg
MODEL_NAME = 'ssd_mobilenet_v1_coco_2017_11_17'
MODEL_FILE = MODEL_NAME + '.tar.gz'
DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'
PATH_TO_FROZEN_GRAPH = MODEL_NAME + '/frozen_inference_graph.pb'
PATH_TO_TEST_IMAGES_DIR = 'images'
```

Configurable Data and Constants at higher level

Continuous Delivery

Continuous Delivery and integration pipeline of this project is based on Git, Travis CI and Heroku. Travis CI is listening for modifications on Git repository. Any modifications in code base will trigger Travis CI, which will initiate a set of tests for validating successful integration of newly added code with already tested production-ready code in Git repository. Upon successful integration, Travis CI will deploy source code on our production server at Heroku.

Following diagram depicts code delivery flow;



Build Management (+ DSL)

For the purposes of build management this project is dependent on Travis CI since builds are being managed at the time of project deployment. Consider following example;

```
language: python
python:
  - "2.7"
  - "3.4"
  - "3.5"
cache: pip
install:
  - pip install -r requirements.txt
script:
  - echo "skipping tests"
  - chmod a+x images/
  - python scripts/clean.py
  # - py.test --cov-report term-missing --cov app -v
deploy:
  provider: heroku
  api_key:
    secure: "a4137ff6-1763-46d8-b8ab-435225dcdf00"
    app: tag-me
```

Build Script for Travis CI

During build deployment, Travis has been instructed to;

1. Check for compatibility of source code with python version 2.7, 3.4 and 3.5
2. Install pre-requisite packages of project found in file requirments.txt
3. Delete contents of directory “images” where images that are to be processed will be stored
4. Deploy fresh build on Heroku instance for application “tag-me” (unique identifier on Heroku.com)

Following is the output of following script;

```
$ pip --version
pip 18.0 from /home/travis/virtualenv/python3.5.6/lib/python3.5/site-packages/pip (python 3.5)
$ pip install -r requirements.txt
$ echo "skipping tests"
skipping tests
The command "echo "skipping tests"" exited with 0.

$ chmod a+x images/
The command "chmod a+x images/" exited with 0.

$ python scripts/clean.py
The command "python scripts/clean.py" exited with 0.

```

Build Script Output Log Part 1

```
498 store build cache
500
501 $ rvm $(travis_internal_ruby) --fuzzy do ruby -S gem install dpl
508
509 Installing deploy dependencies
535 Preparing deploy
542 Deploying application
560 Removing images/test.txt
561 HEAD detached at f85406a
562 Changes not staged for commit:
563   (use "git add/rm <file>..." to update what will be committed)
564   (use "git checkout -- <file>..." to discard changes in working directory)
565
566         deleted:    images/test.txt
567
568 no changes added to commit (use "git add" and/or "git commit -a")
569 Dropped refs/stash@{0} (345b145a77417ffca8f90228d2d70b895614ae15)
570
571 Done. Your build exited with 0.
```

Build Script Output Log Part 1

P.S: Following script is also an example of Domain Specific Language. This script has been written in YAML (*.yml).*

Functional Programming

Throughout this project, good practices for functional programming have been adopted. Following are few examples of such practices;

1. Final Data Structures Usage

Few variables have been made made immutable in code. Hence using as constant.

```
12
13   # vars: constants for model name and remote path for downloadg
14   MODEL_NAME = 'ssd_mobilenet_v1_coco_2017_11_17'
15   MODEL_FILE = MODEL_NAME + '.tar.gz'
16   DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'
17   PATH_TO_FROZEN_GRAPH = MODEL_NAME + '/frozen_inference_graph.pb'
18   PATH_TO_TEST_IMAGES_DIR = 'images'
```

2. Side Effect Free Functions

As discussed earlier, wherever necessary context management has been used. So that its effect could remain locally. Hence using them would not result any side effects.

```
# get detault graph
with detection_graph.as_default():

  # get ref to graph def
  od_graph_def = tf.GraphDef()

  with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid: # read frozen graph in rb mode

    # serialized frozen graph
    serialized_graph = fid.read()

    # parse graph def
    od_graph_def.ParseFromString(serialized_graph) # copy frozen graph meta to prior initliazed tf def
    tf.import_graph_def(od_graph_def, name='') # import graph to tf session
```

3. Use of higher order functions

Higher order functions like map and filter have also been used.

```
dataset = dataset.map(decode_func)
```

4. Clojures/Anonymous Functions

Usage of anonymous functions like “Lamda” have also been made, for getting rid from unnecessary function signature bodies where required.

```
true_fn=lambda: random_source_id
```