denzel Documentation

Release 0.1.5

Elior Cohen

Jan 26, 2019

Contents:

Denzel is a model-agnostic lean framework for fast and easy API deployment of machine learning models.
Denzel is data scientist first; while it leverages production grade tools and practices, its main goal is to abstract the mechanics from the data scientist allowing model deployment with ease.

Warning: Denzel only works on Linux environment at the moment. Windows support will be added - keep yourself updated by checking the *development state* for new features and support.

CHAPTER 1

Introduction

Denzel was designed by data scientists for data scientists.

The framework utilizes production grade tools and practices to allow quick deployment, while abstracting all the mechanics behind the scenes.

To launch a service with API interface, task queue management and monitoring the only thing required from the data scientist is to fill four functions, and denzel will take care of the rest.

Those four functions are `pipeline.load_model()`, `pipeline.verify_input()`, `pipeline.process()` and `pipeline.predict()` (also called *Pipeline Methods*).

1.1 Motivation

Data scientists are not necessarily software engineers. In most companies' structure the data science team is separated from the other engineering teams.

In a standard workflow the data science team will get some data and a problem to solve. The team will then iterate until the point where they believe their solution is worthy and will want to test it out in production environment.

Once this point is reached it is not guaranteed that the engineering teams will have the availability to wrap the solution (model) with a deployment system.

Denzel was created to fill this necessity, to create a framework where a data scientist could deploy a model, in a fast and reliable way.

Need to deploy? Denzel got your back.

1.2 Architecture and Task Flow

Under the hood denzel uses four docker containers and the system overview is as follows:

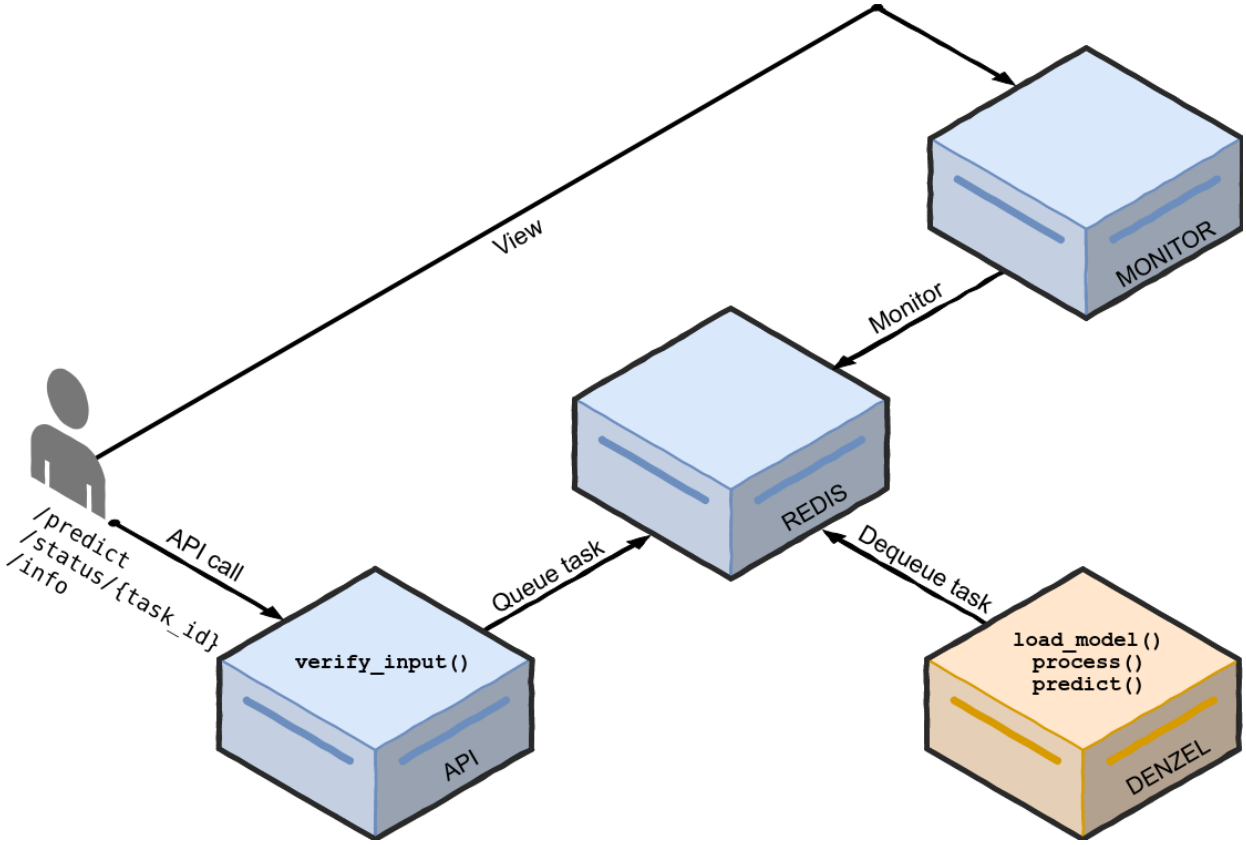


Fig. 1: High level overview of the denzel architecture. Each box is a container (API, Redis, denzel and Monitor). In orange is the main container, where the model will reside. On top of each container we can see the functions that will need to be implemented by the data scientist

Each box, is a docker container with its own purpose.
The API exposes three endpoints to the user, the */predict*, */status/{task_id}* and */info* endpoints.

On system startup (`denzel start`) a worker inside the **denzel container** will use the *`pipeline.load_model()`* function to load the model into memory, and will keep it there until the worker shuts down (should happen only when the system is shut down).
The worker, is always listening for tasks in the queue, which is inside the **Redis container**.
When an end-user sends a POST request to the `/predict` endpoint, the request will first go through the *`pipeline.verify_input()`* function to make sure the schema is as expected by denzel (defined by the data scientist).
If all is well, the request is turned into a task and is sent into the task queue and a response will be sent back to the user with a task ID.
As the task enters the queue, if the worker is not already busy it will consume the task. The task then goes through the *`pipeline.process()`* function, which accepts the output of the *`pipeline.verify_input()`* function and parses it to model-ready data.
If processing is successful, the model-ready data enters the *`pipeline.predict()`* function where all the model magic happens and then a response with the prediction will be sent to a `callback_uri` which was supplied by the end-user initially.
At any time during the lifetime of a task, the end-user can view its status through the */status/{task_id}* endpoint, or through the built-in UI exposed by the **Monitor container**.

Note: By default, denzel will run tasks on parallel, one task per core - for example if the host has 8 cores, that means 8 tasks can be executed in parallel.

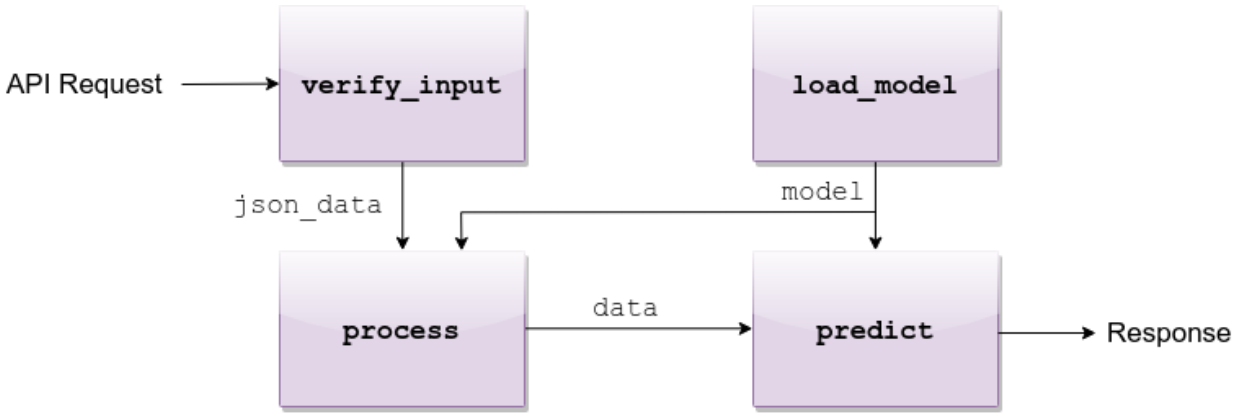


Fig. 2: API request flow through the four methods

1.3 Tasks and Synchrony

Denzel separates tasks from predictions jobs. This is so denzel can support batching, where in one task multiple prediction jobs will be sent.
Consider the following request-body (as JSON) sent by an end-user:

```
{
    "callback_uri": "http://alonzo.trainingday.com/stash",
    "data": {"predict_id_1": {"feature1": 0.45, "feature2": -1.99},
             "predict_id_2": {"feature1": 0.09, "feature2": -6.15}}
}
```

As a response, the end-user will be returned:

```
{
    "task_id": "1ba0dccre8d67dsa",
    "status": "SUCCESS"
}
```

Note: The `"status": "SUCCESS"` means the task has been accepted - **not** that there was a prediction made yet. Essentially it means the request has passed the *`pipeline.verify_input()`* method and has made it into the queue

Here we have a task submitted by the user with two separate prediction jobs.

If all goes well, as the end-user sends the request it will **synchronously** get a response with a task ID, uniquely identifying the task submitted.

After the data has been processed and the prediction has been made, an **asynchronous** response will be sent back to the `callback_uri` and will look something like this:

```
{
    "predict_id_1": "hotdog",
    "predict_id_2": "not_hotdog"
}
```

1.3.1 Why Async Responses?

The use of asynchronous responses is very common in API services.

Denzel, does not want to limit the data scientist and understands that processing and prediction (espcially of batches) might take longer than the end-user response waiting timeout.

Using asynchronous responses, the system virtually unlimited in time it takes to return a response, even though it's recommended to respond as fast as possible.

CHAPTER 2

Tutorial

A step by step tutorial for installing and launching a denzel deployment.
For this tutorial we will train and deploy a Iris classifier, based on the Iris dataset.

2.1 Prerequisites

Python 3 and docker are necessary for denzel.
Optionally, you can set up a virtualenv.

1. Install docker, verify its installation by running

```
$ docker --version
Docker version 18.06.1-ce, build e68fc7a
```

2. Install docker-compose > 1.19.0, verify its installation by running

```
$ docker-compose --version
docker-compose version 1.22.0, build f46880fe
```

3. User must be a part of the `docker` group (be able to run `docker` commands without `sudo`). Details on how to achieve that can be found in the official docs.
4. (Optional) Install virtualenv and create one to work on.

Note:

If opting for `virtualenv` understand that the deployment will not be using this env - it will only be used to install denzel on it.
The deployment itself will use the interpreter and packages installed into the containers (more on this further down).

2.2 Installation

If you are using a virtualenv, make sure you have it active.
To install denzel, simply run

```
$ pip install denzel
```

After this command completes, you should have the `denzel` *Command Line Interface (CLI)* available.
Once available you can always call `denzel --help` for the help menu.

2.3 Toy Model

Before actually using denzel, we need a trained model saved locally.
If you already have one, you can skip this sub-section.

For this tutorial, we are going to use scikit-learn's default SVM classifier and train it on the Iris dataset.
The following script downloads the dataset, trains a model and saves it to disk

```
import pandas as pd
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
import pickle

# -------- Load data --------
IRIS_DATA_URL = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
↪data"
IRIS_DATA_COLUMNS = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
↪'class']

iris_df = pd.read_csv(IRIS_DATA_URL,
                      names=IRIS_DATA_COLUMNS)


# -------- Spit train and test --------
features, labels = iris_df.values[:, 0:4], iris_df.values[:, 4]

test_fraction = 0.2
train_features, test_features, train_labels, test_labels = train_test_split(
    features, labels,
    test_size=test_fraction)

# -------- Train and evaluate --------
model = SVC()
model.fit(X=train_features, y=train_labels)

print(model.score(X=test_features, y=test_labels))
>> 0.9666666666666667
```

(continues on next page)

(continued from previous page)

```
# -------- Save for later --------
SAVED_MODEL_PATH = '/home/creasy/saved_models/iris_svc.pkl'
with open(SAVED_MODEL_PATH, 'wb') as saved_file:
    pickle.dump(
        obj=model,
        file=saved_file)
```

Great, we have a model trained to populate the deployment!

2.4 Starting a denzel Project

To start a project, you first have to run the command *startproject*, as a result denzel will build for you the following skeleton

Note:

Denzel supports GPU deployments. If you are going to run your deployment on a GPU, make sure you install nvidia-docker and use the `--gpu` flag.

```
$ denzel startproject iris_classifier
Successfully built iris_classifier project skeleton
$ cd iris_classifier
$ tree
.
|-- Dockerfile
|-- __init__.py
|-- app
|   |-- __init__.py
|   |-- assets
|   |   `-- info.txt  <-------------------------- Deployment information
|   |-- logic
|   |   |-- __init__.py
|   |   `-- pipeline.py  <----------------------- Pipeline Methods
|   `-- tasks.py
|-- docker-compose.yml
|-- entrypoints
|   |-- api.sh
|   |-- denzel.sh
|   `-- monitor.sh
|-- logs
`-- requirements.txt  <--------------------------- Requirements
```

To make denzel fully operational, the only files we'll ever edit are:

1. `requirements.txt` - Here we'll store all the pip packages our system needs
2. `app/assets/info.txt` - Text file that contains deployment information about our model and system
3. `app/logic/pipeline.py` - Here we will edit the body of the *Pipeline Methods*

These steps, are exactly what this tutorial is all about.

Tip:

A good practice will be to edit only the body of functions in `pipeline.py` and if you wish to add your own custom functions that will be called from within `pipeline.py`, you should put them on a separate file inside the `app/logic` directory and import them.

2.5 Requirements

When we've built our toy model, we used `scikit-learn` so before anything we want to specify this requirement in the `requirements.txt` file.

Open your favorite file editor, and append `scikit-learn`, `numpy` and `scipy` as requirements - don't forget to leave a blank line in the end.

Your `requirements.txt` should look like this

```
# ------------------------------------------------------------
#                           USER GUIDE
# Remember this has to be a lightweight service;
# Keep that in mind when choosing which libraries to use.
# ------------------------------------------------------------
scikit-learn
numpy
scipy
```

Take heed to the comment at the top of the file. Keep your system as lean as possible using light packages and operations in the pipeline methods.

2.6 Define Interface (API)

Our end users will need to know what is the JSON scheme our API accepts, so we will have to define what is the accepted JSON scheme for the */predict* endpoint.

In our *toy model*, we have four features the model expects: 'sepal-length', 'sepal-width', 'petal-length' and 'petal-width'.

Since we are going to return an *async response*, we also need to make sure we include a callback URI in the scheme.

Finally we'll want to support batching, so the following JSON scheme should suffice

```
{
    "callback_uri": <callback_uri>,
    "data": {<unique_id1>: {"sepal-length": <float>,
                            "sepal-width": <float>,
                            "petal-length": <float>,
                            "petal-width": <float>},
```

(continues on next page)

(continued from previous page)

```
            <unique_id2>: {"sepal-length": <float>,
                           "sepal-width": <float>,
                           "petal-length": <float>,
                           "petal-width": <float>},
            ...}
}
```

Also let's include a documentation of this interface and the model version in our `app/assets/info.txt` file that will be available to the end user in the */info* endpoint.
For example we might edit `info.txt` to something like this

```
# =====================  DEPLOYMENT  ======================
```

```
                          v0.1.5

# ========================  MODEL  ========================

Model information:
    Version: 1.0.0
    Description: Iris classifier

For prediction, make a POST request for /predict matching the following scheme

{
    "callback_uri": "http://alonzo.trainingday.com/stash",
    "data": {<unique_id1>: {"sepal-length": <float>,
                            "sepal-width": <float>,
                            "petal-length": <float>,
                            "petal-width": <float>},
             <unique_id2>: {"sepal-length": <float>,
                            "sepal-width": <float>,
                            "petal-length": <float>,
                            "petal-width": <float>},
             ...}
}
```

Looks great, now end users can see this info using GET requests!

2.7 Launch (partial project)

In an ideal scenario, we would launch a project only after we have completed all necessary tasks for a full deployment.

For guidance and simplicity sake of this tutorial, we will launch a partial project and complete tasks gradually.

What we have now is a skeleton, an editted `info.txt` and `requirements.txt` files and we can launch our API, without the functionality of the */predict* endpoint (yet).
Inside project directory run:

```
$ denzel launch

Creating network "iris_classifier_default" with the default driver
Pulling redis (redis:4)...
4: Pulling from library/redis
802b00ed6f79: Pull complete
8b4a21f633de: Pull complete
92e244f8ff14: Pull complete
fbf4770cd9d6: Pull complete
.
.
```

Note: By default denzel will occupy port 8000 for the API and port 5555 for monitoring. If one of them is taken, denzel will let you know and you can opt for other ports - for more info check the *launch* command documentation.

If this is the first time you launch a denzel project, the necessary docker images will be downloaded and built.
What is going on in the background is necessary for building the containers that will power the deployment.

If you are not really familiar with docker you can think of images like classes in programming, they define the structure of an object, and containers are like the instances.
In the context of docker the objects the images define are actually virtual machines and the containers we create from them is where our code will run on.

This whole process might take a few minutes, so sit back and enjoy an Oscar winning performance by the man himself.

Once done if everything went right you should see the end of the output looking like this:

```
Starting redis   ... done
Starting api     ... done
Starting denzel  ... done
Starting monitor ... done
```

This indicates that all the containers (services) were created and are up.
Once they are up the services will start installing the packages we specified in `requirements.txt`, you can view the status of the services by using the *status* command, optionally with the `--live` flag.
If you would run it right away you'd expect to see:

```
$ denzel status
Services:
    denzel - PIP INSTALLING...
    monitor - PIP INSTALLING...
    api - PIP INSTALLING...
    redis - UP
```

When all the installing is done and everything is ready, you'll see all the statuses change to `UP` with an additional line `Worker: worker@iris_classifier - UP` indicating the worker is ready.
If you want to see the messages printed out throughout the installation, you can use the *logs* command.
At any time during the lifetime of your project, if you want to add more pip packages, just insert them to the `requirements.txt` file and use the *updatereqs* command.

Tip:

Using the `denzel status --live` command is a great way to monitor the system. When conducting installations and loading it is a great way to get a high level live view of the system.
For lower level view, examining the outputs of the containers, use the live view of the logs using `denzel logs --live`.

For sanity check, assuming you have deployed locally, open your favorite browser and go to http://localhost:8000/info . You should see the contents of `info.txt` (assuming all services are up).
At any time, you can stop all services using the *stop* command and start them again with the *start* command.
From this moment forward we shouldn't use the *launch* command as a project can and needs to be launched once.
If for any reason you wish to relaunch a project (for changing ports for example) you'd have to first *shutdown* and then *launch* again.

2.8 Pipeline Methods

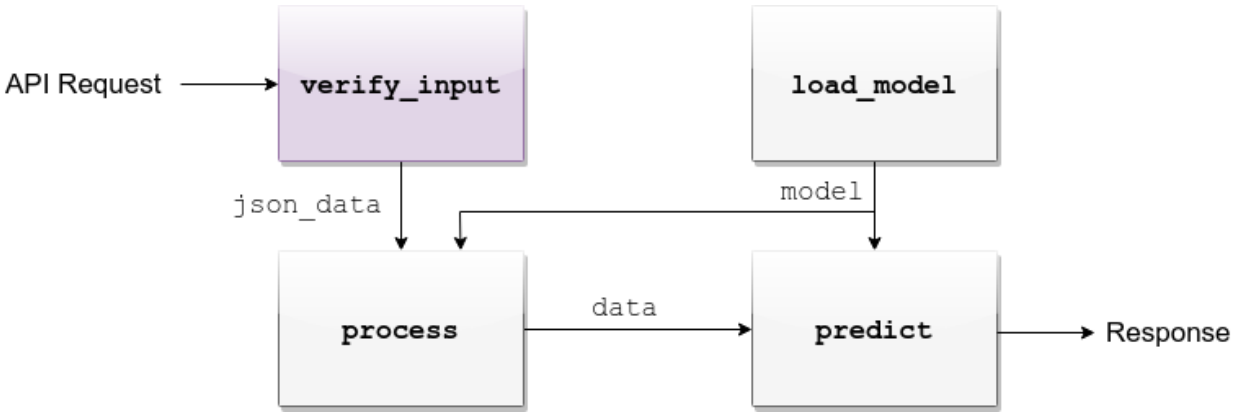
Now is the time to fill the body of the *pipeline methods*. They are all stored inside `app/logic/pipeline.py`.
Open this file in your favorite IDE as we will go through the implementation of these methods.

2.8.1 verify_input

When a user makes a request, the first pipeline method that the request will meet is *verify_input*.
The *verify_input* method is responsible for making sure the JSON data received matches the *interface we defined*.
In order to do that, lets edit the *verify_input* method to do just that:

```
FEATURES = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width']

def verify_input(json_data):
    """
    Verifies the validity of an API request content
```

(continues on next page)



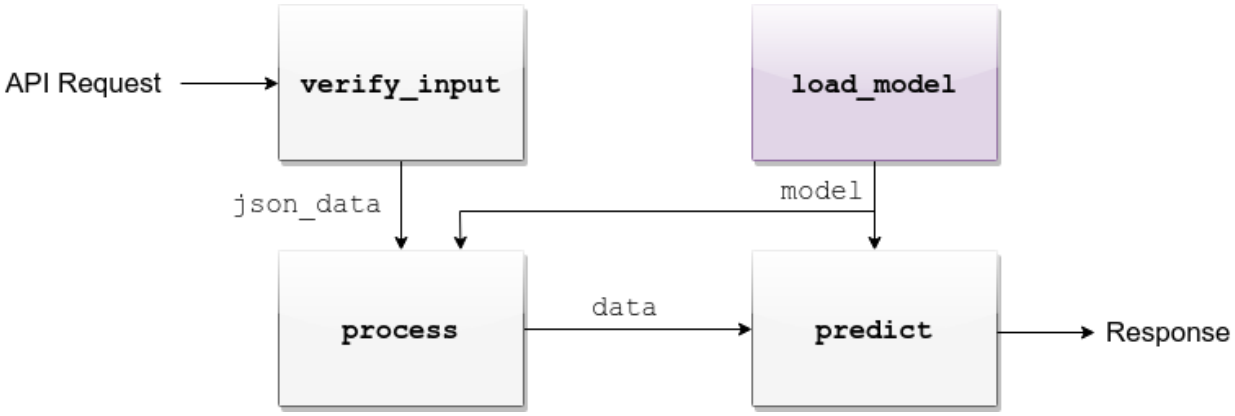
(continued from previous page)

```
    :param json_data: Parsed JSON accepted from API call
    :type json_data: dict
    :return: Data for the the process function
    """

    # callback_uri is needed to sent the responses to
    if 'callback_uri' not in json_data:
        raise ValueError('callback_uri not supplied')

    # Verify data was sent
    if 'data' not in json_data:
        raise ValueError('no data to predict for!')

    # Verify data structure
    if not isinstance(json_data['data'], dict):
        raise ValueError('jsondata["data"] must be a mapping between unique id and␣
↪features')

    # Verify data scheme
    for unique_id, features in json_data['data'].items():
        feature_names = features.keys()
        feature_values = features.values()

        # Verify all features needed were sent
        if not all([feature in feature_names for feature in FEATURES]):
            raise ValueError('For each example all of the features [{}] must be␣
↪present'.format(FEATURES))

        # Verify all features that were sent are floats
        if not all([isinstance(value, float) for value in feature_values]):
            raise ValueError('All feature values must be floats')

    return json_data
```

In the verification process implementation, you may throw any object that inherits from `Exception` and the message attached to it will be sent back to the user in case he tackles that exception.

Tip: For JSON scheme verification, you can consider using the jsonschema library.

2.8.2 `load_model`



load_model is the method responsible for loading our saved model into memory and will keep it there as long as the worker lives.

This method is called when denzel starts up and is called only once - unlike *verify_input*, *process* and *predict* which are called one time per request.

So our model will be accessible for reading, we must copy it into the project directory, preferably to `app/assets`. Once copied there, the assets directory should be as follows:

```
$ cd app/assets/
$ ls -l

total 8
-rw-rw-r-- 1 creasy creasy 1623 Sep 14 14:35 info.txt
-rw-rw-r-- 1 creasy creasy 3552 Sep 14 08:55 iris_svc.pkl
```

Now if we'll look at `app/logic/pipeline.py` we will find the skeleton of *load_model*.

Edit it so it loads the model and returns it, it should look something like:

```
import pickle

.
.

def load_model():
    """
    Load model and its assets to memory
    :return: Model, will be used by the predict and process functions
    """
```

(continues on next page)

(continued from previous page)

```
    with open('./app/assets/iris_svc.pkl', 'rb') as model_file:
        loaded_model = pickle.load(model_file)

    return loaded_model
```

Note:

When using paths on code which is executed inside the containers (like the pipeline methods) the current directory is always the project main directory (where the `requirements.txt` is stored). Hence the saved model prefix above is `./app/assets/...`.

When we edit the pipeline methods, the changes do not take effect until we restart the services.
As we just edited a pipeline method, we should run the *restart* command so the changes apply.
Navigate back into the project main directory and run `denzel restart` and after the services have restarted the changes will take effect.
To verify all went well you can examine the logs by running the *logs* command - if anything went wrong we will see it there (more about that in *Debugging*).

Warning:

When loading heavy models (unlike the tutorial classifier) that take long time to be read, you might want to wait for it to load before making any requests.
To do that, you should watch the output of the *status* command and check if your worker is ready, optionally with the `--live` flag. If your model is indeed taking much time to load, the output should like like follows:

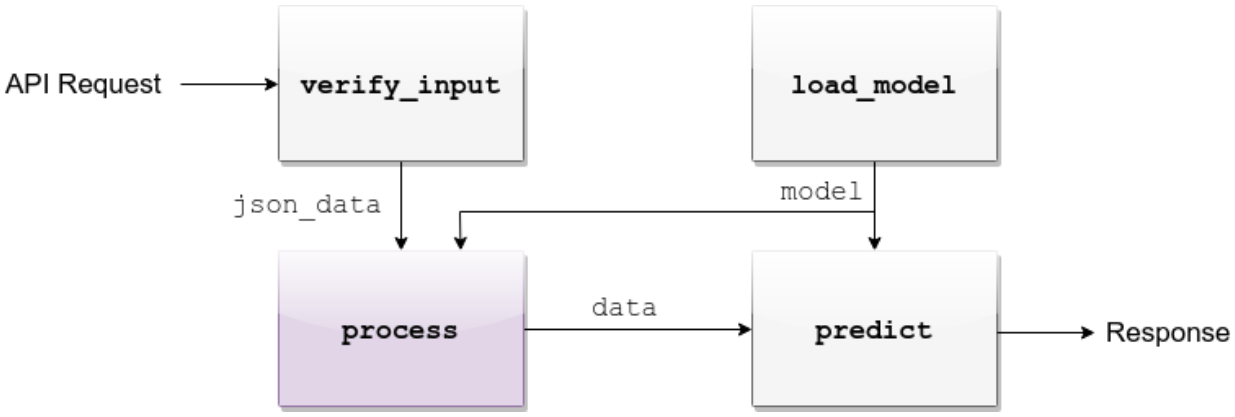
```
$ denzel status

Services:
    denzel - UP
    monitor - UP
    api - UP
    redis - UP
Worker: all - LOADING...
```

This means all the services are up, but API endpoints (as well as monitoring) are not available yet as the worker is still loading.

2.8.3 process

The output of the *verify_input* and *load_model* methods are the input to the *process* method.
The model object itself is not always necessary, but it is there if you want to have some kind of loaded resource available for the processing, in this tutorial we won't use the model in this method.

Now we are in possession of the JSON data, and we are already sure it has all the necessary data for making predictions.

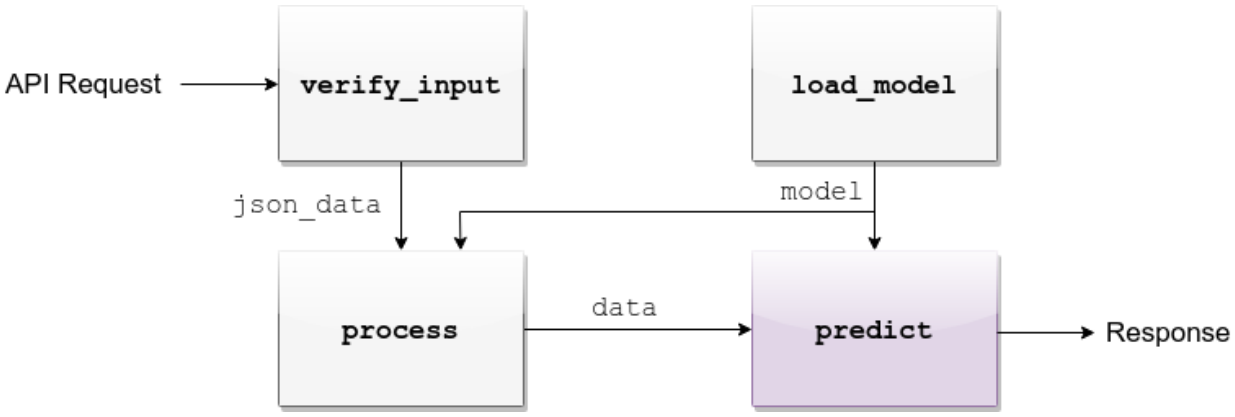


Our model though, does not accept JSON, it expects four floats as input, so in this method we will turn the JSON data into model-ready data.
For our use case, we should edit the function to look as follows:

```
.
.
import numpy as np
.
.

def process(model, json_data):
    """
    Process the json_data passed from verify_input to model ready data

    :param model: Loaded object from load_model function
    :param json_data: Data from the verify_input function
    :return: Model ready data
    """

    # Gather unique IDs
    ids = json_data['data'].keys()

    # Gather feature values and make sure they are in the right order
    data = []
    for features in json_data['data'].values():
        data.append([features[FEATURES[0]], features[FEATURES[1]],␣
↪features[FEATURES[2]], features[FEATURES[3]]])

    data = np.array(data)
    """
    data = [[float, float, float, float],
            [float, float, float, float]]
    """

    return ids, data
```

2.8.4 predict

The output of *process* and *load_model* are the input to the *predict* method.



The final part of a request lifecycle is the actual prediction that will be sent back as response.
In our example in order to do that we would edit the method to look as follows:

```
def predict(model, data):
    """
    Predicts and prepares the answer for the API-caller

    :param model: Loaded object from load_model function
    :param data: Data from process function
    :return: Response to API-caller
    :rtype: dict
    """

    # Unpack the outputs of process function
    ids, data = data

    # Predict
    predictions = model.predict(data)

    # Pack the IDs supplied by the end user and their corresponding predictions in a␣
↪dictionary
    response = dict(zip(ids, predictions))

    return response
```

Warning: The returned value of the *predict* function must be a **dictionary and all of its contents must be JSON serializable**. This is necessary because denzel will parse it into JSON to be sent back to the end user.

And... That's it! Denzel is ready to be fully operational.
Don't forget, after all these changes we must run `denzel restart` so they will take effect.
For reference, the full `pipeline.py` file should look like this

```
import pickle
import numpy as np
```

(continues on next page)

(continued from previous page)

```
FEATURES = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width']

# -------- Handled by api container --------
def verify_input(json_data):
    """
    Verifies the validity of an API request content

    :param json_data: Parsed JSON accepted from API call
    :type json_data: dict
    :return: Data for the the process function
    """

    # callback_uri is needed to sent the responses to
    if 'callback_uri' not in json_data:
        raise ValueError('callback_uri not supplied')

    # Verify data was sent
    if 'data' not in json_data:
        raise ValueError('no data to predict for!')

    # Verify data structure
    if not isinstance(json_data['data'], dict):
        raise ValueError('jsondata["data"] must be a mapping between unique id and␣
↪features')

    # Verify data scheme
    for unique_id, features in json_data['data'].items():
        feature_names = features.keys()
        feature_values = features.values()

        # Verify all features needed were sent
        if not all([feature in feature_names for feature in FEATURES]):
            raise ValueError('For each example all of the features [{}] must be␣
↪present'.format(FEATURES))

        # Verify all features that were sent are floats
        if not all([isinstance(value, float) for value in feature_values]):
            raise ValueError('All feature values must be floats')

    return json_data


# -------- Handled by denzel container --------
def load_model():
    """
    Load model and its assets to memory

    :return: Model, will be used by the predict and process functions
    """
    with open('./app/assets/iris_svc.pkl', 'rb') as model_file:
        loaded_model = pickle.load(model_file)

    return loaded_model


def process(model, json_data):
    """
```

(continues on next page)

(continued from previous page)

```
    Process the json_data passed from verify_input to model ready data

    :param model: Loaded object from load_model function
    :param json_data: Data from the verify_input function
    :return: Model ready data
    """

    # Gather unique IDs
    ids = json_data['data'].keys()

    # Gather feature values and make sure they are in the right order
    data = []
    for features in json_data['data'].values():
        data.append([features[FEATURES[0]], features[FEATURES[1]],␣
↪features[FEATURES[2]], features[FEATURES[3]]])

    data = np.array(data)

    return ids, data


def predict(model, data):
    """
    Predicts and prepares the answer for the API-caller

    :param model: Loaded object from load_model function
    :param data: Data from process function
    :return: Response to API-caller
    :rtype: dict
    """

    # Unpack the outputs of process function
    ids, data = data

    # Predict
    predictions = model.predict(data)

    # Pack the IDs supplied by the end user and their corresponding predictions in a␣
↪dictionary
    response = dict(zip(ids, predictions))

    return response
```

2.9 Using the API to Predict

Now is the time to put denzel into action.

To do that, we must first have some URI to receive the responses (remember, we are using *async responses*).

You can do that by using waithook which is an in browser service for receiving HTTP requests, just what we need - just follow the link, choose a "Path Prefix" (for example `john_q` and press "Subscribe".

Use the link that will be generated for you (http://waithook.com/<chosen_path_prefix>) and keep the browser open as we will receive the responses to the output window.

Next we need to make an actual POST request to the */predict* endpoint. We will do that using curl through the command line.

Tip:

There are more intuitive ways to create HTTP requests than curl. For creating requests through UI you can either use Postman, or through Python using the requests package.

Let's launch a predict request, for two examples from the test set:

Bash

```
$ curl --header "Content-Type: application/json" \
> --request POST \
> --data '{"callback_uri": "http://waithook.com/john_q",'\
> '"data": {"a123": {"sepal-length": 4.6, "sepal-width": 3.6, "petal-length": 1.0,
↪"petal-width": 0.2},'\
> '"b456": {"sepal-length": 6.5, "sepal-width": 3.2, "petal-length": 5.1, "petal-width
↪": 2.0}}}' \
http://localhost:8000/predict
```

Python

```
import requests

data = {
  "callback_uri": "http://waithook.com/john_q",
  "data": {"a123": {"sepal-length": 4.6, "sepal-width": 3.6, "petal-length": 1.0,
↪"petal-width": 0.2},
           "b456": {"sepal-length": 6.5, "sepal-width": 3.2, "petal-length": 5.1,
↪"petal-width": 2.0}}
}

response = requests.post('http://localhost:8000/predict', json=data)
```

If the request has passed the *verify_input* method, you should immediately get a response that looks something like (on curl, you'd see it in your prompt, with `requests` you'll have it in `response.json()`):

```
{"status":"success","data":{"task_id":"19e39afe-0729-43a8-b4c5-6a60281157bc"}}
```

This means that the task has passed verification successfully, already entered the task queue and will next go through *process* and *predict*.

At any time, you can view the task status by sending a GET request to the */status/{task_id}* endpoint.

If you examine waithook in your browser, you will see that a response was already sent back with the prediction, it should looks something like:

```
{
  "method": "POST",
  "url": "/john_q",
```

(continues on next page)

(continued from previous page)


```
  "headers": {
    "User-Agent": "python-requests/2.19.1",
    "Connection": "close",
    "X-Forwarded-Proto": "http",
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "49",
    "Content-Type": "application/json",
    "Host": "waithook.com",
    "X-Forwarded-for": "89.139.202.80"
  },
  "body": "{\"a123\": \"Iris-setosa\", \"b456\": \"Iris-virginica\"}"
}
```


In the `"body"` section, you can see the returned predictions.
If you got this response it means that all went well and your deployment is fully ready.

2.10 Monitoring

Denzel comes with a built in UI for monitoring the tasks and workers.
To use it, once the system is up go to the monitor port (defaults to 5555) on the deployment domain. If deployed locally open your browser and go to http://localhost:5555
You will be presented with a UI that looks something like:

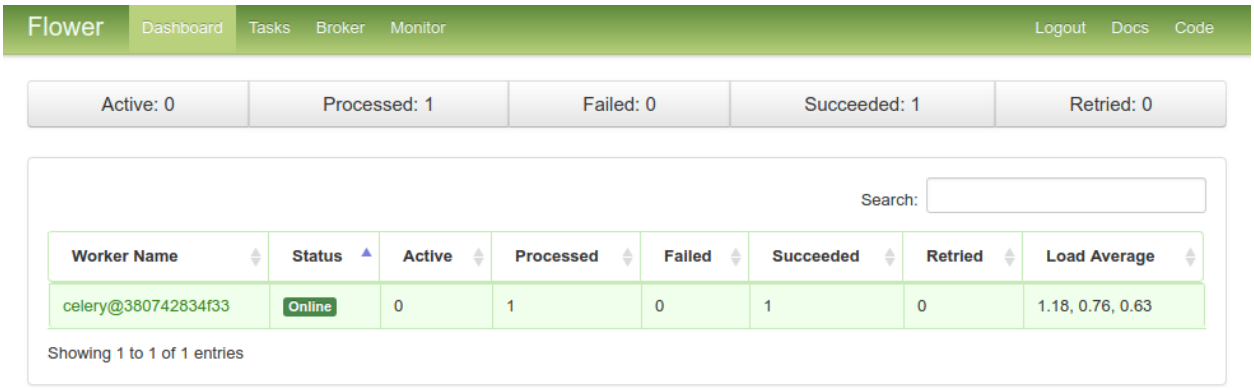


Fig. 1: Example of Flower's monitoring UI

This dashboard is generated by Flower, and gives you access to examine the worker status, tasks status, tasks time and more.

2.11 Debugging

Life is not all tutorials and sometime things go wrong.

Debugging exceptions is dependent of where the exception is originated at.

2.11.1 `verify_input` Exceptions

This method is executed in the API container. If anything goes wrong in this method you will get it as an immediate response to your `/predict` POST request.
For example, lets say we make the same POST request as we did before, but we opt out one of the features in the data.
Given the code we supplied `verify_input` we should get the following response

```
{
 "title": "Bad input format",
 "description": "For each example all of the features [['sepal-length', 'sepal-width',
↪ 'petal-length', 'petal-width']] must be present"
}
```

2.11.2 `load_model` Exceptions

If anything went wrong with the *load_model* method, you will only able to see the traceback and exception on the logs.
Specifically, the denzel service log is where the model is loaded. In this case the exception can be found through the *logs* (to isoltate the relevant container, pass the `--service denzel` option) or *logworker* command.
Check them both as the location of the exception is dependent on its type.

2.11.3 `process` & `predict` Exceptions

If something went wrong in these methods, it necessarily means you made a successful request and passed the *verify_input* method and have received a `"SUCCESS"` status to your response with a task ID.
process and *predict* both get executed on the denzel container. If anything goes wrong inside of them it will be most likely only visible when querying for task status.
For example, if we would forget to import `numpy as np` even though it is in use in the *process* method - we will get a `"SUCCESS"` response for our POST (because we passed the *verify_input* method).
But the task will fail after entering the *process* method - to see the reason, we should query the */status/{task_id}* and we would see the following:

```
{
 "status": "FAILURE",
 "result": {"args":["name 'np' is not defined"]}
}
```

2.12 Deployment

Since denzel is fully containerized it should work on any machine as long as it has docker, docker-compose and Python3 installed.
Also all of the main cloud service providers already support dockerized appications.

After completing all the necessary implementations for deployment covered in this tutorial it is best to check that the system can be launched from scratch.

To do that, we should *shutdown* while purging all images, and relaunch the project - don't worry no code is being deleted during shutdown. This process it to make sure that when we deploy it somewhere else, it will work.

Go to the main project directory and run the following:

```
$ denzel shutdown --purge

Stopping iris_classifier_denzel_1  ... done
Stopping iris_classifier_monitor_1 ... done
Stopping iris_classifier_api_1     ... done
Stopping iris_classifier_redis_1   ... done
Removing iris_classifier_denzel_1  ... done
Removing iris_classifier_monitor_1 ... done
Removing iris_classifier_api_1     ... done
Removing iris_classifier_redis_1   ... done
Removing network iris_classifier_default
Removing image redis:4
Removing image denzel:1.0.0
Removing image denzel:1.0.0
ERROR: Failed to remove image for service denzel:1.0.0: 404 Client Error: Not Found (
↪"No such image: denzel:1.0.0")
Removing image denzel
ERROR: Failed to remove image for service denzel:1.0.0: 404 Client Error: Not Found (
↪"No such image: denzel:1.0.0")

$ denzel launch
Creating network "iris_classifier_default" with the default driver
Pulling redis (redis:4)...
4: Pulling from library/redis
.
.
```

Note:

The "ERROR: Failed to remove. . . ." can be safely ignored. This is a result of the `--purge` flag that tells denzel to remove the denzel image.

Since the image is used by three different containers, it will successfully delete it on the first container but fail on the other two.

After the relaunching is done check again that all endpoints are functioning as expected - just to make sure.

If all is well your system is ready to be deployed wherever, on a local machine, a remote server or a docker supporting cloud service.

To deploy it elsewhere simply copy all the contents of the project directory to the desired destination, verify docker, docker-compose and Python3 are installed, *install denzel* and call `denzel launch` from within that directory.

As a matter of fact, we can skip the installing denzel part when we deploy - more about that in the *Production* section.

2.13 Production

Denzel is essentially a docker-compose application, with an accompanying CLI tool to abstract docker and OS related functionality from the data scientist developing the application.
This means that if you are going to deploy your application on a docker-supporting cloud service, or deliver it to a production developer the denzel package is no longer mandatory.
Because the contents of the project directory already include the `docker-compose.yml`, the `Dockerfile` and all the code needed to run and manage the application - Any service or person that can deal with docker will be able to do so.
Denzel is built that way so that going into production, more advanced docker management tools (like Kubernetes) can be used to apply more advanced production techniques like continuous deployment and scaling.

Denzel takes you from a model, to a containerized and deployable application in a data scientist oriented way - once you have that, the options are endless.

2.14 Deleting

Deleting a denzel project is very simple.
To do so we must call the *shutdown* command, to remove all of the containers. Optionally we could pass the `--purge` flag to remove the underlying images.
Then delete the project directory and the denzel project is fully removed.

CHAPTER 3

Pipeline Methods

The pipeline methods are the blocks that construct the end-user request flow

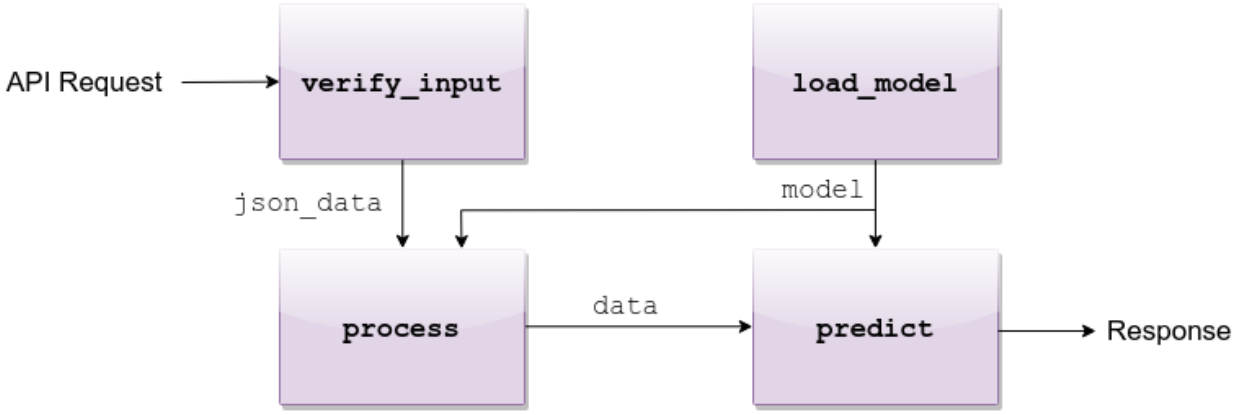


Fig. 1: API request flow through the four functions

3.1 load_model

pipeline.**load_model**()

Load model and its assets to memory

Returns Model, will be used by the predict and process functions

3.2 verify_input

pipeline.**verify_input**(*json_data*)
Verifies the validity of an API request content

Parameters **json_data** (*dict*) – Parsed JSON accepted from API call

Returns Data for the the process function

3.3 process

pipeline.**process**(*model*, *json_data*)
Process the json_data passed from verify_input to model ready data

Parameters

- **model** – Loaded object from load_model function
- **json_data** – Data from the verify_input function

Returns Model ready data

3.4 predict

pipeline.**predict**(*model*, *data*)
Predicts and prepares the answer for the API-caller

Parameters

- **model** – Loaded object from load_model function
- **data** – Data from process function

Returns Response to API-caller

Return type dict

CHAPTER 4

API Endpoints

Denzel exposes three different endpoints for end users.
All endpoints are relative to the host. For example if deployed locally on the default port the endpoint `/info` means `localhost:8000/info`

Endpoints

- */info*
- */predict*
- */status/{task_id}*

4.1 /info

GET /info
Endpoint for deployment information. Basically returns the content of `app/assets/info.txt`.

4.2 /predict

POST /predict
Endpoint for performing predictions

Request Headers

- Accept – application/json

Form Parameters

- **body** – Data necessary for prediction, should match the interface defined

Status Codes

- 200 OK – Request accepted and entered the task queue
- 400 Bad Request – Failed to in the reading / verification process.

4.3 /status/{task_id}

GET /status/(**str:** *task_id*)
Endpoint for checking a task status

Parameters

- **task_id** – Task ID

CHAPTER 5

Command Line Interface (CLI)

Denzel comes with a CLI for the developers use.
At any moment you can use the `--help` flag to see the help menu

```
$ denzel --help

Usage: denzel [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  launch        Builds and starts all services
  logs          Show service logs
  logworker     Show worker log
  restart       Restart services
  shell         Connect to service bash shell
  shutdown      Stops and deletes all services
  start         Start services
  startproject  Builds the denzel project skeleton
  status        Examine status of services
  stop          Stop services
  updatereqs    Update service according to requirements.txt
```

For command specific help you can follow a command with the `--help` flag (ex. `denzel launch --help`)

Commands

- *startproject*

- *launch*
- *shutdown*
- *start*
- *stop*
- *restart*
- *status*
- *logs*
- *logworker*
- *shell*
- *updatereqs*

Note: Except from the *startproject* command, all other commands must be executed from within project directory

5.1 startproject

Usage: `denzel startproject NAME`

Builds the denzel project skeleton.

NAME
Name of the project

--gpu|--no-gpu
Support for NVIDIA GPU

Default: `--no-gpu`

5.2 launch

Usage: `denzel launch [OPTIONS]`

Builds and starts all services.

--api-port `<INTEGER>`
API endpoints port

Default: `8000`

--monitor-port `<INTEGER>`
Monitor UI port

Default: `5555`

5.3 shutdown

Usage: `denzel shutdown [OPTIONS]`

Stops and deletes all services, if you wish only to stop use the *stop* command.

--purge|--no-purge
Discard the docker images

Default: `--no-purge`

5.4 start

Usage: `denzel start`

Start services

5.5 stop

Usage: `denzel stop`

Stop services

5.6 restart

Usage: `denzel restart`

Restart services (equal to calling *stop* and then *start*).

5.7 status

Usage: `denzel status [OPTIONS]`

Examine status of services and worker. Use this to monitor the status of your project.

--live|--no-live
Live status view

Default: `--no-live`

5.8 logs

Usage: `denzel logs [OPTIONS]`

Show service logs

--service `[api|denzel|monitor|redis|all]`
Target service

Default: `all`

--live|--no-live
Follow logs output

Default: `--no-live`

5.9 logworker

Usage: `denzel logworker [OPTIONS]`

Show worker log

`--live|--no-live`
Follow logs output

Default: `--no-live`

5.10 shell

Usage: `denzel shell [OPTIONS]`

Connect to service bash shell. This is only for advanced usage, shouldn't be used in standard scenarios.

`--service` `[api|denzel|monitor|redis]`
Target service

Default: `denzel`

5.11 updatereqs

Usage: `denzel updatereqs`

Update services according to `requirements.txt`. This command always uses the pip `--upgrade` flag, so requirements will always be updated to the latest version. If you wish to install a specific version, specify it in the `requirements.txt` file. This command will initiate a restart so updates will apply.

CHAPTER 6

Showcase

Deployment with denzel is very simple.
Assuming we have saved to disk an already trained model with less than 60 lines you can have it deployed with the following features:

1. Expose an API for users to use your model
2. Allow users to check their prediction request status through API requests
3. Monitor the performance of your deployment through Flower UI
4. Fully containerized system so it could be deployed anywhere

Here is an example of all the code necessary to deploy the Iris classifier from the *tutorial*.

```
import pickle
import numpy as np

FEATURES = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width']

# -------- Handled by api container --------
def verify_input(json_data):
    """ Verifies the validity of an API request content """

    # callback_uri is needed to sent the responses to
    if 'callback_uri' not in json_data:
        raise ValueError('callback_uri not supplied')

    # Verify data was sent
    if 'data' not in json_data:
        raise ValueError('no data to predict for!')

    # Verify data structure
    if not isinstance(json_data['data'], dict):
        raise ValueError('jsondata["data"] must be a mapping between unique id and␣
↪features')
```

(continues on next page)

(continued from previous page)

```
    # Verify data scheme
    for unique_id, features in json_data['data'].items():
        feature_names = features.keys()
        feature_values = features.values()

        # Verify all features needed were sent
        if not all([feature in feature_names for feature in FEATURES]):
            raise ValueError('For each example all of the features [{}] must be␣
↪present'.format(FEATURES))

        # Verify all features that were sent are floats
        if not all([isinstance(value, float) for value in feature_values]):
            raise ValueError('All feature values must be floats')

    return json_data


# -------- Handled by denzel container --------
def load_model():
    """ Load model and its assets to memory """
    with open('./app/assets/iris_svc.pkl', 'rb') as model_file:
        loaded_model = pickle.load(model_file)

    return loaded_model


def process(model, json_data):
    """ Process the json_data passed from verify_input to model ready data """

    # Gather unique IDs
    ids = json_data['data'].keys()

    # Gather feature values and make sure they are in the right order
    data = []
    for features in json_data['data'].values():
        data.append([features[FEATURES[0]], features[FEATURES[1]],␣
↪features[FEATURES[2]], features[FEATURES[3]]])

    data = np.array(data)

    return ids, data


def predict(model, data):
    """ Predicts and prepares the answer for the API-caller """

    # Unpack the outputs of process function
    ids, data = data

    # Predict
    predictions = model.predict(data)

    # Pack the IDs supplied by the end user and their corresponding predictions in a␣
↪dictionary
    response = dict(zip(ids, predictions))
```

(continues on next page)

(continued from previous page)

```
    return response
```

CHAPTER 7

Development State

Denzel is supported by Data Science Group Ltd. and is promised to be kept on open source.
Denzel right now is on alpha. This means that it is fully operational and new features and support will be added to it before moving on to a beta release.

7.1 Upcoming Features

HTTP Routing Table

/info

`GET /info`, 29

/predict

`POST /predict`, 29

/status

`GET /status/(str:task_id)`, 30

Index

Symbols

C

L

N

P

V