

BRENTON MALLEN

---

# DATA SCIENCE: FROM MODEL TO MICROSERVICE



Slides

# BIO

## Masters in Ocean Engineering

Underwater mine detection and classification using sonar image processing

## Data Scientist

## Cyber Security

Performing R&D, ad hoc analyses and building production ML systems for internet bot detection and mitigation

## Agriculture

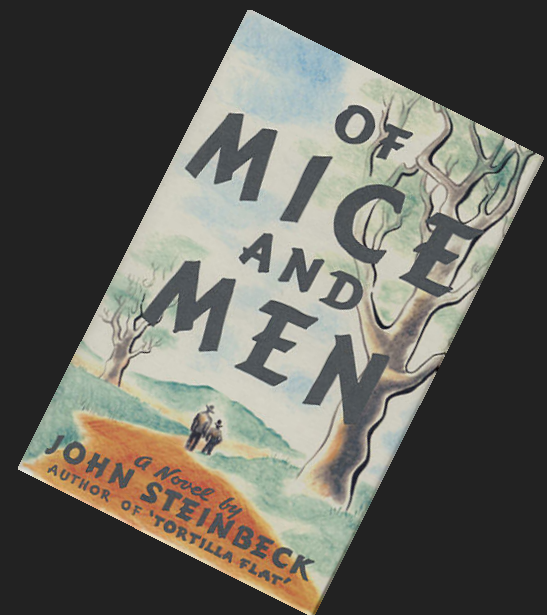
Satellite image processing and time series modeling

## MOTIVATION

Frustration with tutorials not discussing what to do  
with a model after you've trained it

## INTENT

Illustrate an approach to a product development cycle of getting data, building a machine learning model and turning it into something that can be used by others.



# AGENDA

Problem Background

Build a Machine Learning Model

Build a Web App/API (Microservice)

Demo

Code Snippets along the way!



Source Code

## LANGUAGES & TECH

### **Python**

scikit-learn, pandas, flask, zappa

### **HTML**

### **Javascript**

### **Amazon Web Services (AWS)**

Lambda, API Gateway

# THE PROBLEM BACKGROUND

# THE TITANIC PROBLEM

## Objective:

Predict if a passenger would have survived the titanic



<https://www.kaggle.com/c/titanic>



# THE DATA

## Training Set

891 records

Label: Survival

## Test Set

418 records

### Data Variables

Survival

Ticket class

Sex

Age in years

# of siblings / spouses aboard

# of parents / children aboard

Ticket number

Passenger fare

Cabin number

Port of Embarkation

# BUILD A MODEL



Jupyter Notebook

What does every ML model need?



## CHOOSING & ENGINEERING FEATURES

### Removed ship location related features

Not particularly useful without knowledge of the ship layout

### New features:

Age and Fare groups

Is Alone

Data Variables
Ticket class
Sex
Age in years
# of siblings / spouses aboard
# of parents / children aboard
Ticket number
Passenger fare
Cabin number
Port of Embarkation

# FEATURE CHALLENGES

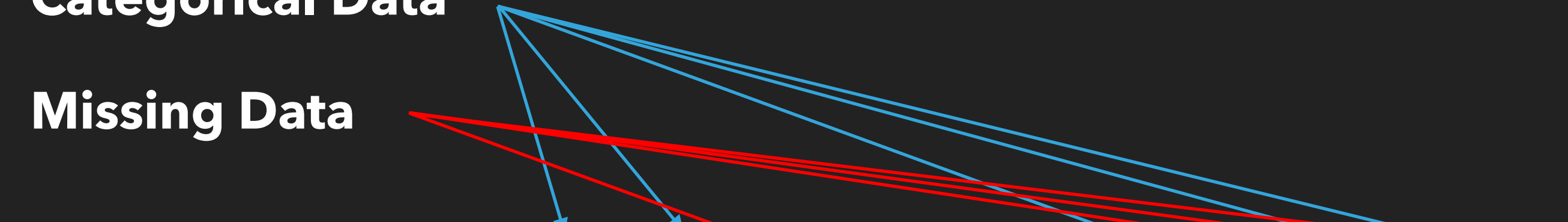
## Small Set

891 Train Records

418 Test Records

## Categorical Data

## Missing Data



PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

# MISSING DATA

## Selecting Most Common

```
most_common_embark = df['Embarked'].mode()[0]  
df['Embarked'] = df['Embarked'].fillna(most_common_embark)
```

## Selecting Median Value

```
test_data['Fare'] = test_data['Fare'].fillna(test_data['Fare'].median())  
test_data['Fare'] = test_data['Fare'].apply(fare_groups)
```

# MISSING DATA

## Predict Using a Classifier\*

```
# Fill missing ages and encode them into age groups
age_features = [f for f in _utils.FEATURES if f != 'Age']
age_clf = _utils.missing_clf(train_data,
                             age_features,
                             'Age'
                             )
train_data['Age'] = (train_data.
                    apply(
                        lambda x: _utils.predict_encode_age(
                            x,
                            features=age_features,
                            clf=age_clf
                        ),
                        axis=1
                    )
                    )
```

Train a classifier on the field with missing data

Use the classifier to fill missing data

\*Using the output of a classifier as features for another classifier can lead to latent interactions and increased tuning complexity

# ENCODING CATEGORICAL FEATURES

## Sex Encoding

```
SEX_MAPPING = {'female': 0, 'male': 1}

# encode the sex data
train_data['Sex'] = train_data['Sex'].map(_utils.SEX_MAPPING)
```

## Embarked Encoding

```
encoding = {f: i for i, f in enumerate(df['Embarked'].unique())}
df['Embarked'] = df['Embarked'].map(encoding)
```



# ENGINEERED FEATURES

## Fare Encoding

```
def fare_groups(fare: float):  
    """  
    This function puts Fares into groups based of  
    a defined interval  
    :param fare:  
    :return:  
    """  
    if fare < 7.78:  
        return 0  
    elif 7.78 <= fare < 8.66:  
        return 1  
    elif 8.66 <= fare < 14.45:  
        return 2  
    elif 14.45 <= fare < 26.0:  
        return 3  
    elif 26.0 <= fare < 52.37:  
        return 4  
    elif 52.37 <= fare < 512.33:  
        return 5  
    else:  
        return 6
```

## Age Encoding

```
def age_groups(age):  
    """  
    This function creates age groups  
    """  
    if age < 10:  
        return 0  
    elif 10 <= age < 18:  
        return 1  
    elif 18 <= age < 26:  
        return 2  
    elif 26 <= age < 36:  
        return 3  
    elif 36 <= age < 48:  
        return 4  
    elif 48 <= age < 56:  
        return 5  
    else:  
        return 6
```

# ENGINEERED FEATURES

## Is Alone

```
def is_alone(row: pd.Series):  
    """  
    This function is used to determine if a passenger was not traveling with  
    anyone else  
    :param row: row of titanic data  
    :return: binary output of whether and passenger was traveling alone  
    """  
    family_size = row['SibSp'] + row['Parch']  
    if family_size == 0:  
        return 1  
    else:  
        return 0
```

If the passenger has no spouse, sibling, parent or child

# FINAL FEATURE SET

Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	Alone
3	1	2	1	0	0	0	1
1	0	4	1	0	5	1	1
3	0	3	0	0	1	0	0
1	0	3	1	0	5	0	1
3	1	3	0	0	1	0	0

# TRAIN A MODEL

```
def train() -> RandomForestClassifier:
    """
    Train a random forest classifier on some training data
    :return: trained classifier
    """
    train_data = process_data(_utils.TRAIN_FILE)
    clf = RandomForestClassifier(n_estimators=300,
                               min_samples_leaf=7,
                               min_samples_split=5,
                               max_features=0.5,
                               oob_score=True,
                               n_jobs=-1,
                               random_state=42
                               )
    clf.fit(train_data[_utils.CLASS_FEATURES], train_data[_utils.LABEL])
    return clf
```

**An attempt to mitigate overfitting due to small sample size**

# MODEL INSPECTION

## Cross Validation

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(survival_clf,
                          encoded_train[CLASS_FEATURES],
                          encoded_train[LABEL],
                          cv=10,
                          scoring='accuracy'
                          )

print(f"""
    Accuracy (95% CI):
    {round(scores.mean(), 3)} (+/- {round(scores.std() * 2, 3)})
    """)
```

Accuracy\* (95% CI):

**0.823 (+/- 0.076)**

\*This is classification accuracy, which is the performance metric used for the Kaggle competition

# MODEL INSPECTION



**Women and Children First**  
by: Fortunino Matania

## Feature Importance

Feature	Importance
Sex	0.51589
Ticket Class	0.15865
Fare	0.11274
Age	0.09403
# of siblings / spouses aboard	0.04831
Embarked	0.03169
# of parents / children aboard	0.02158
Alone	0.01799

# MODEL PERFORMANCE

**Gather Features  
on Test Data**

**Predict Survival  
on Test Data**

```
# Embarked fill and encoding
test_data = raw_test.copy() # retain original since changes are in place
fill_encode_embark(test_data)
# Sex encoding
test_data['Sex'] = test_data['Sex'].map(sex_mapping)
# Age prediction
test_data['Age'] = test_data.apply(
    lambda x: predict_encode_age(x,
                                  features=age_features,
                                  clf=age_clf
                                ),
    axis=1
)
test_data['Fare'] = test_data['Fare'].fillna(test_data['Fare'].median())
test_data['Fare'] = test_data['Fare'].apply(fare_groups)
test_data['Alone'] = test_data.apply(is_alone, axis=1)
test_data['Survived'] = survival_clf.predict(test_data[CLASS_FEATURES])
```

**Test Accuracy: 0.78947\***

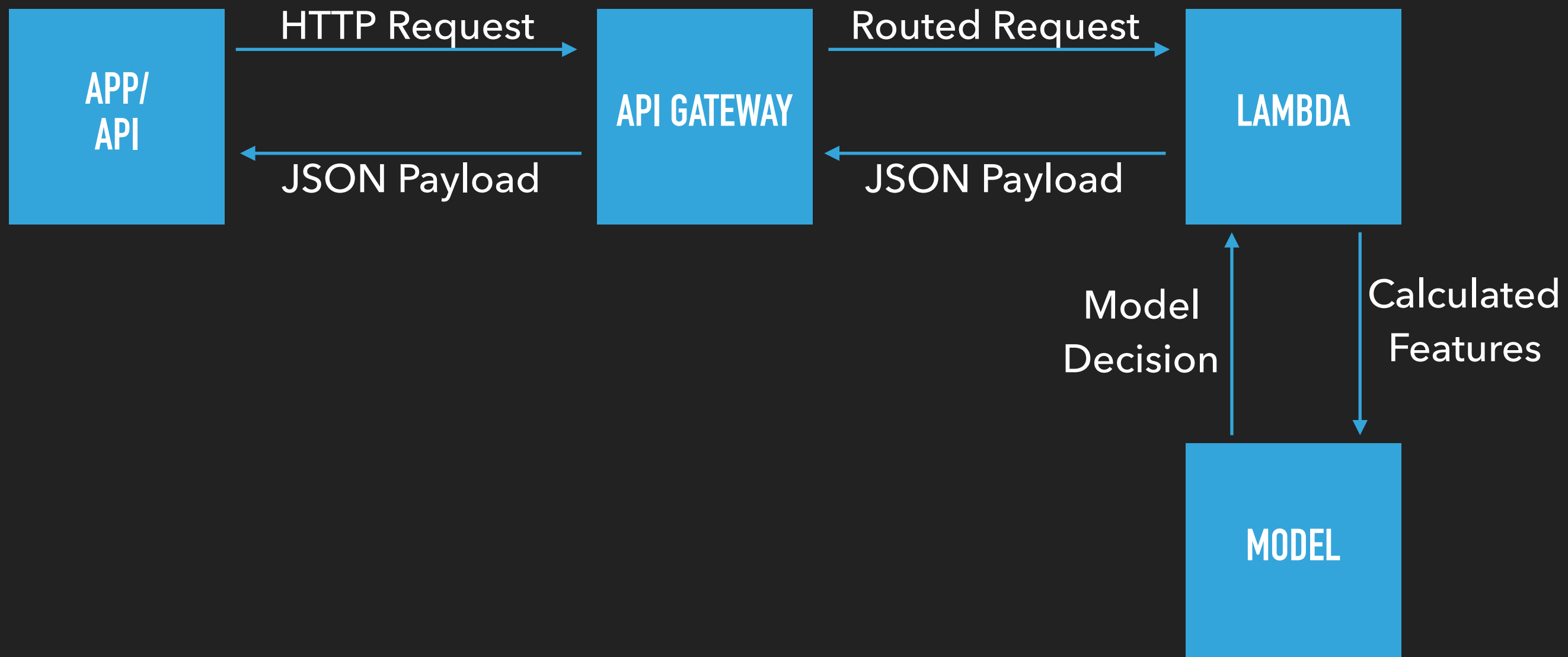
\*Baseline classifier (sex as label) score: 0.76555



**BUILD A WEB APP  
/ API SERVICE**



# ARCHITECTURE DIAGRAM



## API

## Flask App

Resource

Method

```
@app.route('/', methods=['GET'])
def index(name=None):
    """
    Main landing page
    :param name:
    :return:
    """
    return render_template('titanic.html', name=name)

@app.route('/titanic', methods=['POST'])
def predict_survival() -> Response:
    """
    Perform survival prediction based off form input
    :return: flask response with prediction output and status code 200
    """
    clf = _utils.load_model(_utils.MODEL_LOC,
                           _utils.MODEL_NAME)
    prediction = clf.predict(get_features())[0]
    return Response(
        format_prediction(prediction),
        200
    )
```

Return Prediction

⋮

## WEB APP

## Javascript

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <script src="static/js/script.js"></script>
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <title>Titanic Survival</title>
```

```
</head>
```

```
<body>
```

```
<h1>Titanic Survival</h1>
```

```
This is a small application to determine if a passenger would likely survive
the titanic.
```

```
<br>
```

```
<br>
```

```
<div class="container">
```

```
<form action="/dev/titanic" method="POST" enctype="multipart/form-data">
```

```
<table style="width:30%">
```

```
<tr>
```

```
<td><p align="center">Ticket Class:</p></td>
```

```
<td>
```

```
<select name="Pclass">
```

```
<option value="1">First</option>
```

```
<option value="2">Second</option>
```

```
<option value="3">Third</option>
```

```
</select>
```

```
</td>
```

```
</tr>
```

```
⋮
```

```
</table>
```

```
<br>
```

```
<input id="submit" type="submit" name="submit"
  style="height:30px;width:125px"/>
```

```
<br>
```

```
<br>
```

```
<p><strong>Result:</strong></p>
```

```
<pre class="output"></pre>
```

```
</form>
```

## Form &amp; Action

## User Input

## Result

# WEB APP

User Input  
Form

Send Input  
to API

Display Model  
Output

## Titanic Survival

This is a small application to determine if a passenger would likely survive the titanic.

Ticket Class:

Sex:

Age:

Spouse/Sibling Count:

Parent/Child Count:

Ticket Fare:

Embark Location:

Alone:

Submit

Result:

```
{
  "Survival": "Likely"
}
```

Created by [Brenton Mallen](#) ©2018

# WEB APP

## JavaScript

Link to Submit  
Button

Take Form  
Input

Make AJAX  
Call to API

Return Result  
(or Error)

```
$(function() {  
    $('input[type="submit"]').click(function(event) {  
        var $form = $(this).parent();  
        $form.find('.output').text("")  
        $.ajax({  
            url: $form.attr('action'),  
            data: $form.serialize(),  
            type: 'POST',  
            success: function(response) {  
                $form.find('.output').text(JSON.stringify(JSON.parse(response), null, 2))  
                console.log(response);  
            },  
            error: function(error) {  
                $form.find('.output').text(error.responseText)  
                console.log(error);  
            }  
        });  
        event.preventDefault()  
    });  
});
```

# DEPLOYMENT

## Zappa config\*

Application

Environment

IAM Role

Regions

```
{
  "dev": {
    "app_function": "titanic.app",
    "aws_region": "us-east-1",
    "profile_name": "default",
    "project_name": "titanic-survival",
    "runtime": "python3.6",
    "s3_bucket": "titanic-survival",
    "lambda_description": "Function to determine if an passenger would survive the titanic",
    "manage_roles": false,
    "role_name": "titanic-survival-dev-ZappaLambdaExecutionRole",
    "role_arn": "",
    "slim_handler": true
  },
  "dev_ap_northeast_1": {
    "aws_region": "ap-northeast-1",
    "extends": "dev"
  },
  "dev_ap_south_1": {
    "aws_region": "ap-south-1",
    "extends": "dev"
  },
  "dev_ap_southeast_1": {
    "aws_region": "ap-southeast-1",
    "extends": "dev"
  }
},
```

⋮

\* Zappa requires an AWS account as well as an IAM role and policy

# DEPLOYMENT

## Zappa Commands\*

**Deploy**

```
# -----  
#   ZAPPA  
# -----  
.PHONY: deploy  
deploy:  
    (source $(VENV)/activate && zappa deploy $(ZAPPA_ENV))
```

**Update**

```
.PHONY: redeploy  
redeploy:  
    (source $(VENV)/activate && zappa update $(ZAPPA_ENV))
```

**Destroy**

```
.PHONY: remove  
remove:  
    (source $(VENV)/activate && zappa undeploy $(ZAPPA_ENV))
```

**Inspect**

```
.PHONY: logs  
logs:  
    (source $(VENV)/activate && zappa tail $(ENV))
```

\* Commands shown have been compiled into a Makefile for convenience

**DEMO TIME**



## PRODUCTION NEXT STEPS

### **Continuous Integration / Deployment**

AWS CodePipeline/CodeBuild, Jenkins, etc.

### **Monitoring / Dashboards**

AWS Cloudwatch, DataDog, etc.

# POSSIBLE TWEAKS

## Classifier Model

- Perform grid search on hyper-parameters

- Try a different model or feature set

## App / API

- Improve app styling using CSS

- Add DNS to make it more approachable

- Add caching for performance

## ANOTHER APPLICATION

Using this methodology we can make other apps as well

[Board Game Similarity](#)



## QUESTIONS?

