DS7346 Cloud Computing Project

Release AWS Serverless Prediction

Chance Robinson, Jeff Washburn and Sreeni Prabhala

Nov 23, 2020

TABLE OF CONTENTS:

CHAPTER ONE

INTRODUCTION

Wildfires have broken out all over the western region of the United States in 2020, devastating communities and creating smoke plumes that can be seen even from space via satellite images. Some billows of smoke have carried over to places as far away as London, and it has been said that cities such as San Francisco and Seattle have some of the lowest quality of air on the entire planet currently due to the fires. As one of us lives close to the Bobcat Fire in California, which has currently burned over 93 thousand acres of land, the team thought it would be an interesting topic to delve into the data that has been collected in this domain over the past quarter century or so to mine any insights.

CHAPTER

TWO

US WILDFIRES DATA SET

The description below was taken from the Kaggle competition where this data was collected from.

"*This data publication contains a spatial database of wildfires that occurred in the United States from 1992 to 2015. It is the third update of a publication originally generated to support the national Fire Program Analysis (FPA) system. The wildfire records were acquired from the reporting systems of federal, state, and local fire organizations. The following core data elements were required for records to be included in this data publication: discovery date, final fire size, and a point location at least as precise as Public Land Survey System (PLSS) section (1-square mile grid). The data were transformed to conform, when possible, to the data standards of the National Wildfire Coordinating Group (NWCG). Basic error-checking was performed and redundant records were identified and removed, to the degree possible. The resulting product, referred to as the Fire Program Analysis fire-occurrence database (FPA FOD), includes 1.88 million geo-referenced wildfire records, representing a total of 140 million acres burned during the 24-year period.*"

(https://www.kaggle.com/rtatman/188-million-us-wildfires)

Core attributes on each fire captured including: - Discovery Date - Final fire size - Point location (least as precise as Public Land Survey System) section (1-square mile grid)

CHAPTER

THREE

OBJECTIVE ONE

3.1 Goal of this Project

Research of cloud technology on how it can be used to deploy a machine learning model that is fronted by an API endpoint to make predictions on cloud infrastructure.

3.2 Question of Interest

Given the size, location, date and other relevant features from the dataset, can we predict the cause of a wildfire in a cloud based, scalable way?

3.3 Solution Aspect - Methodology

In all, the data set includes 1.88 million geo-referenced wildfire records that equates to 140 million acres burned over the 24-year timeframe.

To predict cause of a fire, used classification modeling - Naive Bayes - Decision Tree

Table 1 describes the features that were used from the dataset and our predictor classification label is the STAT_CAUSE_DESCR

[11]:
```
from IPython.display import Image
Image(filename='./img/us_wildfire_features.png')
```

[11]:

Column Name	Data Type	Description
LATITUDE	Float	Latitude (NAD83) for point location of the fire (decimal degrees).
LONGITUDE	Float	Longitude (NAD83) for point location of the fire (decimal degrees).
DISCOVERY_DATE	Float	Date on which the fire was discovered or confirmed to exist.
FIRE_SIZE	Float	Estimate of acres within the final perimeter of the fire.
STATE	String	Two-letter alphabetic code for the state in which the fire burned (or originated), based on the nominal designation in the fire report.
OWNER_DESCR	String	Name of primary owner or entity responsible for managing the land at the point of origin of the fire at the time of the incident.
DISCOVERY_DOY	Integer	Day of year on which the fire was discovered or confirmed to exist.

Table 1 – Model Features

Table 2 showing the possible classifications

[12]:
```
Image(filename='./img/us_wildfire_labels.png')
```

[12]:

Cause of Fire			
Arson	Campfire	Children	Debris Burning
Equipment Use	Fireworks	Lightning	Miscellaneous
Missing/Undefined	Powerline	Railroad	Smoking
Structure			

Table 2 – Predictor Classification Labels

The following steps performed to create a model and deploy to the AWS cloud: - Ascertain dataset - Explore data doing exploratory data analysis - Create classification models - Naive Bayes - Decision Trees - Measure model performance - Deploy model to AWS infrastructure - Create gateway for REST based API call - Use AWS lambda to deploy model to then fulfill prediction request - Track model execution request using Amazon's Relational Database Service (Amazon RDS)

The remainder of the notebook is the code that goes through the process

3.3.1 Exploratory Data Analysis

Library Imports

[13]:
```
# Base Imports
import sqlite3
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Pre-processing
from sklearn.preprocessing import LabelEncoder

# Metrics and Evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

# Model Selection
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

# Pipeline
from sklearn.pipeline import Pipeline
import joblib

# Estimators
from sklearn.multiclass import OneVsRestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
```

Set Training Parameters

```
[14]: estimator = "decision_tree_classifier"
      train_model = False
```

Load Data

```
[15]: # %%time
      conn = sqlite3.connect('../../../data/FPA_FOD_20170508.sqlite')
      df_fires = pd.read_sql_query("SELECT * FROM 'Fires'", conn)
```

```
[16]: df_fires.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1880465 entries, 0 to 1880464
Data columns (total 39 columns):
 #   Column                      Dtype
---  ------                      -----
 0   OBJECTID                    int64
 1   FOD_ID                      int64
 2   FPA_ID                      object
 3   SOURCE_SYSTEM_TYPE          object
 4   SOURCE_SYSTEM               object
 5   NWCG_REPORTING_AGENCY       object
 6   NWCG_REPORTING_UNIT_ID      object
 7   NWCG_REPORTING_UNIT_NAME    object
 8   SOURCE_REPORTING_UNIT       object
 9   SOURCE_REPORTING_UNIT_NAME  object
 10  LOCAL_FIRE_REPORT_ID        object
 11  LOCAL_INCIDENT_ID           object
 12  FIRE_CODE                   object
 13  FIRE_NAME                   object
 14  ICS_209_INCIDENT_NUMBER     object
 15  ICS_209_NAME                object
 16  MTBS_ID                     object
 17  MTBS_FIRE_NAME              object
 18  COMPLEX_NAME                object
 19  FIRE_YEAR                   int64
 20  DISCOVERY_DATE              float64
 21  DISCOVERY_DOY               int64
 22  DISCOVERY_TIME              object
 23  STAT_CAUSE_CODE             float64
 24  STAT_CAUSE_DESCR            object
 25  CONT_DATE                   float64
 26  CONT_DOY                    float64
 27  CONT_TIME                   object
 28  FIRE_SIZE                   float64
 29  FIRE_SIZE_CLASS             object
 30  LATITUDE                    float64
 31  LONGITUDE                   float64
 32  OWNER_CODE                  float64
 33  OWNER_DESCR                 object
 34  STATE                       object
 35  COUNTY                      object
 36  FIPS_CODE                   object
```

(continues on next page)

(continued from previous page)

```
 37   FIPS_NAME                          object
 38   Shape                              object
dtypes: float64(8), int64(4), object(27)
memory usage: 559.5+ MB
```

[17]:
```
# %%time

df_fires.set_index("OBJECTID", inplace=True)
```

Missing Values

The following columns were found to have missing values, and would not be ideal for most machine learning models.

[18]:
```
na_list = df_fires.columns[df_fires.isnull().any()].tolist()

for i in na_list:
    print(i)
```

```
LOCAL_FIRE_REPORT_ID
LOCAL_INCIDENT_ID
FIRE_CODE
FIRE_NAME
ICS_209_INCIDENT_NUMBER
ICS_209_NAME
MTBS_ID
MTBS_FIRE_NAME
COMPLEX_NAME
DISCOVERY_TIME
CONT_DATE
CONT_DOY
CONT_TIME
COUNTY
FIPS_CODE
FIPS_NAME
```

Total US Wildfires by Cause

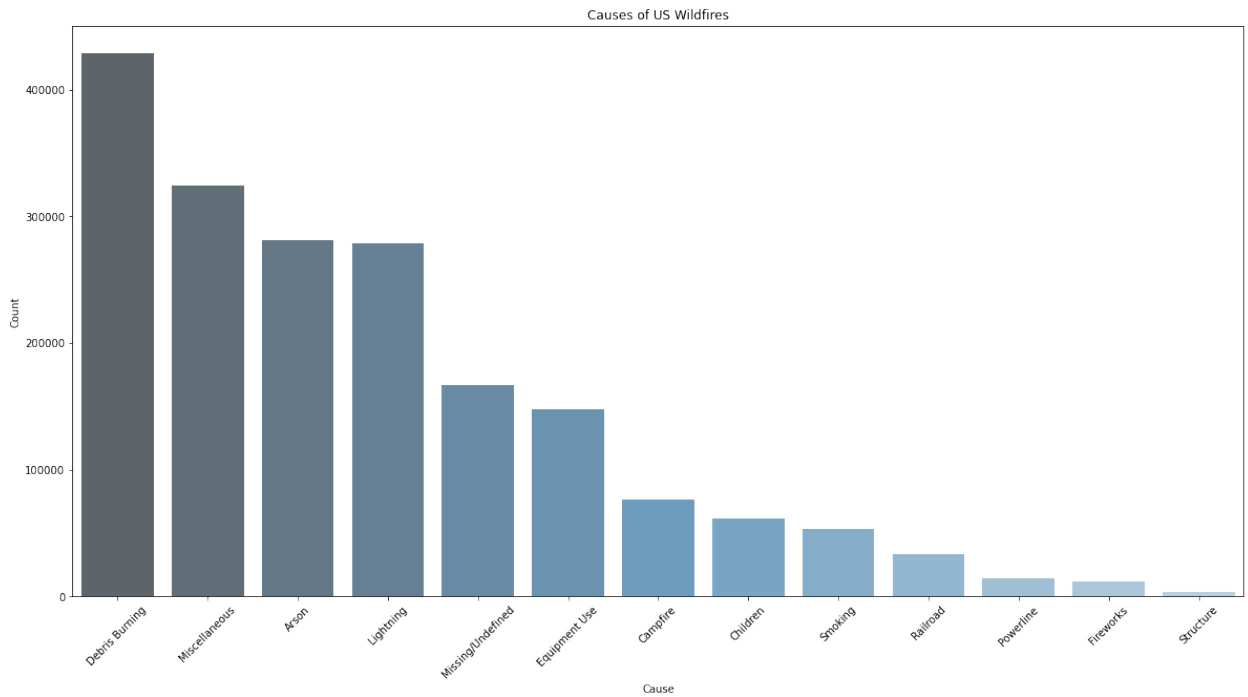
Across all of the states included in the dataset, **debris burning** was the category identified as having caused the most wildfires. The lables are obviously very skewed, so this may need to be taken into account when building our final prediction model. A technique like **SMOTE** might be of use so that the categories with the lower samples are artificially upsampled to match that of the hightest one.

[19]:
```
fig_0 = plt.figure(1, figsize=(20, 10))

chart_1 = fig_0.add_subplot(111)

cause_count  = df_fires['STAT_CAUSE_DESCR'].value_counts()

sns.barplot(cause_count.index, cause_count.values, alpha=0.8, palette="Blues_d")
chart_1.set_title('Causes of US Wildfires')
chart_1.set_xlabel('Cause')
chart_1.set_ylabel('Count')
chart_1.set_xticklabels(chart_1.get_xticklabels(), rotation=45)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Total US Wildfires by Year

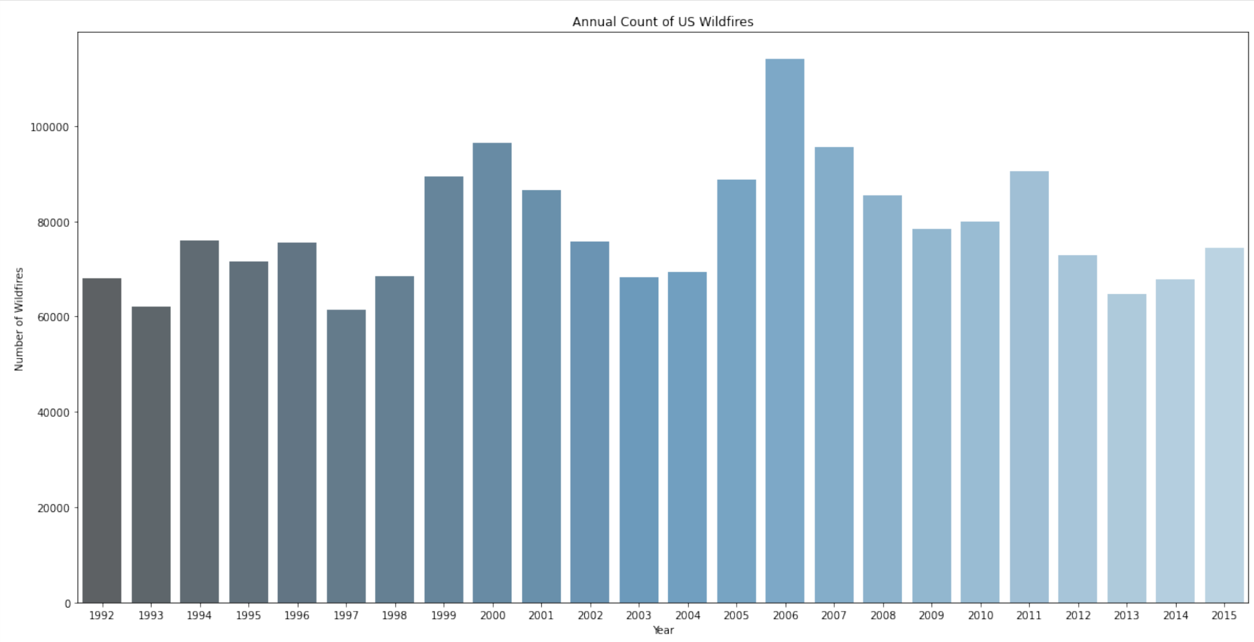
The dataset includes wildfires that occurred in the United States from **1992** to **2015**.

Converting Julian to calendar date using pandas

- https://stackoverflow.com/questions/63434276/converting-julian-to-calendar-date-using-pandas

[20]:
```
df_fires["DISCOVERY_DATETIME"] = pd.to_datetime(df_fires["DISCOVERY_DATE"], unit='D',␣
↪origin='julian')
df_fires['DISCOVERY_DAY_OF_WEEK'] = df_fires["DISCOVERY_DATETIME"].dt.day_name()
```

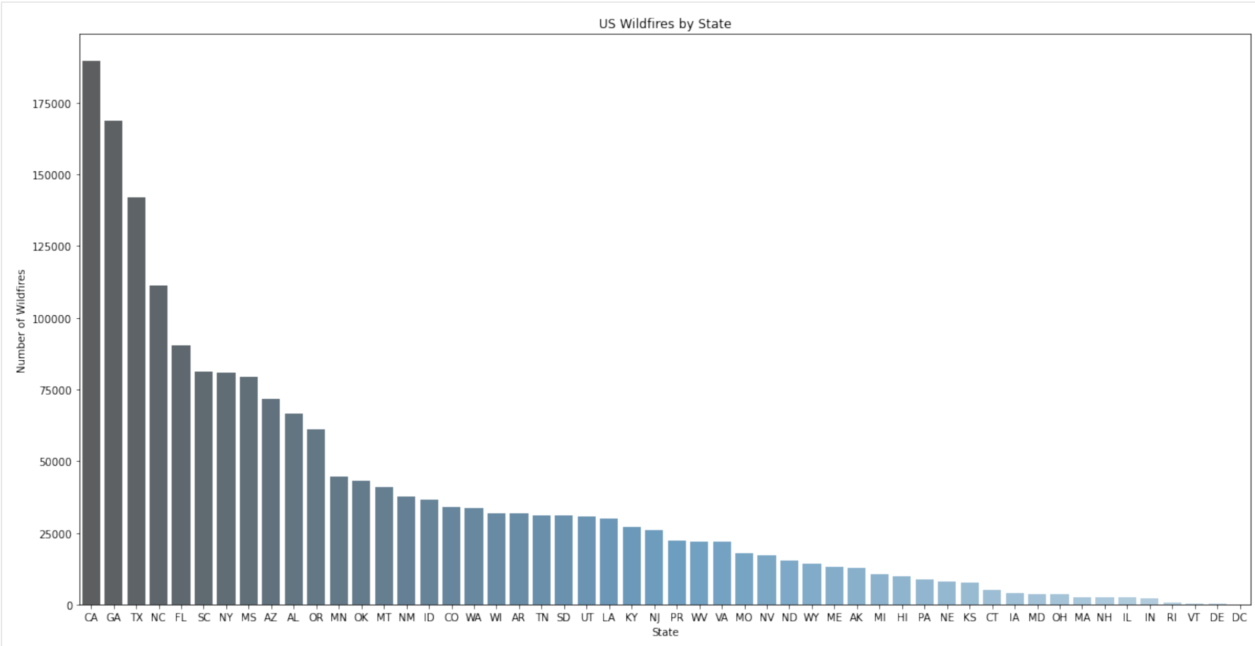
[21]:
```
year_count  = df_fires['FIRE_YEAR'].value_counts().sort_index()
plt.figure(figsize=(20, 10))
sns.barplot(year_count.index, year_count.values, alpha=0.8, palette="Blues_d")
plt.title('Annual Count of US Wildfires')
plt.ylabel('Number of Wildfires')
plt.xlabel('Year')
plt.show()
```



Total US Wildfires by State

California, Georgia and Texas have the highest volume of recorded wildfires. California has the highest population of any state in the union, followed by a distant second with Texas. The counts for those states could be somewhat expected in that they have higher populations. The counts for Georgia however do seem a bit high considering that their state population is lower on the list of most populated states. Further investigation might be warranted there. We may also look to see if states differ in their category rankings.

[22]:
```
state_count  = df_fires['STATE'].value_counts()
plt.figure(figsize=(20, 10))
sns.barplot(state_count.index, state_count.values, alpha=0.8, palette="Blues_d")
plt.title('US Wildfires by State')
plt.ylabel('Number of Wildfires')
plt.xlabel('State')
plt.show()
```



3.4 Modeling

- Naive Bayes
- Decision Tree

[23]:
```
# create an instance of LabelEncoder
label_encoder = LabelEncoder()

# map to numerical values in a new variable
df_fires["STATE_CAT"] = label_encoder.fit_transform(df_fires['STATE'])
df_fires["OWNER_DESCR_CAT"] = label_encoder.fit_transform(df_fires['OWNER_DESCR'])
df_fires["DISCOVERY_DAY_OF_WEEK_CAT"] = label_encoder.fit_transform(df_fires[
↪'DISCOVERY_DAY_OF_WEEK'])
```

[24]:
```
X = df_fires[["LATITUDE", "LONGITUDE", "DISCOVERY_DATE", "FIRE_SIZE", "STATE_CAT",
↪"OWNER_DESCR_CAT", "DISCOVERY_DAY_OF_WEEK_CAT"]]
y = df_fires["STAT_CAUSE_DESCR"]
```

3.4.1 Train / Test Split

[25]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.1,
    random_state=1,
    stratify=y)
```

3.4.2 Train Gaussian Naive Bayes Classifier

[26]:
```
# %%time

if train_model and estimator == "gaussian_nb":

    clf = OneVsRestClassifier(GaussianNB())

    clf.fit(X_train, y_train)
```

3.4.3 Train Decision Tree Classifier

[27]:
```
# %%time

if train_model and estimator == "decision_tree_classifier":

    clf = OneVsRestClassifier(DecisionTreeClassifier(random_state=1,
        splitter='best',
        min_samples_split=5,
        min_samples_leaf=4,
        max_features='auto',
        class_weight=None))

    clf.fit(X_train, y_train)
```

[28]:
```
if train_model and estimator == "decision_tree_classifier":
    joblib.dump(clf, './aws_predict/training/models/decission_tree_classifier.pkl',␣
↪compress=3)
elif train_model and estimator == "gaussian_nb":
    joblib.dump(clf, './aws_predict/training/models/gaussian_nb_classifier.pkl')
else:
    pass
```

3.4.4 Predictions

Once the model has been defined, predictions can be made on new and unseen data.

Gaussian Naive Bayes Classifier

[29]:
```
nb_clf = joblib.load('./aws_predict/training/models/gaussian_nb_classifier.pkl')

pred_test = [[43.235833, -122.466944, 2452859.5, 0.1, 37, 15, 0]]

nb_clf.predict(pred_test)
```

[29]:
```
array(['Lightning'], dtype='<U17')
```

[30]:
```
# %%time

y_pred = nb_clf.predict(X_test)
```

[31]:
```
# %%time

print ('accuracy:', accuracy_score(y_test, y_pred))
```

```
accuracy: 0.31547964072811585
```

[32]:
```
# %%time

print(classification_report(y_test, y_pred))
```

```
                   precision    recall  f1-score   support

            Arson       0.52      0.01      0.02     28145
         Campfire       0.00      0.00      0.00      7614
         Children       0.08      0.14      0.10      6117
   Debris Burning       0.29      0.90      0.44     42903
    Equipment Use       0.36      0.10      0.15     14761
        Fireworks       0.09      0.37      0.14      1150
        Lightning       0.63      0.43      0.51     27847
    Miscellaneous       0.25      0.09      0.13     32381
Missing/Undefined       0.75      0.19      0.31     16672
        Powerline       0.00      0.00      0.00      1445
         Railroad       0.00      0.00      0.00      3345
          Smoking       0.00      0.00      0.00      5287
        Structure       0.00      0.00      0.00       380

         accuracy                           0.32    188047
        macro avg       0.23      0.17      0.14    188047
     weighted avg       0.38      0.32      0.24    188047
```

Decision Tree Classifier

[33]:
```
dt_clf = joblib.load('./aws_predict/training/models/decission_tree_classifier.pkl')

pred_test = [[43.235833, -122.466944, 2452859.5, 0.1, 37, 15, 0]]

dt_clf.predict(pred_test)
```

[33]:
```
array(['Lightning'], dtype='<U17')
```

[34]:
```
# %%time

y_pred = dt_clf.predict(X_test)
```

[35]:
```
# %%time

print ('accuracy:', accuracy_score(y_test, y_pred))
```

```
accuracy: 0.5223640898286067
```

[36]:
```
# %%time

print(classification_report(y_test, y_pred))
```

```
                  precision    recall  f1-score   support

           Arson       0.51      0.48      0.50     28145
        Campfire       0.39      0.28      0.33      7614
        Children       0.24      0.16      0.19      6117
  Debris Burning       0.51      0.56      0.53     42903
   Equipment Use       0.31      0.27      0.29     14761
       Fireworks       0.37      0.31      0.34      1150
       Lightning       0.70      0.75      0.72     27847
   Miscellaneous       0.47      0.49      0.48     32381
Missing/Undefined       0.88      0.89      0.88     16672
       Powerline       0.15      0.12      0.13      1445
        Railroad       0.40      0.40      0.40      3345
         Smoking       0.13      0.09      0.10      5287
       Structure       0.01      0.07      0.02       380

        accuracy                           0.52    188047
       macro avg       0.39      0.37      0.38    188047
    weighted avg       0.52      0.52      0.52    188047
```

CHAPTER

FOUR

OBJECTIVE TWO

Now that a model has been defined, can we make use of cloud resources to make predictions?

4.1 Cloud Deployment

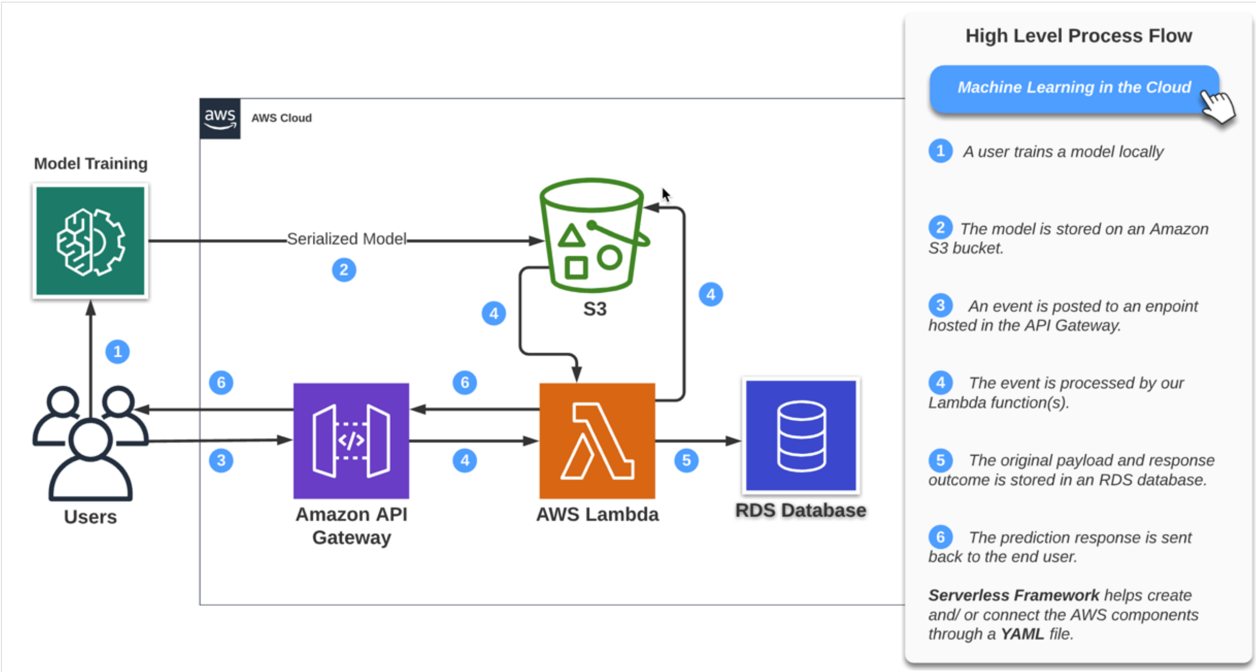
4.1.1 AWS Serverless Process Flow

The below chart visualizes our process flow for making predictions with AWS cloud components.

1. Train a model on the US Wildfire data
2. Upload a serialized (pickled) version to an S3 bucket.
3. POST an event to an endpoint hosed in API Gateway
4. The event is forwarded to a Lambda function, which retrieves the model from our S3 bucket for the purposes of making a prediction.
5. The original payload and response are stored in an RDS database
6. The prediction label and probability is sent back to the client, along with the original payload and the primary key of the record inserted into the database.

[37]:
```
Image(filename='./img/aws_serverless_prediction_flow.png')
```

[37]:



4.1.2 Serverless Framework

Serverless Framework is an open-source framework built with Node.js, and supports a variety of both cloud platforms and programming languages. There are other competing products available, but this has one of the more active communities on GitHub and was also one of the first to market.

Below are some of the options currently supported by the framework, which offers both open-source and professional versions. - Cloud platforms - AWS - Azure - Google Cloud Platform

- Programming languages
  - Node.js
  - Python
  - Java

Another option worth exploring would be the AWS CHALICE product. It was created by AWS and was written specifically for Python, but is also one of the more popular solutions in this space. And considering our project is using this same setup, it would be interesting to have had the time to explore both.

4.1.3 Local Setup and Validation

The `sls invoke local` command can be used to validate that your function is working properly.

You'll notice the orange highlighted deprecation warnings shown below. One important aspect to any project that you plan to release to production is locking down your dependencies so that the program can be reproduced on other setups, especially the one deployed to AWS, and that it is consistent with how it was developed and tested.

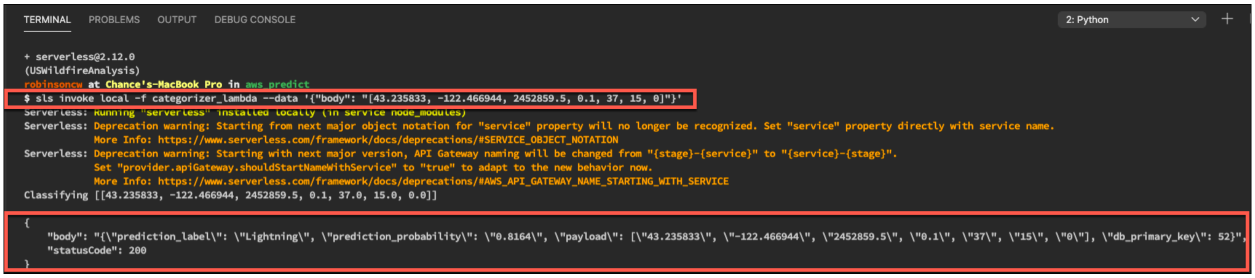
Because we are using the Serverless Framework, we have a number of dependencies to consider.

1. Node.js
2. Serverless python requirements

3. Python libraries
4. AWS Runtime

[38]:
```
Image(filename='./img/sls_predictions.png')
```

[38]:



4.1.4 Building Dependencies with Docker

Using docker to build your packaged dependencies with an environment similar or almost identical to the runtime running in AWS is extremely beneficial.

The following link is a great resource for leveraging docker containers, which as the serverless framework offered, supports a number of different programming runtimes and versions.

https://github.com/lambci/docker-lambda

4.1.5 Serverless Deployment

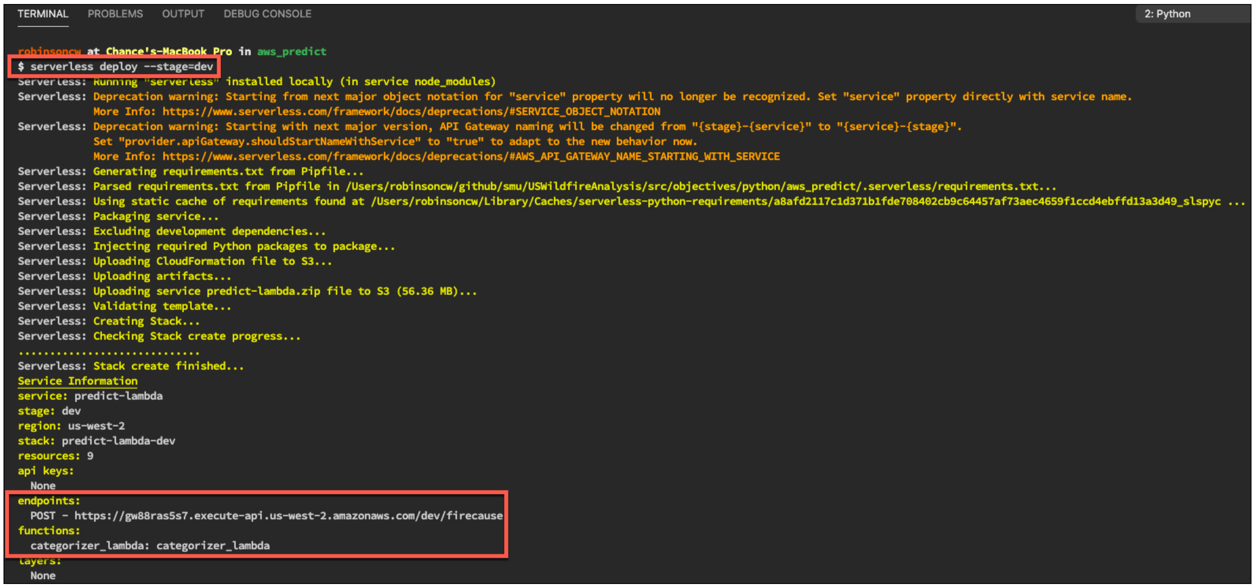
Environments can be staged so that build can be performed in a non-production instance that won't impact your active service.

- `serverless deploy --stage=dev`

Assets can also be removed with a single command, removing all artifacts created by the deployment.

- `serverless remove`

[39]:
```
Image(filename='./img/sls_deployment.png')
```

[39]: 

4.1.6 Prediction Output

Once deployed, a working URL endpoint will be returned which can be used to validate the solution via a `CURL` or other web service request.

Once testing has been performed, the service could be deployed to a production environment and made available for use based on your requirements. The isolated nature of the environments could allow you train and test the model with newer samples, or even change the core code in a safe a structured way before deploying those adjustments to production.

[40]:
```
Image(filename='./img/aws_prediction_output.png')
```

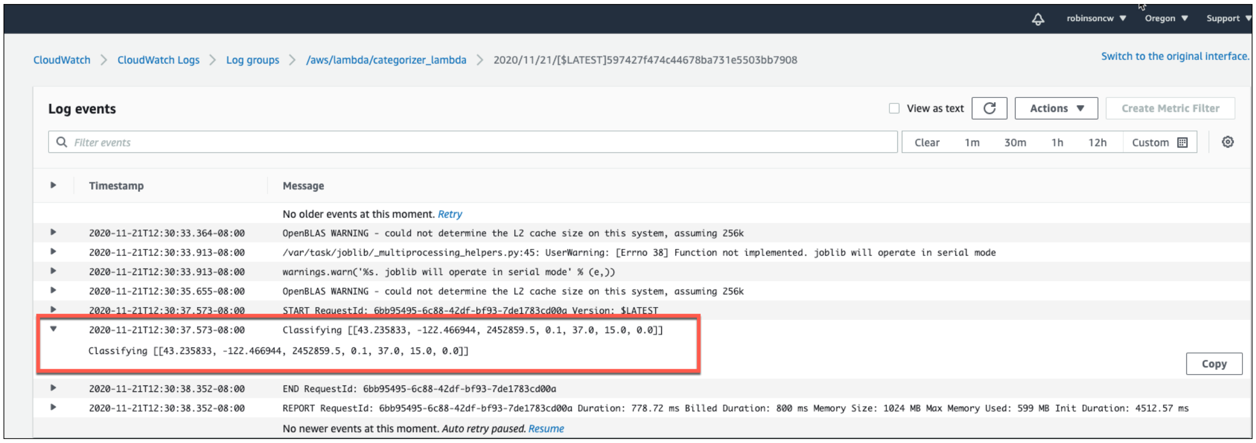
[40]: 

4.1.7 CloudWatch Logging

Logging can be reviewed through the CloudWatch area from the AWS console. A variety of options exist to catch failures gracefully and either attempt to reprocess them or send alerts so that the details can be reviewed.

[41]:
```
Image(filename='./img/aws_cloudwatch_logs.png')
```

[41]:


robinsoncw
Oregon
Support
CloudWatch > CloudWatch Logs > Log groups > /aws/lambda/categorizer_lambda > 2020/11/21/[$LATEST]597427f474c44678ba731e5503bb7908
Switch to the original interface.
Log events
View as text
Actions
Create Metric Filter
Filter events
Clear
1m
30m
1h
12h
Custom
Timestamp
Message
No older events at this moment. Retry
2020-11-21T12:30:33.364-08:00 OpenBLAS WARNING - could not determine the L2 cache size on this system, assuming 256k
2020-11-21T12:30:33.913-08:00 /var/task/joblib/_multiprocessing_helpers.py:45: UserWarning: [Errno 38] Function not implemented. joblib will operate in serial mode
2020-11-21T12:30:33.913-08:00 warnings.warn('%s. joblib will operate in serial mode' % (e,))
2020-11-21T12:30:35.655-08:00 OpenBLAS WARNING - could not determine the L2 cache size on this system, assuming 256k
2020-11-21T12:30:37.573-08:00 START RequestId: 6bb95495-6c88-42df-bf93-7de1783cd00a Version: $LATEST
2020-11-21T12:30:37.573-08:00 Classifying [[43.235833, -122.466944, 2452859.5, 0.1, 37.0, 15.0, 0.0]]
Classifying [[43.235833, -122.466944, 2452859.5, 0.1, 37.0, 15.0, 0.0]]
Copy
2020-11-21T12:30:38.352-08:00 END RequestId: 6bb95495-6c88-42df-bf93-7de1783cd00a
2020-11-21T12:30:38.352-08:00 REPORT RequestId: 6bb95495-6c88-42df-bf93-7de1783cd00a Duration: 778.72 ms Billed Duration: 800 ms Memory Size: 1024 MB Max Memory Used: 599 MB Init Duration: 4512.57 ms
No newer events at this moment. Auto retry paused. Resume

CHAPTER

FIVE

OBJECTIVE THREE

In this section, we will discuss about securing the serverless applications in an enterprise setting. Specifically, we will discuss about the options to secure database credentials and the APIs.

5.1 Secrets Manager

AWS Secrets Manager allows us to protect secrets needed to access applications, services, and other IT resources. It offers built-in integration for databases on RDS, and allows rotating database credentials. In addition, it enables us to control access to secrets using IAM policies.

For our application, below are the steps we have followed to maintain RDS database credentials in Secrets Manager: - Store a new secret - choose Credentials for RDS database; enable encryption - Attach it to RDS DB instance that accesses this credential - In the application, retrieve secret and decrypt using Secrets Manager API - Use the secret to establish connection to the RDS database

Below code snippet identifies the way to get the secret value and decrypt it.

[42]:
```
Image(filename='./img/get_secret_code.png')
```

[42]:
```
def get_secret():
    secret_name = "prod/aws_predict/postgres"
    region_name = "us-west-2"

    # Create a Secrets Manager client
    session = boto3.session.Session()
    client = session.client(service_name="secretsmanager", region_name=region_name)

    get_secret_value_response = client.get_secret_value(SecretId=secret_name)
    secret = get_secret_value_response["SecretString"]
```

5.2 RDS Proxy

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon RDS that makes applications more scalable, more resilient to database failures, and more secure. Serverless applications typically deal with database connections at a high rate causing memory and computing resources contention. RDS Proxy allows pooling and sharing connections, improving the efficiency of resources utilization. In addition, RDS proxy enables managing authentication and access through AWS Secrets Manager and IAM policies.

- High-level steps involved in setting up RDS Proxy for a serverless application are as below:

- Setup network prerequisites, if not already present – VPC, subnets, EC2 instance, and internet gateway
- Setup DB credentials in Secrets Manager
- Setup IAM policy to access proxy through Secrets Manager
- Create RDS Proxy with relevant connectivity details above and required connection pool parameters
- In the application, retrieve secret and decrypt using Secrets Manager API
- Use the secret to establish connection to the RDS database

5.3 API Gateway

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale.

[43]:
```
Image(filename='./img/aws_api_gateway.png')
```

[43]:

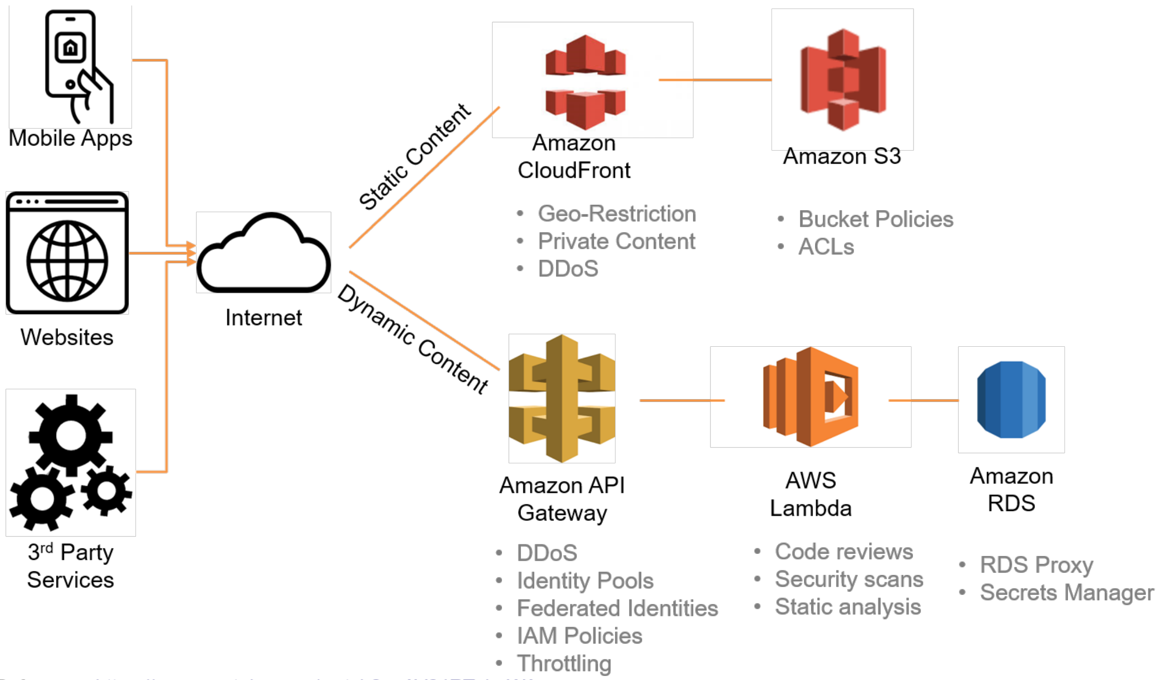
API Gateway: - Creates a unified front-end for microservices - Provides DDoS protection and throttling – through CloudFront and API Gateway Cache - Authenticates and authorizes requests – using Cognito User Pools, Cognito Federated Identities, Custom Authorizers - Throttles, meters, and monetizes API usage – through CloudWatch & Usage Plans

5.4 Securing Serverless Applications

Following diagram depicts the components involved in a typical serverless application architecture and some of the options available for us at various stages to secure the applications.

[44]:
```
Image(filename='./img/securing_serverless_applications.png')
```

[44]:



Reference: https://www.youtube.com/watch?v=AV24RTvbgWA

CHAPTER

SIX

REFERENCES

^ Tatman, Rachael, 2020 *Kaggle: 1.88 Million US Wildfires* URL

^ Weigel, Benjamin, 2018 *PyData Berlin: Deploying a machine learning model to the cloud using AWS Lambda* URL

^ Pirtle, Justin, 2017 *AWS Online Tech Talks: Security Best Practices for Serverless Applications* URL

CHAPTER

SEVEN

APPENDIX

7.1 Data Dictionary

This dataset is an SQLite database that contains the following information:

The source file can be found on Kaggle | 1.88 Million US Wildfires.

Column Name	Data Type	Description
FOD_ID	Integer	Global unique identifier.
FPA_ID	String	Unique identifier that contains information necessary to track back to the original record in the source dataset.
SOURCESYSTEMTYPE	String	Type of source database or system that the record was drawn from (federal, nonfederal, or interagency).
SOURCESYSTEM	String	Name of or other identifier for source database or system that the record was drawn from. See Table 1 in Short (2014), or .pdf, for a list of sources and their identifier.
NWCGREPORTINGAGENCY	String	Active National Wildlife Coordinating Group (NWCG) Unit Identifier for the agency preparing the fire report (BIA = Bureau of Indian Affairs, BLM = Bureau of Land Management, BOR = Bureau of Reclamation, DOD = Department of Defense, DOE = Department of Energy, FS = Forest Service, FWS = Fish and Wildlife Service, IA = Interagency Organization, NPS = National Park Service, ST/C&L = State, County, or Local Organization, and TRIBE = Tribal Organization).
NWCGREPORTINGUNIT_ID	String	Active NWCG Unit Identifier for the unit preparing the fire report.
NWCGREPORTINGUNIT_NAME	String	Active NWCG Unit Name for the unit preparing the fire report.
SOURCEREPORTINGUNIT	String	Code for the agency unit preparing the fire report, based on code/name in the source dataset.
SOURCEREPORTINGUNIT_NAME	String	Name of reporting agency unit preparing the fire report, based on code/name in the source dataset.
LOCALFIREREPORT_ID	String	Number or code that uniquely identifies an incident report for a particular reporting unit and a particular calendar year.
LOCALINCIDENTID	String	Number or code that uniquely identifies an incident for a particular local fire management organization within a particular calendar year.

continues on next page

Table 1 – continued from previous page

Column Name	Data Type	Description
FIRE_CODE	String	Code used within the interagency wildland fire community to track and compile cost information for emergency fire suppression (https://www.firecode.gov/).
FIRE_NAME	String	Name of the incident, from the fire report (primary) or ICS-209 report (secondary).
ICS209INCIDENT_NUMBER	String	Incident (event) identifier, from the ICS-209 report.
ICS209NAME	String	Name of the incident, from the ICS-209 report.
MTBS_ID	String	Incident identifier, from the MTBS perimeter dataset.
MTBSFIRENAME	String	Name of the incident, from the MTBS perimeter dataset.
COMPLEX_NAME	String	Name of the complex under which the fire was ultimately managed, when discernible.
FIRE_YEAR	Integer	Calendar year in which the fire was discovered or confirmed to exist.
DISCOVERY_DATE	Float	Date on which the fire was discovered or confirmed to exist.
DISCOVERY_DOY	Integer	Day of year on which the fire was discovered or confirmed to exist.
DISCOVERY_TIME	String	Time of day that the fire was discovered or confirmed to exist.
STATCAUSECODE	Float	Code for the (statistical) cause of the fire.
STATCAUSEDESCR	String	Description of the (statistical) cause of the fire.
CONT_DATE	Float	Date on which the fire was declared contained or otherwise controlled (mm/dd/yyyy where mm=month, dd=day, and yyyy=year).
CONT_DOY	Float	Day of year on which the fire was declared contained or otherwise controlled.
CONT_TIME	String	Time of day that the fire was declared contained or otherwise controlled (hhmm where hh=hour, mm=minutes).
FIRE_SIZE	Float	Estimate of acres within the final perimeter of the fire.
FIRESIZECLASS	String	Code for fire size based on the number of acres within the final fire perimeter expenditures (A=greater than 0 but less than or equal to 0.25 acres, B=0.26-9.9 acres, C=10.0-99.9 acres, D=100-299 acres, E=300 to 999 acres, F=1000 to 4999 acres, and G=5000+ acres).
LATITUDE	Float	Latitude (NAD83) for point location of the fire (decimal degrees).
LONGITUDE	Float	Longitude (NAD83) for point location of the fire (decimal degrees).
OWNER_CODE	Float	Code for primary owner or entity responsible for managing the land at the point of origin of the fire at the time of the incident.
OWNER_DESCR	String	Name of primary owner or entity responsible for managing the land at the point of origin of the fire at the time of the incident.
STATE	String	Two-letter alphabetic code for the state in which the fire burned (or originated), based on the nominal designation in the fire report.
COUNTY	String	County, or equivalent, in which the fire burned (or originated), based on nominal designation in the fire report.

continues on next page

Table 1 – continued from previous page

Column Name	Data Type	Description
FIPS_CODE	String	Three-digit code from the Federal Information Process Standards (FIPS) publication 6-4 for representation of counties and equivalent entities.
FIPS_NAME	String	County name from the FIPS publication 6-4 for representation of counties and equivalent entities.