

Automated Big Data REST Service Creation in Support of Big Data Applications

Report 9/30/18-8/31/2019

60NANB18D268

G. C. Fox (PI)

G. von Laszewski (co-PI)

Editor

laszewski@gmail.com

<https://github.com/2018-05-30-community/proceedings>

Created by Cloudmesh & Cyberaide Bookmanager, <https://github.com/cyberaide/bookmanager>

AUTOMATED BIG DATA REST SERVICE CREATION IN SUPPORT OF BIG DATA APPLICATIONS

G. C. Fox (Pi) G. von Laszewski (co-PI)

(c) Gregor von Laszewski, Geoffrey C. Fox, 2019

AUTOMATED BIG DATA REST SERVICE CREATION IN SUPPORT OF BIG DATA APPLICATIONS

1 PREFACE

1.1 ABSTRACT ☼

1.1.1 Purpose of the Work Reported

2 TECHNOLOGY ASSESMENT

2.1 Task 2 – Technology Analysis ☼

2.1.1 Previous accomplishments

2.2 RAML ☼

2.3 API Blueprint ☼

2.4 OpenAPI ☼

2.4.1 Swagger CodeGen

2.5 Cloudmesh OpenAPI ☼

3 REST SERVICES

3.1 Task 3: Big Data REST Services ☼

3.2 OpenAPI REST Services with Swagger ☼

3.2.1 Swagger Tools

3.2.2 Swagger Community Tools

3.2.2.1 Converting Json Examples to OpenAPI YAML Models

3.3 OpenAPI 3.0 REST Service via Introspection ☼

3.3.1 Verification

3.3.2 Swagger-UI

3.3.3 Mock service

3.3.4 Exercise

4 REST SPECIFICATIONS

4.1 Task 4 – Generalized Rest Service Specifications ☼

4.2 Amazon Redshift OpenAPI Specification ☼

4.3 Virtual Directory ☼

4.4 Elastic Map Reduce ☼

5 APPLICATIONS

5.1 Task 5 – Application Examples ☼

5.2 S-cone Classification Using REST Services and Machine Learning ☼

5.2.1 Abstract

5.2.2 Introduction

[5.2.3 Data](#)

[5.2.3.1 Preprocessing](#)

[5.2.3.2 Visualization](#)

[5.2.4 Model Discussion](#)

[5.2.4.1 Failures](#)

[5.2.4.2 Activation Function](#)

[5.2.4.3 Decision](#)

[5.2.5 REST Service Implementations](#)

[5.2.6 Specification](#)

[5.3 Rookie Fantasy Football Point Prediction](#)

[5.3.1 Abstract](#)

[5.3.2 Introduction](#)

[5.3.3 Data Set](#)

[5.3.4 KNN Algorithm](#)

[5.3.5 Implementation](#)

[5.3.6 Limitations](#)

[5.3.7 Conclusion](#)

[5.3.8 Specification](#)

[5.4 Analysis of soccer data with kmeans](#)

[5.4.1 Abstract](#)

[5.4.2 Introduction](#)

[5.4.2.1 Sport Analytics](#)

[5.4.2.2 Sensors in Sports](#)

[5.4.3 Soccer Dataset](#)

[5.4.4 Algorithm Discussion](#)

[5.4.4.1 K-means](#)

[5.4.4.2 Spectral Clustering](#)

[5.4.4.3 Dimensional Reduction](#)

[5.4.5 Results](#)

[5.4.6 Analysis](#)

[5.4.7 Specification](#)

[5.5 Tetris Score Analysis Server](#)

[5.5.1 Abstract](#)

[5.5.2 Introduction](#)

[5.5.3 Design](#)

[5.5.4 Architecture](#)

[5.5.5 Dataset](#)

[5.5.6 Results](#)

[5.5.7 Conclusion](#)

[5.5.8 Specification](#)

[5.6 Political Bias and Voting Trends](#) 🍀

[5.6.1 Abstract](#)

[5.6.2 Introduction](#)

[5.6.3 Requirements](#)

[5.6.4 Design](#)

[5.6.4.1 Python](#)

[5.6.4.2 REST Service](#)

[5.6.4.3 Docker](#)

[5.6.5 Dataset](#)

[5.6.6 Results](#)

[5.6.7 Discussion](#)

[5.6.8 Conclusion](#)

[5.6.9 Work Breakdown](#)

[5.6.10 Specification](#)

[5.7 Spam Analysis with Spamalot](#) 🍀

[5.7.1 Abstract](#)

[5.7.2 Introduction](#)

[5.7.3 The Algorithm](#)

[5.7.3.1 Naive Bayes](#)

[5.7.3.1.1 Metrics](#)

[5.7.3.2 Support Vector Machines \(SVM\)](#)

[5.7.4 The Data Set](#)

[5.7.5 Model Results](#)

[5.7.6 Implementation](#)

[5.7.6.1 The Server](#)

[5.7.6.2 The Upload Function and Classification](#)

[5.7.6.3 Specification](#)

[5.7.7 Conclusion](#)

[6 DESSIMINTAION](#)

[6.1 Task 6 – Dissemination](#) 🍀

[6.1.1 Conference Presentation](#)

[6.1.2 Task 6.1 Community Testing](#)

[7 RESOURCES](#)

[7.1 Task 7 – Development Resources](#) 🍀

8 TUTORIALS

8.1 Overview 🍀

8.2 AUTOMATED REST SERVICE GENERATION WITH EVE

8.2.1 Rest Services with Eve 🍀

8.2.1.1 Ubuntu install of MongoDB

8.2.1.2 macOS install of MongoDB

8.2.1.3 Windows 10 Installation of MongoDB

8.2.1.4 Database Location

8.2.1.5 Verification

8.2.1.6 Building a simple REST Service

8.2.1.7 Interacting with the REST service

8.2.1.8 Creating REST API Endpoints

8.2.1.9 REST API Output Formats and Request Processing

8.2.1.10 REST API Using a Client Application

8.2.1.11 Towards cmd5 extensions to manage eve and mongo 📄

8.2.2 HATEOAS 🍀

8.2.2.1 Filtering

8.2.2.2 Pretty Printing

8.2.2.3 XML

8.2.3 Extensions to Eve 🍀

8.2.3.1 Object Management with Eve and Evegenie

8.2.3.1.1 Installation

8.2.3.1.2 Starting the service

8.2.3.1.3 Creating your own objects

8.3 AUTOMATED REST SERVICE GENERATION WITH CODEGEN FOR OPENAPI 2.0

8.3.1 OpenAPI 2.0 Specification 🍀

8.3.1.1 The Virtual Cluster example API Definition

8.3.1.1.1 Terminology

8.3.1.1.2 Specification

8.3.1.2 References

8.3.2 OpenAPI REST Service via Introspection 🍀

8.3.2.1 Verification

8.3.2.2 Mock service

8.3.2.3 Exercise

8.3.3 OpenAPI REST Service via Codegen 🍀

8.3.3.1 Step 1: Define Your REST Service

[8.3.3.2 Step 2: Server Side Stub Code Generation and Implementation](#)

[8.3.3.2.1 Setup the Codegen Environment](#)

[8.3.3.2.2 Generate Server Stub Code](#)

[8.3.3.2.3 Fill in the actual implementation](#)

[8.3.3.3 Step 3: Install and Run the REST Service:](#)

[8.3.3.3.1 Start a virtualenv:](#)

[8.3.3.3.2 Make sure you have the latest pip:](#)

[8.3.3.3.3 Install the requirements of the server side code:](#)

[8.3.3.3.4 Install the server-side code package:](#)

[8.3.3.3.5 Run the service](#)

[8.3.3.3.6 Verify the service using a web browser:](#)

[8.3.3.4 Step 4: Generate Client-Side Code and Verify](#)

[8.3.3.4.1 Client-side code generation:](#)

[8.3.3.4.2 Install the client-side code package:](#)

[8.3.3.4.3 Using the client API to interact with the REST service](#)

[8.3.3.5 Towards a Distributed Client Server](#)

[8.4 AUTOMATED REST SERVICE GENERATION WITH CONEXION FOR OPENAPI 3.0](#)

[8.4.1 REST Specifications](#)

[8.4.1.1 OPENAPI](#)

[8.4.1.1.1 Open API 3.0 Specification \(OAS 3.0\)](#)

[8.4.1.1.1.1 Definitions](#)

[8.4.1.2 RAML](#)

[8.4.1.3 API Blueprint](#)

[8.4.1.4 JsonAPI](#)

[8.4.1.5 Tinyspec](#)

[8.4.1.6 Tools](#)

[8.4.1.6.1 Connexion](#)

[8.4.2 OpenAPI 3.0 REST Service via Introspection](#)

[8.4.2.1 Verification](#)

[8.4.2.2 Swagger-UI](#)

[8.4.2.3 Mock service](#)

[8.4.2.4 Exercise](#)

[8.4.3 REST AI services Example](#)

[8.4.3.1 Service Endpoints/ Paths](#)

[8.4.3.1.1 Path *kmeans/upload*](#)

[8.4.3.1.2 Path *kmeans/fit*](#)

[8.4.3.1.3 Path *kmeans/predict*](#)

[8.4.3.2 Files](#)

[8.4.3.3 Running the example](#)

[8.4.3.4 Notes](#)

[9 REFERENCES](#)

1 PREFACE

1.1 ABSTRACT

This report summarizes the progress made on the project:

Automated Big Data REST Service Creation in Support of Big Data Applications

Report Period: 9/30/18-8/31/2019

Number: 60NANB18D268

1.1.1 PURPOSE OF THE WORK REPORTED

In this research report, we lay out an approach to defining composable Big Data Services in support of the NIST Big Data Reference Architecture (NIST-BDRA).

The work is broken down into the following tasks reflected in the outline of this report:

- **Task 1:** Provide this report that includes the following activities
- **Task 2:** A technology Assessment and Analysis, to identify community efforts that we can leverage and integrate in our activities and identify gaps that this work needs to fill.
- **Task 3:** Identify how we can conveniently create REST services that *are defined in the BDRA Interface definitions.
- **Task 4:** Work with the community to identify extensions to the NIST *BDRA interface definitions that are not covered by the specification*, to validate the approach of using REST services.
- **Task 5:** Identify if the frameworks we use can be used for *implementing Big Data applications.

- **Task 6:** Disseminate the activities
- **Task 7:** Provide a summary for the tools and resources we developed.

2 TECHNOLOGY ASSESSMENT

2.1 TASK 2 – TECHNOLOGY ANALYSIS

Over the last years, the ecosystem regarding Big Data has significantly changed. This is fostered by new technologies in part even driven by Big Data Analysis. Prominent appearance of containers, and GPU's as part of HPC and Cloud infrastructures have to be considered. In this task, we will be analyzing new and influential technologies to be watched and analyzed to identify new trends that influence future developments of Big Data Architectures.

As one of the technologies significantly changed, it was important to revisit the automated generation of REST services. This includes the following:

1. A technology overview that was gradually improved throughout the duration of the funding period.

Here we, for now, focus on the review of OpenAPI like specifications.

2. A section that illustrates the simplicity of creating REST services while leveraging community tools

Here we primarily focus on the creation of REST services through automated introspection of the YAML file that defines the OpenAPI rest service

3. The development of a sophisticated tool that allows merging OpenAPI specifications.

2.1.1 PREVIOUS ACCOMPLISHMENTS

Our previous implementations contained the use of a straightforward mechanism to define REST services. Although this approach is much simpler than the approach we suggest now, it lacked interoperability with language and framework independent implementations as it was strictly based on technology only supported in Python. This approach is also dated as the tools that we used

became deprecated, and a new approach was needed.

We have thus evaluated several frameworks to easily develop REST services while at the same time look for specification focus that is language-independent and may be adapted by the community in general. From these frameworks, we primarily looked at Swagger/OpenAPI, RAML, and API Blueprint. From those we settled on OpenAPI due to the following reasons

- The specification is done now by the community
- Through previous versions e.g., Swagger, OpenAPI has been proven to work
- The specification created nice-looking Web pages
- The specification can be hosted on GitHub (as do others also)
- The documents about swagger say the community is large
- The constructs provide sufficient richness for the specification of REST APIs

In addition, we developed a set of scripts that explored if OpenAPI to code (in our case python) can successfully be used to implement multiple independent and dependent REST services that are automatically derived from the specification itself. In addition, instantiations or implementations of the operations into the specification based on source code management has been evaluated.

This has been successfully demonstrated in the following ways.

1. The creation of a joint specification generator of a service that merges two independent OpenAPI specifications
2. The creation of an example that uses the specification to generate internal methods based on introspection of the code as provided by the swagger community

During the last six month, we identified that this approach required considerable programming expertise and turned out to be out of reach for many application developers with little experience in programming. Thus we developed a new approach that has been very successful and is documented in this report.

2.2 RAML

According to [1] RESTful API Modeling Language (RAML) manages the API lifecycle from design to sharing. It allows to focus on the parts that are needed, is reusable, and has a machine-readable format that is also human-friendly.

RAML has a significant, but much smaller community than OpenAPI. In addition to expressing REST API's RAML can also be used for expressing other APIs.

Due to the dominance of OpenAPI the RAML team has also joined the OpenAPI initiative [2]. This makes it evident that OpenAPI is, at this time, the preferred choice for us to define API's based on template formats.

2.3 API BLUEPRINT

The API Blueprint is presented at [3]. It focuses on the collaborative specification of API's. It is an open specification and is under MIT license with its specifications being posted on GitHub:

- <https://github.com/apiaryio/api-blueprint-rfcs/blob/master/rfcs/0001-rfc-process.md>
- <https://github.com/apiaryio/api-blueprint-rfcs/blob/master/rfcs/0002-authentication.md>
- <https://github.com/apiaryio/api-blueprint-rfcs/blob/master/rfcs/0003-authentication-basic.md>
- <https://github.com/apiaryio/api-blueprint-rfcs/blob/master/rfcs/0004-request-parameters.md>
- <https://github.com/apiaryio/api-blueprint-rfcs/blob/master/rfcs/0005-authentication-oauth2.md>

Additionally, we find the following useful information:

- A tutorial: <https://apiblueprint.org/documentation/tutorial.html>
- The documentation: <https://apiblueprint.org/documentation/>
- A list of tools: <https://apiblueprint.org/tools.html>

According to [4] “the API Blueprint lacks advanced construct and code level tooling. Because of this, its adoption has been slow, dwarfed when compared to Swagger or RAML”.

2.4 OPENAPI

The OpenAPI specification was derived initially from a specification called Swagger. Since then, the community has engaged in separating the specification, which is now known under the name OpenAPI.

The organization that contributed Swagger to OpenAPI is now focussing on building tools around it to make the creation of OpenAPIs easier (<https://swagger.io/>). A statement from the Web site reflects this:

“OpenAPI specification (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API”.

Previous efforts we conducted used OpenAPI v2. This version is also known as Swagger due to the company project that this version was developed under. The choice for OpenAPI v2 was motivated due to the fact that the specification was widely used, and the tools for it were sufficiently mature. However, the community developed the new OpenAPI v3.

The different versions available are available at

- <https://github.com/OAI/OpenAPI-Specification/tree/master/versions>

In particular the specifications for the following versions are available:

- v3.0.0:
 - <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md>
- v3.0.1:
 - <https://github.com/OAI/OpenAPI->

[Specification/blob/master/versions/3.0.1.md](#)

- v3.0.3:
 - <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md>

A significant effort was spent to keep up with these versions. The specifications were all updated to OpenAPI v3, and feedback to the NIST BDRA Working group. It took more than one-month effort to convert all specifications from v2 to v3.

2.4.1 SWAGGER CODEGEN

Our code was initially relying on Swagger CodeGen <https://swagger.io/tools/swagger-codegen/>. This tool creates and derives a comprehensive API based on the OpenAPI specification. We originally developed all backend services using this tool but found that

- the tool did not work well when we switched to OpenAPI v3
- when using it with contributors to our efforts, the development of code was too complicated. We guided on how to services that are automatically derived from the YAML service, while associating simply python functions as operationID's.

Thus although we did not initially plan for it, we were forced to look for alternatives addressing these two issues. Initially, none of the tools were sufficient to support OpenAPI v3. We conducted tool analyses in January 2019, and in July 2019. In July, we found that the support for OpenAPI v3 by the community had reached critical support, and it was possible for us to tr to v3. However, we decided at that time to drop our use of codegen as it was not mature enough for us and was too complicated for our community.

The later was mainly motivated by a fact that we found out after the funding period of this project ended, and we wished we would have known earlier. We were approached In October by the top contributor to swagger codegen. He alerted us that a new fork was developed by him and others which is now

available at

- <https://github.com/OpenAPITools/openapi-generator/blob/master/docs/qna.md>

Naturally, it was too late for us to evaluate this new version. However, our new method is better suited for the activities we undertook. The reasons for the switch include:

- For a long time there was confusion about the support of codegen for OpenAPI 2.0 vs 3.0.
- Beginner students could not easily use swagger codegen, so we wrote a *custom* template how to generate API's from a specification while at the same time providing extensive documentation on how to do it.
- Besides us noticing that codegen was complex and, at the time not provide sufficient support for v3. This was also mentioned by the developers motivating their new branch.
- The availability of Conexions ??? automated introspection to create an OpenAPI v3 Web service is sufficient for our prototyping efforts.
- When discussing the two approaches with students, they prefer the use of the connexion server and the integration of operationIDs, which we gave simple examples to. Also the simple examples we provided as teaching tool to the community were much appreciated and could users get started within minutes.

When comparing the efforts for one versus the other, we informally observed:

- Swagger codegen needed 1-2 month for students to comprehend, while some students never mastered it. We observed that no one has two month time for learning this, so the codegen solution was not appropriate even if it would have worked.
- Instead of swagger codegen we now use the implicit creation of the API directly from the YAML file via connexion. This was a winning transition as we now can create APIs in 1 week, and all students can do it.

- We also found a lack of documentation for the new codegen (which may have been fixed by now) in regards to samples discussing data upload, and authentication, as well as authorization examples.

2.5 CLOUDMESH OPENAPI

Our previous approach as outlined in

- <https://github.com/cloudmesh-community/book/blob/master/chapters/rest/swagger-codegen.md>

revealed that it was too complicated for inexperienced developers to use. Since then have devised a new usage workflow that we have documented in

- <https://github.com/cloudmesh-community/book/blob/master/chapters/rest/swagger-introspection.md>

and also included in this report. Furthermore, we have developed a sophisticated extension to our multi-cloud environment called Cloudmesh that allows the instantiation as well as the combination of OpenAPI specification and start them as a REST prototype service.

This tool is available at

- <https://github.com/cloudmesh/cloudmesh-openapi>

It has the following manual page:

```
Usage:
  openapi merge [SERVICES...] [--dir=DIR] [--verbose]
  openapi list [--dir=DIR]
  openapi description [SERVICES...] [--dir=DIR]
  openapi md FILE [--indent=INDENT]
  openapi codegen [SERVICES...] [--srcdir=SRCDIR]
                                [--destdir=DESTDIR]
  openapi server start YAML [--directory=DIRECTORY]
                             [--port=PORT] [--server=SERVER]
                             [--verbose]
  openapi server stop YAML

Arguments:
  DIR    The directory of the specifications
  FILE   The specification
  SRCDIR The directory of the specifications
  DESTDIR The directory where the generated code should be put

Options:
  --verbose          specifies to run in debug mode
                    [default: False]
  --port=PORT       the port for the server [default: 8080]
  --directory=DIRECTORY the directory in which the server is run
                    [default: ./]
```

`--server=SERVER` the server [default: flask]

Description:

This command does some useful things.

3 REST SERVICES

3.1 TASK 3: BIG DATA REST SERVICES

REST Services have become very popular as part of the Web to offer services to the community. We are leveraging this popularity and develop several rest services in support of Big Data Architectures. The REST services are composable and allow reuse in big data applications to build quickly frameworks that support the analysis.

3.2 OPENAPI REST SERVICES WITH SWAGGER

Swagger <https://swagger.io/> is a tool for developing API specifications based on the OpenAPI Specification (OAS). It allows not only the specification but the generation of code based on the specification in a variety of languages.

Swagger itself has several tools which together build a framework for developing REST services for a variety of languages.

3.2.1 SWAGGER TOOLS

The primary Swagger tools of interest are:

Swagger Core

includes libraries for working with Swagger specifications
<https://github.com/swagger-api/swagger-core>.

Swagger Codegen

allows generating code from the specifications to develop Client SDKs, servers, and documentation. <https://github.com/swagger-api/swagger-codegen>

Swagger UI

is an HTML5 based UI for exploring and interacting with the specified APIs <https://github.com/swagger-api/swagger-ui>

Swagger Editor

is a Web-browser based editor for composing specifications using YAML <https://github.com/swagger-api/swagger-editor>

Swagger Hub

is a Web service to collaboratively develop and host OpenAPI specifications <https://swagger.io/tools/swaggerhub/>

The developed APIs can be hosted and further developed on an online repository named SwaggerHub <https://app.swaggerhub.com/home> The convenient online editor is available which also can be installed locally on a variety of operating systems including macOS, Linux, and Windows.

3.2.2 SWAGGER COMMUNITY TOOLS

notify us about other tools that you find and would like us to mention here.

3.2.2.1 Converting Json Examples to OpenAPI YAML Models

Swagger toolbox is a utility that can convert JSON to swagger compatible YAML models. It is hosted online at

- <https://swagger-toolbox.firebaseio.com/>

The source code to this tool is available on GitHub at

- <https://github.com/essuraj/swagger-toolbox>

It is crucial to make sure that the JSON model is configured correctly. As such, each datatype must be wrapped in “quotes” and the last element must not have a , behind it.

In case you have large models, we recommend that you gradually add more and

more features so that it is easier to debug in case of an error. This tool is not designed to provide back a full-featured OpenAPI, but help you get started deriving one.

Let us look at a small example. Let us assume we want to create a REST service to execute a command on the remote service. We know this may not be a good idea if it is not secured correctly, so be extra careful. A good way to simulate this is just to use a return string instead of executing the command.

Let us assume the JSON schema looks like:

```
{
  "host": "string",
  "command": "string"
}
```

The output the swagger toolbox creates is

```
---
required:
- "host"
- "command"
properties:
  host:
    type: "string"
  command:
    type: "string"
```

As you can see it is far from complete, but it could be used to get you started.

Based on this tool develop a rest service to which you send a schema in JSON format from which you get back the YAML model.

3.3 OPENAPI 3.0 REST SERVICE VIA INTROSPECTION

The simplest way to create an OpenAPI service is to use the connexion service and read in the specification from its YAML file. It then introspects and dynamically creates methods that are used for the implementation of the server.

The full example for this is available in

- <https://github.com/cloudmesh-community/book/tree/master/examples/rest/cpu>

An extensive documentation is available at

- <https://media.readthedocs.org/pdf/connexion/latest/connexion.pdf>

This example returns the cpu information of a computer to dynamically demonstrate how simple it is to generate in python a REST service from an OpenAPI specification.

Our requirements.txt file includes

```
flask
connexion[swagger-ui]
```

as dependencies. The `server.py` file simply contains the following code:

```
from flask import jsonify
import connexion

# Create the application instance
app = connexion.App(__name__, specification_dir="./")

# Read the yaml file to configure the endpoints
app.add_api("cpu.yaml")

# create a URL route in our application for "/"
@app.route("/")
def home():
    msg = {"msg": "It's working!"}
    return jsonify(msg)

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

This will run our REST service under the assumption we have a `cpu.yaml` and a `cpu.py` files as our YAML file calls out methods from `cpu.py`

The YAML file looks as follows.

```
openapi: 3.0.2
info:
  title: cpuinfo
  description: A simple service to get cpuinfo as an example of using OpenAPI 3.0
  license:
    name: Apache 2.0
    version: 0.0.1

servers:
  - url: http://localhost:8080/cloudmesh

paths:
  /cpu:
    get:
      summary: Returns cpu information of the hosting server
      operationId: cpu.get_processor_name
      responses:
        '200':
          description: cpu info
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/cpu"

components:
```

```

schemas:
  cpu:
    type: "object"
    required:
      - "model"
    properties:
      model:
        type: "string"

```

Here we implement a get method and associate it with the URL `/cpu`. The `operationid`, defines the method that we call which, as we used the local directory, is included in the file `cpu.py`. This is controlled by the prefix in the operation id.

A straightforward function to return the CPU information is defined in `cpu.py` which we list next

```

import os, platform, subprocess, re
from flask import jsonify

def get_processor_name():
    if platform.system() == "Windows":
        p = platform.processor()
    elif platform.system() == "Darwin":
        command = "/usr/sbin/sysctl -n machdep.cpu.brand_string"
        p = subprocess.check_output(command, shell=True).strip().decode()
    elif platform.system() == "Linux":
        command = "cat /proc/cpuinfo"
        all_info = subprocess.check_output(command, shell=True).strip().decode()
        for line in all_info.split("\n"):
            if "model name" in line:
                p = re.sub(".*model name.*:", "", line, 1)
    else:
        p = "cannot find cpuinfo"
    pinfo = {"model": p}
    return jsonify(pinfo)

```

We have implemented this function to return a *jsonified* information from the dict `pinfo`.

To simplify working with this example, we also provide a makefile for OSX that allows us to call the server and the call to the server in two different terminals

```

define terminal
  osascript -e 'tell application "Terminal" to do script "cd $(PWD); $1"'
endef

install:
  pip install -r requirements.txt

demo:
  $(call terminal, python server.py)
  sleep 3
  @echo "===== "
  @echo "Get the info"
  @echo "===== "
  curl http://localhost:8080/cloudmesh/cpu
  @echo
  @echo "===== "

```

When we call

```
make demo
```

our demo is run.

3.3.1 VERIFICATION

It is essential to be able to verify if a YAML file is correct. To identify this, the easiest method is to use the swagger editor. There is an online version available at:

- <https://editor.swagger.io/>

Go to the Web site, remove the current petstore example, and paste your YAML file in it. Debug messages are helping you to correct things.

A terminal-based command may also be helpful but is a bit difficult to read.

```
$ connexion run cpu.yaml --stub --debug
```

3.3.2 SWAGGER-UI

Swagger comes with a convenient UI to invoke REST API calls using the Web browser rather than relying on the curl commands.

Once the request and response definitions are correctly specified, you can start the server by,

```
$ python server.py
```

Then the UI would also be spawned under the service URL *http://[service url]/ui/*

Example: <http://localhost:8080/cloudmesh/ui/>

3.3.3 MOCK SERVICE

In some cases, it may be useful to develop the API without having yet developed methods that you call with the OperationI. In this case, it is useful to run a mock service. You can invoke such a service with

```
$ connexion run cpu.yaml --mock=all -v
```

3.3.4 EXERCISE

OpenAPI.Conexion.1:

Modify the makefile, so it also works on ubuntu, but do not disable the ability to run it correctly on OSX. Tip use if's in makefiles base on the OS. You can look at the makefiles that create this book as an example. Find alternatives to starting a terminal in Linux.

OpenAPI.Conexion.2:

Modify the makefile, so it also works on Windows 10, but do not disable the ability to run it correctly on OSX. Tip use ifs in makefiles. You can look at the makefiles that create this book as example. Find alternatives to start a PowerShell or cmd.exe in windows. Maybe you need to use GitBash.

OpenAPI.Conexion.3:

Implement a swagger specification of an issue related to the NIST BDRA. Implement it. Please remember this could prepare you for a project good topics include:

- *virtual compute service interfacing with AWS, Azure, Google or OpenStack*
- *virtual directory service interfacing with google drive, box, GitHub, iCloud, FTP, scp, and others*

As there are so many possibilities to contribute, come up in class with one specification and then implement it for different providers. The difficulty here is that it is not done for one IaaS, but for all of them and all can be integrated.

This exercise is typically growing to be part of your class project.

OpenAPI.Conexion.4:

Develop instructions on how to integrate the OpenAPI service framework in a WSGI based Web service. Chose a service you like so

that the service could run in production.

OpenAPI.Conexion.5:

Develop instructions on how to integrate the OpenAPI service framework in Tornado so the service could run in production.

4 REST SPECIFICATIONS

4.1 TASK 4 – GENERALIZED REST SERVICE SPECIFICATIONS



Based on the lessons from the exposure to the community, we have received contributions for the definition of REST services addressing database access, virtual directories, and map-reduce as follows:

- A database access with an example to AWS Redshift <https://github.com/cloudmesh/cloudmesh-redshift/blob/master/cloudmesh/redshift/openapi/redshift.yaml>
- Virtual Directory <https://github.com/cloudmesh/cloudmesh-storage/blob/master/cloudmesh/vdir/openapi/vdir.yaml>
- Map/Reduce with an example of using AWS emr <https://github.com/cloudmesh/cloudmesh-emr/blob/master/cloudmesh/emr/openapi/emr.yaml>

Due to the time frame these specifications were developed and OpenAPI tools not yet working vor OpenAPI v3 the specifications are provided in OPenAPI v2. A future activity could include converting them to OpenAPI v3.

The specifications are included as an example next.

4.2 AMAZON REDSHIFT OPENAPI SPECIFICATION

```
swagger: "2.0"
info:
  description: "This is an OpenAPI for Amazon RedShift"
  version: "1.0.0"
  title: "AWS RedShift"
  termsOfService: "IU"
  contact:
    email: "joshish@iu.edu"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
host: "localhost:8080"
basePath: "/cloudmesh/redshift/v1"
tags:
- name: "cluster"
  description: "RedShift Clusters"
```

```

externalDocs:
  description: "Get information and operate on Clusters"
  url: "http://swagger.io"
schemes:
- "http"
consumes:
- "application/json"
produces:
- "application/json"
paths:
  /clusters:
    get:
      tags:
      - "cluster"
      summary: "Describe all clusters"
      description: "Detailed description of all cluster attributes"
      operationId: "cloudmesh.redshift.Provider.describe_clusters"
      produces:
      - "application/json"
      responses:
        200:
          description: "successfully listed clusters"
          schema:
            $ref: "#/definitions/RedShiftCluster"
        400:
          description: "Invalid status value"
  /cluster/{clusterId}:
    get:
      tags:
      - "cluster"
      summary: "Describe cluster by ID"
      description: "Returns a single cluster description"
      operationId: "cloudmesh.redshift.Provider.describe_cluster"
      produces:
      - "application/json"
      parameters:
      - name: "clusterId"
        in: "path"
        description: "ID of cluster"
        required: true
        type: "string"
      responses:
        200:
          description: "successfully listed cluster"
          schema:
            $ref: "#/definitions/RedShiftCluster"
        400:
          description: "Invalid ID supplied"
        404:
          description: "Cluster not found"
    post:
      tags:
      - "cluster"
      summary: "Creates a cluster"
      description: ""
      operationId: "cloudmesh.redshift.Provider.create_multi_node_cluster"
      produces:
      - "application/json"
      parameters:
      - name: "clusterId"
        in: "path"
        description: "ID of cluster to be created"
        required: true
        type: "string"
      - name: "dbName"
        in: "query"
        description: "Name of the DB"
        required: true
        type: "string"
      - name: "masterUserName"
        in: "query"
        description: "Master user name"
        required: true
        type: "string"
      - name: "password"
        in: "query"
        description: "Master user password"
        required: true
        type: "string"
      - name: "nodeType"
        in: "query"

```

```

    description: "Type of the node of the cluster"
    type: "string"
    default: 'dc2.large'
  - name: "clusterType"
    in: "query"
    description: "Type of the cluster"
    type: "string"
    default: 'multi-node'
  - name: "nodeCount"
    in: "query"
    description: "Count of nodes in cluster"
    type: "number"
    default: 2
  responses:
    405:
      description: "Invalid input"
delete:
  tags:
  - "cluster"
  summary: "Deletes a cluster"
  description: "Delete a cluster"
  operationId: "cloudmesh.redshift.Provider.delete_cluster"
  produces:
  - "application/json"
  parameters:
  - name: "clusterId"
    in: "path"
    required: true
    type: "string"
  responses:
    400:
      description: "Invalid ID supplied"
    404:
      description: "Cluster not found"
/ccluster/{clusterId}/resize:
  patch:
    tags:
    - "cluster"
    summary: "Increases cluster nodes"
    description: ""
    operationId: "cloudmesh.redshift.Provider.resize_cluster_to_multi_node"
    produces:
    - "application/json"
    parameters:
    - name: "clusterId"
      in: "path"
      description: "ID of cluster to resize"
      required: true
      type: "string"
    - name: "clusterType"
      in: "query"
      description: "Type of the cluster"
      required: false
      type: "string"
      default: "multi-node"
    - name: "nodeCount"
      in: "query"
      description: "Count of nodes to resize cluster to"
      required: false
      type: "number"
      default: 2
    - name: "nodeType"
      in: "query"
      description: "Type of nodes "
      required: false
      type: "string"
      default: 'dc2.large'
    responses:
      200:
        description: "successful operation"
        schema:
          $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/changenodetype:
  patch:
    tags:
    - "cluster"
    summary: "Change node type"
    description: ""
    operationId: "cloudmesh.redshift.Provider.resize_cluster_node_types"
    produces:
    - "application/json"

```

```

parameters:
- name: "clusterId"
  in: "path"
  description: "ID of cluster to update"
  required: true
  type: "string"
- name: "clusterType"
  in: "query"
  description: "Type of the cluster"
  required: true
  type: "string"
- name: "nodeType"
  in: "query"
  description: "Type of the node"
  required: false
  type: "string"
  default: 'dc2.large'
responses:
  200:
    description: "successful operation"
    schema:
      $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/rename:
patch:
  tags:
  - "cluster"
  summary: "Rename cluster"
  description: "Rename the cluster"
  operationId: "cloudmesh.redshift.Provider.rename_cluster"
  produces:
  - "application/json"
  parameters:
  - name: "clusterId"
    in: "path"
    description: "ID of cluster to rename"
    required: true
    type: "string"
  - name: "newId"
    in: "query"
    description: "New ID for the cluster"
    required: true
    type: "string"
  responses:
    200:
      description: "successful operation"
      schema:
        $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/changepassword:
patch:
  tags:
  - "cluster"
  summary: "Change master password"
  description: "Change the password for the cluster"
  operationId: "cloudmesh.redshift.Provider.modify_cluster"
  produces:
  - "application/json"
  parameters:
  - name: "clusterId"
    in: "path"
    description: "ID of cluster to update"
    required: true
    type: "string"
  - name: "newPass"
    in: "query"
    description: "New Password"
    required: true
    type: "string"
  responses:
    200:
      description: "successfully change password"
      schema:
        $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/allowaccess:
patch:
  tags:
  - "cluster"
  summary: "Allow external access to cluster"
  description: "Allow external programs (eg Python) access to the Redshift cluster for queries"
  operationId: "cloudmesh.redshift.Provider.allow_access"
  produces:
  - "application/json"

```

```

parameters:
- name: "clusterId"
  in: "path"
  description: "ID of cluster to allow access to"
  required: true
  type: "string"
responses:
  200:
    description: "successfully allowed access"
    schema:
      $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/runDDL:
  patch:
    tags:
      - "cluster"
    summary: "Runs DDL on cluster"
    description: "Run SQL Statements like CREATE TABLE, ALTER TABLE, DROP TABLE on the database"
    operationId: "cloudmesh.redshift.Provider.runddl"
    produces:
      - "application/json"
    parameters:
      - name: "clusterId"
        in: "path"
        description: "ID of cluster "
        required: true
        type: "string"
      - name: "dbName"
        in: "query"
        description: "DB Name"
        required: true
        type: string
      - name: "host"
        in: "query"
        description: "Host name"
        required: true
        type: "string"
      - name: "port"
        in: "query"
        description: "Port number"
        required: false
        type: "integer"
        default: 5439
      - name: "userName"
        in: "query"
        description: "User name"
        required: true
        type: "string"
      - name: "password"
        in: "query"
        description: "Password"
        required: true
        type: "string"
      - name: "sql_file_contents"
        in: "query"
        description: "Contents of an SQL file"
        required: true
        type: "string"
        format: base64
    responses:
      200:
        description: "successfully ran the DDL statements"
        schema:
          $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/runDML:
  patch:
    tags:
      - "cluster"
    summary: "Runs DML on cluster"
    description: "Run SQL Statements like INSERT, UPDATE, DELETE for data in the database"
    operationId: "cloudmesh.redshift.Provider.rundml"
    produces:
      - "application/json"
    parameters:
      - name: "clusterId"
        in: "path"
        description: "ID of cluster "
        required: true
        type: "string"
      - name: "dbName"
        in: "query"
        description: "DB Name"

```

```

    required: true
    type: string
  - name: "host"
    in: "query"
    description: "Host name"
    required: true
    type: "string"
  - name: "port"
    in: "query"
    description: "Port number"
    required: false
    type: "integer"
    default: 5439
  - name: "userName"
    in: "query"
    description: "User name"
    required: true
    type: "string"
  - name: "password"
    in: "query"
    description: "Password"
    required: true
    type: "string"
  - name: "sql_file_contents"
    in: "query"
    description: "Contents of an SQL file"
    required: true
    type: "string"
    format: base64
responses:
  200:
    description: "successfully ran the DML file"
    schema:
      $ref: "#/definitions/RedShiftCluster"
/ccluster/{clusterId}/runQuery:
  patch:
    tags:
      - "cluster"
    summary: "Runs a query on the cluster"
    description: "Run SQL SELECT Statement"
    operationId: "cloudmesh.redshift.Provider.runselectquery_text"
    produces:
      - "application/json"
    parameters:
      - name: "clusterId"
        in: "path"
        description: "ID of cluster "
        required: true
        type: "string"
      - name: "dbName"
        in: "query"
        description: "DB Name"
        required: true
        type: string
      - name: "host"
        in: "query"
        description: "Host name"
        required: true
        type: "string"
      - name: "port"
        in: "query"
        description: "Port number"
        required: false
        type: "integer"
        default: 5439
      - name: "userName"
        in: "query"
        description: "User name"
        required: true
        type: "string"
      - name: "password"
        in: "query"
        description: "Password"
        required: true
        type: "string"
      - name: "queryText"
        in: "query"
        description: "SQL query"
        required: true
        type: "string"
        format: base64

```



```

responses:
  200:
    description: "successfully run the query"
    schema:
      $ref: "#/definitions/RedShiftCluster"

definitions:
  RedShiftCluster:
    type: "object"
    required:
      - "clusterId"
    properties:
      clusterId:
        type: "string"

```

4.3 VIRTUAL DIRECTORY

```

swagger: "2.0"
info:
  version: "0.0.1"
  title: "vdir"
  description: "A service for cloudmesh virtual directory"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "cloudmesh virtual directory REST Service"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/cloudmesh"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /mkdir:
    post:
      tags:
        - VDIR
      operationId: vdir_openapi.mkdir
      description: "Returns new directory"
      parameters:
        - in: body
          name: params
          description: "Provide the directory name in body of the request"
          schema:
            $ref: "#/definitions/PUT"
      produces:
        - "application/json"
      responses:
        "200":
          description: "Makes new directory"
  /cd:
    get:
      tags:
        - VDIR
      operationId: vdir_openapi.cd
      description: "Returns list of endpoints in the specified directory"
      parameters:
        - in: query
          name: dir
          description: "Provide the directory name in body of the request"
          type: string
          required: false
      produces:
        - "application/json"
      responses:
        "200":
          description: "navigate to directory"
          schema:
            $ref: "#/definitions/LIST"
  /ls:
    get:
      tags:
        - VDIR
      operationId: vdir_openapi.ls

```

```

description: "Returns list of documents in the specified directory"
parameters:
  - in: query
    name: dir
    description: "Specify directory to list contents of"
    type: string
    required: false
produces:
  - "application/json"
responses:
  "200":
    description: "list documents"
    schema:
      $ref: "#/definitions/LIST"
/add:
  post:
    tags:
      - VDIR
    operationId: vdir_openapi.add
    description: "Returns new document"
    parameters:
      - in: body
        name: params
        description: "Provide the location of the file and link in the body of the request"
        schema:
          $ref: "#/definitions/ADD"
    produces:
      - "application/json"
    responses:
      "200":
        description: "add file link"
/delete:
  get:
    tags:
      - VDIR
    operationId: vdir_openapi.delete
    description: "Returns list of documents in the specified directory"
    parameters:
      - in: query
        name: dir_or_name
        description: "Provide the directory or link name in body of
          the request"
        type: string
        required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "delete links or directories"
        schema:
          $ref: "#/definitions/LIST"
/status:
  get:
    tags:
      - VDIR
    operationId: vdir_openapi.status
    description: "Returns specified document"
    parameters:
      - in: query
        name: dir_or_name
        description: "Provide the directory or link name in body of
          the request"
        type: string
        required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "get status of link or directory"
        schema:
          $ref: "#/definitions/LIST"
/get:
  get:
    tags:
      - VDIR
    operationId: vdir_openapi.get
    description: "Returns specified file"
    parameters:
      - in: query
        name: name
        description: "Provide the link name in body of the request"

```

```

    type: string
    required: true
  - in: query
    name: destination
    description: "Provide the destination directory name
      in the body of the request"
    type: string
    required: true
  produces:
  - "application/json"
  responses:
    "200":
      description: "download the specified file"
      schema:
        $ref: "#/definitions/LIST"
definitions:
  PUT:
    type: "object"
    required:
    - "dir"
    properties:
      dir:
        type: "string"
  LIST:
    type: "object"
    required:
    - "results"
    properties:
      results:
        type: "string"
  ADD:
    type: "object"
    required:
    - "endpoint"
    - "dir_and_name"
    properties:
      endpoint:
        type: "string"
      dir_and_name:
        type: "string"

```

4.4 ELASTIC MAP REDUCE

```

swagger: "2.0"
info:
  version: "0.0.1"
  title: "Amazon AWS EMR"
  description: "An OpenAPI Service to manage AWS EMR clusters."
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: ""
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/api"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /list_clusters:
    get:
      operationId: cloudmesh.emr.openapi.cloudmesh.emr.list_clusters
      description: "Returns details of clusters visible to this account."
      parameters:
        - name: status
          description: "The status of the cluster to search for."
          in: query
          type: string
          required: false
      produces:
        - "application/json"
      responses:
        "200":
          description: "Cluster Listing"

```

```
    schema:
      $ref: "#/definitions/emr"
/list_instances:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.list_instances
    description: "Returns instance details for a given cluster, status,
      and type."
    parameters:
      - name: cluster
        description: "The ClusterID to list instances for."
        in: query
        type: string
        required: true
      - name: status
        description: "The state of instance to search for."
        in: query
        type: string
        required: false
      - name: type
        description: "The type of instance to search for."
        in: query
        type: string
        required: false
    produces:
      - "application/json"
    responses:
      "200":
        description: "Instance Listing"
        schema:
          $ref: "#/definitions/emr"
/list_steps:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.list_steps
    description: "Returns details of the steps running on the cluster."
    parameters:
      - name: cluster
        description: "The ClusterID to list steps for."
        in: query
        type: string
        required: true
      - name: status
        description: "The status of the step to search for."
        in: query
        type: string
        required: false
    produces:
      - "application/json"
    responses:
      "200":
        description: "Step Listing"
        schema:
          $ref: "#/definitions/emr"
/describe:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.describe
    description: "Describes a cluster."
    parameters:
      - name: cluster
        description: "The ClusterID to list instances for."
        in: query
        type: string
        required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "Cluster Description"
        schema:
          $ref: "#/definitions/emr"
/stop:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.stop
    description: "Stops a cluster."
    parameters:
      - name: cluster
        description: "The ClusterID to stop."
        in: query
        type: string
        required: true
    produces:
      - "application/json"
```

```

responses:
  "200":
    description: "Cluster Stopped"
    schema:
      $ref: "#/definitions/emr"
/start:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.start
    description: "Stops a cluster."
    parameters:
      - name: name
        description: "The name of the cluster to start."
        in: query
        type: string
        required: true
      - name: count
        description: "The number of servers to use for the cluster."
        in: query
        type: string
        required: true
      - name: master
        description: "The instance type to use for the Master node."
        in: query
        type: string
        required: false
      - name: node
        description: "The instance type to use for the Worker nodes."
        in: query
        type: string
        required: false
    produces:
      - "application/json"
    responses:
      "200":
        description: "Cluster Started"
        schema:
          $ref: "#/definitions/emr"
/upload:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.upload
    description: "Upload a file to an S3 bucket for processing."
    parameters:
      - name: file
        description: "The filename to upload from the REST server."
        in: query
        type: string
        required: true
      - name: bucket
        description: "The bucket to upload the file to."
        in: query
        type: string
        required: true
      - name: bucketname
        description: "The name to save the file as in the bucket."
        in: query
        type: string
        required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "Cluster Started"
        schema:
          $ref: "#/definitions/emr"
/copy:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.copy
    description: "Copy a file from an s3 bucket to the master node's
      hadoop directory."
    parameters:
      - name: cluster
        description: "The ClusterID to copy to."
        in: query
        type: string
        required: true
      - name: bucket
        description: "The bucket to use as the source."
        in: query
        type: string
        required: true
      - name: bucketname

```



```
      description: "The name to bucket object to copy."
      in: query
      type: string
      required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "Cluster Started"
        schema:
          $ref: "#/definitions/emr"
/run:
  get:
    operationId: cloudmesh.emr.openapi.cloudmesh.emr.run
    description: "Submit a file to Spark."
    parameters:
      - name: cluster
        description: "The ClusterID to use."
        in: query
        type: string
        required: true
      - name: bucket
        description: "The bucket to use as the source."
        in: query
        type: string
        required: true
      - name: bucketname
        description: "The name of the program file to run."
        in: query
        type: string
        required: true
    produces:
      - "application/json"
    responses:
      "200":
        description: "Cluster Started"
        schema:
          $ref: "#/definitions/emr"
definitions:
  emr:
    type: "object"
    required:
      - "model"
    properties:
      model:
        type: "string"
```

5 APPLICATIONS

5.1 TASK 5 – APPLICATION EXAMPLES

It is crucial to verify the applicability of the general framework, while we test the reusability of the proposed software system. We show this on multiple well-understood applications identified by such as the NIST fingerprint and face detection examples. Following these initial verification and demonstration projects, we expand usability studies into IaaS, PaaS, and other applications. This is fostered in tight collaboration with our educational dissemination activities, as described next. The importance of building a tight integration between education and specification is that students provided a meaningful and critical step in the verification and fine-tuning of the development and documentation activities.

5.2 S-CONE CLASSIFICATION USING REST SERVICES AND MACHINE LEARNING

Brandon Fischer, Ryan Danehy
bfisch9@iu.edu, rdanehy@iu.edu
Indiana University Bloomington
hid: sp19-222-89 sp19-222-102
github: 
code: 

Keywords: S-cones, Scikit, Support Vect Machine, Neural Network, WebPlotViz, ISOS (Inner Segment - Outer Segment junction), Cone Outer segment tip (COST).

5.2.1 ABSTRACT

We worked in partnership with Dr. Don Miller's lab from the IU School of Optometry to create a binary classifier which is trained to differentiate (and

generate a count of) S-cones from L and M cones in 3D retinal imaging. We deployed an RBF-kernel SVM to classify S-cones vs non-S-cones. This project has clinical significance in the tracking of progression of the disease Retinitis Pigmentosa (RP). In RP, S-cones can be seen migrating from their natural positions, and eventually disappearing entirely in retinal scans. Our service could be extended from purely classifying/counting S-cones to tracking the rate of their movement and determining the progression/severity of the disease in a given patient.

5.2.2 INTRODUCTION

Cones or Cone cells are photoreceptor cells in the retinas of humans. They are responsible for color vision and work best in bright lights. S-cone cells differ from M-cones and L-cones based on the light wavelengths they are sensitive to. For example S-cones are sensitive to short-wavelengths, M-cones to medium-wavelengths, and L-cones to Long-wavelengths [5]. Short-wavelengths correspond with 'blue', medium with 'green', and long with 'red', therefore it is believed that the study of these cones could lead to new insights into diseases such as red-green colorblindness.

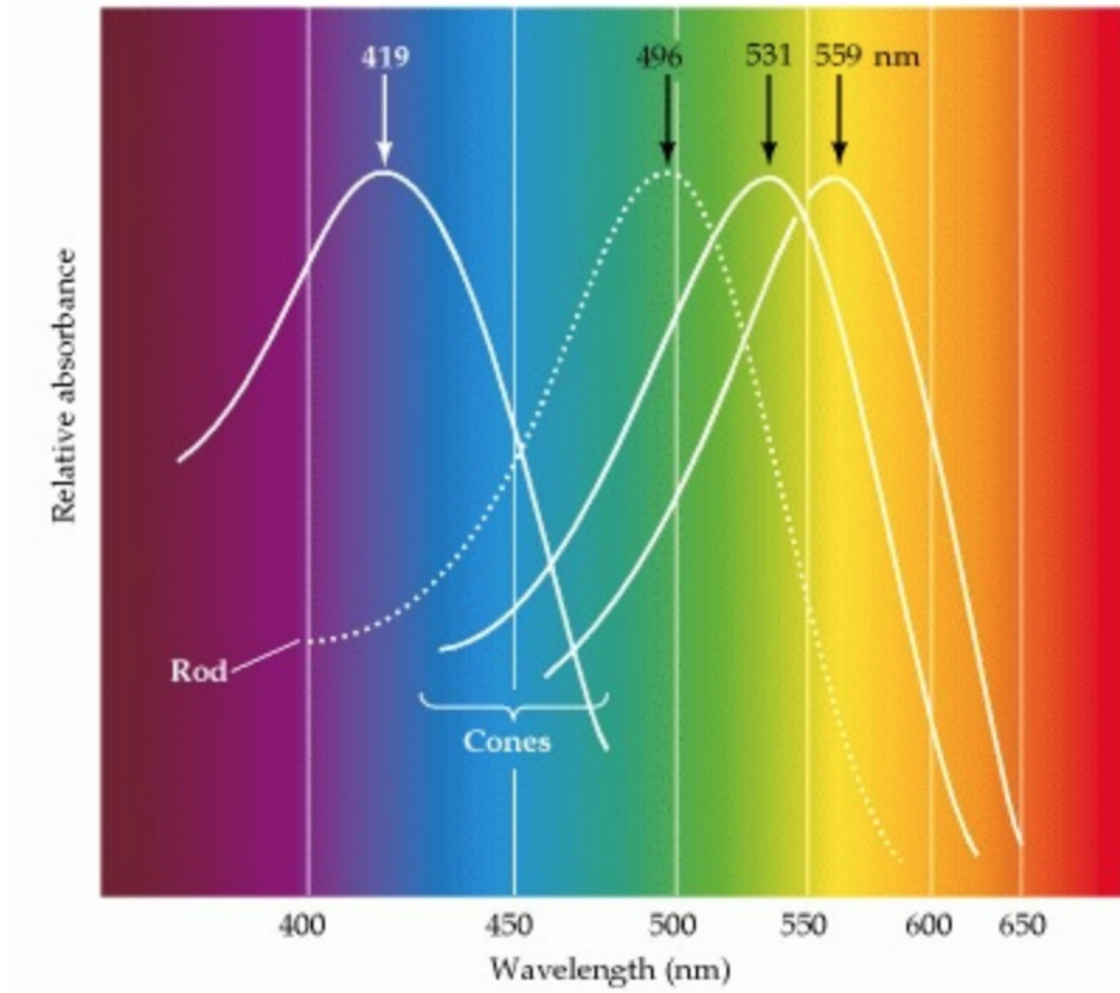


Figure 1: Spectrum of Wavelengths

Caption: Color vision. The absorption spectra of the four photopigments in the normal human retina. The solid curves indicate the three kinds of cone opsins; the dashed curve shows rod rhodopsin for comparison. Absorbance is defined as the log value of the intensity of incident light divided by intensity of transmitted light [5].

Individual cones are entirely color blind in that their response is simply a reflection of the number of photons they capture, regardless of the wavelength of the photon. Therefore it is impossible to determine why a change in the effectiveness of a particular cone occurred. This question can only be resolved by comparing the activity in different classes of cones. Comparisons of the responses of individual cone cells, and cells at higher levels in the visual pathway are clearly involved in how the visual system extracts color information

from spectral stimuli. However, understanding of the neural mechanisms that underlie color perception has been elusive to the scientific community [Figure 1](#).

“This diagram was produced based on histological sections from a human eye to determine the density of the cones. The diagram represents an area of about 1° of visual angle. The number of S-cones was set to 7% based on estimates from previous studies. The L-cone:M-cone ratio was set to 1.5. This is a reasonable number considering that recent studies have shown wide ranges of cone ratios in people with normal color vision. In the central fovea an area of approximately 0.34° is S-cone free. The S-cones are semi-regularly distributed and the M- and L-cones are randomly distributed. Throughout the whole retina the ratio of L- and M- cones to S-cones is about 100:1 [6].”

There are two main reflection sites inside the cone photoreceptor cells that line the back of the eye. The first one occurs at what is called the inner segment – outer segment junction (ISOS) and the second one (which occurs directly behind the first one) occurs at what is called the cone outer segment tip (COST). Cones can be classified by the comparison of the inner segment length vs outer segment length. For example, histologically S-cones have a longer inner segment and a shorter outer segment. The ultimate goal of this project was to design an effective and fast method of classification of S-cones, using REST Services to facilitate user interaction with the model.

5.2.3 DATA

Datasets were provided by Dr. Miller’s lab which included the 3D coordinates and aperture size of each cone detected within the retinal scan. Using this information, we were able to differentiate the S-cones from the others due to their deeper position and wider aperture compared to the other cell types. Our starting dataset includes information from the images of three patients’ retinas, with a mix of healthy and colorblind individuals. Additional data was collected as needed. The data provided was delivered in a fashion to not jeopardize the patients anonymity nor were we given any personal data that could or would put a patient’s privacy in concern.

After the initial model was trained, unlabeled data is given a classification through a rest service, and the count and locations of S-cones was returned via

another rest service. The service also allows the retraining of the model on new datasets, and outputs the corresponding metrics on the newly trained model. This allows the model to be updated and re-trained as more data becomes available.

Our data includes eight features: X-Coordinate of cone (Coord_X), Y Coordinate of cone (Coord_Y), Retina Depth location of Inner- Outer Segment (ISOS_Z), X Coordinate of ISOS (ISOS_size_X), Y Coordinate of ISOS (ISOS_size_Y), Retina Depth location of COST (COST_Z), X Coordinate of COST (COST_size_X), and Y Coordinate of COST (COST_size_Y). These features were extracted from 3D imaging of the retinal hence the three dimensional parameter types. ISOS_z is the retinal depth location of ISOS and COST_z is the retinal depth location of COST.

5.2.3.1 Preprocessing

Our raw data had some observations that were unknown or missing certain datapoints, and as such were marked “Nan” in the original dataset. We preprocessed the data in order to exclude feature vectors which included Nan for any feature value. The preprocessing that we performed on the data can be seen in the read_data.py file. Machine learning models can be very sensitive to scaling and in order to prevent this we normalized the data. Normalization rescales the data to be in the range of 0 to 1, thus eliminating any possible feature scaling within our data. We performed normalization on our data using Scikit learn’s preprocessing.normalize() function. This function scales the input individually to unit norms (vector length). In order to guarantee the quality of our data before training a model, we standardized our data. Standardization transforms data to have a mean of zero and standard deviation of 1. Standardization was performed by using scikit learn’s StandardScaler(). The function first_model and retrain_model in model.py show this normalization and standardization.

5.2.3.2 Visualization

We visualized our data using WebPlotViz which results can be seen using the following [link](https://spidal-gw.dsc.soic.indiana.edu/public/groupdashboard/E222). <https://spidal-gw.dsc.soic.indiana.edu/public/groupdashboard/E222>. From the WebPlotViz visualizations it can be noted that the data is not clearly separated into clusters

nor in a regular shape. It also important to notice how there is no clear distinction on which features are weighted heavier than others in classifying S-cone from M and L-cones. However, histological studies have shown that the biggest differentiation between the different cones types is the difference between ISOS_Z - COST_Z [7]. This difference signifies the physical length of an important component of the cone photoreceptors. In one of the visualizations we plotted X_coordinate vs Y-coordinate vs (COST_Z- ISOS_Z). In this plot it is not glaringly obvious that (COST_Z- ISOS_Z) is the most important feature, but there does seem to be a noticeable correlation. The lack of an obviously dominant feature led us to the conclusion that for our model to train the best no weights should be applied (*Not sure that's accurate, I think DNN will always give weights*)

5.2.4 MODEL DISCUSSION

In our final project we decided to use a neural network using scikit learn's MLPClassifier(). We had several reasons for picking a neural network model, one being the fact that it is supervised learning. Supervised learning makes sense for our project because in our training data we were given labels for every cell observed. Supervised learning also made sense given that our goal was to predict the type of cone for a given cell, and making predictions is usually the goal behind supervised learning algorithms. Additionally, neural networks are ideal for solving non-linear classification problems, which is precisely what we are trying to solve [8].

5.2.4.1 Failures

We started out trying to use a support vector machine algorithm (SVM) as our model of choice, but several problems became evident while trying to implement the SVM. We originally chose Scikit's svm.SVC() algorithm, but after testing it became apparent the model was not accurate at all averaging an F1 score below .3. We then experimented with altering the parameters of the algorithm, including changing from a linear to non-linear SVM model. We found that changing the kernel to 'RBF' produced an F1 score of 1.0. This makes sense given that our data is grouped in a nonlinear way and 'RBF' are used for nonlinear solutions.

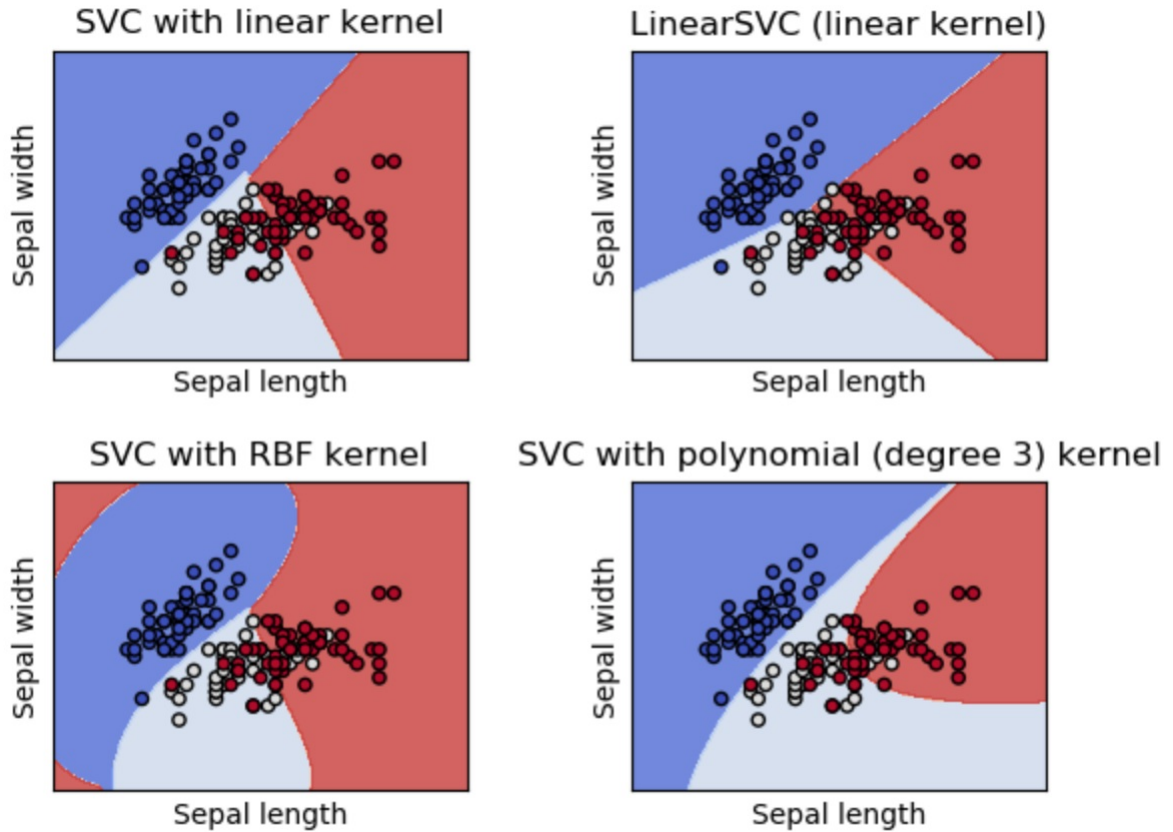


Figure 2: RBF Kernel example

However, an F1 score of 1.0 is a possible symptom of an over-fitting problem. Over-fitting is when an algorithm matches so perfectly to the training data that the model does not generalize well to classification of additional datasets.

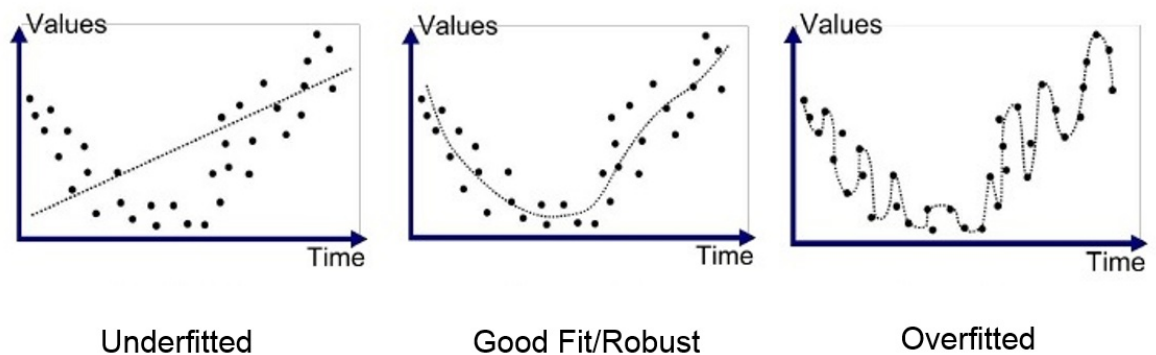


Figure 3: Overfitting example

To determine whether there is overfitting or not we decided to decrease the amount of data used for training. Decreasing the amount of data used for training should only slightly decrease the performance of the model unless there is

overfitting. If there was overfitting then they would be a drastic difference since they would be less data to fit perfectly too. We trained with roughly 60% of the data rather than 80%. The new model that was trained with 60% of the data performed horribly, confirming the overfitting problem. Thus we decided to move on to the implementation of a neural network.

5.2.4.2 Activation Function

When deciding to implement a multi-layer perceptron (neural network) it is important to consider and analyze what activation function will work best for your data solution. The types of activation functions have very important influences on the networks' learning speeds, classification correct rates and non-linear mapping precision. Activation functions determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1. An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model, which places an increased computational strain on the activation function, and its derivative function. Backpropagation is an algorithm which traces back from the output of the model, through the different neurons which were involved in generating that output, back to the original weight applied to each neuron. Backpropagation suggests an optimal weight for each neuron, which results in the most accurate prediction [9].

5.2.4.3 Decision

In order to decide what activation function fit our model best we experimented with a few activation functions. Firstly we experimented with a logistics function which performed poorly averaging an F1 score around .5. Next we tried using the RELU function which performed significantly better averaging a F1 score around .9. However the best was the Hyperbolic Tangent (Tanh), which averaged f1 score around .95. The advantages tanh provides is that its zero centered meaning it makes it easier to model strongly positive, negative, or

neutral inputs. As previously mentioned our input (features) are very neutral making this activation function perfect for our model [9].

5.2.5 REST SERVICE IMPLEMENTATIONS

REST is an abstraction of the basic HTTP methods (like GET, POST, etc) which is used to build APIs that behave in predictable ways. The basic behaviors of REST services are called CRUD: Create, Read, Update, and Delete [10]. These map to the HTTP methods POST, GET, PUT, and DELETE, respectively. When a server is launched that implements REST conventions, the fields of the URL submitted to the website will contain endpoints and/or parameters which are used to specify the desired behavior.

The interaction between client and server for our service in particular involves the use of 3 yaml specified endpoints: /, /app/run, and /app/retrain. These can be found in our master.yaml file. The '/' endpoint is a simple read action from the client, and a rendered html page with information about our service is returned to the client.

The /app/run endpoint specifies another read action, and returns a rendered html template to the client which will allow a user to upload one or more .csv files containing data which they want classified. When the user clicks the upload button on this page, a create action is used to send the data files from client to server. The files which are uploaded are then run through the model to have their S-cones counted, and then an html file containing feedback is dynamically generated. A rendered version is returned to the client.

Similarly, the /app/retrain endpoint specifies a read action and returns a rendered file-upload html template to the client. The client user will upload files with which to train a new model, and then by clicking the upload button they again generate a create action. These uploaded files are then used to train a new model, which is created, saved, and tested. The metrics calculated as a result of this testing, accuracy, precision, recall, and F1 score, are dynamically written to an html file, which is then rendered and returned to the client.

5.2.6 SPECIFICATION

```
swagger: "2.0"
info:
```

```
version: "0.0.1"
title: "cone classifier"
description: "classify cones from csv data"
termsOfService: "http://swagger.io/terms/"
contact:
  name: "Ryan Danehy and Brandon Fischer"
license:
  name: "Apache"
host: "localhost:4555"

consumes:
- "text/html"
produces:
- "text/html"

basePath: "/app"

paths:
  /run:
    get:
      operationId: scripts.uploadrun.display
      description: "Displays upload file page"
      responses:
        "200":
          description: "Upload page displayed successfully"

  /upload_file:
    post:
      operationId: scripts.uploadrun.upload
      description: "Generate post request to actually upload file"
      responses:
        "201":
          description: "File upload successful"

  /retrain:
    get:
      operationId: scripts.uploadrerun.display
      description: "Displays upload new training file page"
      responses:
        "200":
          description: "Upload page displayed successfully"


  /upload_file_retrain:
    post:
      operationId: scripts.uploadrerun.upload
      description: "Generate post request to actually upload file"
      responses:
        "201":
          description: "File upload successful"
```

5.3 ROOKIE FANTASY FOOTBALL POINT PREDICTION

Andrew Gotts, Ethan Japundza, Brian Schwantes
adgotts@iu.edu, ejapundz@iu.edu, bschwant@iu.edu

Indiana University

hid: sp19-222-94, sp19-222-90, sp19-222-92

github: 

code: 

Keywords: KNN, Fantasy Football, REST, Docker, Yaml

5.3.1 ABSTRACT

As the number of fantasy football players increases dramatically every year, we saw an opportunity to create a service that will help users draft better teams. While NFL veterans have gameplay for fantasy enthusiasts to evaluate, incoming rookies with a lack of professional experience make it difficult for fans to evaluate whether or not their teams picks will be successful in the NFL. To solve this problem, we implemented a REST service that gets an aggregate of combine and fantasy football data from Google-Drive. Our service then utilizes the K-Nearest Neighbor machine learning algorithm on our data sets, outputting our projections for which 2019 NFL Rookies will score the most fantasy football points based on their metrics from the combine.

5.3.2 INTRODUCTION

Who should be the top pick in this years fantasy football draft? This is a question that has plagued fantasy football enthusiasts since its creation in 1962. But why does it even matter, isn't fantasy football just a game? According to Joris Drayer, a professor at Temple University in Sports Marketing and Analytics, almost 30 million Americans and Canadians actively participate in fantasy sports leagues every year. Drayer goes on to discuss the economic impact that fantasy football has on the sports industry, estimating it to be nearly \$4.5 billion [11]. With such a large amount of people and money involved, the technologies created to help players be successful are in a position to revolutionize the fantasy football market. Utilizing the power of machine learning, we were able to create the 'Rookie Fantasy Football Point Predictor'. A service for those trying to make an educated pick on rookie players with no professional experience.

5.3.3 DATA SET

Our project uses information from three data sets consisting of the 2000-2018_NFL_Combine statistics, the 2019_NFL_Combine statistics for this year's rookies, and the 2001-2018_Fantasy_Football data for every active NFL player. These datasets include six combine drills:

- 40-Yard Dash: Measures a player's explosion, burst, and acceleration.
- Three-Cone Drill: Highlights a players ability to shift directions at a high

- speed.
- Shuttle Run: Demonstrates short area lateral quickness, and the speed at which a player can change directions.
 - Vertical Jump: Tests for lower body extension and power.
 - Broad Jump: Evaluates an athlete's lower-body explosion, and lower-body strength.
 - Bench Press: Demonstrates a players strength and endurance.

From these drills measurables are obtained that can serve as a predictor for the rookies effectiveness in the NFL. For certain positions, like wide receiver, the shuttle run is significant as this drill will often determine whether or not the defender is able to get separation between his defender at the line of scrimmage. While for other positions like quarterback, the shuttle run has less importance as they are rarely making quick and decisive cuts while playing in NFL games [12]. Due to the difference in how each drill can serve as a predictor for different positions, we found the best way to analyze the incoming rookies within the 2019_NFL_Combine dataset was to compare their performance to previous NFL players within the 2000-2018_NFL_Combine dataset. From this comparison, our goal was to both rank and predict the average fantasy points-per-game for the incoming rookie class based on how their comparisons performed in the 2001-2018_Fantasy_Football dataset. We found the best way to make this correlation was to employ the K-Nearest Neighbor machine learning algorithm for our service.

5.3.4 KNN ALGORITHM

The machine learning technique KNN (K-Nearest Neighbors) is a technique often used for classification or linear regression predictive problems. The algorithm relies on feature similarity, or the process of checking how an input sample will resemble the training set [13]. Using the Euclidean distance formula, KNN calculates which features of the input sample are nearest to the training set. [Figure 4](#) is a graphical illustration of this process.

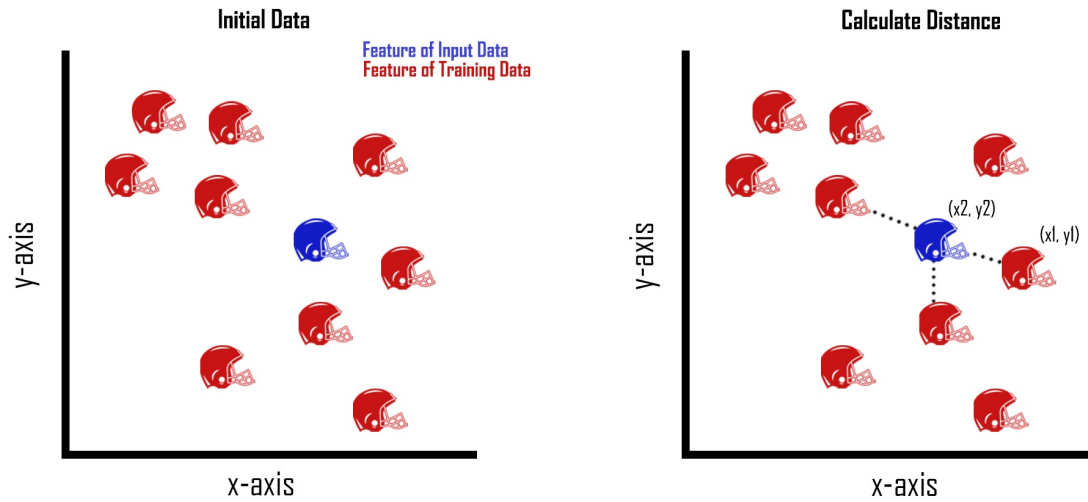


Figure 4: Euclidean distance formula

The Euclidean distance formula, [Equation 1](#), is used within KNN to calculate the distance between a feature of the input data, blue helmet, and k instances of that same feature within the training data, red helmets. Choosing the value of k is dependent on the training datasets used and the desired outcome. Compared to other machine learning algorithms, KNN has an advantage as it does not make assumptions about the data it uses, it is simple, highly effective, and it is versatile. However to its disadvantage, the KNN algorithm requires that we store all of the training dataset, which will often cause the algorithm to be computationally expensive and slow [13].

$$EuclideanDistance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (1)$$

Before our implementation of the KNN algorithm, we found that choosing a value of three to represent our k would best optimize the results of the program. In the python file `k-nearest`, we call and train the KNN algorithm with the NFL combine results from the last 18 years. Using the incoming rookies `2019_NFL_Combine` statistics as the input data, the program calculates the Euclidean distance between each feature of the input data and the accumulated combine results of previous years. This calculation yields the three closest veteran players that each NFL rookie performed the most similar to during the combine. While the results of our implementation are accurate and informative,

the speed disadvantage of the KNN algorithm becomes apparent within our program. Despite making adjustments to speed up the algorithm, such as only running KNN on the instances of the input data called for opposed to running it on all of the input data and filtering it afterward to get a tailored output, the program still runs slowly. This is due to the KNN algorithm requiring us to store the entire dataset before it runs computations on it. Despite our best efforts of optimization, our program still has a run speed of about five seconds.

Another disadvantage of the KNN algorithm and other machine learning techniques is their tendency to un-intentionally weight certain features of datasets due to size and scale differences. To account for this, we have to normalize the datasets we pass into the algorithm, creating an equal ‘weight’ for the Euclidean distance to calculate. In Python using the PANDAS library, or Python Data Analysis Library, allows us to do just that, and easily import the csv’s needed to execute the KNN algorithm.

From this library we use the read.csv function to read the NFL csv, comma-separated value, files into a data frame. A data frame is a two-dimensional tabular data structure with labeled axes, in which arithmetic operations align on both row and column labels. We use this data frame to isolate only our numeric features within the dataset, allowing us to quickly use only the features from the NFL combine that are measurables. As discussed earlier, in order for our output to be accurate the numeric features within each dataset must be normalized to create an equal weight for each combine measurement. Using the PANDAS data frame we achieved this by taking the numeric features and subtracting them from their mean. Once this was done we divide them by the standard deviation of the dataset, resulting in a very neat and evenly weighted data frame to pass into the KNN module.

5.3.5 IMPLEMENTATION

The core of this project is written in Python, which allows us to package all components needed to run our service easily. The use of “in_it” files creates a sensible directory structure that ensures a user can import both functions and data if they see them useful for personal applications. The versatility of python enabled us to create functions that execute the machine learning on our NFL datasets, along with a server that hosts these functions tied together through a

REST service.

REST, or Representational State Transfer, is a style of architecture used when creating web services. From the textbook, REST is described as being “based on stateless, client-server, cacheable communications protocol.” Applications using this architecture typically use HTTP protocol with basic methods like GET, PUT, and POST. These methods allow applications using the REST architecture to perform the four CRUD operations, which are to create, read, update, and delete resources. User created functions can be combined with these operations to create a cloud service that completes predefined tasks between a server and a client. However, to establish this direct link between a server and a client a YAML file is required [14].

A YAML file is written in a readable data serialization language that can be used to configure endpoints for a server. These endpoints are used to dictate what URLs are valid within the server, and what data, or data types, will be displayed for a particular path. YAML also allows specifications to be added to endpoints such as operations, HTML responses, and HTTP methods. Future additions to endpoints are simple to add without affecting current endpoints, which makes future updates straight forward with few worries about breaking past versions. This provides flexibility that is complemented by an easy to read syntax.

Our project uses REST architecture, in conjunction with a YAML file, to package our python modules so that they are able to run on a local machine. Within the YAML file, there are predefined paths a user can enter for different purposes. For our project, the path can be altered by changing ‘/results/’ to view fantasy football point predictions based on a given position.

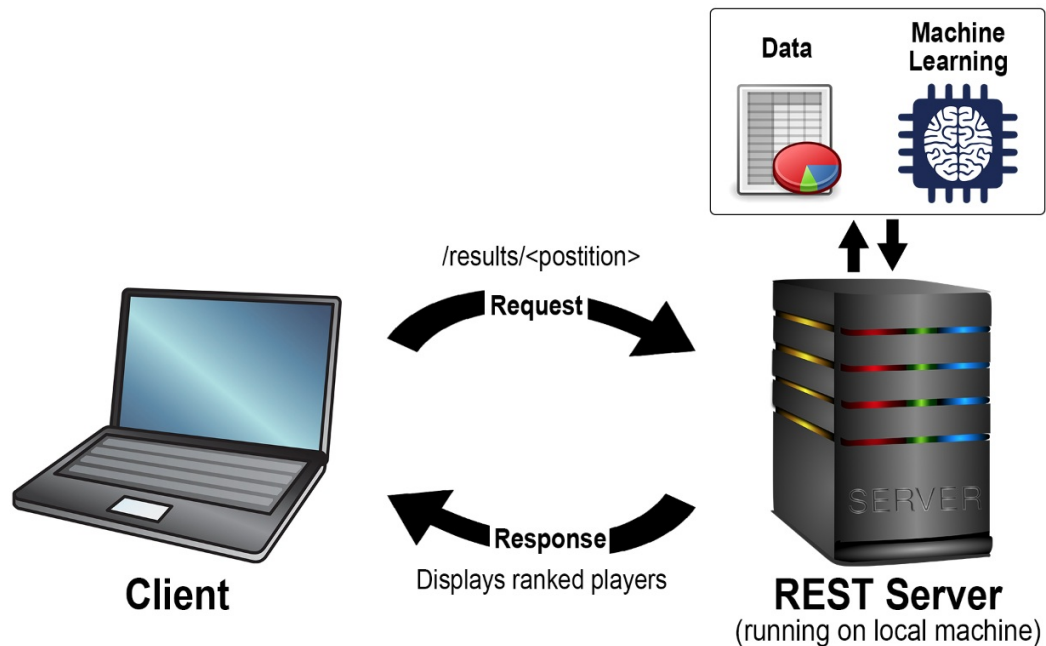


Figure 5: REST Service

Visualized in [Figure 5](#), when an offensive position is entered, the HTTP request is then sent to the REST server. From here, our service calls the python function `ff_prediction(position)` within the `nfl-analysis.py` file. First datasets are retrieved from a Google drive. Next, the machine learning is executed and the players are ranked through our comparative analysis and exported to a csv file. We then convert the final csv file to a html table, where it is printed to the users screen at the endpoint defined in the yaml file. While this is functional, we aren't able to ensure consistency across multiple development and release cycles, which highlights the advantage of using a software like Docker to standardize the environment of our project.

Docker is a tool that allows users to create, deploy and run applications within virtual containers. This technology is extremely useful when developing applications because it allows one to combine all of the needed components into one package [15]. Docker also ensures parity, which allows your images to run the same regardless of the server or laptop it is running on. This eliminates bugs that will occur on only certain computers, creating a standardized environment. We utilized Docker within our project by implementing compatibility through the use of a Dockerfile and a Makefile which makes it easy for users to build the

Docker container, run the server within the Docker container, and remove the Docker images. The inclusion of the Makefile simplifies the process for users looking to run our service and remove it once they are finished with it, ensuring no compatibility issues. The user only needs the Dockerfile and the Makefile to use our service to its fullest extent.

5.3.6 LIMITATIONS

As mentioned before, the NFL combine has six tests that measure speed, agility, and strength. We used that data in conjunction with each player's height and weight when finding their "nearest neighbors." However, not all players, rookies and current, completed every drill at the combine. This can result in inaccurate rankings, as KNN is only finding comparisons within one drill. Unfortunately, this gave players who skipped drills a boost in our rankings, as they tended to dominate in the drills they did choose to participate in. Another hindrance is that we had no way of knowing what NFL team each rookie will be drafted to, and whether or not they will be becoming a starter or second string player. The role the rookie has within his team is a huge factor in Fantasy Football, and not being able to take this into account creates a limitation on how accurate our model can be. Going forward, one way we could attempt to increase the accuracy of our predictions is to compensate for both of these factors, and to utilize rookies' college football statistics. While using college statistics was in our initial plans, we ultimately decided that the differences in conference and division could provide skewed data resulting in inaccurate predictions. This would not be impossible to account for, but it was not realistic to achieve this in the timeframe we were given.

5.3.7 CONCLUSION

As the number of Fantasy Football players continues to increase, we set out to answer the age-old question, who should be my top pick in this year's fantasy football draft? To help our fellow fantasy football players answer this question, we provide users of our service with a way to make educated draft picks on rookies that are likely to succeed in the NFL. To accomplish this goal, we used nearly 20 years worth of NFL combine and fantasy football data to create a service using REST, Python, Yaml, and Docker. Utilizing these technologies along with artificial intelligence, we were able to successfully rank rookies by

their position, accompanied by our prediction of average fantasy points per game.

We hope to test the accuracy of our service throughout the course of 2019 to see if this years rookies will hold up to our predictions. If we were to address some of our limitations, adding factors such as the role a rookie will play on his team, we would be able to continuously update our service and make adjustments. Analyzing these results would help us make it more accurate for next year's draft, and if we find our algorithm to be successful we have an avenue for deploying it into the real world. Providing users with educated sports picks through machine learning could revolutionize fantasy sports; and in a market that has nearly a \$4.5 billion impact across the sports industry, this information has the opportunity to produce serious revenue. We hope to continue building on top of this foundation, and refine our model with hopes of eventually tapping into this market.

5.3.8 SPECIFICATION

```
swagger: "2.0"
info:
  version: "0.0.1"
  title: "NFL FF Points"
  description: "A service that provides Fantasy Football point predictions for upcoming NFL rookies"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "NFL FF Prediction Service"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/cloudmesh/nfl-analysis/v1"
paths:
  /results/{position}:
    get:
      tags:
        - NFL
      parameters:
        - name: position
          in: path
          description: position
          type: string
          required: true
      operationId: nfl-analysis.ff_prediction
      produces:
        - text/html
      responses:
        "200":
          description: "FF data"
          schema:
            $ref: "#/definitions/NFL"
definitions:
  NFL:
    type: "string"
    properties:
      model:
        type: "string"
```

5.4 ANALYSIS OF SOCCER DATA WITH KMEANS

Jesus Badillo, Xandria McDowell, Ben Yeagley
jebadi@iu.edu, xmcdowel@iu.edu, byeagley@indiana.edu
Indiana University Bloomington,
hid: sp19-222-92
github: [🌸](#)
code: [🌸](#)

5.4.1 ABSTRACT

Data collected by wearbale devices used by the Indiana University Women's soccer team was used to classify the six unique offensive and defensive positions excluding the goaltender. Successful classification of positions from wearable data using machine learning techinques could provide talent evaluators and coaches with an extra tool to help increase on field productivity. The k-means algorithm was used in order to classifiy the data from wearble devices into six unique positions. This is the first step towards creating a workflow for using machine learning as a talent evaluation tool for soccer players. Once the ability to successfully classify data from wearable devices into unique positions additional layers of complexity can be added, like comparing professional athletes or collegiate athletes to prospects. A framework for the implemntation of classfiying soccer postions has been implemented with Scikit-learn libraries in python3. OpenAPI Specification was used in order to create a service that exploits the Scikit-learn libraries through an easy user interface.

5.4.2 INTRODUCTION

Can data gathered from athletic practices and games be used to create benchmarks for how an athlete playing a specific position should perform? Recent advancements in wearable technologies has allowed data from sensors to instantly improve how machines and people perform. These advancements are now being used in sports to make computations to player data to get future insight on how to improve the overall team. For example, the popular movie Money Ball focused on how analytics can be used to assemble a competitive professional baseball team. This type of analysis can show users what datapoints are shared by top players allowing coaches to pick their players accordingly. Soccer uses sports data to improve the physical attributes, such as stamina and pace, of each individual player since it is a more technical sport that cannot just

use quantitative data to make decisions. The field of sports analytics has been changed by machine learning techniques that have been adapted to use the hard numerical data received from edge-computing devices to improve teams.

5.4.2.1 Sport Analytics

The rapid innovation of technology has infiltrated many fields including sports analytics because it has created more data that can show coaches what athletes need to improve on. Edge computing is the technology that has led to the massive increase in more refined sports data because of its ability to provide a real-time numerical representation of what happened on the field. With this numerical data, actions can be taken to rapidly improve each player specifically on the areas where they need immediate attention. Edge computing is computing that is done near the source of the data, and it uses the cloud's ability to compute from a remote source to get instant data from a device. Edge-computing has allowed sensors to get more accurate data without disturbing the players which has given many sports a new way to get an advantage over their competitors.

5.4.2.2 Sensors in Sports

Sensors have become more prominent in sports due to recent advancements in biomedical engineering which have made sensory technology better to create more informative data. These new innovations in sensory technology can now measure data points such as “temperature, oxygen saturation levels, and heart rate (SpO2) through photo optic sensors in wearable rings and wrist devices”[16]. These new advancements are necessary because now, more than ever, “the differences between athletes are becoming more and more slight”[16] which has led coaches and trainers to look to data analytics to get an advantage. By putting small sensors into players' training equipment, we can detect important aspects of playing a sport such as muscle exertion, heart rate, and respiration. Sensors can provide immediate feedback to the athletes. For example, when attached to player attire, “accelerometers and conductive materials can measure posture and provide real-time feedback to athletes so they can perfect their form”[16].

Soccer has recently began to adopt the use of technology in many different ways such as the Adidas MiCoach ball, which is a regular sized soccer ball with a built

in sensor that can detect the power, spin, and accuracy of a player's kick. This ball allows players and coaches to see how the athlete shoots the ball through an iOS or Android device to improve the player's shooting form. This type of sensor is mainly used for beginners, professionals tend to focus on the more physical side of data collection such as wearable sensors that track physical stats. The Catapult Sports Playertek is a compression vest with a built-in sensor that has an accelerometer, magnetometer, and a GPS that can track athletes' performance during practices. The IU Women's soccer team uses sensors like the Playertek to get athlete data because it allows the coaches to know what physical attributes to track for each player when they are training or playing.

5.4.3 SOCCER DATASET

Practice and game data from the IU Women's Soccer team was used in this analysis. The results of this analysis aim to leverage 'sensed' athlete data to show coaches and players if players are performing optimally. The Women's team data has 34 individual columns such as sprints, accelerations in varying ranges, and heart rate that are weighted based on how much time the individual played. Some data points, such as sprints, can be used to improve the athlete in a certain aspect of their sport, in this case by increasing the number of sprints they have during a game. Using data in this way can help the player improve their speed, but it will do very little to improve their overall ability. Instead the Kmeans Clustering method will show the entire team how they compare to each other and the ideal player of their respected position, so that the entire team can improve. By using Kmeans on the data each player can improve how they play, not just how fast they can run. The Kmeans algorithm is optimal for this dataset because it uses eight different datapoints to determine how well each player is playing their position compared to how they should be. This method provides a literal target from which players can see which areas they specifically need improvement on to reach the centroid, or the ideal player for that position.

Clustered data can be used to compare players to other players who play the same position in different divisions, levels, etc. This method can sometimes be inefficient because what is *good* can vary a lot in soccer, due to it being based on skill rather than physical attributes. For example, some defenders do not run very much at all, but instead rely on their ability to read player movements to stop the opposing players from getting past them. This would be a great data point

that could likely be used to create more accurate models that can help the players distinguish themselves from the other positions in terms of how they should be performing. The problem is that sensors are limited to numerical data, so this type of ability cannot be measured. To avoid any of these limitations, the Women's Soccer team data has many of the same data points measured in different ranges. For example, the *Number of Accelerations* category has eight different ranges which are either a deceleration or an acceleration. This way of splitting data could transcend the inability to measure the *intangibles* because it measures the same data points at different instances which tend to vary by position.

The Women's team data was obtained from the 2018 season where they faced off with many other Big 10 schools. There were eleven datasets which each contained a row for each of the players where you could see their names and all of their data points for that particular game. Since each dataset had only 23 data points we decided to join all of the datasets into one big dataset where the players' names were removed and replaced by the player's position into a column called *Class*. The original datasets that were given to us did not contain how many minutes each player played, which could alter how the kmeans algorithm clusters the players. The algorithm would likely cluster the players wrongly if the *Minutes Played* column was not added because it would give the players who are substitutes higher values without taking into account the amount of time that they played during a game. In soccer, the amount of time you play can especially effect many of your stats because the sport is largely based on endurance with a few short bursts of sprints. The substitute players who had no game time were removed from the dataset because they could alter how the algorithm clusters a good stat vs. a bad stat. Once the extraneous data points were removed there were only 170 rows left which all had a label of either Mid-fielder, Forward, or Defender. The label was used as the prediction variable to give insight as to where the kmeans-computed clusters should be centered. The Kmeans algorithm uses the labels to predict the outcome of each of the 170 rows' stats against the kmeans-computed stats.

5.4.4 ALGORITHM DISCUSSION

5.4.4.1 K-means

K-Means is an unsupervised machine learning algorithm that takes an unlabeled dataset and finds groups within data without defined categories. Unsupervised is learning from data that are unlabeled meaning you don't know what the values of the output data might be. The idea is to look at the average or mean values that can be clustered and how they can be related to the k groups. K is also known as centroids which is a data point at the center of each cluster. Each cluster is placed in a certain position because depending on how close the clusters are to each other will change the results. Which is why the placement of the cluster are important because the layout of centroids will determine the results. In K-Means the centroids are randomly selected and then used as the cluster of each point. After the centroid is placed then all the data point is split up into the correct centroids based on the data point that they are. Now that the data points have a home the next step is re-calculation of k to optimize the position of the centroid. This process of optimization continues to happen until centroids have stabilized or a certain number of iterations have occurred. When using this algorithm, the hope is that the results can explain why common parameter values are in the same group. This algorithm takes numerical values or data points that describe a coordinated value within a data set and cluster them according to the given k. For example, K- Means is beneficial for this project because it's able to focus on a particular data set for the women's soccer team at Indiana University (IU) and compare that to users data. K-Means coordinates can represent and describe many different things such as elections, medical test, sports teams, or wine data. Typical you use K-means on a dataset to get the model of the data, this is used to see the behaviors and analyze the patterns the data set. When using clustering methods variance becomes an important topic trying to understand the math behind this algorithm. Variance is a measurement of how far data values are from the mean. By look at the variance equation, it shows that it is a measure of the square difference between the data points and the mean value and then the average. When using the K-means algorithm one of the key points is to measure the closeness of the data to its average which led into cluster variance which divides the data into clusters, each will have its own average. While there are many different machine learning algorithms K-Means worked best because it takes a unsupervised data set which was need in this case. While K-Means gives an optimal cluster model, spectral cluster is better. The K-means objective function is given in [Equation 2](#).

$$\sum_{j=1}^k \sum_{i=1}^n \|x_i^{(j)} - c_j\|^2 \quad (2)$$

5.4.4.2 Spectral Clustering

Before landing on K-means to be the machine learning algorithm, spectral clustering was the set machine learning algorithm for finding out how close people were playing against Indiana University womens soccer team. A commonly used algorithm for classification that has become popular in recent years is spectral clustering [17]. This algorithm can be effectively solved by using standard linear algebra methods and, software. For the most part, it is known to outperform K-means which is what was seen with this data set. Spectral clustering uses the data points as nodes of a graph, which are then treated as partitioning points and mapped to a low-dimensional space making it easier to form clusters. Spectral clustering follows an approach where if the points are extremely close to each other they are put into the same cluster, even if the distance between them is within a two point range if they are not connected they are not clustered. While k-means points can be within the same range and still fall within the same group because it is measured by the distance between the data. This data set has many points where they are considered within range of each other which is why spectral clustering gave better results. In spectral clustering the data is connected within a way that leads to the data points falling within the same cluster. The problem arises when using spectral clustering because the project aims to show people how they played compared to Indiana University soccer players but when trying to show their data and the soccer players data on the same plot issues arise. This happens because you can not get the labels using spectral clusters so the use would not know where they where. spectral equation below.

$$\left(\frac{1}{2}\right) \sum_{i,j=1}^n w_{ij}(f_i - f_j)^2$$

5.4.4.3 Dimensional Reduction

Dimensional reduction was something that was needed for this data set because there were many points within one players statistics. Having multiple points, was too much for K-means to give explainable results. Dimension reduction allowed

the data set to be reduced to a certain number of random variables by obtaining a set of key variables. Looking at dimension reduction from the math perspective there are the non-linear and linear methods. Typically, the Linear method is used because it is easier to implement. In the linear calculation results in each variable being apart of a linear combination of the initial variables such that $k \geq p$ see [Equation 3](#).

$$S_i = w_i x_i + \dots w_i p x p \text{ for } I = 1, \dots, k \quad (3)$$

where

$$S = Wx$$

Wx represents the weighted matrix in linear conversion. $Ap * k$ is the same as $x = As$ which are new variables or s identified as hidden or latent variables according to [18]. When using the matrix X in terms of $n \times p$ [Equation 4](#) is applied.

$$S_{ij} = w_{1j} W_{1j} + \dots w_{1p} X_{pJ} \text{ for } I = 1, \dots, k \text{ and } j = 1, \dots, n \quad (4)$$

According to [18](#) this is the easiest linear technique to use.

5.4.5 RESULTS

Normally a clustering model cannot be scored for its accuracy; there are no true labels in unsupervised learning to compare with the clustering models predicted labels [19]. However, in our case each of the fitness data points had the player name associated with it, so we were able to replace each name in the dataset with their position on the field (the predicted value). Converting these positions to numeric representations gave us a list of true labels to use for scoring. We found each players position by taking the position listed on the team roster page. Of course, some players may play in different positions throughout the season, so some of the labels may be incorrect. Our labelling process was the best we could do with the information available. We focused on three evaluations for our model: completeness, homogeneity, and v-measure. Completeness is a measure of how well each class was clustered together [20], while homogeneity scores the clusters based on what degree in contains only one class [21]. The harmonic mean of the completeness and homogeneity scores gives the v-measure score

[22] [23]. Each is considered perfect if their score is 1.0. Conversely, 0.0 is the worst possible score. The results of our clustering model can be seen in [Table 1](#). had a completeness score of .260, homogeneity score of .227, and a v-measure score of .242. This means our model was more effective at clustering the classes together than having each cluster contain purely one class. Overall, the model scored poorly. Its scatter plot [\[24\]](#):

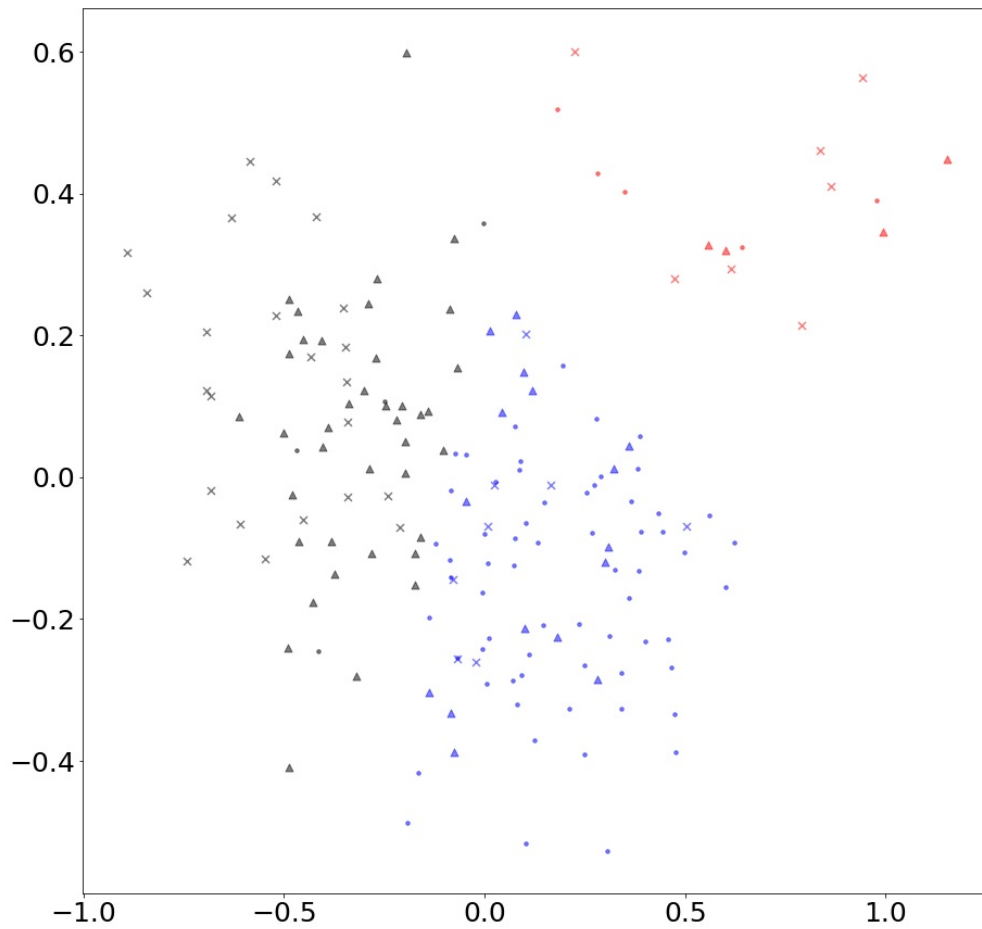


Figure 6: Scatter plot of clustered results

Looking at the scatter plot showing the clusters and the data points true labels, we can see that the poor score can be mainly attributed to the general failure to classify forwards (x's) correctly. The defenders (dots) and midfielders (triangles) have fairly defined clusters themselves, and viewing the scatter plot without the any forward data points makes it even more apparent.

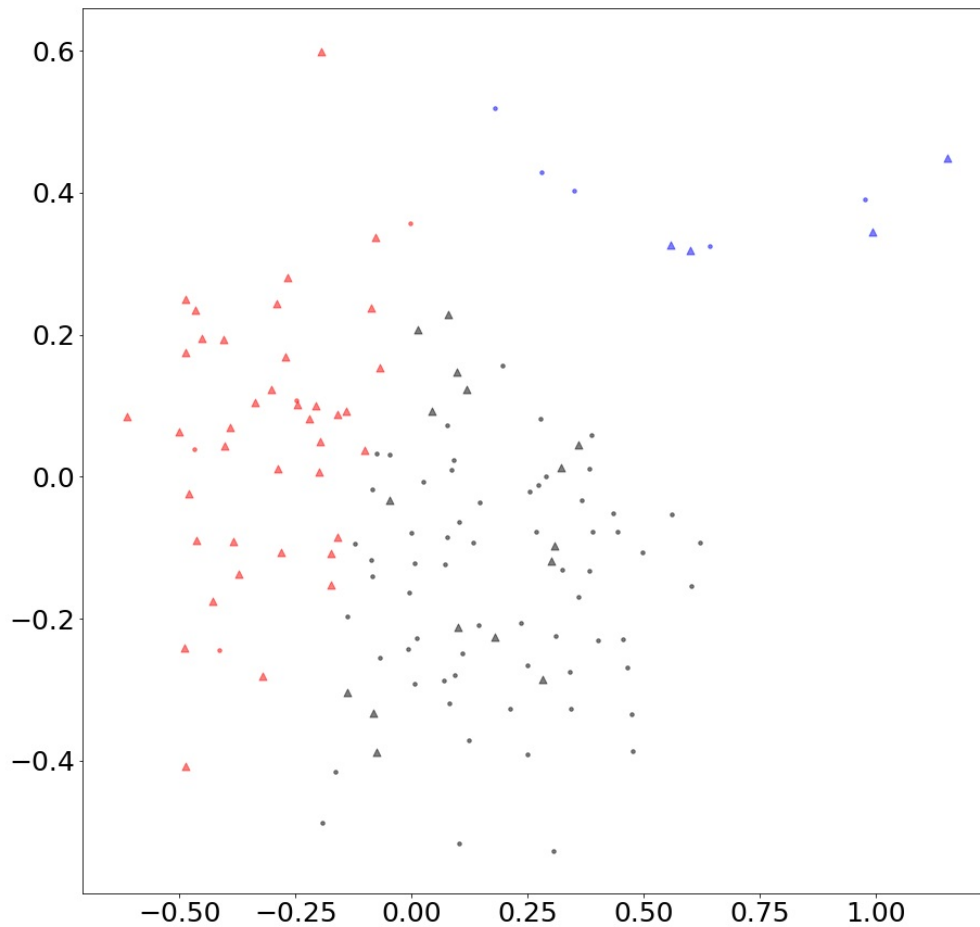


Figure 7: Scatter plot clustered results without forwards

The black cluster was split almost 50/50 between forwards and midfielders despite them being decently separated groups, mainly because the model identified the outliers of the dataset as the red cluster. It's possible that if the dataset was cleaned up by removing the outliers before clustering, the model would've separated the classes much more accurately. Outliers as defined by our boxplot are shown here [\[25\]](#) [\[26\]](#):

A Boxplot of the data after normalization is given in [Figure 8](#).

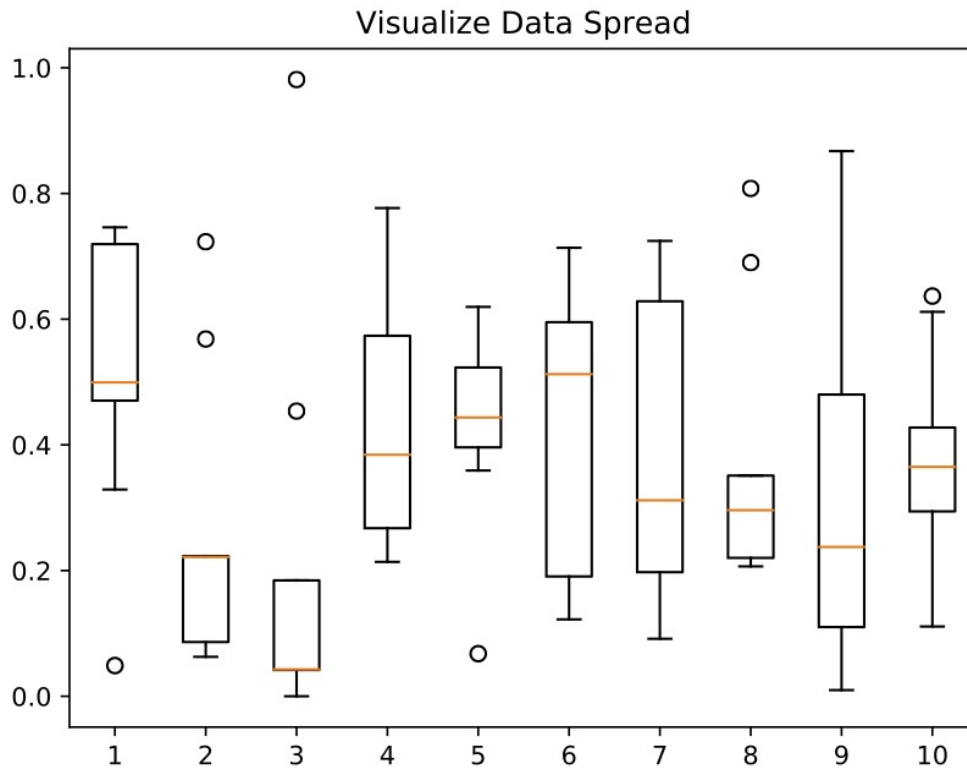


Figure 8: Boxplot after normalization

Our method for removing the outliers could have involved calculating a Z-Score for each data point, which gives the amount of standard deviations a point is from the mean of the dataset, and then getting rid of points with a Z-Score above a certain amount of standard deviations [27]. However, it's important to determine if the points in that top-right cluster would even qualify as true outliers or if they are just points the clustering model is unable to predict. We might need to investigate those points in the original dataset to see if there is a discernable pattern to them because theoretically, the data shouldn't have any anomalies unless the measuring device malfunctioned. It could even be a specific game that led to those points being separated; the game could have been much more intense than normal, leading to higher numbers for each player that participated. Disregarding them just to improve the models accuracy may not be the right choice.

Table 1: Results of our k-means analysis
completeness homogeneity v-score

5.4.6 ANALYSIS

A necessary question to ask is: is the low v-measure score of the model due to possible errors in the dataset like those discussed above, an incorrect choice of machine learning method, a general unpredictability of player position with the type of data given, or some other reason? To examine the second possibility (wrong method), we split up the data into training and testing using `train_test_split` from `sklearn`, and then we added a variety of other classification techniques to the project code [28]. The first technique was Multinomial Logistic Regression, a method that uses a linear combination of the predictor variables to model the dependent variable (position) [29]. The model had an accuracy score of .63 for our training data and .60 for our test data. The next technique we employed was a Decision Tree Classifier, which handles multi-class predictions naturally. Decision trees use the observed variables from the dataset to make choices (branches of the tree), leading to a conclusion about the predicted value (leaves of the tree) [30]. We gave our decision tree a max depth of eight as its only parameter. It had an accuracy score of .80 for the training data and .65 for the test data. Next, our K-Nearest Neighbors Classification model functions by taking a vote of the points around the point to be classified and assigning the point whichever class that had the most votes [31]. The number of neighbors used can be specified in the function call, and we found eight neighbors worked best. Our k-NN model had an accuracy score of .76 for the training data and .74 for the test data. We have a Linear Discriminant Analysis classifier as well, with the number of components set at two [32] [33]. Its accuracy score was .69 for the training data and .60 for the test data. We also had a Gaussian Naïve Bayes classifier despite the fact that our data doesn't have features that can be considered independent, but it still serves the purpose of comparison to the clustering model [34] [35] [36]. Its accuracy score was .73 for the training data and .63 for the test data. Finally, we had an SVM classifier added to our project code, another method better suited for just two classes but used anyway [37] [38]. Our SVM model had an accuracy score of .58 for the training data and .63 for the test data. Overall, the highest scoring model for the test data was K-Nearest Neighbors, yet its .74 score for the test data still was not very high. It is difficult to make a direct comparison between the k-NN model and our original clustering model for multiple reasons. Their scores don't really represent the

same idea, as accuracy isn't well defined when it comes to clustering. Clustering is more generic than k-NN classification too, because its goal isn't to predict classes for each individual data point. Instead, the goal of clustering is to group the data into distinct sets and see how these sets align with real-world observation and truth. Failed clustering (or poor clustering in our case) means our dataset couldn't fully differentiate positions with the variables measured. An aspect sorely needed to further separate the classes would be GPS data to find the average location on the field for each player, or a heat map of their movement throughout the game.

Table 2: Machine Learning Table of Reults

Machine Learning Method	Accuracy (training)	Accuracy (testing)
Logistic Regression	.63	.60
Decision Tree	.80	.65
K-Nearest Neighbors	.76	.74
Linear Discriminant Analysis	.69	.60
Gaussian Naive Bayes	.73	.63
SVM	.58	.63

5.4.7 SPECIFICATION

```

swagger: "2.0"
info:
  version: "0.0.1"
  title: "Clustering Athlete Data"
  description: "A simple service that gets game data from a cloud storage service to show athletic performance"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "Cloudmesh REST Service with AI"
  license:
    name: "Xandria McDowell, Ben Yeagley, and Jesus Badillo"

host: "localhost:8080"
basePath: "/project19"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /data/output/<output>:
    get:
      tags:
        - DATA
      operationId: functions.download
      description: "Downloads the dataset from the URL"
      produces:
        - "application/json"
      responses:
        "200":

```



```
      description: "Data info"
      schema: {}

/data/kmeans/<datafile>:
  get:
    tags:
      - KMEANS_PLOT
    operationId: functions.kmeans_plot
    description: "Filter the dataset, normalize, and perform Kmeans"
    produces:
      - "application/json"
    responses:
      "200":
        description: "Data info"
        schema: {}

/data/user/<datafile>:
  get:
    tags:
      - USER_DATA
    operationId: functions.user_plot
    description: "user data"
    produces:
      - "application/json"
    responses:
      "200":
        description: "Data info"
        schema: {}

/data/boxplot/<filename>:
  get:
    tags:
      - USER_DATA
    operationId: functions.boxplot
    description: "user data"
    produces:
      - "application/png"
    responses:
      "200":
        description: "fig info"
        schema: {}
```

5.5 TETRIS SCORE ANALYSIS SERVER

Zach Levy
zwlevy@iu.edu
Indiana University
hid: sp19-222-97
github: 
code: 

Keywords: Tetris, linear regression, decision tree, gini coefficient, correlation coefficient, determination coefficient, Python

5.5.1 ABSTRACT

Tetris is a popular video game played by many professional and casual players

worldwide. Using data recorded from players in from the Tetris World Championship and analyzing the qualities between various factors such as playtime, score, level, and more, the aim is to possibly extract meaningful relationships and evaluate them using standard regression models. The usage of a linear regression model and decision trees are used to explain if any relationships between these exist, and whether said relationships are meaningful. From the extrapolated data, the aim is to use statistical coefficients, such as the correlation coefficient and Gini coefficient from the models to do this.

5.5.2 INTRODUCTION

Released originally on the Electronika 60 in the USSR on June 6, 1984, Tetris has come from a small programming project from Soviet game designer Alexey Pajitnov to one of the most world-renowned puzzle video games of all time. Although deviations have spawned from the game, the primary rules of Tetris are rather simple. Blocks, arranged into sequences of four called tetrominoes, move vertically from top to bottom. The player can reposition and reorient the blocks, and the goal is to get a set of blocks all on a bottom line, in which the blocks will disappear and add to the score. After many of these occur, the game will speed up, making it harder to arrange blocks correctly. The game is over when the blocks reach to top of the screen. Ported to a wide variety of platforms, Tetris has become a part of popular video game culture and has sold millions of copies. Psychological studies done on people who play the game have lead to the discovery that people who play for prolonged periods of time may see mild remote memories of the images of the tetrominoes moving [39]. Additionally, some studies suggest that Tetris may help reduce mental stress and help individuals cope with post-traumatic stress disorder [40].

5.5.3 DESIGN

The primary design for this project comes in two stages:

1. The creation of a regression model that most suitable fits the dataset. This should most likely be a linear regression model or a data tree.
2. The creation of a RESTful server that could retrieve the raw data sets, perform analysis, and return the data to a requested user.

The first part of the project required me to program a way to retrieve the transcribed data. Due to the data on the website being only images of spreadsheets rather than the spreadsheets themselves, this was necessary [41]. By importing the Python libraries csv and requests, the program can retrieve this data from a Google Drive downloadable link as TETRIS_DOWNLOAD.csv.

By using a decision tree and a linear regression model, we can better understand which relationships between variables are meaningful and which are not. The analysis of the data was first done with the usage of a linear regression model. For this, I took the features to be year, rounds won, and composite score. The label in question was the resulting ranking of players. The second model used was a data tree classifier. This would help with the previous linear regression model, since we can observe how the Gini coefficients - numbers that represent the inequality distribution of data tree nodes that help color the importance of certain variables - are resulted from the usage of a decision tree. Gini coefficients are assigned to each branch and leaf of a decision tree and evaluate the decision inequality when evaluating whether an entry is one or another. If the nodes expectedly go to zero, it is a positive indicator that the model fits well.

5.5.4 ARCHITECTURE

There are two primary structures within this experiment. The first is the linear regression model module. This module can retrieve data from a Google Drive link and downloads it locally. Using a Representational state transfer (REST) service, it can send over results of analysis on the data set. It does this by running HTTP requests to the server that contains an API to manage the endpoints of communication and send HTTP responses back to a client. The scipy library makes it very simple to perform linear regression. It gives three distinct graphs - divided into subplots on a single image - and uses matplotlib to make graphs that are easily human-readable.

1. Rank vs. Year
2. Rank vs. Score
3. Rank vs. Rounds Won

As for the data tree module it will simply read the data the same way as the linear regression module and create a .dot chart from the results, which can be

saved remotely just like the graphs within the linear regression module. The Python library graphviz makes for easy use of creating the decision tree to reflect the data used within this experiment, while libraries such as csv, io, os and requests allow the modules to download and read the extrapolated data with ease.

5.5.5 DATASET

The data used within this project was primarily extrapolated from data of the winner results from the Tetris World Championships. Due to the fact that previous years represent data as only image screenshots of spreadsheets, the data had to be manually transcribed into other comma-separated value spreadsheets. The data was extrapolated from the spreadsheets into a Google Spreadsheet and saved into a comma-separated value file. Extrapolated entries included a player's name, composite score, rank, rounds won, and the year the game took place.

5.5.6 RESULTS

To put the results into better interpretation, there are various mathematical ideas that can help explain the actual importance or usefulness of relationships shown. Most importantly when it comes to our experiment is the following for linear regression models and data trees.

- **Correlation Coefficient.** Denoted by R , a correlation coefficient measures the strength of linear relationships of a scatter plot. If R is close to -1, then a strong negative linear relationship exists. Likewise a +1 indicates a strong positive linear relationship. However, if R is closer to zero, then it indicates that a linear relationship isn't very present [42].

R is calculated as follows:
$$R = \frac{\sum((x-\bar{x})(y-\bar{y}))}{\sum(x-\bar{x})^2 \sum(y-\bar{y})^2}$$

- **Determination Coefficient.** Denoted by R^2 , the coefficient of determination how much variation of data has been explained by the model. The closer this value is to one, the better. An R^2 that is less than zero indicates that it is a poorly fit model, and the reason this is such is that we have used bad constraints or a made poor choice in model [43].

R^2 is calculated as follows: $R^2 = 1 - \frac{SS_{Res}}{SS_{Total}}$

The following table shows various points of data found based on the graphs.

Table 3: Linear Regressor

Graph Type	Slope	Intercept	R Value	R Squared	P Value	Standard Error
Rank vs. Year	0.480	-950.059	0.100	0.010	0.129	0.129
Rank vs. Score	-8.474	24.648	-0.492	0.242	1.495	0.001
Rank vs. Rounds Won	-6.928	29.911	-0.896	0.802	4.607	0.226

With the correlation coefficients of the graph, most of the relationships we found did not have much importance. As we can tell, the R value of Rank vs. Year was closer to zero, indicating that there is not any strength in a linear relationship with the two statistics. In the Rank vs. Score graph it indicates a middle-level of strength with $R = -0.492$. This suggests a negative linear relationship that isn't so strong, but it is expected since ranks are actually lower in number, meaning 1st place is actually the highest, but 32nd place has a higher number. That is why when discussing them we look at the R^2 value instead, which is positive. This carries over when discussing with Rank vs. Rounds Won, as $R = -0.896$. As expected, a person with a higher rank has won more rounds, so this is as expected.

Moving on, we can discuss the other model used, the decision tree. The decision tree helps us understand the actual meaning of any relationships between variables and whether or not they are of any real importance. The primary way that we evaluate this is by using the following:

- **Gini Coefficient.** Denoted by its name, *Gini*, Gini coefficients are used with decision tree algorithms to measure the effectiveness of branches in a data tree. In a good model, we expect these to be closer to zero towards the leaves of a data tree [44].

Gini is calculated as follows: $Gini = 1 - \sum_{k=1}^n p^2k$

Here is a graph that represents the decision tree found within the experiment.

Figure 9: A more readable image can be found at <https://i.imgur.com/mLXv6sU.png>

At the root node the Gini coefficient was 0.238 for a sample size of 232 samples. Moving towards the true side of the graph, we found that Gini coefficients gradually become closer to zero, with last branching node to only contain a Gini coefficient of 0.037. It should be noted that the data tree node is very lopsided. This could possibly indicate that the computational cost per each decision is rather high [44].

5.5.7 CONCLUSION

The deductions made from both of these experiments must be approached differently, since they are separate models. According to the chart, we can see that the R squared value for Rank vs. Year became 0.010, for Rank vs. Score it was 0.242, and for Rank vs. Rounds Won it was 0.803. Table 3. This should come as no real surprise, since we expect that any person who won more rounds has a higher rank than the others who did not win. We know that from the rather low values comparatively for Score and Year that there does not seem to be a strong linear relationship between Rank vs. Year and Rank vs. Score.

As for the data tree model, it was discovered that the Gini coefficients for the model were indeed closer to zero but that the data tree itself was heavily lopsided Figure 9. This suggests that although there may be vague linear relationship when classifying the elements of the data tree, it has high cost for its usage, suggesting it may not fit to be a very useful model. The Gini coefficient at the first branch was 0.238, and the Gini coefficient at the last one was 0.37. Pruning of unnecessary leaves, mainly any branches whose Gini coefficients are already zero, could help this matter [44].

5.5.8 SPECIFICATION

```
swagger: "2.0"
info:
  version: "1.0.0"
  description: "Analyzes Tetris games from the Tetris World Championships"
  termsOfService: "http://swagger.io/terms"
  contact:
    name: "Tetris Score Analyzer"
    email: "zwlevy@iu.edu"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0"
host: "localhost:8080"
```

```
basePath: "/cloudmesh/ai/tetris"
schemes:
- "http"
consumes:
- "application/json"
produces:
- "application/json"
paths:
  /lin_reg/download/<output>:
    get:
      operationId: project-code.linear_regressor.download
      produces:
      - "application/json"
      responses:
        "200":
          description: "OK"
          schema: {}



  /lin_reg/read:
    get:
      operationId: project-code.linear_regressor.read_csv
      produces:
      - "application/txt"
      responses:
        "200":
          description: "OK"
          schema: {}

  /lin_reg/analyze:
    get:
      operationId: project-code.linear_regressor.linear_regression
      produces:
      - "application/png"
      responses:
        "200":
          description: "OK"
          schema: {}

  /dt/download/<output>:
    get:
      operationId: project-code.data_tree.download
      produces:
      - "application/json"
      responses:
        "200":
          description: "OK"
          schema: {}

  /dt/analyze:
    get:
      operationId: project-code.data_tree.data_tree
      produces:
      - "application/png"
      responses:
        "200":
          description: "OK"
          schema: {}
```

5.6 POLITICAL BIAS AND VOTING TRENDS

Mercedes Olson Jarod Saxberg
mercolso@iu.edu jsaxberg@iu.edu
Indiana University
hid: sp19-222-96 sp19-222-100
github: 
code: 

5.6.1 ABSTRACT

Sklearn's K-Nearest Neighbors algorithm was used to train a classifier which determines if a presidential candidate would win or lose an election based on their support or opposition to ten issues-healthcare, military, education, taxing the wealthy/businesses, women's rights, globalism, gun rights, infrastructure, minority rights, and immigration. Correlations were attempted to be made between the algorithm results and actual presidential election results.

5.6.2 INTRODUCTION

K-Nearest Neighbors (KNN) is a basic classification algorithm. It is non-parametric, so it does not make any underlying assumptions about the distribution of data. A workflow for a typical KNN algorithm is as follows:

- Store training samples in an array
- Calculate Euclidean distance between the test data point and all training data points
- Make a set of the smallest distances obtained
- Return the majority label from the set

The algorithm finds the training data point that is most similar to the testing data point. Once found, it determines the classification of the test data from the classification from the nearest training data. This was used with a dataset containing ten features that will determine one classification. The features and classification are aligned so that the algorithm will attempt to predict if a candidate would win an election based on their support or opposition of the ten features.

5.6.3 REQUIREMENTS

In order to run this project, docker must be installed and working. Once complete, a docker account must be created to use docker. After creating an account, download the Makefile and Dockerfile to a new directory. To build the docker container, run the command `make docker-all`. The Docker container will download the necessary python libraries found within requirements.txt and then start the REST service on its own. The endpoints will then be available to access

through a web browser or curl client.

5.6.4 DESIGN

The design of this project is quite simple for people to use. From a high-level perspective, all a user needs to do is build a docker container and proceed to the endpoints defined. To break this down, the project is split up into three parts: the python code containing the machine learning logic, the REST API connecting the python code to a website that can be navigated to, and a docker container that houses all of the files necessary for the project. Below each part will be discussed in more detail.

5.6.4.1 Python

This project makes use of sklearn's K-Nearest Neighbors algorithm [45] to perform machine learning on the dataset. Sklearn allows easy use of many machine learning algorithms. Using their K-Nearest Algorithm is as simple as using the lines:

```
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)

pred = classifier.predict(x_test)
```

With the use of that code, everything within the scope of this project can be completed. When users pass their own hypothetical candidate with ten arguments, the arguments are added to an array that is then passed through the predict function and the classifier determines if the candidate would win or lose based on the given arguments.

5.6.4.2 REST Service

The REST Service allows simple connection between backend python code and a frontend website/API where users can pass data in and receive it back as a json formatted object. People can easily access the classifier through the API without the need to build it and run it on their own. Endpoints are developer-designed urls that can be navigated to so users can run the classifier and other useful features. The basepath is what prefixes all other endpoints; it can be thought of as a top level directory and all of the endpoints are files within the directory

structure. The specification in YAML that showcases the basepath and endpoints used can be found in [Section 5.6.10](#). For readability and length the endpoint that allows a user to perform a custom classification used abbreviations. To clarify the abbreviations the end point is provided next with the abbreviations spelled out.

```
/run/custom/<neighbors>/<healthcare>/<military>/<education>/<tax
wealthy>/<womens rights>/<globalism>/<gun
rights>/<infrastructure>/<minority rights>/<immigration>
```

5.6.4.3 Docker

Docker containers were used in this project to create an image that held all of the requirements to run this project. Containers are good to use because it eliminates the requirement of downloading all the necessary files to a personal computer and allows them to be easily removed by deleting the image. It also allows one common operating system—in this case, Ubuntu—to be used across all projects, thus eliminating the troubles that can arise from a mixture of operating systems.

5.6.5 DATASET

The dataset for this project had to be built from scratch as there were not any easily accessible datasets with the features desired. In order to create the dataset, google sheets was used to input data, and then the spreadsheet was downloaded as a csv file. After converting the dataset to a csv file, it was then uploaded to a website that can be accessed through the python requests library. The information obtained from the Federal Election Commissions [\[46\]](#) website included the top 4 presidential candidates based off the total votes and percentage of votes received from the years 1988 to 2016 with their respective parties. After selecting the candidates ten topics were chosen. The topics include healthcare, military, education, taxing the wealthy, women’s rights, globalism, gun rights, infrastructure, minority rights, and immigration. Each presidential candidate received a 1 or 0 if they supported or opposed a topic, based on research from the website OnTheIssues [\[47\]](#) and wikipedia pages about the respective campaigns. An example would be if the candidate had a 0 in the 3rd element and 1 in the 5th element, their actions in the past show opposition to education and support for women’s rights. One final thing to take into consideration when making this data is personal bias and the limitations of records can affect the results.

5.6.6 RESULTS

It is difficult to draw conclusive results from this machine learning application primarily because of the limited size of the dataset. There are only 32 entries in the dataset, so the algorithm does not have a lot of data to learn from. Running the algorithm multiple times returns multiple different values for the precision, recall, and f1-score. It is primarily dependent on which entries are chosen for the testing and training datasets. The algorithm could be improved with additional candidates dating further back from 1988, but the data becomes more sparse pre 1988.

Using the KNN algorithm from sklearn requires an argument for `n_neighbors` which will affect the accuracy of the algorithm. A simple loop was developed to determine the best `n_neighbor` argument for each iteration of testing and training datasets. The loop determined the error rate that each possible `n_neighbor` argument produced, and creates a graph plotting the `n_neighbor` argument against the error rate. [Figure 10](#) is a sample graph generated by one iteration of the testing and training datasets.

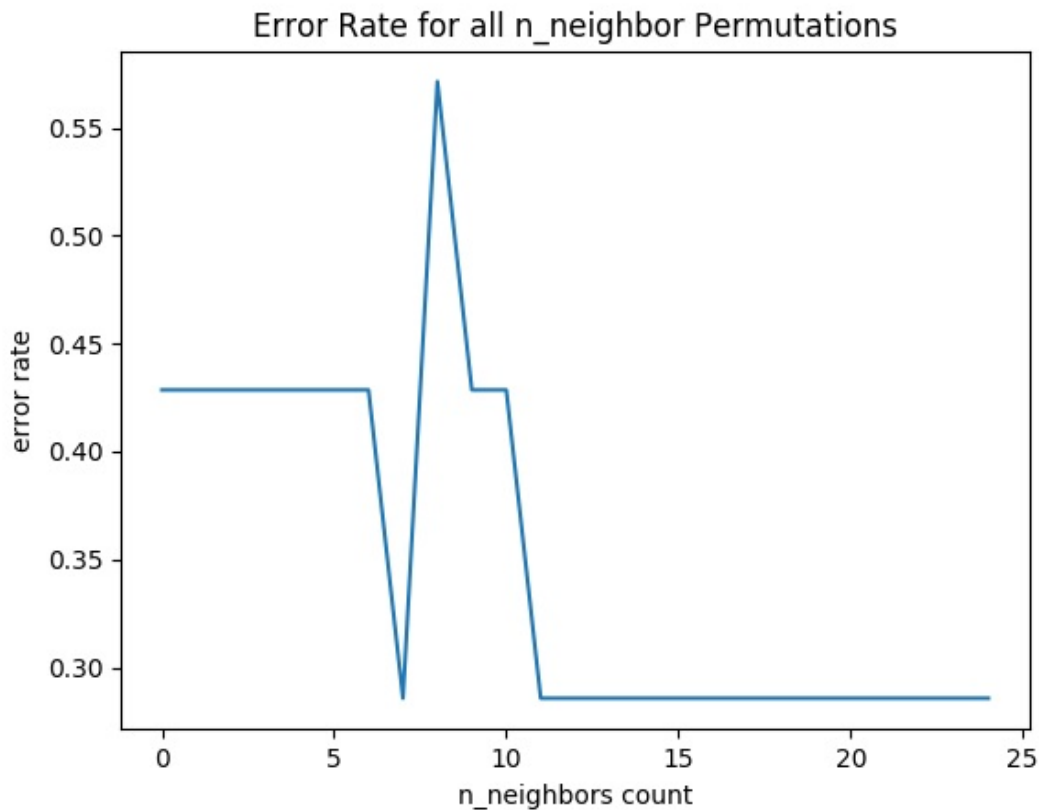


Figure 10: Error Rate vs. Number of Neighbors

5.6.7 DISCUSSION

Machine learning and artificial intelligence are both hot topics having grown exponentially in popularity during the past ten years—many people are racing to find new ways to apply these algorithms to real-world scenarios. An area of interest for this project is how machine learning applies to politics, specifically data-driven politics. The application has been difficult, however. As shown in [48], it is difficult to draw correlations between analysis results and electoral outcomes. Prior research has been done connecting measures of public opinion measured from polls with sentiment measured from text as shown in [49], and it shows promise having correlations as high as 80%. The research shows how sentiment from text could be used as a substitute for typical polling and predict movement in the polls; however, using Twitter as the source for text nowadays would create issues as the amount of bots on Twitter has grown exponentially since this research was done in 2011. There are other more reliable ways to obtain data for use with machine learning that are not affected in the ways that

social media is. Combining typical polling with machine learning could be a path forward that might produce positive results, and help politicians run their campaigns. Machine learning can also help voters make more informed decisions about the candidates available for them to vote on and the stances they hold on certain issues. There are many different applications of machine learning to political data, it is just a matter of finding the best approach and refining the algorithm until error rates are low.

5.6.8 CONCLUSION

As stated in the results section, it is difficult to draw conclusions to the accuracy of this application of machine learning because there is only 32 entries total in the dataset. The precision, recall, and f1-scores varied each time the algorithm ran. Likewise, the best `n_neighbors` argument varied with each testing and training dataset used.

To improve the results, more candidates from past elections could be added—it would require more in depth research on the topics they support as the information is more difficult to access. It would also improve over time as the amount of candidates increases and their campaigns would be better documented. The predictions will gradually get more accurate as time goes on and more data is collected.

5.6.9 WORK BREAKDOWN

Going through candidates and creating the dataset was an even contribution from both contributors. The coding aspect was split in a way that Jarod set up the KNN algorithm and Mercedes set up the training and testing part of the algorithm. The dockerfile was worked on by both authors, and running and testing the docker containers was done on Mercedes's computer. The report was broken down in segments with Mercedes writing the initial outline and Jarod working on the abstract. Both collaborators worked evenly on writing and editing the rest of the sections of the report.

5.6.10 SPECIFICATION

```
swagger: "2.0"
info:
  version: "0.0.1"
```

```

title: "presidential support"
description: "Attempts to determine how much support a candidate will receive based on their viewpoints"
license:
  name: "Apache"
host: "localhost:8080"
basePath: "/cloudmesh/ai/voting"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /run/custom/<neighbors>/<hlt>/<mil>/<edu>/<tax>/<wmr>/<glb>/<gnr>/<inf>/<mnr>/<img>:
    get:
      tags:
        - RUN_CUSTOM
      operationId: run.run_custom
      description: "Runs an analysis based on given arguments"
      produces:
        - "application/json"
      responses:
        "200":
          description: "Run custom analysis"
          schema: {}
  /run/test:
    get:
      tags:
        - RUN_TEST
      operationId: run.run_test
      description: "Runs an analysis based on test/train data from dataset."
      produces:
        - "application/json"
      responses:
        "200":
          description: "Run test analysis"
          schema: {}
  /run/neighbors:
    get:
      tags:
        - RUN_NEIGHBORS
      operationId: run.neighbors
      description: "Determines best neighbor argument to use for KNN algorithm"
      produces:
        - "application/json"
      responses:
        "200":
          description: "Run neighbor search"
          schema: {}
  /data/download/<output>:
    get:
      tags:
        - DATA
      operationId: data.download
      description: "Downloads data from an external location"
      produces:
        - "application/json"
      responses:
        "200":
          description: "Data info"
          schema: {}
  /data/show/graph:
    get:
      tags:
        - DATA_GRAPH
      operationId: data.graph
      description: "Shows the graph generated from the neighbors endpoint"
      produces:
        - "application/png"
      responses:
        "200":
          description: "Show graph"
          schema: {}

```

5.7 SPAM ANALYSIS WITH SPAMALOT

Eric Bower, Tyler Zhang
epbower@iu.edu, tjzhang@iu.edu
Indiana University Bloomington
hid: sp19-222-101
Github: [🌸](#)
code: [🌸](#)

Keywords: Spam

5.7.1 ABSTRACT

Spam emails are an issue in cybersecurity because they can contain phishing scams or malware that can steal private information from the user, which can often lead to identity theft. These emails are particularly dangerous because many are seemingly innocuous. We are exploring the possibility of predicting malicious intent in emails by using a machine learning algorithm. We chose to use the Support Vector Machines (SVM) algorithm to classify spam emails due to its superior classification performance over another commonly used classification model, the Naive Bayes algorithm. Successfully classifying a malicious email can prevent harmful situations for users.

5.7.2 INTRODUCTION

The goal is the creation of a service that can classify spam emails. By the term “spam email”, we refer to emails that are sent with malicious intent against the recipient. Emails without malicious intent will be referred to as a “ham email”. Common forms of malicious intent that steal or restrict a user’s personal data include phishing and malware.

Phishing describes a process in which an attacker impersonates a trustworthy third party in an attempt to obtain sensitive information [50]. In a recent phishing attack, a link was sent to Snapchat users telling them to enable two-step authentication. This link collected the login information of more than 50,000 Snapchat users. Users were under the impression that they were making their private information more secure, but they instead exposed their personal information to hackers, demonstrating how phishing attacks can be particularly

deceptive.

Spam emails can also have dangerous attachments that contain malware. When these attachments are downloaded, the malware is unloaded onto the computer system. A common example of malware is called ransomware, which encrypts local files and network files, effectively preventing the user from accessing their own files. The ransomware will ask the user to pay in exchange for decrypting their files. Even if the user pays, the attacker still has control over the user's data, which may lead to identity theft [51].

An email classifier can prevent users from being hit by phishing scams and malware. We used a machine learning algorithm to classify malicious emails. An important consideration for the user is to minimize misclassification errors. When creating our service, we need to be careful to minimize the number of spam emails that are erroneously labeled as ham and the number of ham emails that are erroneously labeled as spam. If misclassification occurs, the user is at risk of opening dangerous spam emails and missing important ones.

5.7.3 THE ALGORITHM

We considered the Naive Bayes and Support Vector Machine (SVM) algorithms for our implementation of spam classification. Naive Bayes and SVM are both supervised learning algorithms, which means that they are trained with data that is already labeled. Both algorithms have strengths and weaknesses. The Naive Bayes algorithm generally is faster and less computationally complex [52]. SVM is slower than Naive Bayes but typically tends to be more statistically robust [53]. We experimented with both algorithms and chose which one to use based on their performance and respective statistics.

5.7.3.1 Naive Bayes

Naive Bayes classifiers are very commonly used for spam filtering and document classification problems [54]. The Naive Bayes algorithm relies on Bayes' probability theorem, which expresses a relationship between the probability of the occurrence of an event c given the occurrence of other events, x_1 through x_n [52]. Representing E as (x_1, x_2, \dots, x_n) , the probability of an event c given E is given in [Equation 5](#) from [52].

$$P(c|E) = \frac{P(E|c)P(c)}{P(E)} \quad (5)$$

In terms of classification, the vector E would be the features of the data point, and c is the classification of that data point (either ham or spam). To create a binary classifier, with two classifications being $c = spam$ and $c = ham$, the classification of a data point with a feature vector E is given in [Equation 6](#) from [\[52\]](#).

$$f_b(E) = \frac{P(c = spam|E)}{P(c = ham|E)} \geq 1 \quad (6)$$

The terms $P(c = spam|E)$ and $P(c = ham|E)$ are both calculated using [Equation 5](#). The classification is spam if $f_b(E)$ is greater than or equal to one, and ham if $f_b(E)$ is less than one. Other researchers have found that the Naive Bayes classification model has decent precision and recall values when classifying spam emails. The following [Figure 11](#) shows the results of such an experiment of a Naive Bayes classifier run on a data set of 2893 total messages:

Filter Configuration	λ	No. of attrib.	Spam Recall	Spam Precision	Weighted Accuracy	Baseline W. Acc.	TCR
(a) bare	1	50	81.10%	96.85%	96.408%	83.374%	4.63
(b) stop-list	1	50	82.35%	97.13%	96.649%	83.374%	4.96
(c) lemmatizer	1	100	82.35%	99.02%	96.926%	83.374%	5.41
(d) lemmatizer + stop-list	1	100	82.78%	99.49%	97.064%	83.374%	5.66
(a) bare	9	200	76.94%	99.46%	99.419%	97.832%	3.73
(b) stop-list	9	200	76.11%	99.47%	99.401%	97.832%	3.62
(c) lemmatizer	9	100	77.57%	99.45%	99.432%	97.832%	3.82
(d) lemmatizer + stop-list	9	100	78.41%	99.47%	99.450%	97.832%	3.94
(a) bare	999	200	73.82%	99.43%	99.912%	99.980%	0.23
(b) stop-list	999	200	73.40%	99.43%	99.912%	99.980%	0.23
(c) lemmatizer	999	300	63.67%	100.00%	99.993%	99.980%	2.86
(d) lemmatizer + stop-list	999	300	63.05%	100.00%	99.993%	99.980%	2.86

Figure 11: NB Ling-Spam Results[\[55\]](#)

5.7.3.1.1 Metrics

In [Figure 11](#), two important performance metrics of the Naive Bayes algorithm were recall and precision, which are both standard metrics used to evaluate the effectiveness of a model. Recall is defined as the number of true positives divided by the sum of true positives and false negatives. To put that definition in

the context of a spam classifier, recall measures the ratio of spam emails correctly identified as spam compared to the number of actual spam emails in the data set.

Precision is defined as the number of true positives divided by the sum of true positives and false positives. To put that definition in the context of a spam classifier, precision measures the ratio of spam emails correctly identified as spam compared to the number of emails the classifier thinks are spam. For both metrics, higher percentages show a superior model. The figure [Figure 11](#) demonstrates decently high percentages for recall and precision when using Naive Bayes as a spam email classifier.

5.7.3.2 Support Vector Machines (SVM)

SVM is also used to work in text classification [\[54\]](#). SVM models construct hyper-planes in the feature space of the dataset which can be used for classification. The hyperplane is chosen by finding the optimal plane that maximizes its margins of separation between points of all classes [\[56\]](#). In other words, data points are separated from the others based on their features in an optimal manner.

To better illustrate how SVM works, one can imagine all points of a data set plotted based on their attributes. The data points that belong to a certain classification will generally be plotted together in regions because they normally have similar attributes. A hyperplane is an imaginary divider between these classification regions that is mathematically calculated based on distance. The following figure [Figure 12](#) shows a visualization of a two-dimensional hyperplane:

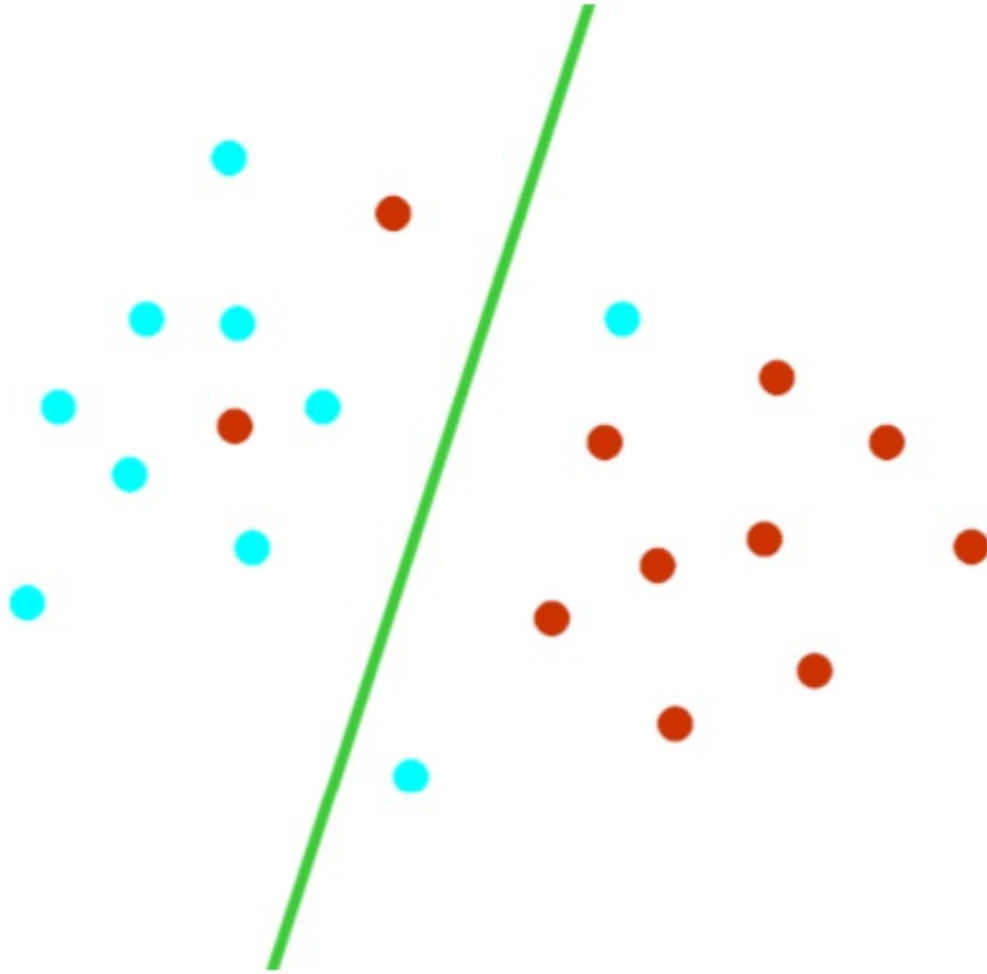


Figure 12: SVM 2D Visualization[56]

This figure shows a divider between general regions of points classified as blue and points classified as red. Notice that the hyperplane is imperfect; there are some red points on the blue side of the hyperplane, and there are some blue points on the red side. Such error is inevitable, but it can be minimized mathematically. To make a classification on a new piece of data, the algorithm plots the attributes of a new point and makes a prediction based on which side of the hyperplane the point falls on.

SVM algorithms are very effective classifiers when working with datasets that utilize a large number of features [56]. In the case of emails, we can calculate the frequency of words contained in the email, which will be a vector with a large number of features [54]. As a result, SVM is an appropriate model to classify spam emails due to the high dimensional dataset.

5.7.4 THE DATA SET

The data set used to train the machine learning algorithm is taken from a project that tested the effectiveness of five different variations of Naive Bayesian classifiers on classifying spam emails. It contains the text of ham and spam messages from a member of Enron corpus with random ham-spam ratio [57]. These emails are all labeled as ham or spam.

The raw messages of each email generally contain too much information to be considered useful to train the classifier. In particular, raw email message text has many characters that contribute little to the classification of the email. To counteract this, we removed all non-alphabetical characters from the data set. Also, we removed any words that were only one character long, such as ‘a’ and ‘I’. Finally, we removed all instances of the word ‘the’, which turned out to be one of the most common words in all the training emails.

It has been shown that the process of lemmatization improves classification performance for spam filtering [55]. Lemmatization is the process of grouping together variations of the same root word. For instance, a lemmatizer would group all instances of the words “include”, “includes”, and “included” in the same category. The data set that we used to train our algorithm was already lemmatized, so we did not need to go through this process ourselves.

Once the email texts were filtered, we then created vectors of word frequencies from each email. These are treated as the features for the machine learning model. Our dataset was already labeled as “spam” or “ham”, making it possible to train the SVM and Naive Bayes supervised learning models. We shuffled the list of word frequency vectors from the dataset and randomly chose 80% of these vectors to train both the SVM and Naive Bayes models. We used the remaining 20% of these vectors to assess the quality of each model.

5.7.5 MODEL RESULTS

The confusion matrix generated from the Naive Bayes algorithm is represented in [Figure 13](#).

Total Emails: 1226		Predicted	
		Ham	Spam
Actual	Ham	812	18
	Spam	203	193

Figure 13: NB Confusion Matrix

The confusion matrix generated from the SVM algorithm is represented in [Figure 14](#).

Total Emails: 1226		Predicted	
		Ham	Spam
Actual	Ham	694	136
	Spam	53	343

Figure 14: SVM Confusion Matrix

We can use the above confusion matrices to obtain metrics for each model. Recall measures the ratio of spam emails correctly identified as spam compared to the number of actual spam emails in the data set. For the Naive Bayes model, this value is $\frac{193}{203+193} = 0.487$. For the SVM model, this value is $\frac{343}{53+343} = 0.866$.

Precision measures the ratio of spam emails correctly identified as spam compared to the number of emails the classifier thinks are spam. For the Naive Bayes model, this value is $\frac{193}{18+193} = 0.915$. For the SVM model, this value is $\frac{343}{136+343} = 0.716$.

Although the Naive Bayes method has higher precision than the SVM, it has significantly lower recall. The impact this would have on the user is dangerous: the confusion matrix [Figure 13](#) shows that the Naive Bayes model flags a great number of spam emails as ham, exposing the user to more malicious emails. On the other hand, the SVM method has a significantly higher recall value, but at a lower precision value. The consequences of this are not as dire for the user: the confusion matrix [Figure 14](#) shows the SVM model flags more ham emails as spam. While this is inconvenient for the user, the SVM method is safer because it would expose fewer malicious emails to the user, at the cost of flagging benign emails as spam more often. For this reason, we decided to use the SVM model for our classification method.

5.7.6 IMPLEMENTATION

5.7.6.1 The Server

We used Swagger OpenAPI to create our server. This package lets us conveniently define our server's endpoints and REST API operations in a concise YAML file. In addition to Swagger OpenAPI, we also greatly utilized the Flask package. In our YAML file, our server has one endpoint called 'upload', which acts as a REST POST operation. The `upload()` function corresponding to this endpoint is defined in a file called `gatherData.py`. From the Flask package, `upload()` uses `Flask.request()` to read in a user's text file. Once the file has been accepted, the text file is processed and the classification occurs. The results of the classification are then returned and displayed to the user as an HTML page using `Flask.render_template()`. The `upload()` function and the classification process is explained in more detail in the next section.

It should be noted that the 'upload' endpoint cannot be reached directly by URL; it can only be accessed by pressing a JavaScript button on the home page of the server, which then calls the `Flask.request()` function and starts the classification process. If a user attempts to access the 'upload' endpoint via URL instead of clicking the button, they will be taken to an error page. Once a user successfully sees the classification results, they are given the option to return to the home page via another JavaScript button, where they can upload another email for classification.

We created a Dockerfile and a Makefile that contains all the necessary commands needed to host the server. This Dockerfile runs Ubuntu and runs commands to clone files from our Github to get the service running. Running the server in a container such as Docker is beneficial because the installation will not interfere with the host system, and the container is easy to remove once installed.

The following diagram [Figure 15](#) shows the basic workflow of our server:

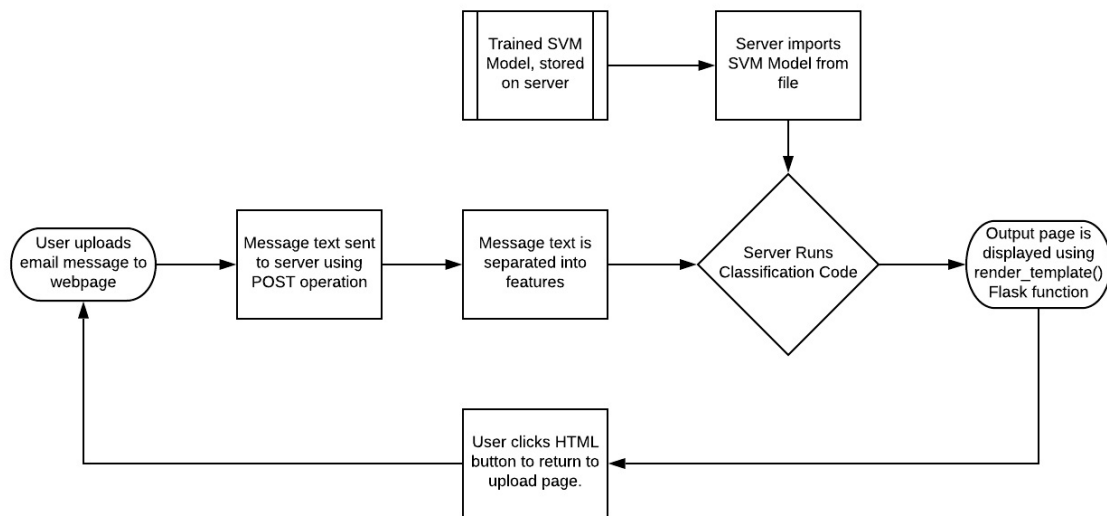


Figure 15: Classification Flowchart

5.7.6.2 The Upload Function and Classification

The `upload()` function handles file uploading and makes other function calls to perform classification. The file is first retrieved by utilizing the `Flask.request()` function from the Flask package. This file must have a `.txt` extension and should contain the body of the email to be classified.

Once the file is uploaded, `upload()` sends the file to a function called `extract_features()`. The `extract_features()` function converts the email text into a word frequency vector. This vector is treated as the features of the email and is used as an input for the SVM model to classify the email as spam or ham.

Our server contains a previously-trained SVM model file, so it will not need to retrain the model each time a user uploads a new file. We used the `sklearn`

package to train an SVM model on a large number of emails, creating a large database of word frequency vectors and their corresponding labels. Using the pickle package, we saved this model as a file on the server. When the `extract_features()` function successfully returns a feature vector for the user's uploaded email, the server uses the pickle package to load the saved SVM model and labels the user's email as ham or spam.

The final pieces of information that the `upload()` function calculates are performance statistics for the SVM model to create a confusion matrix, from which the user can calculate model metrics such as precision and recall. This gives the user insight on the performance of the SVM model. Once the server has made a prediction and has performance statistics, it calls `Flask.render_template()` to display an HTML file with the returned variables. This is the page the user sees after uploading their email file.

5.7.6.3 Specification

```
swagger: "2.0"
info:
  version: "0.0.1"
  title: "spamInfo"
  description: "An intelligent system to determine whether a given email is spam or not"
  contact:
    name: "Spamalot"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "text/html"
paths:
  /upload:
    post:
      operationId: py_scripts.gatherData.upload
      description: "Uploads a file"
      responses:
        "201":
          description: "Upload"
```

5.7.7 CONCLUSION

Our implementation of a spam email classifier is a building block for a more sophisticated spam filter. Currently, our service is only capable of making predictions on one email at a time, and each prediction requires the user to upload a new file containing the email text. A more sophisticated spam filter would be integrated into an email app and could automatically classify incoming

emails as spam. Additionally, our service's classification ability is imperfect. As seen in [Figure 11](#), other researchers achieve superior classification performance using Naive Bayes, so there is room for improvement.

In its current state, our implementation is best suited for single users. But with some further development, such as adding the ability to classify multiple emails at once and improving our service's classification performance, a spam filter would be a useful tool for company employees. Our service could prevent employees from opening dangerous emails that could steal company information.

6 DESSIMINTAION

6.1 TASK 6 – DISSEMINATION

The work we are proposing is integrated into various dissemination activities. This includes integration of the activities as part of classes taught at the graduate level at Indiana University to test out and improve the components developed. Educational components were designed in a reusable fashion and added to a Handbook of Cloud Computing and Big Data for Graduates and Undergraduates distributed in epub format.

- Graduate: <https://github.com/cloudmesh-community/book/blob/master/vonLaszewski-cloud.epub>
- Undergraduate: <https://github.com/cloudmesh-community/book/blob/master/vonLaszewski-e222.epub>

The component is now also available as a compact topical focus area allowing flexible integration into other educational activities such as tutorials and presentations focusing on specific aspects of the work conducted.

Such components are available as markdown documents at the following locations.

- <https://github.com/cloudmesh-community/book/blob/master/chapters/nist/bdra.md>
- <https://github.com/cloudmesh-community/book/blob/master/chapters/rest/rest.md>
- <https://github.com/cloudmesh-community/book/blob/master/chapters/rest/swagger-codegen.md>

To make the system genuinely reusable, the bookmanager can integrate the components in a customizable document that can be generated by choice for tutorials, classes, and individual learning experiences. The tool is called bookmanager and available on PyPI at

- <https://pypi.org/project/cyberaide-bookmanager/>

6.1.1 CONFERENCE PRESENTATION

The work we conducted in regards to automated REST services creation for Big Data was presented at PyData in Indianapolis Oct 11th

- <https://2019.indypy.org/pydata/>

6.1.2 TASK 6.1 COMMUNITY TESTING

The most crucial part is that we verified with more than 20 students on the graduate and undergraduate level that the approach we have chosen is valid and can be replicated even with inexperienced students that do not have any prior exposure to the technology and also have only a limited amount of time exploring the technology. The examples in this handbook include at this time results from undergraduate students showcasing the REST services that developed with the help of our tutorial and tools.

7 RESOURCES

7.1 TASK 7 – DEVELOPMENT RESOURCES

We have set up some significant development resources for this project. This includes the deployment of a

- Swagger cluster, and a
- Kubernetes cluster

In addition, the projects that we developed have been using docker containers on the developers' computers. Using containers has the advantage that such services can be hosted in a portable fashion on any major resource infrastructure.

Additional resources have been used on the following cloud infrastructures:

- Chameleon Cloud (chameleoncloud.org)
- Aws
- Azure
- Google

We also experimented with IBM Watson which also could have been used for this project, but as we had ample resources, we did not fully explore this option.

8 TUTORIALS

8.1 OVERVIEW

In this Section we present a number of tutorials that are related to this work.

This includes the following:

- **Automated REST service generation with eve:** A tutorial to create rest services with eve. This tutorial used the first version of the BDRA vol 8 to automatically generate REST services from examples. Although this was the easiest way to generate REST services, the community development of eve was halted for a significant period of time and the community moved on to OpenAPI v1
- **Automated REST service generation with codegen:** A tutorial to create rest services with swagger codegen. This tutorial used the second version of the BDRA vol 8 to automatically generate REST services from examples which was based on OpenAPI v2.
- **Automated REST service generation with OpenAPIv3 and connexion:** A tutorial to create rest services with connexion using OpenAPIv3. This tutorial used the current version of the BDRA vol 8 to automatically generate REST services from examples which was based on OpenAPI v2.

8.2 AUTOMATED REST SERVICE GENERATION WITH EVE

8.2.1 REST SERVICES WITH EVE

Next, we focus on how to make a RESTful web service with Python Eve. Eve makes the creation of a REST implementation in python easy. More information about Eve can be found at:

- <http://python-eve.org/>

Although we do recommend Ubuntu 17.04, at this time, there is a bug that forces us to use 16.04. Furthermore, we require you to follow the instructions on how to install pyenv and use it to set up your python environment. We recommend that you use either python 2.7.14 or 3.6.4. We do not recommend you to use anaconda as it is not suited for cloud computing but targets desktop computing. If you use pyenv you also avoid the issue of interfering with your system-wide python install. We do recommend pyenv regardless if you use a virtual machine or are working directly on your operating system. After you have set up a proper python environment, make sure you have the newest version of pip installed with

```
$ pip install pip -U
```

To install Eve, you can say

```
$ pip install eve
```

As Eve also needs a backend database, and as MongoDB is an obvious choice for this, we have first to install MongoDB. MongoDB is a Non-SQL database which helps to store lightweight data easily.

8.2.1.1 Ubuntu install of MongoDB

On Ubuntu, you can install MongoDB as follows.

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
--recv 2930ADAE8CAF5059EE73BB4B58712A2291FA4AD5
$ echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu \
xenia/mongodb-org/3.6 multiverse" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-3.6.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

8.2.1.2 macOS install of MongoDB

On macOS you can use the command.

```
$ brew update
$ brew install mongodb
```

8.2.1.3 Windows 10 Installation of MongoDB

A student or student group of this class are invited to discuss on Piazza on how

to install mongoDB on Windows 10 and come up with an easy installation solution. Naturally, we have the same 2 different ways on how to run mongo. In user space or in the system. As we want to make sure your computer stays secure. The solution must have an easy way on how to shut down the Mongo services.

An enhancement of this task would be to integrate this function into Cloudmesh cmd5 with a command *mongo* that allows for easier starting and stopping the service from *cms*.

8.2.1.4 Database Location

After downloading Mongo, create the *db* directory. This is where the Mongo data files will live. You can create the directory in the default location and assure it has the right permissions. Make sure that the */data/db* directory has the right permissions by running.

8.2.1.5 Verification

In order to check the MongoDB installation, please run the following commands in one terminal:

```
$ mkdir -p ~/cloudmesh/data/db
$ mongod --dbpath ~/cloudmesh/data/db
```

In another terminal, we try to connect to mongo and issue a mongo command to show the databases:

```
$ mongo --host 127.0.0.1:27017
$ show databases
```

If they execute without errors, you have successfully installed MongoDB. To stop the running database instance, run the following command. simply CTRL-C the running mongod process

8.2.1.6 Building a simple REST Service

In this section, we focus on creating a simple rest service. To organize our work we create the following directory:

```
$ mkdir -p ~/cloudmesh/eve
$ cd ~/cloudmesh/eve
```

As Eve needs a configuration and it is read in by default from the file `settings.py` we place the following content in the file `~/cloudmesh/eve/settings.py`:

```
MONGO_HOST = 'localhost'
MONGO_PORT = 27017
MONGO_DBNAME = 'student_db'
DOMAIN = {
    'student': {
        'schema': {
            'firstname': {
                'type': 'string'
            },
            'lastname': {
                'type': 'string'
            },
            'university': {
                'type': 'string'
            },
            'email': {
                'type': 'string',
                'unique': True
            }
        },
        'username': {
            'type': 'string',
            'unique': True
        }
    }
}
RESOURCE_METHODS = ['GET', 'POST']
```

The DOMAIN object specifies the format of a `student` object that we are using as part of our REST service. In addition, we can specify `RESOURCE_METHODS` which methods are activated for the REST service. This way, the developer can restrict the available methods for a REST service. To pass along the specification for MongoDB, we simply specify the hostname, the port, as well as the database name.

Now that we have defined the settings for our example service, we need to start it with a simple python program. We could name that program anything we like, but often it is called simply `run.py`. This file is placed in the same directory where you placed the **settings.py**. In our case, it is in the file `~/cloudmesh/eve/run.py` and contains the following python program:

```
from eve import Eve
app = Eve()

if __name__ == '__main__':
    app.run()
```

This is the most minimal application for Eve, which uses the `settings.py` file for its configuration. Naturally, if we were to change the configuration file and, for example change the DOMAIN and its schema, we would naturally have to

remove the database previously created and start the service new. This is especially important as during the development phase, we may frequently change the schema and the database. Thus it is convenient to develop necessary cleaning actions as part of a Makefile, which we leave as easy exercise for the students.

Next, we need to start the services which can easily be achieved in a terminal while running the commands:

Previously we started the MongoDB service as follows:

```
$ mongod --dbpath ~/cloudmesh/data/db/
```

This is done in its own terminal so that we can observe the log messages easily. Next, we start in another window the Eve service with

```
$ cd ~/cloudmesh/eve
$ python run.py
```

You can find the codes and commands up to this point in the following document.

8.2.1.7 Interacting with the REST service

Yet, in another window, we can now interact with the REST service. We can use the command line to save the data in the database using the REST API. The data can be retrieved in XML or JSON format. JSON is often more convenient for debugging as it is easier to read than XML.

Naturally, we need first to put some data into the server. Let us assume we add the user Albert Zweistein.

```
$ curl -H "Content-Type: application/json" -X POST \
-d '{"firstname":"Albert","lastname":"Zweistein", \
  "school":"ISE","university":"Indiana University", \
  "email":"albert@iu.edu", "username": "albert"}' \
http://127.0.0.1:5000/student/
```

To achieve this, we need to specify the header using **H** tag, saying we need the data to be saved using JSON format. And **X** tag says the HTTP protocol, and here we use POST method. And the tag **d** specifies the data and make sure you use JSON format to enter the data. Finally, the REST API endpoint to which we must save data. This allows us to save the data in a table called **student** in

MongoDB within a database called **eve**.

In order to check if the entry was accepted in mongo and included in the server issue the following command sequence in another terminal:

```
$ mongo
```

Now you can query mongo directly with its shell interface.

```
> show databases
> use student_db
> show tables # query the table names
> db.student.find().pretty() # pretty will show the json in a clear way
```

Naturally, this is not necessary for A REST service such as eve as we show you next how to gain access to the data via mongo while using REST calls. We can simply retrieve the information with the help of a simple URI:

```
$ curl http://127.0.0.1:5000/student?firstname=Albert
```

Naturally, you can formulate other URLs and query attributes that are passed along after the `?`.

This will now allow you to develop sophisticated REST services. We encourage you to inspect the documentation provided by Eve to showcase additional features that you could be used as part of your efforts.

Let us explore how to use additional REST API calls properly. We assume you have MongoDB up and running. To query the service itself, we can use the URI on the Eve port

```
$ curl -i http://127.0.0.1:5000
```

Your payload should look like the one listed next if your output is not formatted like this try adding `?pretty=1`

```
$ curl -i http://127.0.0.1:5000?pretty=1
```

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 150
Server: Eve/0.7.6 Werkzeug/0.11.15 Python/2.7.16
Date: Wed, 17 Jan 2018 18:34:07 GMT
```

```
{
  "_links": {
    "child": [
      {
        "href": "student",
        "title": "student"
      }
    ]
  }
}
```

```
} 1
```

Remember that the API entry points include additional information such as links and a child, and href.

Set up a python environment that works for your platform. Provide explicit reasons why anaconda and other prepackaged python versions have issues for cloud-related activities. When may you use anaconda and when should you not use anaconda. Why would you want to use pyenv?

What is the meaning and purpose of links, child, and href

In this case, how many child resources are available through our API?

Develop a REST service with Eve and start and stop it

Define curl calls to store data into the service and retrieve it.

Write a Makefile, and in it, a target clean that cleans the database. Develop additional targets such as start and stop, that start and stop the MongoDB but also the Eve REST service

Issue the command

```
$ curl -i http://127.0.0.1:5000/people
```

What does the `_links` section describe?

What does the `_items` section describe?

```
{
  "_items": [],
  "_links": {
    "self": {
      "href": "people",
      "title": "people"
    },
    "parent": {
      "href": "/",
      "title": "home"
    }
  },
  "_meta": {
    "max_results": 25,
    "total": 0,
    "page": 1
  }
}
```

8.2.1.8 Creating REST API Endpoints

Next, we want to enhance our example a bit. First, let us get back to the eve working directory with

```
$ cd ~/cloudmesh/eve
```

Add the following content to a file called **run2.py**

```
from eve import Eve
from flask import jsonify
import os
import getpass
app = Eve ()
@app.route('/student/albert')
def alberts_information():
    data = {
        'firstname': 'Albert',
        'lastname': 'Zweistsein',
        'university': 'Indiana University',
        'email': 'albert@example.com'
    }
    try:
        data['username'] = getpass.getuser()
    except:
        data['username'] = 'not-found'
    return jsonify(**data)

if __name__ == '__main__':
    app.run(debug=True, host="127.0.0.1")
```

After creating and saving the file. Run the following command to start the service

```
$ python run2.py
```

After running the command, you can interact with the service while entering the following url in the web browser:

```
http://127.0.0.1:5000/student/alberts
```

You can also open up a second terminal and type in it

```
$ curl http://127.0.0.1:5000/student/alberts
```

The following information will be returned:

```
{
  "firstname": "Albert",
  "lastname": "Zweistain",
  "university": "Indiana University",
  "email": "albert@example.com",
  "username": "albert"
}
```

This example illustrates how easy it is to create REST services in python while

combining information from a dict with information retrieved from the system. The important part is to understand the decorator **app.route**. The parameter specifies the route of the API endpoint, which is be the address appended to the base path, <http://127.0.0.1:5000>. We must return a *jsonified* object, which can easily be done with the `jsonify` function provided by Flask. As you can see, the name of the decorated function can be anything you look. The route specifies how we access it from the service.

8.2.1.9 REST API Output Formats and Request Processing

Another way of managing the data is to utilize class definitions and response types that we explicitly define.

If we want to create an object like Student, we can first define a python class. Create a file called **student.py**. Please, note the get method that returns simply the information in the dict for the class. It is not related to the REST get function.

```
class Student(object):
    def __init__(self, firstname, lastname, university, email):
        self.firstname = firstname
        self.lastname = lastname
        self.university = university
        self.email = email
        self.username = 'undefined'
    def get(self):
        return self.__dict__
    def setUsername(self, name):
        self.username = name
        return name
```

Next, we define a REST service with Eve as shown in the following listing

```
from eve import Eve
from student import Student
import platform
import psutil
import json
from flask import Response
import getpass
app = Eve()
@app.route('/student/albert', methods=['GET'])
def processor():
    student = Student("Albert",
                     "Zweistein",
                     "Indiana University",
                     "albert@example.edu")

    response = Response()
    response.headers["Content-Type"] = "application/json; charset=utf-8"

    try:
        student.setUsername(getpass.getuser())
        response.headers["status"] = 200
    except:
        response.headers["status"] = 500
```

```
response.data = json.dumps(student.get())
return response

if __name__ == '__main__':
    app.run(debug=True, host='127.0.0.1')
```

In contrast to our earlier example, we are not using the jsonify object but create a response explicitly that we return to the clients. The response includes a header that we return the information in JSON format, a status of 200, which means the object was returned successfully, and the actual data.

8.2.1.10 REST API Using a Client Application

O This example is not tested. Please provide feedback and improve.

In the Section [Rest Services with Eve](#) we created our own REST API application using Python Eve. Now once the service is running, we need to learn how to interact with it through clients.

First, go back to the working folder:

```
$ cd ~/cloudmesh/eve
```

Here we create a new python file called **client.py**. The file include the following content.

```
import requests
import json

def get_all():
    response = requests.get("http://127.0.0.1:5000/student")
    print(json.dumps(response.json(), indent=4, sort_keys=True))

def save_record():
    headers = {
        'Content-Type': 'application/json'
    }

    data = '{"firstname":"Gregor",
            "lastname":"von Laszewski",
            "university": "Indiana University",
            "email":"jane@iu.edu",
            "username": "jane"}'

    response = requests.post('http://localhost:5000/student/',
                             headers=headers,
                             data=data)

    print(response.json())

if __name__ == '__main__':
    save_record()
    get_all()
```

Run the following command in a new terminal to execute the simple client by

```
$ python client.py
```

Here when you run this class for the first time, it runs successfully, but if you tried it for the second time, it would give you an error. Because we did set the email to be a unique field in the schema when we designed the settings.py file in the beginning. So if you want to save another record, you must have entries with unique emails. To make this dynamic, you can include an input reading by using the terminal to get the student data first, and instead of the static data you can use the user input data from the terminal to get dynamic data. But for this exercise, we do not expect that or any other form data functionality.

To get the saved data, you can comment on the record saving function and uncomment the get all function. In python, commenting is done by using #.


This client is using the **requests** python library to send GET, POST and other HTTP requests to the server so you can leverage build in methods to simplify your work.

The `get_all` function provides a way to get the output to the console with all the data in the student database. The `save_record` function provides a way to save data in the database. You can create dynamic functions in order to save dynamic data. However, it may take some time for you to apply as an exercise.

Write a RESTful service to determine a useful piece of information off of your computer i.e. disk space, memory, RAM, etc. In this exercise what you need to do is use a python library to extract data about computer information mentioned previously and sent this information to the the user once the user calls an API endpoint like `http://localhost:5000/performance/ram`, it must return the RAM value of the given machine. For each information like disk space, RAM, etc. you can use an endpoint per each feature needed. As a tip for this exercise, use the `psutil` library in python to retrieve the data, and then get this information into a string then populate a class called Computer and try to save the object likewise.

8.2.1.11 Towards cmd5 extensions to manage eve and mongo



 *Part of this section related to the management of the MongoDB service is done by the `cm4` command we will be developing as part of this class `cms mongo admin` that does all of the things explained next and more.*

Naturally, it is of advantage to have in `cms` administration commands to manage `mongo` and `eve` from `cmd` instead of targets in the Makefile. Hence, we **propose** that the class develops such an extension. We create in the repository the extension called `admin` and hope that students through collaborative work and pull requests complete such an `admin` command.

The proposed command is located at:

- <https://github.com/cloudmesh/cloudmesh.rest/blob/master/cloudmesh/admin>

It will be up to the class to implement such a command. Please coordinate with each other.

The implementation based on what we provided in the Make file seems straight forward. A great extension is to load the definitions of the objects or `eve` e.g., `settings.py` not from the class, but a place in `.cloudmesh`. I propose to place the file at:

```
~/cloudmesh/db/settings.py
```

the location of this file is used when the `Service` class is initialized with `None`. Prior to starting the service, the file needs to be copied there. This could be achieved with a `set` command.

8.2.2 HATEOAS

In the previous section, we discussed the basic concepts of RESTful web service. Next, we introduce you to the concept of HATEOAS

HATEOAS stands for Hypermedia as the Engine of Application State, and the default configuration enables this within `Eve`. It is useful to review the terminology and attributes used as part of this configuration. HATEOAS explains how REST API endpoints are defined, and it provides a clear description of how

the API can be consumed through these terms:

_links

Links describe the relation of the current resource being accessed to the rest of the resources. It is like if we have a set of links to the set of objects or service endpoints that we are referring in the RESTful web service. Here an endpoint refers to a service call that is responsible for executing one of the CRUD operations on a particular object or set of objects. More on the links, the links object contains the list of serviceable API endpoints or a list of services. When we are calling a GET request or any other request, we can use these service endpoints to execute different queries based on the user purpose. For instance, a service call can be used to insert data or retrieve data from a remote database using a REST API call. About databases, we discuss in detail in another chapter.

title

The title in the rest endpoint is the name or topic that we are trying to address. It describes the nature of the object by a single word. For instance student, bank-statement, salary, etc. can be a title.

parent

The term parent refers to the very initial link or an API endpoint in a particular RESTful web service. Generally, this is denoted with the primary address, like <http://example.com/api/v1/>.

href

The term href refers to the url segment that we use to access a particular REST API endpoint. For instance “student?page=1” will return the first page of student list by retrieving a particular number of items from a remote database or a remote data source. The full URL looks like this, “<http://www.exampleapi.com/student?page=1>”.

In addition to these fields, eve automatically create the following information when resources are created as showcased of

- <http://python-eve.org/features.html>

Field	Description
<code>_created</code>	item creation date.
<code>_updated</code>	item last updated on.
<code>_etag</code>	ETag, to be used for concurrency control and conditional requests.
<code>_id</code>	unique item key, also needed to access the individual item endpoint.

Pagination information can be included in the `_meta` field.

8.2.2.1 Filtering

Clients can submit query strings to the rest service to retrieve resources based on a filter. This also allows the sorting of the results queried. One nice feature about using mongo as a backend database is that Eve allows not only python conditional expressions, but also mongo queries.

A number of examples to conduct such queries include:

```
$ curl -i -g http://eve-demo.herokuapp.com/people?where={%22lastname%22:%20%22Doe%22}
```

A python expression

```
$ curl -i http://eve-demo.herokuapp.com/people?where=lastname=="Doe"
```

8.2.2.2 Pretty Printing

Pretty printing is typically supported by adding the parameter `?pretty` or `?pretty=1`

If this does not work, you can always use python to beautify a JSON output with

```
$ curl -i http://localhost/people?pretty
```

or

```
$ curl -i http://localhost/people | python -m json.tool
```

8.2.2.3 XML

If for some reason you like to retrieve the information in XML you can specify this for example through curl with an Accept header

```
$ curl -H "Accept: application/xml" -i http://localhost
```

8.2.3 EXTENSIONS TO EVE

Several extensions have been developed by the community. This includes eve-swagger, eve-sqlalchemy, eve-elastic, eve-mongoengine, eve-neo4j, eve.net, eve-auth-jwt, and flask-sentinel.

Naturally, there are many more.

Students have the opportunity to pick one of the community extensions and provide a section for the handbook.

Pick one of the extension, research it and provide a small section for the handbook, so we add it.

8.2.3.1 Object Management with Eve and Evegenie

<http://python-eve.org/>

Eve makes the creation of a REST implementation in python easy. We will provide you with an implementation example that showcases that we can create REST services without writing a single line of code. The code for this is located at <https://github.com/cloudmesh/rest>

This code will have a master branch but will also have a dev branch in which we will add gradually more objects. Objects in the dev branch will include:

- virtual directories
- virtual clusters
- job sequences
- inventories

You may want to check our ongoing development work in the dev branch. However, for the purpose of this class, the master branch is sufficient.

8.2.3.1.1 Installation

First, we have to install MongoDB. The installation will depend on your operating system. For the use of the REST service, it is not essential to integrate MongoDB into the system upon reboot, which is the focus of many online documents. However, for us, it is better if we can start and stop the services explicitly for now.

On ubuntu, you need to do the following steps:

🕒 TODO: Section can be contributed by a student.

On Windows 10, you need to do the following steps:

🕒 TODO: Section can be contributed by a student. If you elect Windows 10. You could be using the online documentation provided by starting it on Windows, or running it in a docker container.

On macOS you can use home-brew and install it with:

```
$ brew update  
$ brew install mongodb
```

In future, we may want to add SSL authentication in which case you may need to install it as follows:

```
$ brew install mongodb --with-openssl
```

8.2.3.1.2 Starting the service

We have provided a convenient Makefile that currently only works for macOS. It is easy for you to adapt it to Linux. Certainly, you can look at the targets in the makefile and replicate them one by one. Important targets are `deploy` and `test`.

When using the makefile, you can start the services with:

```
$ make deploy
```

It starts two terminals. In one you see the mongo service, in the other, you see the eve service. The eve service takes a file called `sample.settings.py` that is based on `sample.json` for the start of the eve service. The mongo service is configured

in such a way that it only accepts incoming connections from the localhost, which is sufficient for our case. The mongo data is written into the `$(USER)/.cloudmesh` directory, so make sure it exists.

To test the services, you can say:

```
$ make test
```

You will see several JSON messages be written to the screen.

8.2.3.1.3 Creating your own objects

The example demonstrated how easy it is to create a MongoDB and an eve rest service. Now let us use this example to create your own. For this, we have modified a tool called evegenie to install it onto your system.

The original documentation for evegenie is located at:

- <http://evegenie.readthedocs.io/en/latest/>

However, we have improved evegenie while providing a command line tool based on it. The improved code is located at:

- <https://github.com/cloudmesh/evegenie>

You clone it and install on your system as follows:

```
$ cd ~/github
$ git clone https://github.com/cloudmesh/evegenie
$ cd evegenie
$ python setup.py install
$ pip install .
```

This should install in your system evegenie. YOU can verify this by typing:

```
$ which evegenie
```

If you see the path evegenie is installed. With evegenie installed its usage is simple:

```
$ evegenie

Usage:
evegenie --help
evegenie FILENAME
```

It takes a JSON file as input and writes out a settings file for the use in eve. Let us assume the file is called sample.json than the settings file is called sample.settings.py. Having the evegenie program allows us to generate the settings files easily. You can include them in your project and leverage the Makefile targets to start the services in your project. In case you generate new objects, make sure you rerun evegenie, kill all previous windows in which you run eve and mongo and restart. In case of changes to objects that you have designed and run previously, you also need to delete the MongoDB database.

8.3 AUTOMATED REST SERVICE GENERATION WITH CODEGEN FOR OPENAPI 2.0

8.3.1 OPENAPI 2.0 SPECIFICATION

Swagger provides through its specification the definition of REST services through a YAML or JSON document.

When following the API-specification-first approach to defining and developing a RESTful service, the first and foremost step is to define the API conforming to the OpenAPI specification, and then using codegen tools to conveniently generate server-side stub code, client code, documentation, in the language you desire. In Section [REST Service Generation with OpenAPI](#) we have introduced the codegen tool and how to use that to generate server-side and client-side code and documentation. In this Section [The Virtual Cluster example API Definition](#) we will use a slightly more complex example to show how to define an API following the OpenAPI 2.0 specification. The example is to retrieve the virtual cluster (VC) object from the server.

The OpenAPI specification is formerly known as Swagger RESTful API Documentation Specification. It defines a specification to describe and document a RESTful service API. It is also known under version 3.0 of swagger. However, as the tools for 3.0 are not yet completed, we continue for now to use version swagger 2.0, until the transition has been completed. This is especially of importance, as we need to use the swagger codegen tool, which currently supports only up to specification v2. Hence we are at this time using OpenAPI/Swagger v2.0 in our example. There are some structure and syntax

changes in v3, while the essence is very similar. For more details of the changes between v3 and v2, please refer to [A document published on the Web titled *Difference between OpenAPI 3.0 and Swagger 2.0*](#).

You can write the API definition in JSON or YAML format. Let us discuss this format briefly and focus on YAML as it is easier to read and maintain.

On the root level of the YAML document, we see fields like *swagger*, *info*, and others. Among these fields, ***swagger***, ***info***, and ***path*** are **required**. Their meaning is as follows:

swagger

specifies the version number. In our case a string value '2.0' is used as we are writing the definition conforming to the v2.0 specification.

info

defines metadata information related to the API. E.g., the API *version*, *title* and *description*, *termsOfService* if applicable, *contact* information and *license*, etc. Among these attributes, ***version*** and ***title*** are required while others are optional.

path

defines the actual endpoints of the exposed RESTful API service. Each endpoint has a *field pattern* as the key and a *Path Item Object* as the value. In this example we have defined */vc* and */vc/{id}* as the two service endpoints. They will be part of the final service URL, appended after the service *host* and *basePath*, which is explained later.

Let us focus on the *Path Item Object*. It contains one or more supported *operations* on the service endpoint. An *operation* is keyed by a valid HTTP operation verb, e.g., one of **get**, **put**, **post**, **delete**, or **patch**. It has a value of *Operation Object* that describes the operations in more detail.

The *Operation Object* will always **require** a *Response Object*. A *Response Object* has a *HTTP status code* as the key, e.g., **200** as successful return; **40X** as authentication and authorization related errors; and **50x** as other

server-side servers. It can also have a default response keyed by **default** for undeclared HTTP status return code. The *Response Object* value has a **required** *description* field, and if anything is returned, a *schema* indicating the object type to be returned, which could be a primitive type, e.g., *string*, or an *array* or customized *object*. In case of *object* or an *array* of *object*, use *\$ref* to point to the definition of the object. In this example, we have

\$ref: “#/definitions/VC”

to point to the *VC* definition in the *definitions* section in the same specification file, which will be explained later.

Besides the required field, the *Operation Object* **can** have *summary* and *description* to indicate what the operation is about; and *operationId* to uniquely identify the operation; and *consumes* and *produces* to indicate what MIME types it expects as input and for returns, e.g., *application/json* in most modern RESTful APIs. It can further specify what input parameter is expected using *parameters*, which requires a *name* and *in* fields. *name* specifies the name of the parameter, and *in* specifies from where to get the parameter and its possible values are *query*, *header*, *path*, *formData* or *body*. In this example in the */vc/{id}* path we obtain the *id* parameter from the URL path which has the *path* value. When the *in* has *path* as its value, the *required* field is required and has to be set as *true*; when the *in* has value other than *body*, a *type* field is required to specify the type of the parameter.

While the three root-level fields mentioned previously are required, in most cases, we also use other optional fields.

host

to indicate where the service is to be deployed, which could be *localhost* or a valid IP address or a DNS name of the host where the service is to be deployed. If another port number other than *80* is to be used, write the port number as well, e.g., *localhost:8080*.

schemas

to specify the transfer protocol, e.g., *HTTP* or *https*.

basePath

to specify the common base URL to be appended after the *host* to form the base path for all the endpoints, e.g., */api* or */api/1.0/*. In this example with the values specified we would have the final service endpoints *http://localhost:8080/api/vcs* and *http://localhost:8080/api/vc/{id}* by combining the *schemas*, *host*, *basePath* and *paths* values.

consumes and produces

can also be specified on the top level to specify the default MIME types of the input and return if most *paths* and the defined operations have the same.

definitions

as used in in the *paths* field, in order to point to a customized object definition with a *\$ref* keyword.

The *definitions* field contains the object definition of the customized objects involved in the API, similar to a class definition in any Object Oriented programming language. In this example, we defined a *VC* object, and hierarchically a *Node* object. Each object defined is a type of *Schema Object* in which many field could be used to specify the object (see details in the REF link at the top of the document), but the most frequently used ones are:

type

to specify the type and in the customized definition case the value is mostly *object*.

required

field to list the names of the required attributes of the object.

properties

has the detailed information of each attribute/property of the object, e.g., *type*, *format*. It also supports hierarchical object definition so a property of

one object could be another customized object defined elsewhere while using *schema* and *\$ref* keyword to point to the definition. In this example we have defined a *VC*, or virtual cluster, object, while it contains another object definition of

Node

as part of a cluster.

8.3.1.1 The Virtual Cluster example API Definition

8.3.1.1.1 Terminology

VC

A virtual cluster, which has one Front-End (FE) management node and multiple compute nodes. A VC object also has *id* and *name* to identify the VC, and *nnodes* to indicate how many compute nodes it has.

FE

A management node from which to access the compute nodes. The FE node usually connects to all the compute nodes via a private network.

Node

A computer node object that the info *ncores* to indicate the number of cores it has, and *ram* and *localdisk* to show the size of RAM and local disk storage.

8.3.1.1.2 Specification

```
swagger: "2.0"
info:
  version: "1.0.0"
  title: "A Virtual Cluster"
  description: "Virtual Cluster as a test of using swagger-2.0 specification and codegen"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "IU ISE software and system team"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/api"
schemes:
  - "http"
```

```

consumes:
- "application/json"
produces:
- "application/json"
paths:
/vcs:
  get:
    description: "Returns all VCs from the system that the user has access to"
    produces:
      - "application/json"
    responses:
      "200":
        description: "A list of VCs."
        schema:
          type: "array"
          items:
            $ref: "#/definitions/VC"
/vcs/{id}:
  get:
    description: "Returns all VCs from the system that the user has access to"
    operationId: getVCById
    parameters:
      - name: id
        in: path
        description: ID of VC to fetch
        required: true
        type: string
    produces:
      - "application/json"
    responses:
      "200":
        description: "The vc with the given id."
        schema:
          $ref: "#/definitions/VC"
      default:
        description: unexpected error
        schema:
          $ref: '#/definitions/Error'
definitions:
VC:
  type: "object"
  required:
    - "id"
    - "name"
    - "nnodes"
    - "FE"
    - "computes"
  properties:
    id:
      type: "string"
    name:
      type: "string"
    nnodes:
      type: "integer"
      format: "int64"
    FE:
      type: "object"
      schema:
        $ref: "#/definitions/Node"
    computes:
      type: "array"
      items:
        $ref: "#/definitions/Node"
  tag:
    type: "string"
Node:
  type: "object"
  required:
    - "ncores"
    - "ram"
    - "localdisk"
  properties:
    ncores:
      type: "integer"
      format: "int64"
    ram:
      type: "integer"
      format: "int64"
    localdisk:
      type: "integer"
      format: "int64"

```

```
Error:
  required:
  - code
  - message
  properties:
  code:
    type: integer
    format: int32
  message:
    type: string
```

8.3.1.2 References

[The official OpenAPI 2.0 Documentation](#)

8.3.2 OPENAPI REST SERVICE VIA INTROSPECTION

The simplest way to create an OpenAPI service is to use the connexion service and read in the specification from its YAML file. It is then introspected, and dynamically methods are created that are used for the implementation of the server.

The full example for this is available in

- <https://github.com/cloudmesh-community/nist/tree/master/examples/flask-connexion-swagger>

An extensive documentation is available at

- <https://media.readthedocs.org/pdf/connexion/latest/connexion.pdf>

This example returns the cpu information of a computer to dynamically demonstrate how simple it is to generate in python a REST service from an OpenAPI specification.

Our requirements.txt file includes

```
flask
connexion[swagger-ui]
```

as dependencies. The `server.py` file simply contains the following code:

```
from flask import jsonify
import connexion

# Create the application instance
app = connexion.App(__name__, specification_dir="./")
```

```

# Read the yaml file to configure the endpoints
app.add_api("cpu.yaml")

# create a URL route in our application for "/"
@app.route("/")
def home():
    msg = {"msg": "It's working!"}
    return jsonify(msg)

if __name__ == "__main__":
    app.run(port=8080, debug=True)

```

This will run our REST service under the assumption we have a `cpu.yaml` and a `cpu.py` files as our yaml file calls out methods from `cpu.py`

The YAML file looks as follows.

```

swagger: "2.0"
info:
  version: "0.0.1"
  title: "cpuinfo"
  description: "A simple service to get cpuinfo as an example of using swagger-2.0 specification and codegen"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "Cloudmesh REST Service Example"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/cloudmesh"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /cpu:
    get:
      tags:
        - CPU
      operationId: cpu.get_processor_name
      description: "Returns cpu information of the hosting server"
      produces:
        - "application/json"
      responses:
        "200":
          description: "CPU info"
          schema:
            $ref: "#/definitions/CPU"
definitions:
  CPU:
    type: "object"
    required:
      - "model"
    properties:
      model:
        type: "string"

```

Here we implement a get method and associate it with the URL `/cpu`. The `operationid`, defines the method that we call which, as we used the local directory, is included in the file `cpu.py`. This is controlled by the prefix in the operation id.

A straightforward function to return the CPU information is defined in `cpu.py` which we list next

```

import os, platform, subprocess, re
from flask import jsonify

def get_processor_name():
    if platform.system() == "Windows":
        p = platform.processor()
    elif platform.system() == "Darwin":
        command = "/usr/sbin/sysctl -n machdep.cpu.brand_string"
        p = subprocess.check_output(command, shell=True).strip().decode()
    elif platform.system() == "Linux":
        command = "cat /proc/cpuinfo"
        all_info = subprocess.check_output(command, shell=True).strip().decode()
        for line in all_info.split("\n"):
            if "model name" in line:
                p = re.sub(".*model name.*:", "", line, 1)
    else:
        p = "cannot find cpuinfo"
    pinfo = {"model": p}
    return jsonify(pinfo)

```

We have implemented this function to return a jsonified information from the dict pinfo.

To simplify working with this example, we also provide a makefile for OSX that allows us to call the server and the call to the server in two different terminals

```

define terminal
    osascript -e 'tell application "Terminal" to do script "cd $(PWD); $1"'
endef

install:
    pip install -r requirements.txt

demo:
    $(call terminal, python server.py)
    sleep 3
    @echo "===== "
    @echo "Get the info"
    @echo "===== "
    curl http://localhost:8080/cloudmesh/cpu
    @echo
    @echo "===== "

```

When we call

```
make demo
```

our demo is run.

8.3.2.1 Verification

It is important to be able to verify if a YAML file is correct. To identify this, the easiest method is to use the swagger editor. There is an online version available at:

- <https://editor.swagger.io/>

Go to the Web site, remove the current petstore example, and paste your YAML file in it. Debug messages are helping you to correct things.

A terminal-based command may also be helpful but is a bit difficult to read.

```
$ connexion run cpu.yaml --stub --debug
```

8.3.2.2 Mock service

In some cases, it may be useful to develop the API without having yet developed methods that you call with the OperationI. In this case, it is useful to run a mock service. YOU can invoke such a service with

```
$ connexion run cpu.yaml --mock=all -v
```

8.3.2.3 Exercise

OpenAPI.Conexion.1:

Modify the makefile, so it also works on ubuntu, but do not disable the ability to run it correctly on OSX. Tip use if's in makefiles base on the OS. You can look at the makefiles that create this book as an example. Find alternatives to starting a terminal in Linux.

OpenAPI.Conexion.2:

Modify the makefile, so it also works on Windows 10, but do not disable the ability to run it correctly on OSX. Tip use ifs in makefiles. You can look at the makefiles that create this book as example. Find alternatives to start a PowerShell or cmd.exe in windows. Maybe you need to use GitBash.

OpenAPI.Conexion.3:

Implement a swagger specification of an issue related to the NIST BDRA. Implement it. Please remember this could prepare you for a project good topics include:

- *virtual compute service interfacing with AWS, azure, google or*

OpenStack

- *virtual directory service interfacing with google drive, box, GitHub, iCloud, FTP, scp, and others*

As there are so many possibilities to contribute, come up in class with one specification and then implement it for different providers. The difficulty here is that it is not done for one IaaS, but for all of them and all can be integrated.

This exercise is typically growing to be part of your class project.

OpenAPI.Conexion.4:

Develop instructions on how to integrate the OpenAPI service framework in a WSGI based Web service. Chose a service you like so that the service could run in production.

OpenAPI.Conexion.5:

Develop instructions on how to integrate the OpenAPI service framework in Tornado so the service could run in production.

8.3.3 OPENAPI REST SERVICE VIA CODEGEN



[REST 36:02 Swagger](#)

In this section, we are discussing how to use OpenAPI 2.0 and Swagger Codegen to define and develop a REST Service.

We assume you have been familiar with the concept of REST service, OpenAPI, as discussed in section [Overview of Rest](#).

In the next section, we further look into the Swagger/OpenAPI 2.0 specification [Swagger Specification](#) and use a slightly more complex example of walking you through the design of a RESTful service following the OpenAPI 2.0 specifications.

We use a simple example to demonstrate the process of developing a REST

service with Swagger/OpenAPI 2.0 specification and the tools related to it. The general steps are:

- Step 1 (Section [Step 1: Define Your REST Service](#)). Define the REST service conforming to Swagger/OpenAPI 2.0 specification. It is a YAML document file with the basics of the REST service defined, e.g., what resources it has and what actions are supported.
- Step 2 (Section [Step 2: Server Side Stub Code Generation and Implementation](#)). Use Swagger Codegen to generate the server-side stub code. Fill in the actual implementation of the business logic portion in the code.
- Step 3 (Section [Step 3: Install and Run the REST Service](#)). Install the server-side code and run it. The service is then available.
- Step 4 (Section [Step 4: Generate Client Side Code and Verify](#)). Generate client-side code. Develop code to call the REST service. Install and run to verify.

8.3.3.1 Step 1: Define Your REST Service

In this example we define a simple REST service that returns the hosting server's basic CPU info. The example specification in yaml is as follows:

```
swagger: "2.0"
info:
  version: "0.0.1"
  title: "cpuinfo"
  description: "A simple service to get cpuinfo as an example of using swagger-2.0 specification and codegen"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "Cloudmesh REST Service Example"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/api"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /cpu:
    get:
      description: "Returns cpu information of the hosting server"
      produces:
        - "application/json"
      responses:
        "200":
          description: "CPU info"
          schema:
```



```
    $ref: "#/definitions/CPU"
definitions:
  CPU:
    type: "object"
    required:
      - "model"
    properties:
      model:
        type: "string"
```

8.3.3.2 Step 2: Server Side Stub Code Generation and Implementation

With the REST service having been defined, we can now generate the server-side stub code easily.

8.3.3.2.1 Setup the Codegen Environment

You need to [install the Swagger Codegen tool](#) if not yet done so. For macOS we recommend that you use the homebrew install via

```
$ brew install swagger-codegen
```

On Ubuntu you can install swagger as follows (update the version as needed):

```
$ mkdir ~/swagger
$ cd ~/swagger
$ wget https://oss.sonatype.org/content/repositories/releases/io/swagger/swagger-codegen-cli/2.3.1/swagger-codegen-cli-2.
$ alias swagger-codegen="java -jar ~/swagger/swagger-codegen-cli-2.3.1.jar"
```

Add the alias to your `.bashrc` or `.bash_profile` file. After you start a new terminal you can use in that terminal now the command

```
swagger-codegen
```

For other platforms, you can just use the `.jar` file, which can be downloaded from [this link](#).

Also, make sure Java is installed in your system. To have a well-defined location, we recommend that you place the code in the directory `~/cloudmesh`. In this directory, you also place the file `cpu.yaml`.

8.3.3.2.2 Generate Server Stub Code

After you have the codegen tool ready, and with Java 7 or 8 installed in your system, you can run the following to generate the server-side stub code:

```
$ swagger-codegen generate \  
-i ~/cloudmesh/cpu.yaml \  
-l python-flask \  
-o ~/cloudmesh/swagger_example/server/cpu/flaskConnexion \  
-D supportPython2=true
```

or if you have not created an alias

```
$ java -jar swagger-codegen-cli.jar generate \  
-i ~/cloudmesh/cpu.yaml \  
-l python-flask \  
-o ~/cloudmesh/swagger_example/server/cpu/flaskConnexion \  
-D supportPython2=true
```

In the specified directory under *flaskConnexion*, you find the generated python flask code with python 2 compatibility. It is best to place the swagger code under the directory `~/cloudmesh` to have a location where you can easily find it. If you want to use Python 3 make sure to choose the appropriate option. To switch between python 2 and Python 3 we recommend that you use a python virtual environment.

8.3.3.2.3 Fill in the actual implementation

Under the *flaskConnexion* directory, you find a *swagger_server* directory, under which you find directories with *models* defined and *controllers* code stub resides. The models' code are generated from the definition in Step 1. On the controller code, though, we need to fill in the actual implementation. You may see a `default_controller.py` file under the *controllers* directory in which the resource and action is defined but yet to be implemented. In our example, you find such a function definition which we list next:

```
def cpu_get(): # noqa: E501  
    """cpu_get  
  
    Returns cpu info of the hosting server # noqa: E501  
  
    :rtype: CPU  
    """  
    return 'do some magic!'
```

Please note the `do some magic!` string at the return of the function. This ought to be replaced with actual implementation of what you would like your REST call to be really doing. In reality, this could be some call to a backend database or datastore, a call to another API; or even calling another REST service from another location. In this example, we simply retrieve the CPU model information from the hosting server through a simple python call to illustrate this principle. Thus you can define the following code:

```
import os, platform, subprocess, re

def get_processor_name():
    if platform.system() == "Windows":
        return platform.processor()
    elif platform.system() == "Darwin":
        command = "/usr/sbin/sysctl -n machdep.cpu.brand_string"
        return subprocess.check_output(command, shell=True).strip()
    elif platform.system() == "Linux":
        command = "cat /proc/cpuinfo"
        all_info = subprocess.check_output(command, shell=True).strip()
        for line in all_info.split("\n"):
            if "model name" in line:
                return re.sub(".*model name.*:", "", line, 1)
        return "cannot find cpuinfo"
```

And then change the `cpu_get()` function to the following:

```
def cpu_get(): # noqa: E501
    """cpu_get

    Returns CPU info of the hosting server # noqa: E501

    :rtype: CPU
    """
    return CPU(get_processor_name())
```

Please note we are returning a CPU object as defined in the API and later generated by the codegen tool in the *models* directory.

It is best *not* to include the definition of `get_processor_name()` in the same file as you see the definition of `cpu_get()`. The reason for this is that in case you need to regenerate the code, your modified code will naturally be overwritten. Thus, to minimize the changes, we do recommend to maintain that portion in a different filename and import the function to keep the modifications small.

At this step, we have completed the server-side code development.

8.3.3.3 Step 3: Install and Run the REST Service:

Now we can install and run the REST service. It is strongly recommended that you run this in a pyenv or a virtualenv environment.

8.3.3.3.1 Start a virtualenv:

In case you are not using pyenv, please use virtual env as follows:

```
$ virtualenv RESTServer
$ source RESTServer/bin/activate
```

8.3.3.3.2 Make sure you have the latest pip:

```
$ pip install -U pip
```

8.3.3.3 Install the requirements of the server side code:

```
$ cd ~/cloudmesh/swagger_example/server/cpu/flaskConnexion  
$ pip install -r requirements.txt
```

8.3.3.4 Install the server-side code package:

Under the same directory, run:

```
$ python setup.py install
```

8.3.3.5 Run the service

Under the same directory:

```
$ python -m swagger_server
```

You should see a message like this:

```
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

8.3.3.6 Verify the service using a web browser:

Open a web browser and visit:

- <http://localhost:8080/api/cpu>

to see if it returns a JSON object with CPU model info in it.

Assignment: How would you verify that your service works with a `curl` call?

8.3.3.4 Step 4: Generate Client-Side Code and Verify

In addition to the server-side code, swagger can also create a client side code.

8.3.3.4.1 Client-side code generation:

Generate the client-side code in a similar fashion as we did for the server-side code:

```
$ java -jar swagger-codegen-cli.jar generate \
```

```
-i ~/cloudmesh/cpu.yaml \  
-l python \  
-o ~/cloudmesh/swagger_example/client/cpu \  
-D supportPython2=true
```

8.3.3.4.2 Install the client-side code package:

Although we could have installed the client in the same python pyenv or virtualenv, we showcase here that it can be installed in a completely different environment. That would make it even possible to use a Python 3 based client and a Python 2 based server, showcasing interoperability between python versions (although we just use python 2 here). Thus we create a new python virtual environment and conduct our install.

```
$ virtualenv RESTClient  
$ source RESTClient/bin/activate  
$ pip install -U pip  
$ cd swagger_example/client/cpu  
$ pip install -r requirements.txt  
$ python setup.py install
```

8.3.3.4.3 Using the client API to interact with the REST service

Under the directory *swagger_example/client/cpu* you will find a README.md file which serves as API documentation with the example client code in it. E.g., if we save the following code into a .py file:

```
from __future__ import print_function  
import time  
import swagger_client  
from swagger_client.rest import ApiException  
from pprint import pprint  
# create an instance of the API class  
api_instance = swagger_client.DefaultApi()  
  
try:  
    api_response = api_instance.cpu_get()  
    pprint(api_response)  
except ApiException as e:  
    print("Exception when calling DefaultApi->cpu_get: %s\n" % e)
```

We can then run this code to verify the calling to the REST service actually works. We are expecting to see a return similar to this:

```
{'model': 'Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz'}
```

Obviously, we could have applied additional cleanup of the information returned by the python code, such as removing duplicated spaces.

8.3.3.5 Towards a Distributed Client Server

Although we develop and run the example on one localhost machine, you can separate the process into two separate machines. E.g., on a server with external IP or even DNS name to deploy the server-side code, and on a local laptop or workstation to deploy the client-side code. In this case, please make changes on the API definition accordingly, e.g., the **host** value.

8.4 AUTOMATED REST SERVICE GENERATION WITH CONEXION FOR OPENAPI 3.0

8.4.1 REST SPECIFICATIONS

RESTful services have undoubtedly become the de-facto software architectural style for creating Web services. A REST API specification would define the attributes and constraints to be used in the web service. There have been multiple specifications that have been in use such as [OpenAPI \(formally called Swagger\) \[58\]](#), [RAML \[59\]](#), [tinyspec \[60\]](#), and [API Blueprint \[61\]](#).

8.4.1.1 OPENAPI

Over the years, Open API specification has become the most popular with a much larger community behind it. Therefore, this section would focus on the latest specification, [OpenAPI 3.0 \(OAS 3.0\) \[62\]](#).

According to the [OAS documentation \[63\]](#), it allows users to,

- Describe endpoints and operations on each endpoint
- Specify operation parameters, inputs, and outputs for each operation
- Handle authentication
- Describe contact, license, terms of use and other information

API specifications can be written in YAML or JSON. OAS also comes with a rich toolkit that includes [Swagger Editor \[64\]](#), [Swagger UI \[65\]](#) and [Swagger Codegen \[66\]](#), that creates an end-to-end development environment, even for the users who are new to OAS.

Section [OpenAPI Specification](#) details more on the OAS 2.0 specification.

8.4.1.1.1 Open API 3.0 Specification (OAS 3.0)

OAS 3.0 key definitions are depicted in [Figure 16](#)

OpenAPI 3

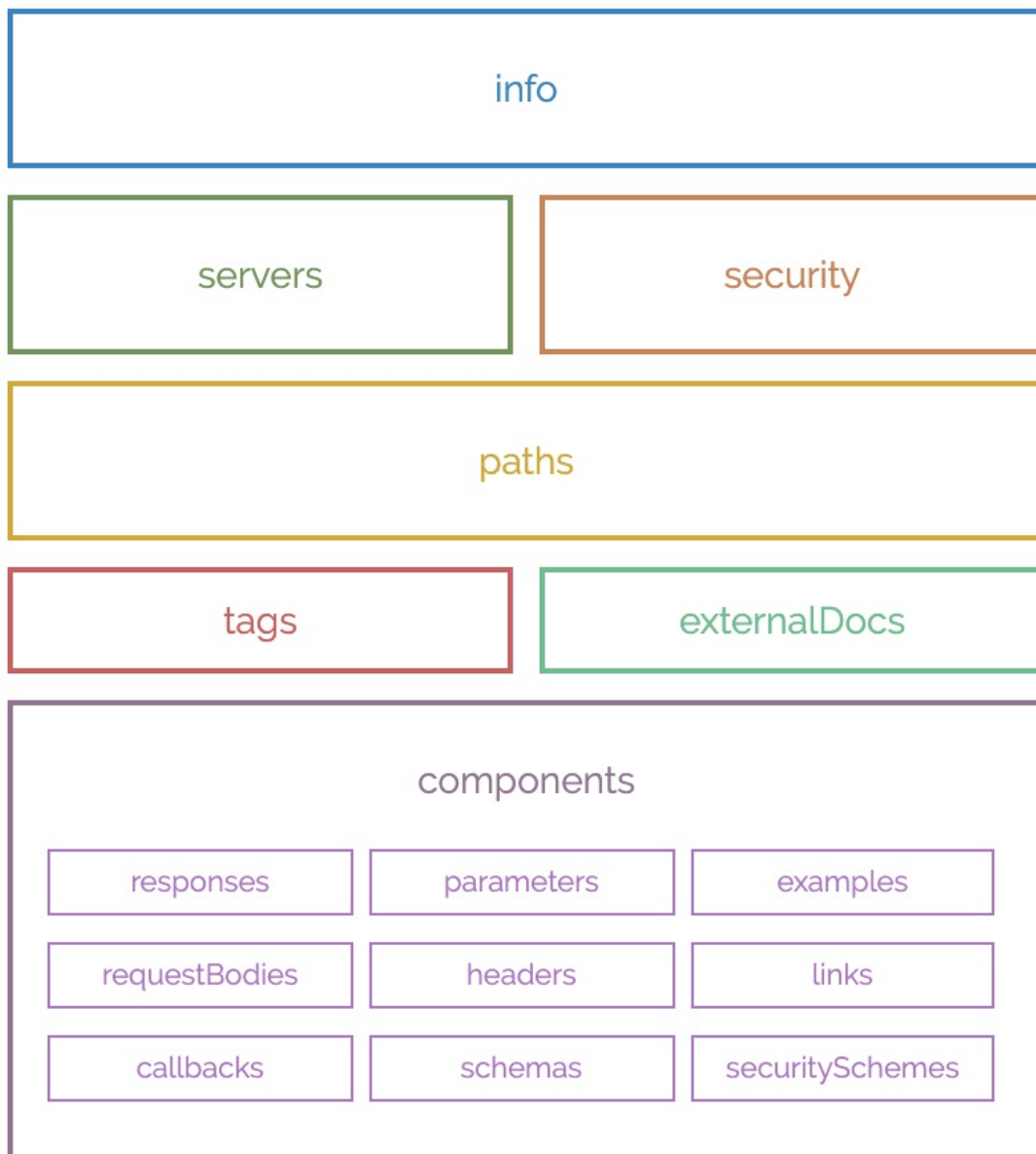


Figure 16: Components of OAS 3.0 [Source](#)

The basic structure of the definitions would look like this. The sample REST service exposes `http://localhost:8080/cloudmesh` basepath. Under that base path, an endpoint has been exposed as `cloudmesh/cpu`, which would return CPU information of the server. It uses a predefined schema to return the results, which is defined under the `components/schemas`. See the Section [OpenAPI REST](#)

[Service via Introspection](#) for the detailed example.

```
openapi: 3.0.2
info:
  title: cpuinfo
  description: A simple service to get cpuinfo as an example of using OpenAPI 3.0
  license:
    name: Apache 2.0
    version: 0.0.1

servers:
  - url: http://localhost:8080/cloudmesh

paths:
  /cpu:
    get:
      summary: Returns cpu information of the hosting server
      operationId: cpu.get_processor_name
      responses:
        '200':
          description: cpu info
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/cpu"

components:
  schemas:
    cpu:
      type: "object"
      required:
        - "model"
      properties:
        model:
          type: "string"
```

8.4.1.1.1.1 Definitions

Metadata:

OAS 3.0 requires a specification definition at the start under the *openapi* field.

```
openapi: 3.0.2
```

Next, metadata can be specified under *info* field such as *title*, *version*, *description*, etc. Additionally, license, contact information can also be specified. *title* and *version* are mandatory fields under *info*.

```
info:
  title: cpuinfo
  description:
    A simple service to get cpuinfo as an example of using OpenAPI 3.0
  license:
    name: Apache 2.0
  version: 0.0.1
```

Servers:

The *servers* section defines the server URLs with the basepath. Optionally, a *description* can be added.

```
servers:
  - url: http://localhost:8080/cloudmesh
    description: Cloudmesh server basepath
```

Paths:

The *paths* section specifies all the endpoints exposed by the API and the HTTP operations supported by these endpoints.

```
paths:
  /cpu:
    get:
      summary: Returns cpu information of the hosting server
      operationId: cpu.get_processor_name
      responses:
        '200':
          description: cpu info
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/cpu"
```

Operation ID:

When using introspection for REST services (using Connexion), we would need to point to the operation that would ultimately carry out the request. This operation is specified by the *operationID*.

```
...
paths:
  /cpu:
...
    operationId: cpu.get_processor_name
```

Parameters:

If the service endpoint accepts URL parameters (ex: */cpu/cache/{cache_level}* or */cpu?arch=x86*), headers or cookies, those can also be specified under a *path*.

```
paths:
  /cpu/cache/{cache_level}:
    get:
      summary: Returns the cache size of the specified level
      parameters:
        - name: cache_level
          in: path
          required: true
          description: Parameter description in CommonMark or HTML.
          schema:
            type: string
            minimum: 1
      responses:
        '200':
          description: OK
```

Request Body:

When a request is sent with a body, such as *POST*, that is specified in the

requestBody under a *path*.

```
paths:
  /upload:
    post:
      summary: upload input
      requestBody:
        content:
          multipart/form-data:
            schema:
              type: object
              properties:
                file:
                  type: string
                  format: binary
      responses:
        '200':
          description: OK
```

Responses:

For each path, *responses* can be specified with the corresponding status codes such as 200 OK or 404 Not Found. A response may return a response body, that can be defined under *content*.

```
...
paths:
  /cpu:
...
  responses:
    '200':
      description: cpu info
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/cpu"
```

Schemas:

The *components/schemas* section allows users to define schemas for inputs or outputs that can be referenced via *\$ref* tag.

```
...
paths:
  /cpu:
...
  responses:
    '200':
      description: cpu info
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/cpu"
...

components:
  schemas:
    cpu:
      type: "object"
      required:
        - "model"
      properties:
        model:
          type: "string"
```

Authentication:

Under the *components* sections, *securitySchemes* can also be specified such as Basic Auth.

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic

security:
  - BasicAuth: []
```

According to the current OAS 3.0, supported authentication methods are,

- HTTP authentication: Basic, Bearer, and others.
- API key as a header or query parameter or in cookies
- OAuth2
- OpenID Connect Discovery

8.4.1.2 RAML

[RAML](#) [59] (RESTful API Modeling Language) is a specification proposed in 2013, and it is based on YAML format. The specification is managed by the RAML Worker Group. It initially came out as a proprietary vendor language (specification) but later was open-sourced. As of Sep 2019, the latest specification is [RAML 1.0](#) [67].

Following is an example RAML specification from the [RAML.org](#)

```
##%RAML 1.0
title: Hello world # required title

/helloworld: # optional resource
  get: # HTTP method declaration
    responses: # declare a response
      200: # HTTP status code
        body: # declare the content of the response
          application/json: # media type
            type: | # structural definition of a response (schema or type)
              {
                "title": "Hello world Response",
                "type": "object",
                "properties": {
                  "message": {
                    "type": "string"
                  }
                }
              }
          example: | # example of how a response looks
            {
              "message": "Hello world"
            }
          }
```

In the current context, the industry seems to be adopting OpenAPI more than RAML. Consequently, some of the main contributors of RAML, such as MuleSoft, have joined the Open API Initiative since 2017. Hence, it is safe to conclude that Open API would be the dominant REST API specification in the web services domain.

Furthermore, there are tools available to switch between the specifications, such as [RAML Web API Parser](#), which can convert RAML to Open API and vice-versa.

8.4.1.3 API Blueprint

[API Blueprint](#) [61] is another specification available currently which uses Markdown syntax. As of Sep 2019 the latest version available is 1A-rev9.

8.4.1.4 JsonAPI

As the name suggests, [JSON API](#)[68] attempts to leverage web services specifications using JSON format. It reached a stable version 1.0 in May 2015, but there have been no revisions since then.

8.4.1.5 Tinyspec

[Tinyspec](#)[60] is a lightweight alternative to Open API. It has not been able to enter into the mainstream.

8.4.1.6 Tools

There are a number of tools available in the REST Web services specification domain. A classification of REST tools can be found in the Section `sec:rest_classification?` section.

8.4.1.6.1 Connexion

[Connexion](#)[69] is one such tool that is based on Open API, and it is widely used in the Python environment. This framework allows users to define Web services

in Open API and then map those services to Python functions conveniently. We would be using Connexion when we create REST services using introspection [Section 8.4.2](#).

Here is an example from the [Connexion official website](#) [69].

```
openapi: "3.0.0"

info:
  title: Hello World
  version: "1.0"
servers:
  - url: http://localhost:9090/v1.0

paths:
  /greeting/{name}:
    post:
      summary: Generate greeting
      description: Generates a greeting message.
      operationId: hello.post_greeting
      responses:
        200:
          description: greeting response
          content:
            text/plain:
              schema:
                type: string
                example: "hello dave!"
      parameters:
        - name: name
          in: path
          description: Name of the person to greet.
          required: true
          schema:
            type: string
            example: "dave"
```

This service would map to the following *post_greeting* Python function.

```
import connexion

def post_greeting(name: str) -> str:
    return 'Hello {name}'.format(name=name)

if __name__ == '__main__':
    app = connexion.FlaskApp(__name__, port=9090, specification_dir='openapi/')
    app.add_api('helloworld-api.yaml', arguments={'title': 'Hello World Example'})
    app.run()
```

8.4.2 OPENAPI 3.0 REST SERVICE VIA INTROSPECTION

The simplest way to create an OpenAPI service is to use the connexion service and read in the specification from its YAML file. It then introspects and dynamically creates methods that are used for the implementation of the server.

The full example for this is available in

- <https://github.com/cloudmesh->

[community/book/tree/master/examples/rest/cpu](https://github.com/connexion/connexion/tree/master/examples/rest/cpu)

An extensive documentation is available at

- <https://media.readthedocs.org/pdf/connexion/latest/connexion.pdf>

This example returns the cpu information of a computer to dynamically demonstrate how simple it is to generate in python a REST service from an OpenAPI specification.

Our requirements.txt file includes

```
flask
connexion[swagger-ui]
```

as dependencies. The `server.py` file simply contains the following code:

```
from flask import jsonify
import connexion

# Create the application instance
app = connexion.App(__name__, specification_dir="./")

# Read the yaml file to configure the endpoints
app.add_api("cpu.yaml")

# create a URL route in our application for "/"
@app.route("/")
def home():
    msg = {"msg": "It's working!"}
    return jsonify(msg)

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

This will run our REST service under the assumption we have a `cpu.yaml` and a `cpu.py` files as our YAML file calls out methods from `cpu.py`

The YAML file looks as follows.

```
openapi: 3.0.2
info:
  title: cpuinfo
  description: A simple service to get cpuinfo as an example of using OpenAPI 3.0
  license:
    name: Apache 2.0
  version: 0.0.1

servers:
  - url: http://localhost:8080/cloudmesh

paths:
  /cpu:
    get:
      summary: Returns cpu information of the hosting server
      operationId: cpu.get_processor_name
      responses:
```

```

'200':
  description: cpu info
  content:
    application/json:
      schema:
        $ref: "#/components/schemas/cpu"

components:
  schemas:
    cpu:
      type: "object"
      required:
        - "model"
      properties:
        model:
          type: "string"

```

Here we implement a get method and associate it with the URL /cpu. The operationid, defines the method that we call which, as we used the local directory, is included in the file `cpu.py`. This is controlled by the prefix in the operation id.

A straightforward function to return the CPU information is defined in `cpu.py` which we list next

```

import os, platform, subprocess, re
from flask import jsonify

def get_processor_name():
    if platform.system() == "Windows":
        p = platform.processor()
    elif platform.system() == "Darwin":
        command = "/usr/sbin/sysctl -n machdep.cpu.brand_string"
        p = subprocess.check_output(command, shell=True).strip().decode()
    elif platform.system() == "Linux":
        command = "cat /proc/cpuinfo"
        all_info = subprocess.check_output(command, shell=True).strip().decode()
        for line in all_info.split("\n"):
            if "model name" in line:
                p = re.sub(".*model name.*:", "", line, 1)
    else:
        p = "cannot find cpuinfo"
    pinfo = {"model": p}
    return jsonify(pinfo)

```

We have implemented this function to return a *jsonified* information from the dict pinfo.

To simplify working with this example, we also provide a makefile for OSX that allows us to call the server and the call to the server in two different terminals

```

define terminal
  osascript -e 'tell application "Terminal" to do script "cd $(PWD); $1"'
endef

install:
  pip install -r requirements.txt

demo:
  $(call terminal, python server.py)
  sleep 3
  @echo "=====
  @echo "Get the info"
  @echo "=====

```



```
curl http://localhost:8080/cloudmesh/cpu
@echo
@echo "====="
```

When we call

```
make demo
```

our demo is run.

8.4.2.1 Verification

It is essential to be able to verify if a YAML file is correct. To identify this, the easiest method is to use the swagger editor. There is an online version available at:

- <https://editor.swagger.io/>

Go to the Web site, remove the current petstore example, and paste your YAML file in it. Debug messages are helping you to correct things.

A terminal-based command may also be helpful but is a bit difficult to read.

```
$ connexion run cpu.yaml --stub --debug
```

8.4.2.2 Swagger-UI

Swagger comes with a convenient UI to invoke REST API calls using the Web browser rather than relying on the curl commands.

Once the request and response definitions are correctly specified, you can start the server by,

```
$ python server.py
```

Then the UI would also be spawned under the service URL *http://[service url]/ui/*

Example: <http://localhost:8080/cloudmesh/ui/>

8.4.2.3 Mock service

In some cases, it may be useful to develop the API without having yet developed methods that you call with the OperationI. In this case, it is useful to run a mock service. You can invoke such a service with

```
$ connexion run cpu.yaml --mock=all -v
```

8.4.2.4 Exercise

OpenAPI.Conexion.1:

Modify the makefile, so it also works on ubuntu, but do not disable the ability to run it correctly on OSX. Tip use if's in makefiles base on the OS. You can look at the makefiles that create this book as an example. Find alternatives to starting a terminal in Linux.

OpenAPI.Conexion.2:

Modify the makefile, so it also works on Windows 10, but do not disable the ability to run it correctly on OSX. Tip use ifs in makefiles. You can look at the makefiles that create this book as example. Find alternatives to start a PowerShell or cmd.exe in windows. Maybe you need to use GitBash.

OpenAPI.Conexion.3:

Implement a swagger specification of an issue related to the NIST BDRA. Implement it. Please remember this could prepare you for a project good topics include:

- *virtual compute service interfacing with AWS, Azure, Google or OpenStack*
- *virtual directory service interfacing with google drive, box, GitHub, iCloud, FTP, scp, and others*

As there are so many possibilities to contribute, come up in class with one specification and then implement it for different providers. The difficulty here is that it is not done for one IaaS, but for all of them and all can be integrated.

This exercise is typically growing to be part of your class project.


OpenAPI.Conexion.4:

Develop instructions on how to integrate the OpenAPI service framework in a WSGI based Web service. Chose a service you like so that the service could run in production.

OpenAPI.Conexion.5:

Develop instructions on how to integrate the OpenAPI service framework in Tornado so the service could run in production.

8.4.3 REST AI SERVICES EXAMPLE

Now we present a more involved example which uses OpenAPI 3.0 specification to invoke  [K-means Clustering](#) routine in scikit-learn ???. Scikit-learn k-means user-guide can be found Scikit-learn K-Means package [70].

This involves the following.

- Upload a file with points to create the k-means clustering model.
- Method to call scikit-learn KMeans module
- Upload a file with points that need to be predicted and return a file with the predicted cluster IDs.
- Additionally, scikit-learn KMeans module provides routines to get the cluster centers, labels, etc. which can also be exposed as REST services.

To create the REST services, we would be using OpenAPI 3.0 REST service via introspection.

8.4.3.1 Service Endpoints/ Paths

8.4.3.1.1 Path *kmeans/upload*

A POST request with a file containing points to create the k-means clustering model. POST content would be *multipart/form-data*.

For an example consider the following 6 points in XY dimensions,

```
1, 2
1, 4
1, 0
10, 2
10, 4
10, 0
```

Curl command:

```
$ curl -X POST "http://localhost:8080/kmeans/upload" \
  -H "accept: application/json" \
  -H "Content-Type: multipart/form-data" \
  -F "file=@model.csv;type=text/csv"
```

Service implementation would look like this. The file content is received as a [werkzeug.datastructures.FileStorage](#) subject in *Flask*, which can be used to stream into the filesystem. The backend keeps two dicts to map Job ID to file and vice-versa (*inputs* and *inputs_r*).

```
def upload_file(file=None):
    filename = file.filename

    in_file = INPUT_DIR + '/' + filename
    if not os.path.exists(in_file):
        file.save(in_file) # save the input file

    if in_file not in inputs_r:
        job_id = len(inputs)
        inputs.update({job_id: in_file})
        inputs_r.update({in_file: job_id})
    else:
        job_id = inputs_r[in_file]

    return jsonify({'job_id': job_id, 'filename': filename})
```

If the request is successful, a *JSON* will be returned with the file name and the associated job ID. Job ID can be considered ID that would connect, inputs to the models, and the predicted outputs.

```
{
  "filename": "model.csv",
  "job_id": 0
}
```

8.4.3.1.2 Path *kmeans/fit*

A POST request with a *JSON* body containing Job ID and model parameters that need to pass on to the scikit-learn KMeans model initialization such as, number of clusters (*n_clusters*), maximum iterations (*max_iter*), etc.

Example:

```
{
```

```
"job_id": 0,
"model_params": {
  "n_clusters": 3
}
}
```

curl command:

```
$ curl -X POST "http://localhost:8080/kmeans/fit" \
-H "accept: text/csv" \
-H "Content-Type: application/json" \
-d "{\"job_id\":0,\"model_params\":{\"n_clusters\":3}}"
```

Service implementation looks like this. POST request body will be populated as a dict and passed on to the method by Flask (*body*). Once the model is fitted, it will be put into an in-memory dict (*models*) against its Job ID. Labels will be written to disk as a file, and the content will be returned as a CSV.

```
def kmeans_fit(body):
    print(body)

    job_id = body['job_id']

    if job_id not in inputs or not os.path.exists(inputs[job_id]):
        abort(500, "input file missing for job id " + str(job_id))
        return
    in_file = inputs[job_id]

    X = np.genfromtxt(in_file, delimiter=",") # create the model

    params = dict(default_model_params)
    params.update(body['model_params'])

    kmeans = KMeans(**params).fit(X)

    models.update({job_id: kmeans}) # add the model in to the dict

    labels = OUTPUT_DIR + "/" + str(job_id) + ".labels"
    np.savetxt(labels, kmeans.labels_, delimiter=",")

    return send_file(labels)
```

The response CSV file will be returned with the corresponding labels for the input points.

```
1.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
0.0000000000000000e+00
0.0000000000000000e+00
2.0000000000000000e+00
```

8.4.3.1.3 Path *kmeans/predict*

A POST request with a file containing the points to be predicted and the corresponding Job ID as *multipart/form-data*.

```
job_id=0
```

Points to be predicted

```
0, 0  
12, 3
```

curl command:

```
$ curl -X POST "http://localhost:8080/kmeans/predict" \  
-H "accept: text/csv" \  
-H "Content-Type: multipart/form-data" \  
-F "job_id=0" \  
-F "file=@predict.csv;type=text/csv"
```

Service implementation looks like this. Note that there is a strange behavior in *Flask* with *Connexion* where the file content will be passed on to the *file* object as a [werkzeug.datastructures.FileStorage](#) object, but the Job ID is passed as a dict to *body* object.

```
def kmeans_predict(body, file=None):  
    job_id = int(body['job_id'])  
  
    if job_id in models:  
        p_file = OUTPUT_DIR + '/' + str(job_id) + '.p'  
        file.save(p_file)  
  
        p = np.genfromtxt(p_file, delimiter=',') # read the predictions  
  
        result = models[job_id].predict(p)  
  
        print(result)  
  
        res_file = OUTPUT_DIR + "/" + str(job_id) + ".out"  
        np.savetxt(res_file, result, delimiter=",")  
  
        return send_file(res_file)  
  
    else:  
        abort(500, "model not found for job id " + str(job_id))  
    return
```

The response would send out the corresponding labels of the passing points as a CSV file.

```
1.0000000000000000e+00  
0.0000000000000000e+00
```

8.4.3.2 Files

Files of this example can be found [here](#).

- Open API 3 service definitions - [api.yaml](#)
- Flask server - [server.py](#)
- Kmeans service implementation - [kmeans.py](#)

- Python requirements - [requirements.txt](#)
- Example files [model.csv](#) and [predict.csv](#)

8.4.3.3 Running the example

- Go to the example directory.
- Activate the Python3 venv used for *Cloudmesh*
- Install requirements.txt

```
$ pip install -r requirements.txt
```

- Start the server

```
$ python server.py
```

- Upload a file

```
$ curl -X POST "http://localhost:8080/kmeans/upload" \
-H "accept: application/json" \
-H "Content-Type: multipart/form-data" \
-F "file=@model.csv;type=text/csv"
```

- Fit the kmeans model

```
$ curl -X POST "http://localhost:8080/kmeans/fit" \
-H "accept: text/csv" \
-H "Content-Type: application/json" \
-d "{\"job_id\":0,\"model_params\":{\"n_clusters\":3}}"
```

- Predict using the fitted kmeans model

```
$ curl -X POST "http://localhost:8080/kmeans/predict" \
-H "accept: text/csv" \
-H "Content-Type: multipart/form-data" \
-F "job_id=0" \
-F "file=@predict.csv;type=text/csv"
```

- Additionally, you can access the Swagger UI for *kmeans* service in your Flask server from [here](#)

8.4.3.4 Notes

- Above services can easily be combined in the backend to accept a model file, together with a prediction input
- File and to return the predicted output file (synchronous operation). But usually, we can expect AI jobs to be long-running, hence the services would

need to be handled asynchronously.

- Additionally, once a model is fitted, users should be able to reuse the model for multiple predictions. Hence it is sensible to separate model fitting and predictions into separate services.

9 REFERENCES

The following references are collected automatically from multiple sources. 

- [1] “RAML.” [Online]. Available: <https://raml.org/>
- [2] Sagger, Available: <https://swagger.io/blog/news/mulesoft-joins-the-openapi-initiative/>
- [3] “API blueprint.” [Online]. Available: <https://apiblueprint.org/>
- [4] K. Sandoval, “Top specification formats for rest apis.” Web Page, Sep-2015 [Online]. Available: <https://nordicapis.com/top-specification-formats-for-rest-apis/>
- [5] P. Gouras, “The role of s-cones in human vision,” *Documenta ophthalmologica*, vol. 106, no. 1, pp. 5–11, 2003.
- [6] “Rods and cones.” Website [Online]. Available: https://www.cis.rit.edu/people/faculty/montag/vandplite/pages/chap_9/ch9p1.htm
- [7] D. Mustafi, A. H. Engel, and K. Palczewski, “Structure of cone photoreceptors,” *Progress in retinal and eye research*, vol. 28, no. 4, pp. 289–302, 2009.
- [8] M. Simoneau and J. Price, “Neural networks provide solutions to real-world problems: Powerful new algorithms to explore, classify, and identify patterns in data.” Website, 1998 [Online]. Available: <https://www.mathworks.com/company/newsletters/articles/neural-networks-provide-solutions-to-real-world-problems-powerful-new-algorithms-to-explore-classify-and-identify-patterns-in-data.html>
- [9] F. Manessi and A. Rozza, “Learning combinations of activation functions,” *CoRR*, vol. abs/1801.09403, 2018 [Online]. Available: <http://arxiv.org/abs/1801.09403>
- [10] “Python rest apis with flask, connexion, and sqlalchemy.” Website, 2018

[Online]. Available: <https://realpython.com/flask-connexion-rest-api/>

[11] J. Drayer, S. L. Shapiro, B. Dwyer, A. L. Morse, and J. White, “The effects of fantasy football participation on nfl consumption: A qualitative analysis,” *Sport Management Review*, vol. 13, no. 2, pp. 129–141, 2010.

[12] B. Gagnon, “What is the most important combine event for each nfl position?” 2018 [Online]. Available: <https://bleacherreport.com/articles/2760924-what-is-the-most-important-combine-event-for-each-nfl-position#slide6>

[13] T. Srivastava, “Introduction to k-nearest neighbors: Simplified (with implementation in python),” 2018 [Online]. Available: <https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/>

[14] C. Pautasso, “RESTful web service composition with bpel for rest,” *Data & Knowledge Engineering*, vol. 68, no. 9, pp. 851–866, 2009.

[15] E. Novoseltseva, “Top 10 benefits of docker,” 2017 [Online]. Available: <https://dzone.com/articles/top-10-benefits-of-using-docker>

[16] SPORTTECHIE, “Sensors are taking over sports,” 2016.

[17] U. Von Luxburg, “A tutorial on spectral clustering,” *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[18] I. K. Fodor, “A survey of dimension reduction techniques,” Lawrence Livermore National Lab., CA (US), 2002.

[19] Available: http://www.fon.hum.uva.nl/praat/manual/k-means_clustering_1_How_does_k-means_clustering_work_.html

[20] *scikit*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.completeness_score.html

[21] *scikit*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.homogeneity_score.html#skle

- [22] “Harmonic mean,” *Wikipedia*. Wikimedia Foundation, Mar-2019 [Online]. Available: https://en.wikipedia.org/wiki/Harmonic_mean
- [23] *scikit*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.homogeneity_completeness_
- [24] “Kmeans clustering and 3D plotting,” *Kaggle*. [Online]. Available: <https://www.kaggle.com/timi01/k-means-clustering-and-3d-plotting>
- [25] K. Nishida, “Visualizing k-means clustering results to understand the characteristics of clusters better,” *learn data science*. learn data science, Nov-2018 [Online]. Available: <https://blog.exploratory.io/visualizing-k-means-clustering-results-to-understand-the-characteristics-of-clusters-better-b0226fb3dd10>
- [26] G. Seif, “5 quick and easy data visualizations in python with code,” *Towards Data Science*. Towards Data Science, Mar-2018 [Online]. Available: <https://towardsdatascience.com/5-quick-and-easy-data-visualizations-in-python-with-code-a2284bae952f>
- [27] N. Sharma, “Ways to detect and remove the outliers,” *Towards Data Science*. Towards Data Science, May-2018 [Online]. Available: <https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba>
- [28] “Multiclass classification,” *Wikipedia*. Wikimedia Foundation, Apr-2019 [Online]. Available: https://en.wikipedia.org/wiki/Multiclass_classification
- [29] AbhishekAbhishek, “Training logistic regression using scikit learn for multi-class classification,” *Stack Overflow*. [Online]. Available: <https://stackoverflow.com/questions/32074630/training-logistic-regression-using-scikit-learn-for-multi-class-classification>
- [30] “Decision tree learning,” *Wikipedia*. Wikimedia Foundation, Apr-2019 [Online]. Available: https://en.wikipedia.org/wiki/Decision_tree_learning
- [31] “K-nearest neighbors algorithm,” *Wikipedia*. Wikimedia Foundation, Apr-2019 [Online]. Available: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

- [32] Available: <https://multivariatestatsjl.readthedocs.io/en/latest/mclda.html>
- [33] “Linear discriminant analysis,” *Wikipedia*. Wikimedia Foundation, Apr-2019 [Online]. Available: https://en.wikipedia.org/wiki/Linear_discriminant_analysis#Linear_discriminant
- [34] “Naive bayes for machine learning,” *Machine Learning Mastery*. Sep-2016 [Online]. Available: <https://machinelearningmastery.com/naive-bayes-for-machine-learning/>
- [35] “1.9. Naive bayes,” *scikit*. [Online]. Available: https://scikit-learn.org/stable/modules/naive_bayes.html
- [36] *scikit*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
- [37] “SVM multi-class classification,” *Apache Ignite Documentation*. [Online]. Available: <https://apacheignite.readme.io/docs/svm-multi-class-classification>
- [38] “Sklearn.svm.SVC,” *scikit*. [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
- [39] M. Stickgold Malia, “Replaying the game: Hypnagogic images in normals and amnesics,” Oct. 2000 [Online]. Available: <https://science.sciencemag.org/content/sci/290/5490/350.full.pdf>
- [40] B. News, “Tetris ’helps to reduce trauma’.” Web Page, Jan-2007 [Online]. Available: <http://news.bbc.co.uk/2/hi/health/7813637.stm>
- [41] C. T. W. Championship, “Results.” Web Page, 2018 [Online]. Available: <https://thectwc.com/results/>
- [42] P. S. University, “Correlation coefficient r.” Web Page, 2018 [Online]. Available: <https://newonlinecourses.science.psu.edu/stat462/node/96/>
- [43] Macklin, “Regression performance metrics.” Web Page; Indiana University School of Informatics, Computing, & Engineering, Sep-2018.

- [44] Macklin, “Introduction to decision trees.” Web Page; Indiana University School of Informatics, Computing, & Engineering, Oct-2018.
- [45] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [46] F. E. Commission, “Election results.” <https://transition.fec.gov/pubrec/electionresults.shtml>, 2019.
- [47] S. M. Group, “Candidates on the issues.” <https://www.ontheissues.org/default.htm>, 2018.
- [48] D. Gayo-Avello, P. T. Metaxas, and E. Mustafaraj, “Limits of electoral predictions using twitter,” in *Fifth international aai conference on weblogs and social media*, 2011.
- [49] B. O’Connor, R. Balasubramanyan, B. R. Routledge, and N. A. Smith, “From tweets to polls: Linking text sentiment to public opinion time series,” in *Fourth international aai conference on weblogs and social media*, 2010.
- [50] T. N. Jagatic, N. A. Johnson, M. Jakobsson, and F. Menczer, “Social phishing,” *Communications of the ACM*, vol. 50, no. 10, pp. 94–100, 2007.
- [51] L. Bridges, “The changing face of malware,” *Network Security*, vol. 2008, no. 1, pp. 17–20, 2008.
- [52] H. Zhang, “The optimality of naive bayes,” *A A*, vol. 1, no. 2, p. 3, 2004.
- [53] D. Sculley and G. M. Wachman, “Relaxed online svms for spam filtering,” in *Proceedings of the 30th annual international acm sigir conference on research and development in information retrieval*, 2007, pp. 415–422.
- [54] A. Khorsi, “An overview of content-based spam filtering techniques,” *Informatica*, vol. 31, no. 3, 2007.
- [55] I. Androutsopoulos, J. Koutsias, K. V. Chandrinou, G. Paliouras, and C. D. Spyropoulos, “An evaluation of naive bayesian anti-spam filtering,” *arXiv preprint cs/0006013*, 2000.

- [56] S. R. Gunn and others, “Support vector machines for classification and regression,” *ISIS technical report*, vol. 14, no. 1, pp. 5–16, 1998.
- [57] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with naive bayes-which naive bayes?” in *CEAS*, 2006, vol. 17, pp. 28–69.
- [58] OpenAPI Initiative, “The openapi specification.” Web Page [Online]. Available: <https://github.com/OAI/OpenAPI-Specification>
- [59] RAML, “RAML version 1.0: RESTful api modeling language.” Web Page [Online]. Available: <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md>
- [60] tinyspec, “Tinyspec.” Web Page [Online]. Available: <https://github.com/Ajaxy/tinyspec>
- [61] api blueprint, “API blueprint. A powerful high-level api description language for web apis.” Web Page [Online]. Available: <https://apiblueprint.org/>
- [62] OpenAPI Initiative, “Announcing the official release of openapi 3.0.” Web Page, 2017 [Online]. Available: <https://swagger.io/blog/news/announcing-openapi-3-0/>
- [63] OpenAPI Initiative, “The openapi docs.” Web Page [Online]. Available: <https://swagger.io/docs/specification/about/>
- [64] “Swagger inspector.” [Online]. Available: <http://editor.swagger.io/>
- [65] S. Software, “Swagger ui.” Web Page [Online]. Available: <https://swagger.io/docs/open-source-tools/swagger-ui/usage/installation/>
- [66] S. Software, “Swagger codegen documentation.” Web Page [Online]. Available: <https://swagger.io/docs/open-source-tools/swagger-codegen/>
- [67] RAML, “RAML.” Web Page [Online]. Available: <https://raml.org/>
- [68] Yehuda Katz, “JSON:API.” Web Page [Online]. Available: <https://jsonapi.org/>

[69] Zalando SE, “Connexion.” Web Page [Online]. Available: <https://github.com/zalando/connexion>

[70] scikit-learn developers, Web Page [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>