Machine Learning Case Studies

# Lesson 1: Population Segmentation

## Introducing Cezanne & Dan

<https://www.youtube.com/watch?time_continue=7&v=2K8KFEUxNbw&feature=emb_logo>

If you have questions or Cezanne and Dan, or want to stay up-to-date with their latest projects, consider:

* Following [Dan on Twitter](https://twitter.com/dmbanga)
* Following [Cezanne on Twitter](https://twitter.com/cezannecam)

## Expert Interview: AWS SageMaker

In these exclusive interview segments, learn about SageMaker and how it is applied to real-world use cases. One of the values that SageMaker and Udacity share is that they want to make machine learning **accessible**. We do this through education, and they do it through making tools and scalable infrastructure available to learners and engineers.

Later in this course, you'll learn about how SageMaker has developed over time and hear some predictions about the future of ML-powered technology.

*Please view the segments that seem interesting to you!*

### How do you define SageMaker?

<https://www.youtube.com/watch?v=JWRtWcd92E4&feature=emb_logo>

### What applications does SageMaker make possible?

<https://www.youtube.com/watch?v=iXN30g70PJ0&feature=emb_logo>

### Why should students gain skills in SageMaker and cloud services?

<https://www.youtube.com/watch?v=Hp6qTdiqU3g&feature=emb_logo>

## Course Outline

Throughout this course, we’ll be focusing on deployment tools and the machine learning workflow; answering a few big questions along the way:

* How do you decide on the correct machine learning algorithm for a given task?
* How can we utilize cloud ML services in SageMaker to work with interesting datasets or improve our algorithms?

To approach these questions, we’ll go over a number of real-world **case studies**, and go from task and problem formulation to deploying models in SageMaker. We’ll also utilize a number of SageMaker’s built-in algorithms.

### Case Studies

Case studies are in-depth examinations of specific, real-world tasks. In our case, we’ll focus on three different problems and look at how they can be solved using various machine learning strategies. The case studies are as follows:

***Case Study 1 - Population Segmentation using SageMaker***

You’ll look at a portion of [US census data](https://www.census.gov/data.html) and, using a combination of unsupervised learning methods, extract meaningful components from that data and group regions by similar census-recorded characteristics. This case study will be a deep dive into Principal Components Analysis (PCA) and K-Means clustering methods, and the end result will be groupings that are used to inform things like localized marketing campaigns and voter campaign strategies.

***Case Study 2 - Detecting Credit Card Fraud***

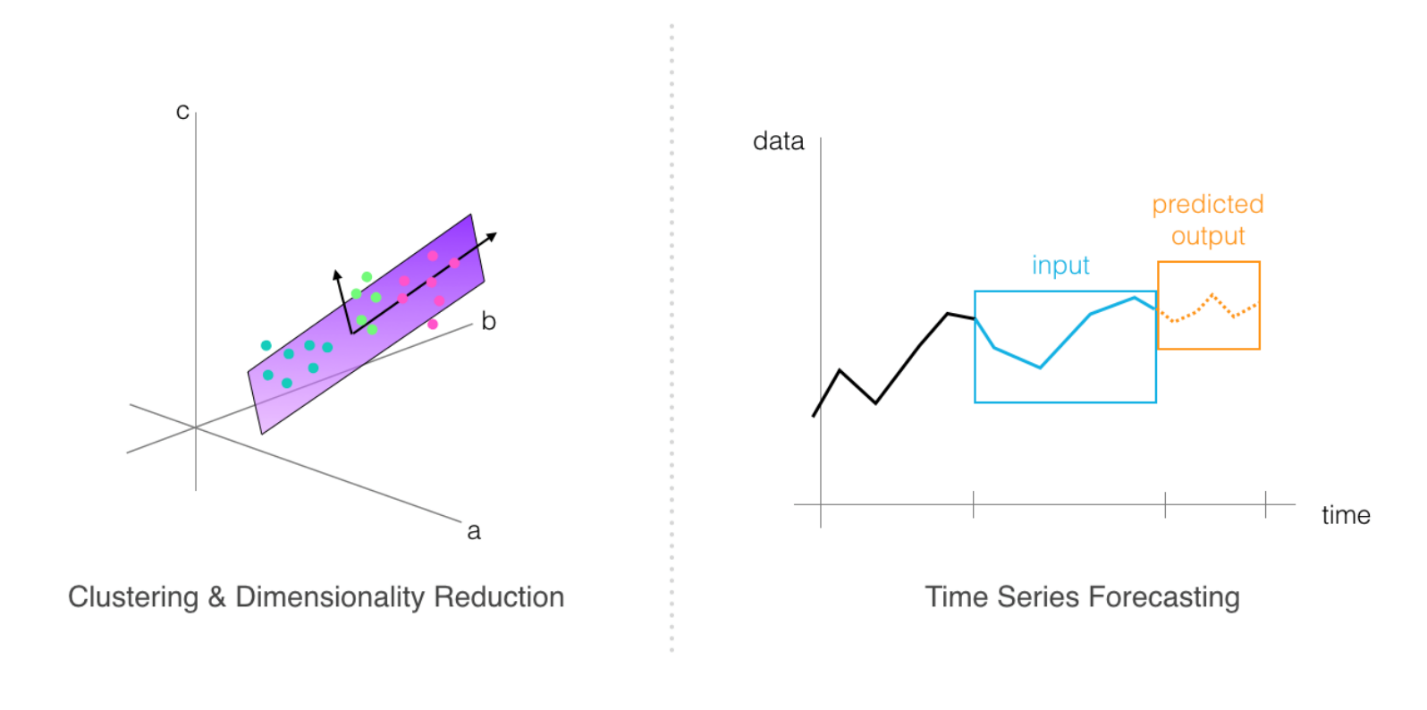
This case will demonstrate how to use supervised learning techniques, specifically SageMaker’s LinearLearner, for fraud detection. The payment transaction dataset we'll work with is unbalanced, with many more examples of valid transactions vs. fraudulent, and so you will investigate methods for compensating for this imbalance and tuning your model to improve its performance according to a specific product goal.

***Custom Models - Non-Linear Classification***

Adding on to what you have learned in the credit card fraud case study, you will learn how to manage cases where classes of data are not separable by a linear line. You'll train and deploy a custom, PyTorch neural network for classifying data.

***Case Study 3 - Time-Series Forecasting***

This case demonstrates how to train SageMaker's DeepAR model for forecasting predictions over time. Time-series forecasting is an active area of research because a good forecasting algorithm often takes in a number of different features and accounts for seasonal or repetitive patterns. In this study, you will learn a bit about creating features out of time series data and formatting it for training.



***Project: Plagiarism Detection***

You'll apply the skills that you've learned to a final project; building and deploying a plagiarism classification model. This project will test your ability to do [text] data processing and feature extraction, your ability to train and evaluate models according to an accuracy specification, and your ability to deploy a trained model to an endpoint.

By the end of this course, you should have all the skills you need to build, train and deploy models to solve tasks of your own design!

## Unsupervised vs Supervised Learning

<https://www.youtube.com/watch?time_continue=7&v=9M6T9Bx3oNA&feature=emb_logo>

## Model Design

<https://www.youtube.com/watch?time_continue=1&v=zxNoSTZ3s90&feature=emb_logo>

## Population Segmentation

<https://www.youtube.com/watch?time_continue=1&v=3pXFLrnk7q0&feature=emb_logo>

## K-Means Clustering

To perform population segmentation, one of our strategies will be to use k-means clustering to group data into similar clusters. To review, the k-means clustering algorithm can be broken down into a few steps; the following steps assume that you have n-dimensional data, which is to say, data with a discrete number of features associated with it. In the case of housing price data, these features include traits like house size, location, etc. **features** are just measurable components of a data point. K-means works as follows:

You select k, a predetermined number of clusters that you want to form. Then k points (centroids for k clusters) are selected at random locations in feature space. For each point in your training dataset:

1. You find the centroid that the point is closest to
2. And assign that point to that cluster
3. Then, for each cluster centroid, you move that point such that it is in the center of all the points that are were assigned to that cluster in step 2.
4. Repeat steps 2 and 3 until you’ve either reached convergence and points no longer change cluster membership \_or\_ until some specified number of iterations have been reached.

This algorithm can be applied to any kind of unlabelled data. You can watch a video explanation of the k-means algorithm, as applied to color image segmentation, below. In this case, the k-means algorithm looks at R, G, and B values as features, and uses those features to cluster individual pixels in an image!

### Color Image Segmentation

<https://www.youtube.com/watch?v=Cf_LSDCEBzk&feature=emb_logo>

### Data Dimensionality

One thing to note is that it’s often easiest to form clusters when you have low-dimensional data. For example, it can be difficult, and often noisy, to get good clusters from data that has over 100 features. In high-dimensional cases, there is often a dimensionality reduction step that takes place before data is analyzed by a clustering algorithm. We’ll discuss PCA as a dimensionality reduction technique in the practical code example, later.

## The Github Repository

<https://www.youtube.com/watch?time_continue=200&v=w2GBAnhUlOw&feature=emb_logo>

You can find a link to all of the exercise and project code for this course in the repository: <https://github.com/udacity/ML_SageMaker_Studies>. Copy and paste this repository into the Github clone option when you create your notebook instance!

**Note:** Once a notebook instance has been set up, by default, it will be **InService** which means that the notebook instance is running. This is important to know because the cost of a notebook instance is based on the length of time that it has been running. This means that once you are finished using a notebook instance you should **Stop** it so that you are no longer incurring a cost. Don't worry though, you won't lose any data provided you don't delete the instance. Just start the instance back up when you have time and all of your saved data will still be there.

## Notebook: Population Segmentation, Exercise

Now, you're ready to approach the task of population segmentation! As you follow along with this lesson, you are encouraged to open the referenced SageMaker notebooks. We will present a solution to you, but please try to work on a solution of your own, when prompted. Much of the value in this experience will come from experimenting with the code, **in your own way**.

To open this notebook:

* *Navigate to your SageMaker notebook instance, in the [SageMaker console](https://console.aws.amazon.com/sagemaker/" \t "_blank), which has been linked to the main [Github exercise repository](https://github.com/udacity/ML_SageMaker_Studies" \t "_blank)*
* *Activate the notebook instance (if it is in a "Stopped" state), and open it via Jupyter*
* *Click on the exercise notebook in the Population\_Segmentation directory.*

You may also directly view the exercise and solution notebooks via the repository at the following links:

* [Exercise notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Population_Segmentation/Pop_Segmentation_Exercise.ipynb)
* [Solution notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Population_Segmentation/Pop_Segmentation_Solution.ipynb)

**The solution notebook is meant to be consulted if you are stuck or want to check your work.**

### Notebook Outline

We'll go over the following steps to complete the notebook.

* Load in and explore population data
* Perform dimensionality reduction with a deployed PCA model
* Cluster components with K-Means
* Visualize the results

### Later: Delete Resources

At the end of this exercise, and intermittently, you will be reminded to delete your endpoints and resources so that you do not incur any extra processing or storage fees!

## Exercise: Data Loading & Processing

<https://www.youtube.com/watch?time_continue=144&v=YlG9T17KcbU&feature=emb_logo>

## Solution: Data Loading & Processing

<https://www.youtube.com/watch?time_continue=65&v=2jUouM70A1I&feature=emb_logo>

## Exercise: Range of Values

One thing you may have noticed, especially when creating density plots and comparing feature values, is that that the values in each feature column are in quite a wide range; some are very large numbers or small, floating points.

Now, our end goal is to cluster this data and clustering relies on looking at the perceived similarities and differences between features! So, we want our model to be able to look at these columns and **consistently** measure the relationships between features.

*To make sure the feature measurements are consistent and comparable, you’ll scale all of the*numerical*features into a range between 0 and 1. This is a pretty typical****normalization****step.*

Below, is what the exercise looks like in the main notebook.

### EXERCISE: Normalize the data

You need to standardize the scale of the numerical columns in order to consistently compare the values of different features. You can use a MinMaxScaler to transform the numerical values so that they all fall between 0 and 1.

*# scale numerical features into a normalized range, 0-1*

*# store them in this dataframe*

counties\_scaled = **None**

**Try to complete this task on your own**, in the exercise notebook, and if you get stuck or want to consult a solution, I’ll go over my solution, next.

## Solution: Range of Values

<https://www.youtube.com/watch?time_continue=2&v=UDWwdG4e1a0&feature=emb_logo>

## PCA

Principal Component Analysis (PCA) attempts to reduce the number of features within a dataset while retaining the “principal components”, which are defined as weighted combinations of existing features that:

1. Are uncorrelated with one another, so you can treat them as independent features, and
2. Account for the largest possible variability in the data!

So, depending on how many components we want to produce, the first one will be responsible for the largest variability on our data and the second component for the second-most variability, and so on. Which is exactly what we want to have for clustering purposes!

PCA is commonly used when you have data with many many features.

You can learn more about the details of the PCA algorithm in the video, below.

<https://www.youtube.com/watch?time_continue=6&v=uyl44T12yU8&feature=emb_logo>

*Now, in our case, we have data that has 34-dimensions and we’ll want to use PCA to find combinations of features that produce the most variability in the population dataset.*

The idea is that components that cause a larger variance will help us to better differentiate between data points and (therefore) better separate data into clusters.

So, next, I’ll go over how to use **SageMaker’s built-in PCA model** to analyze our data.

## PCA Estimator & Training

<https://www.youtube.com/watch?time_continue=6&v=HGEqgi2MKcU&feature=emb_logo>

## Exercise: PCA Model Attributes & Variance

<https://www.youtube.com/watch?v=dumVafbS7pk&feature=emb_logo>

## Solution: Variance

<https://www.youtube.com/watch?time_continue=1&v=C-BRBjxlUuE&feature=emb_logo>

## Component Makeup

<https://www.youtube.com/watch?time_continue=1&v=fiSr_Xjm3qI&feature=emb_logo>

## Exercise: PCA Deployment & Data Transformation

<https://www.youtube.com/watch?time_continue=1&v=qsnpHHuwbbA&feature=emb_logo>

## Solution: Creating Transformed Data

<https://www.youtube.com/watch?time_continue=24&v=4l2UHyyVV7Y&feature=emb_logo>

## Exercise: K-means Estimator & Selecting K

### Creating a KMeans Estimator

Now that we’ve run the original data through PCA and moved from 34-dimensional data to 7-dimensional component data, we’re better prepared to actually cluster this data! So, now, I’ll ask you to use these components that we’ve gotten from our training data and cluster counties using k-means.

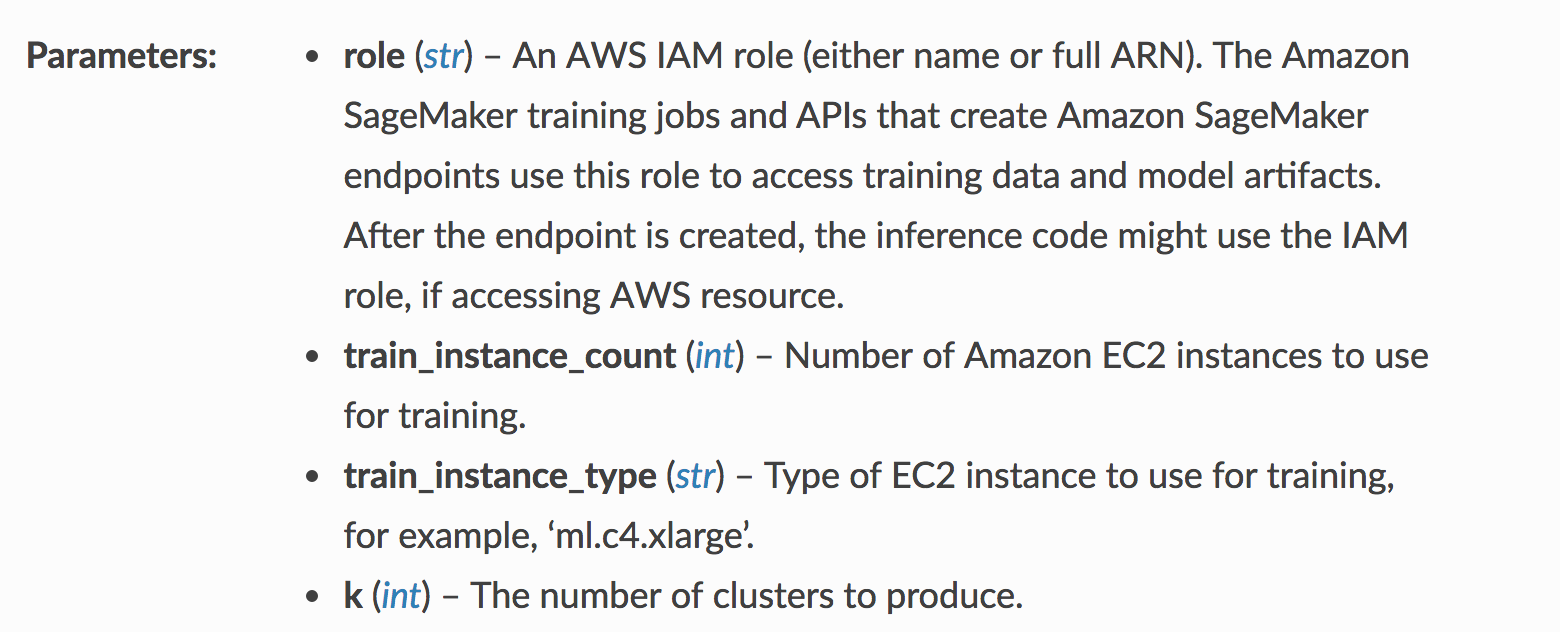
You'll instantiate a KMeans estimator, by specifying specific model arguments and passing them into a KMeans constructor ([documentation, here](https://sagemaker.readthedocs.io/en/stable/kmeans.html)). Knowing how to read documentation is an important skill for learning to create models on your own!

Here is what this exercise looks like in the main, exercise notebook:

#### EXERCISE: Define a k-means model

Your task will be to instantiate a k-means model. A KMeans estimator requires a number of parameters to be instantiated, which allow us to specify the type of training instance to use, and the model hyperparameters.

*# define a KMeans estimator*



Some parameters in the KMeans documentation

### General Estimator Parameters

From the documentation, you can see that you'll need to specify the IAM role (which we defined when creating the notebook instance), and details about the instance type to use for training.

Most of SageMaker's built-in algorithms are based off of an [EstimatorBase object](https://sagemaker.readthedocs.io/en/stable/estimators.html" \l "sagemaker.estimator.EstimatorBase" \t "_blank), which allows you to specify additional parameters. It is good practice to be specific about two additional parameters:

* **output\_path (str)** – S3 location for saving the training result (model artifacts and output files). If not specified, results are stored to a default bucket.
* **sagemaker\_session (sagemaker.session.Session)** – Session object which manages interactions with Amazon SageMaker APIs and any other AWS services needed.

### Model-Specific Parameters

You'll also notice a parameter k for the number of clusters this model should produce as output. This parameter is specific to the k-means model, and for different models, you'll see different required model parameters.

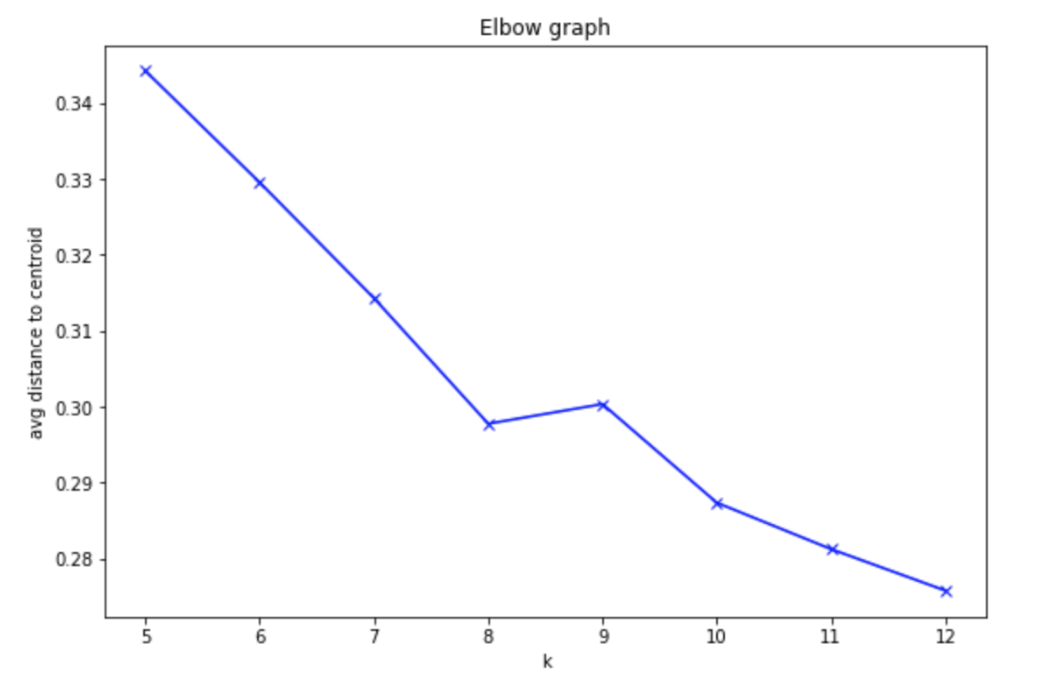
### Choosing a "Good" K

One method for choosing a "good" k, is to choose based on empirical data.

* A bad k would be one so high that only one or two very close data points are near it, and
* Another bad k would be one so low that data points are really far away from the centers.

You want to select a k such that data points in a single cluster are close together but that there are enough clusters to effectively separate the data. You can approximate this separation by measuring how close your data points are to each cluster center; the average centroid distance between cluster points and a centroid. After trying several values for k, the centroid distance typically reaches some "elbow"; it stops decreasing at a sharp rate and this indicates a good value of k.

The graph below indicates the average distance—between our component data and cluster centroids—for a value of k between 5 and 12.



### Training Job

After creating a KMeans estimator, I also want you to proceed with creating a training job. You'll have to format your data correctly for this job and make sure you are passing in the reduced-dimensionality training data. It may be helpful to reference the PCA training job code.

Here is what these exercises look like in the exercise notebook:

#### EXERCISE: Create formatted, k-means training data

Just as before, you should convert the counties\_transformed df into a numpy array and then into a RecordSet. This is the required format for passing training data into a KMeans model.

*# convert the transformed dataframe into record\_set data*

#### EXERCISE: Train the k-means model

Pass in the formatted training data and train the k-means model.

%%time

*# train kmeans*

After you are done with these steps, you can move on to model deployment!

## Exercise: K-means Predictions (clusters)

### Getting Predicted Clusters

After you've trained your KMeans estimator, you can deploy it and apply it to our data to get resultant clusters.

#### EXERCISE: Deploy the k-means model

Deploy the trained model to create a kmeans\_predictor.

%%time

*# deploy the model to create a predictor*

kmeans\_predictor = **None**

#### EXERCISE: Pass in the training data and assign predicted cluster labels

After deploying the model, you can pass in the k-means training data, as a numpy array, and get resultant, predicted cluster labels for each data point.

*# get the predicted clusters for all the kmeans training data*

cluster\_info=**None**

If you finish this exercise, you should be able to proceed with some interesting visualizations that give you the ability to explore how counties are clustered and what that means as far as features that define the similarity between counties.

### Shutting Down the Endpoint

After you successfully make predictions and assign each county to a cluster, you can delete your KMenas endpoint.

## Solution: K-means Predictor

<https://www.youtube.com/watch?time_continue=1&v=0xx2p2vnCg0&feature=emb_logo>

## Exercise: Get the Model Attributes

## Model Attributes & Explainability

Explaining the result of the modeling is an important step in making use of our analysis. By combining PCA and k-means, and the information contained in the model attributes within a SageMaker trained model, you can learn about a population and remark on some patterns you've found, based on the data.

To access the k-means model attributes, you'll find the following guidance in the main, exercise notebook.

### EXERCISE: Access the k-means model attributes

Extract the k-means model attributes from where they are saved as a TAR file in an S3 bucket.

You'll need to access the model by the k-means training job name, and then unzip the file into model\_algo-1. Then you can load that file using MXNet, as before.

*# download and unzip the kmeans model file*

*# use the name model\_algo-1*

*# get the trained kmeans params using mxnet*

kmeans\_model\_params = **None**

print(kmeans\_model\_params)

Save the model attributes as kmeans\_model\_params; you should see that there is only 1 set of model parameters contained within the k-means model: the **cluster centroid locations** in PCA-transformed, component space.

### Cluster Centroids

You know that each of the counties in our US county data, is assigned to a cluster and that indicates something about groupings found by k-means and similarities between counties in the same cluster. But, what exactly are the features that these clusters have in common?

*For example, how is cluster 1 any different than cluster 2?*

This is what we aim to define by looking at the location of cluster centroids in component space. Since each cluster is defined in component space, we can look at how eighty each component is in defining a certain cluster. Then, we can go one step further and map the components back to the original data features. I encourage you to look at the visualization code in the exercise notebook and watch the next video to see some complete code.

## Solution: Model Attributes

<https://www.youtube.com/watch?v=VS-hVhsCBPw&feature=emb_logo>

## Clean up Resources

It is good practice to always clean up and delete any resources that you are no longer using. That is, after you complete an exercise, and you are done with predictions and data analysis, you should get rid of any:

* Data source in S3 that you are no longer using
* Endpoint configuration files that you no longer need
* Endpoints that you will no longer use
* CloudWatch logs that are no longer useful

### Deleting Endpoints

In the notebook, we have usually included code to delete your endpoints after creating some predictions, for example:

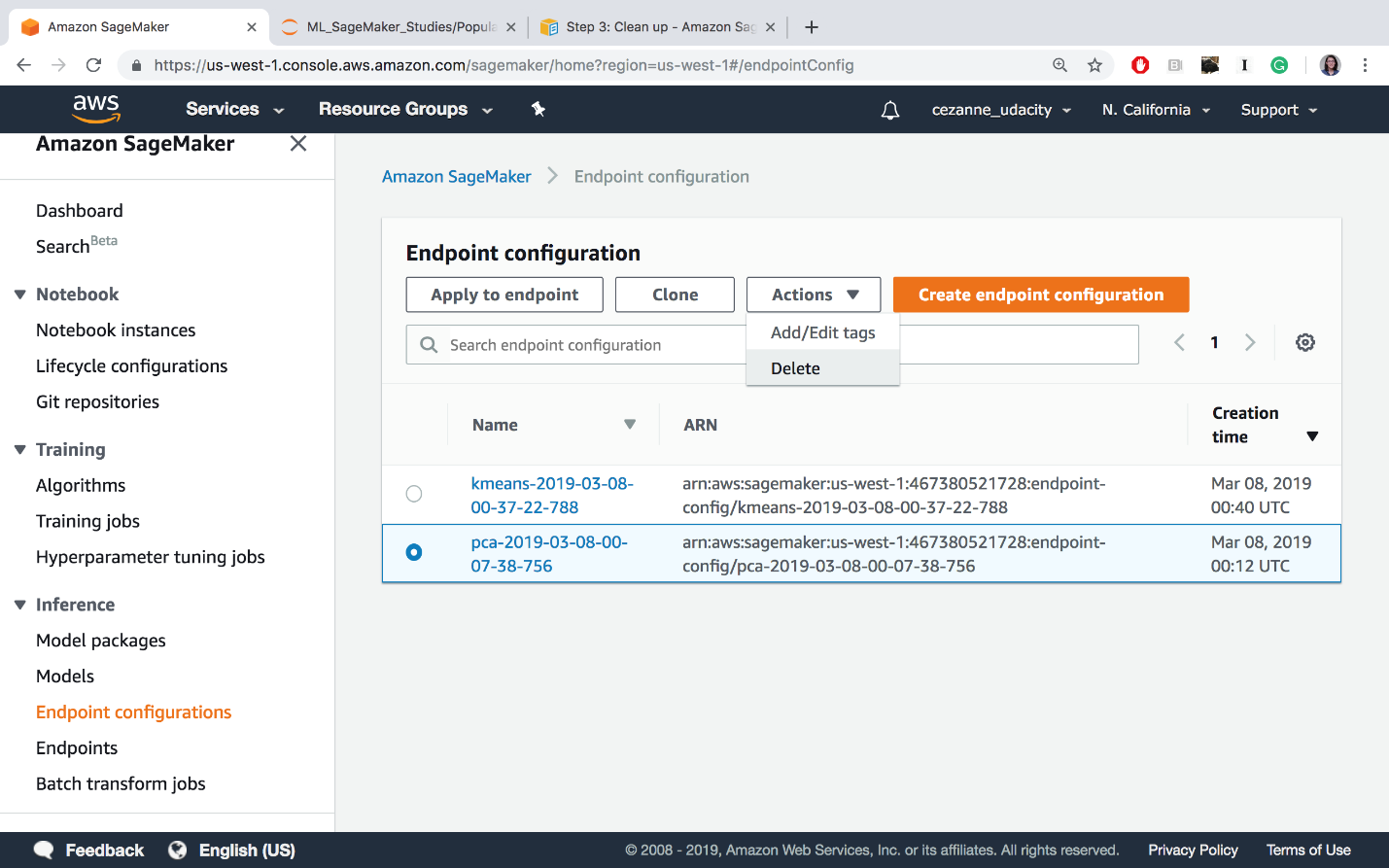
*# delete predictor endpoint*

session.delete\_endpoint(predictor.endpoint)

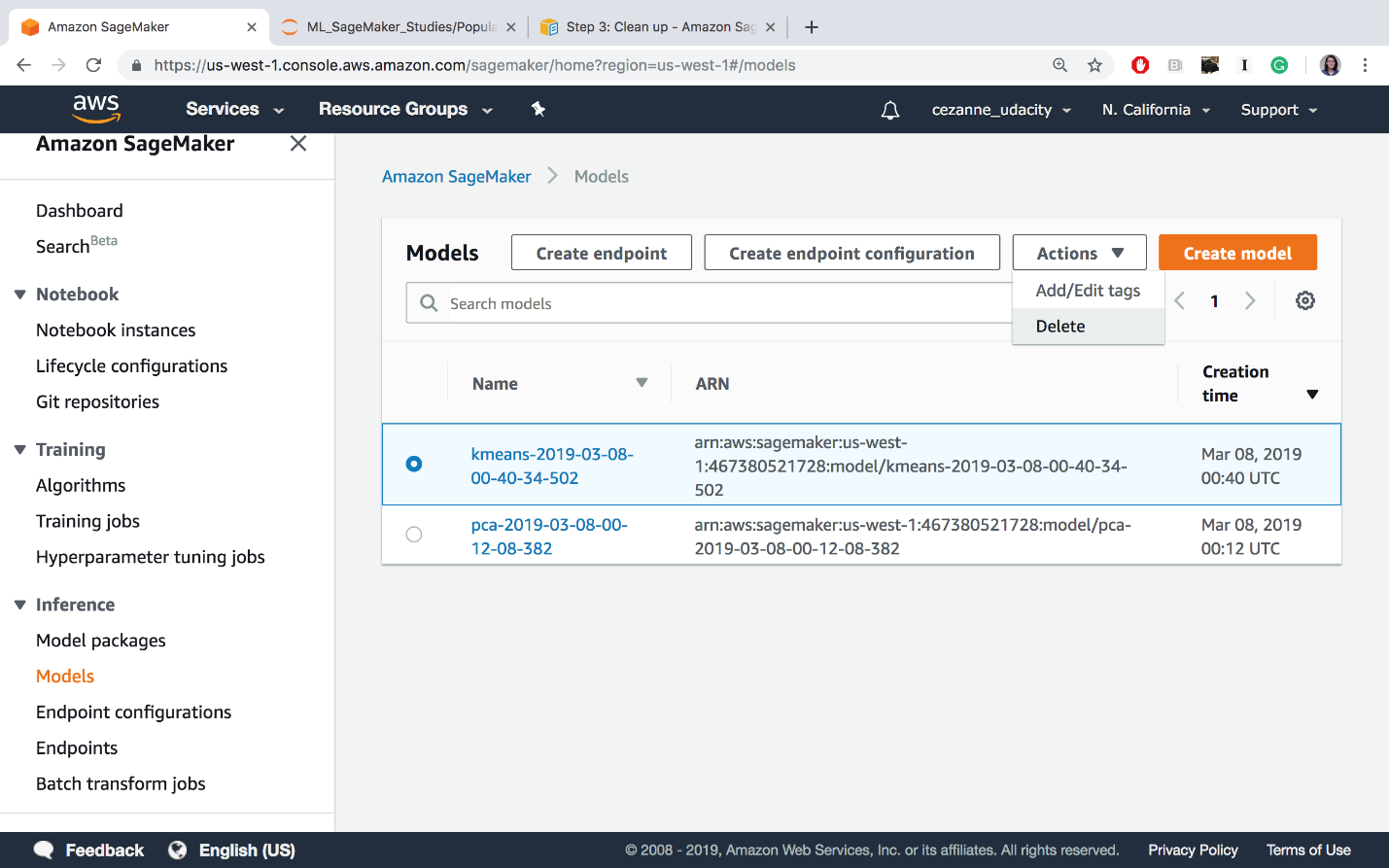
### Thorough Clean up

You can find a link for instructions on cleaning up all your resources, [in this documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html) and I will go over some of these details, next.

* Open the Amazon SageMaker console at <https://console.aws.amazon.com/sagemaker/> and delete the following resources:
* The endpoint configuration.
* The model.

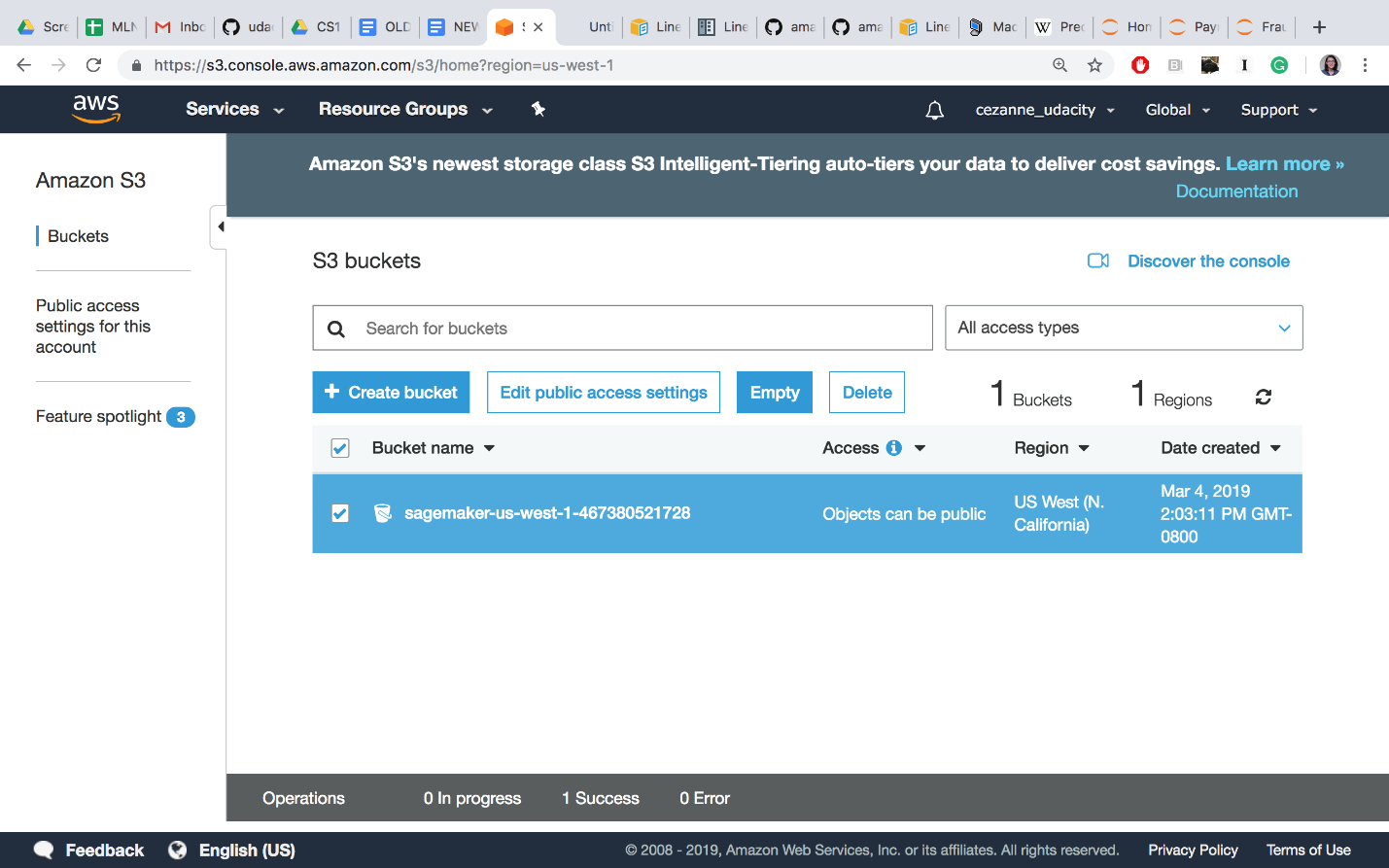


Delete endpoint config files.



Deleting models

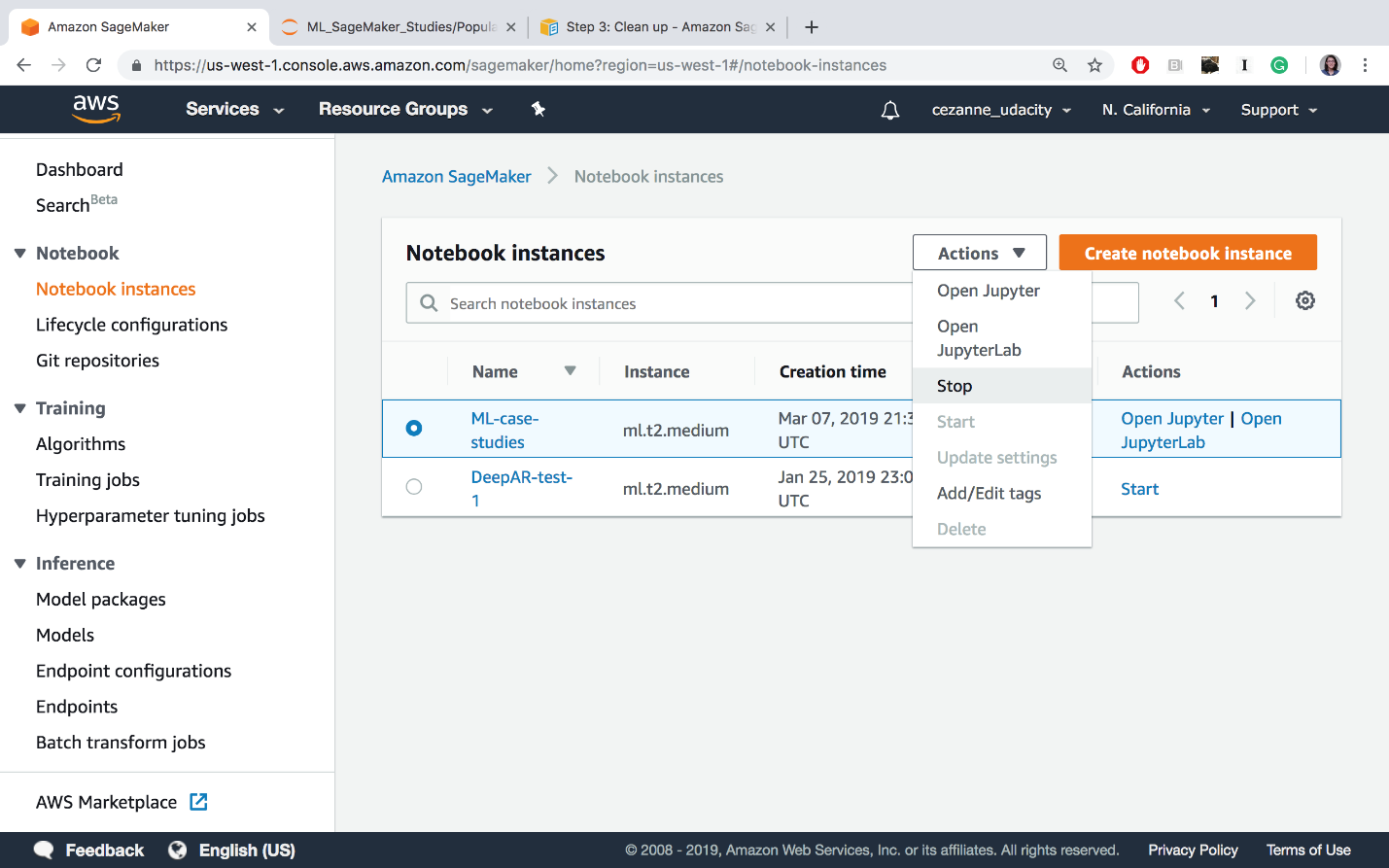
* Open the Amazon S3 console at <https://console.aws.amazon.com/s3/> and delete or empty the bucket that you created for storing model artifacts and the training dataset.



Delete or empty your S3 bucket (empty is recommended until the end of the course, when you should delete this bucket entirely)

* Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/> and delete all of the log groups that have names starting with /aws/sagemaker/.

At the end of this course, you may also choose to delete the entire notebook instance and IAM Role, but you may keep these as is, for now. In between lessons, if you are taking a break, you may want to **Stop** your notebook and pause it from continuously running.



Stopping the ML-case-studies notebook

Cleaning up resources at the end of an exercise or lesson is a great practice to get into!

### IMPORTANT

**To avoid incurring additional charges, it is suggested that you DELETE any unused notebooks and data resources on S3 and CloudWatch.**

## AWS Workflow & Summary

<https://www.youtube.com/watch?time_continue=1&v=vMLN832942E&feature=emb_logo>

# Lesson 2: Payment Fraud Detection

## Preview: Fraud Detection

<https://www.youtube.com/watch?time_continue=2&v=zDnyR5Tci5M&feature=emb_logo>

## Exercise: Payment Transaction Data

<https://www.youtube.com/watch?time_continue=22&v=bF65I3J6aqQ&feature=emb_logo>

## Solution: Data Distribution & Splitting

<https://www.youtube.com/watch?time_continue=4&v=Cjn82LqTB00&feature=emb_logo>

## LinearLearner & Class Imbalance

<https://www.youtube.com/watch?v=pjs5pP9OOMc&feature=emb_logo>

## Exercise: Define a LinearLearner

### Instantiate a LinearLearner

Now that you've uploaded your training data, it's time to define and train a model! In the main exercise notebook, you'll define and train the SageMaker, built-in algorithm, LinearLearner.

#### EXERCISE: Create a LinearLearner Estimator

You've had some practice instantiating built-in models in SageMaker. All estimators require some constructor arguments to be passed in.

*See if you can complete this task, instantiating a LinearLearner estimator, using only the [LinearLearner documentation](https://sagemaker.readthedocs.io/en/stable/linear_learner.html" \t "_blank) as a resource.*

You'll find that this estimator takes in a lot of arguments, but not all are required. My suggestion is to start with a simple model and utilize default values where applicable. Later, we will discuss some specific hyperparameters and their use cases.

#### Instance Types

It is suggested that you use instances that are available in the free tier of usage: 'ml.c4.xlarge' for training and 'ml.t2.medium' for deployment.

Here is what the exercise code looks like in the main notebook:

*# import LinearLearner*

**from** sagemaker **import** LinearLearner

*# instantiate LinearLearner*

Try to complete this code on your own, and I'll go over one possible solution, next!

## Solution: Default LinearLearner

<https://www.youtube.com/watch?time_continue=5&v=WaqDbA_5dNE&feature=emb_logo>

## Exercise: Format Data & Train the LinearLearner

## Train your Estimator

After defining a model, you can format your training data and call .fit() to train the LinearLearner.

In the notebook, these exercises look as follows:

### EXERCISE: Convert data into a RecordSet format

Prepare the data for a built-in model by converting the train features and labels into numpy array's of float values. Then you can use the record\_set function to format the data as a RecordSet and prepare it for training!

*# create RecordSet of training data*

formatted\_train\_data = **None**

### EXERCISE: Train the Estimator

After instantiating your estimator, train it with a call to .fit(), passing in the formatted training data.

%%time

*# train the estimator on formatted training data*

Complete this code, and you may check out a solution, next!

## Solution: Training Job

<https://www.youtube.com/watch?time_continue=1&v=-whnaHFkPxU&feature=emb_logo>

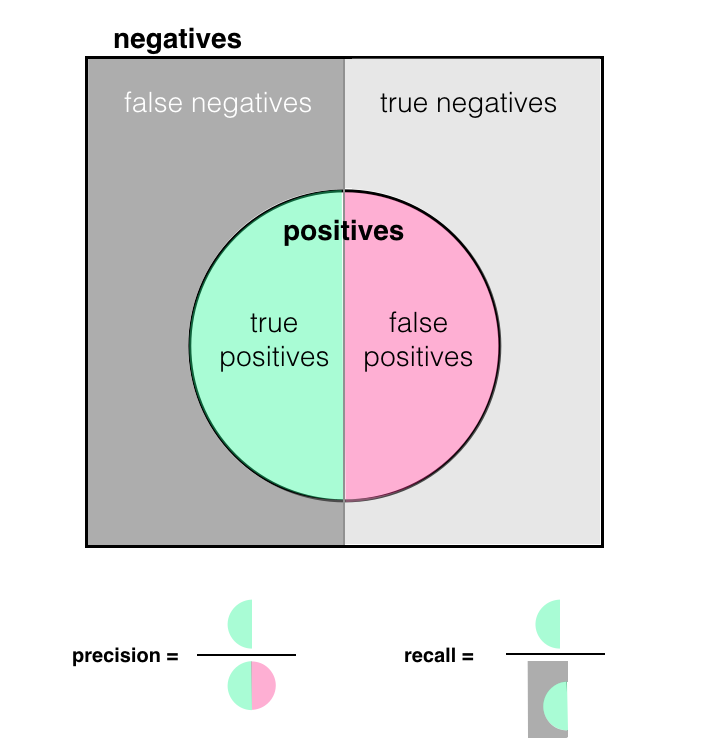
## Precision & Recall

Precision and recall are just different metrics for measuring the "success" or performance of a trained model.

* **precision** is defined as the number of true positives (truly fraudulent transaction data, in this case) over all positives, and will be the higher when the amount of false positives is low.
* **recall** is defined as the number of true positives over true positives plus false negatives and will be higher when the number of false negatives is low.

Both take into account true positives and will be higher for high, positive accuracy, too.

I find it helpful to look at the below image to wrap my head around these measurements:



In many cases, it may be worthwhile to optimize for a higher recall or precision, which gives you a more granular look at false positives and negatives.

## Exercise: Deploy Estimator

### Deploy an Endpoint and Evaluate Predictions

Finally, you are ready to deploy your trained LinearLearner and see how it performs according to various metrics. As you evaluate this model, I want you to think about:

* Which metrics best define success for this model?
* Is it important that we catch all cases of fraud?
* Is it important to prioritize a smooth user experience and never flag valid transactions?

The answers to these questions may vary based on use case!

In the main exercise notebook, you'll see the following instructions for deploying an endpoint and using it to make predictions:

#### EXERCISE: Deploy the trained model

Deploy your model to create a predictor. We'll use this to make predictions on our test data and evaluate the model.

%%time

*# deploy and create a predictor*

linear\_predictor = **None**

#### Evaluating Your Model

Once your model is deployed, you can see how it performs when applied to the test data. Let's first test our model on just one test point, to see the resulting list.

*# test one prediction*

test\_x\_np = test\_features.astype('float32')

result = linear\_predictor.predict(test\_x\_np[0])

print(result)

You should proceed with investigating and evaluating the model test results. And next, I will discuss the results I got after deploying.

#### Shutting Down an Endpoint

*As always, after deploying a model and making/saving predictions, you are free to delete your model endpoint and clean up that resource.*

## Solution: Deployment & Evaluation

<https://www.youtube.com/watch?time_continue=14&v=ZknaWInjSa4&feature=emb_logo>

## Model Improvements

<https://www.youtube.com/watch?time_continue=3&v=JjZMuUnxKw4&feature=emb_logo>

## Improvement, Model Tuning

<https://www.youtube.com/watch?time_continue=2&v=bb7zG0TdtRM&feature=emb_logo>

## Exercise: Improvement, Class Imbalance

### Model Improvement: Accounting for Class Imbalance

We have a model that is tuned to get a higher recall, which aims to reduce the number of **false negatives**. Earlier, we discussed how class imbalance may actually bias our model towards predicting that all transactions are valid, resulting in higher **false negatives and true negatives**. It stands to reason that this model could be further improved if we account for this imbalance!

To account for class imbalance during training of a binary classifier, LinearLearner offers the hyperparameter, positive\_example\_weight\_mult, which is the weight assigned to positive (fraudulent data) examples when training a binary classifier. The weight of negative examples (valid data) is fixed at 1.

From the [hyperparameter documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ll_hyperparameters.html) on positive\_example\_weight\_mult, it reads:

*"If you want the algorithm to choose a weight so that errors in classifying negative vs. positive examples have equal impact on training loss, specify balanced."*

In the main exercise notebook, your exercises from defining to deploying an improved model looks as follows:

#### EXERCISE: Create a LinearLearner with a positive\_example\_weight\_mult parameter

In addition to tuning a model for higher recall, you should add a parameter that helps account for class imbalance.

*# instantiate a LinearLearner*

*# include params for tuning for higher recall*

*# \*and\* account for class imbalance in training data*

linear\_balanced = **None**

#### EXERCISE: Train the balanced estimator

Fit the new, balanced estimator on the formatted training data.

%%time

*# train the estimator on formatted training data*

#### EXERCISE: Deploy and evaluate the balanced estimator

Deploy the balanced predictor and evaluate it. Do the results match with your expectations?

%%time

*# deploy and create a predictor*

balanced\_predictor = **None**

An important question here, when evaluating your model, is: **Do the results match with your expectations?** Much like in a scientific experiment it is good practice to start with a hypothesis that drives your idea for improving a model; if the trained model reacts in a different way than you expect (i.e. the model metrics are worse), it is worth revisiting your assumptions and approach.

Try to complete all these tasks, and if you get stuck, you can reference the solution video, next!

#### Shutting Down the Endpoint

*Remember to delete your deployed, model endpoint after you finish with evaluation.*

## Solution: Accounting for Class Imbalance

<https://www.youtube.com/watch?v=ncoPZdiVLJg&feature=emb_logo>

## Exercise: Define a Model w/ Specifications

### Model Design

Now that you've seen how to tune and balance a LinearLearner, it is your turn to put together all that you've learned and build a new model, based on a real, business problem. This exercise is meant to be more open-ended, so that you get practice with the steps involved in designing a model and deploying it. In this exercise you'll:

* Create a LinearLearner model, according to specifications
* Train and deploy the model
* Evaluate the results
* Delete the endpoint (after evaluation)

Here is what you'll see in the main exercise notebook:

#### EXERCISE: Train and deploy a LinearLearner with appropriate hyperparameters, according to the given scenario

**Scenario:**

*A bank has asked you to build a model that optimizes for a good user experience; users should only ever have up to about 15% of their valid transactions flagged as fraudulent.*

This requires that you make a design decision: Given the above scenario, **what metric (and value)** should you aim for during training?

You may assume that performance on a training set will be within about 5-10% of the performance on a test set. For example, if you get 80% on a training set, you can assume that you'll get between about 70-90% accuracy on a test set.

**Your final model should account for class imbalance and be appropriately tuned.**

%%time

*# instantiate and train a LinearLearner*

*# include params for tuning for higher precision*

*# \*and\* account for class imbalance in training data*

%%time

*# deploy and evaluate a predictor*

*## IMPORTANT*

*# delete the predictor endpoint after evaluation*

In this case, I will not be walking through a detailed solution (and there are multiple ways to approach this task and come up with a solution), but you can see one example solution in the solution notebook and on the next page.

#### Final Cleanup!

After completing these tasks, double check that you have deleted **all** your endpoints, and associated files. I'd also suggest manually deleting your S3 bucket, models, and endpoint configurations directly from your AWS console. You can find thorough cleanup instructions, [in the documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html).

## One Solution: Tuned and Balanced LinearLearner

To optimize for few false positives (misclassified, valid transactions), I defined a model that accounts for class imbalance and optimizes for a **high precision**.

Let's review the scenario:

*A bank has asked you to build a model that optimizes for a good user experience; users should only ever have up to about 15% of their valid transactions flagged as fraudulent.*

My thoughts: If we're allowed about 15/100 incorrectly classified valid transactions (false positives), then I can calculate an approximate value for the precision that I want as: 85/(85+15) = 85%. I'll aim for about 5% higher during training to ensure that I get closer to 80-85% precision on the test data.

%%time

*# One possible solution*

*# instantiate and train a LinearLearner*

*# include params for tuning for higher precision*

*# \*and\* account for class imbalance in training data*

linear\_precision = LinearLearner(role=role,

train\_instance\_count=1,

train\_instance\_type='ml.c4.xlarge',

predictor\_type='binary\_classifier',

output\_path=output\_path,

sagemaker\_session=sagemaker\_session,

epochs=15,

binary\_classifier\_model\_selection\_criteria='recall\_at\_target\_precision',

target\_precision=0.9,

positive\_example\_weight\_mult='balanced')

*# train the estimator on formatted training data*

linear\_precision.fit(formatted\_train\_data)

This model trains for a fixed precision of 90%, and, under that constraint, tries to get as high a recall value as possible. After training, I deployed the model to create a predictor:

%%time

*# deploy and evaluate a predictor*

precision\_predictor = linear\_precision.deploy(initial\_instance\_count=1, instance\_type='ml.t2.medium')

*INFO:sagemaker:Creating model with name: linear-learner-2019-03-11-04-07-10-993 INFO:sagemaker:Creating endpoint with name linear-learner-2019-03-11-03-36-56-524*

Then evaluated the model by seeing how it performed on test data:

print('Metrics for tuned (precision), LinearLearner.\n')

*# get metrics for balanced predictor*

metrics = evaluate(precision\_predictor,

test\_features.astype('float32'),

test\_labels,

verbose=**True**)

These were the results I got:

Metrics for tuned (precision), LinearLearner.

prediction (col) 0.0 1.0

actual (row)

0.0 85276 26

1.0 31 110

Recall: 0.780

Precision: 0.809

Accuracy: 0.999

As you can see, we still misclassified 26 of the valid results and so I may have to go back and up my aimed-for precision; the recall and accuracy are not too bad, considering the precision tradeoff.

Finally, I made sure to **delete the endpoint** after I was doe with evaluation.

*## IMPORTANT*

*# delete the predictor endpoint*

delete\_endpoint(precision\_predictor)

*Deleted linear-learner-2019-03-11-03-36-56-524*

## Summary and Improvements

<https://www.youtube.com/watch?v=VsjDz3agnhQ&feature=emb_logo>

# Lesson 3: Interview Segment: SageMaker as a Tool and the Future of ML

## Interview Segment: Developing SageMaker

### Expert Interview: AWS SageMaker

Learn about how SageMaker has developed over time, in response to signals from their developer community.

*Please view the segments that seem interesting to you!*

#### How does AWS decide which features to add to their cloud services?

<https://www.youtube.com/watch?v=KYG_LWDhg4I&feature=emb_logo>

#### What is the goal behind the SageMaker [example repository](https://github.com/awslabs/amazon-sagemaker-examples)?

<https://www.youtube.com/watch?time_continue=1&v=9HSJp_i9LFw&feature=emb_logo>

#### What are some use cases that work well (or don't) with SageMaker?

<https://www.youtube.com/watch?time_continue=1&v=9HSJp_i9LFw&feature=emb_logo>

## Interview Segment: New Features

### How do you label data at scale?

<https://www.youtube.com/watch?time_continue=8&v=G_E5N6k2knA&feature=emb_logo>

### What predictions do you have for the future of this technology? What features might be prioritized in the next few years?

<https://www.youtube.com/watch?time_continue=1&v=git73JsQC1Y&feature=emb_logo>

## Interview Segment: Further Learning

### Do you have advice for someone who wants to become an ML engineer?

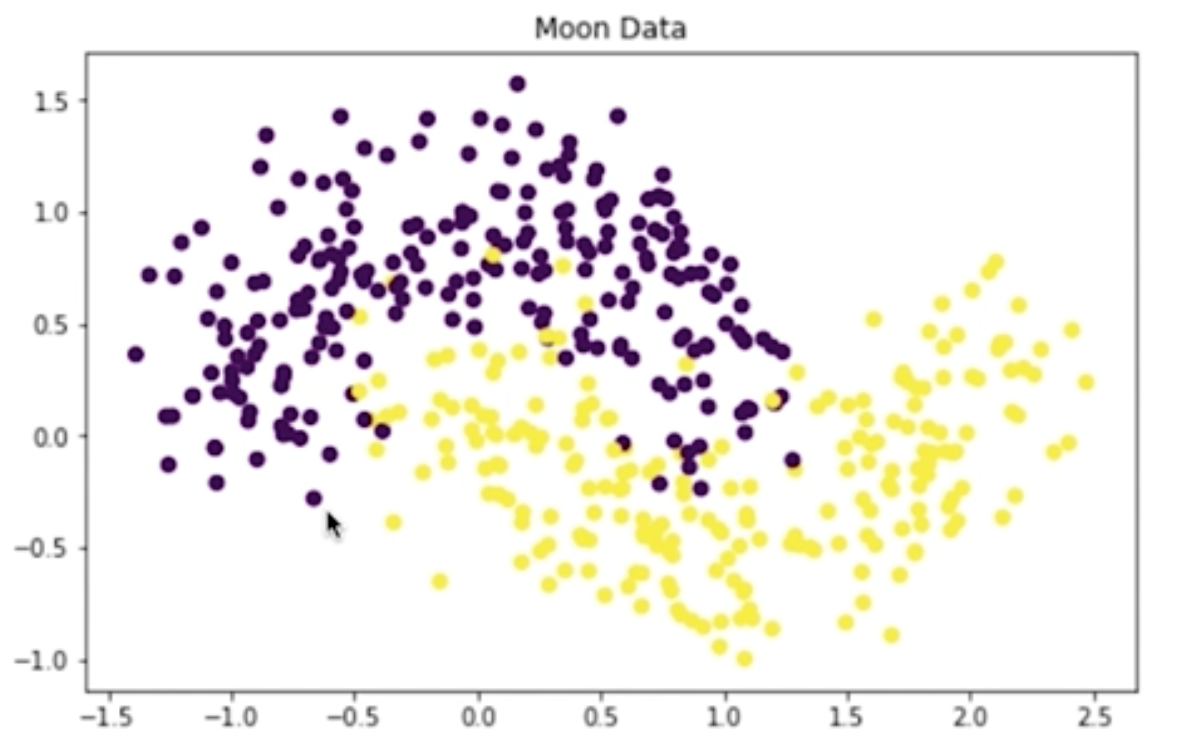
<https://www.youtube.com/watch?v=Wgq4eukacqE&feature=emb_logo>

# Lesson 4: Deploying Custom Models

## Pre-Notebook: Custom Models & Moon Data

### Notebook: Custom Models & Moon Data, Exercise

Next, you'll approach the task of building and training a custom PyTorch classifier to classify data! Specifically, you'll be tasked with classifying "moon data," which is 2-dimensional data whose classes are distributed to look a bit like moons in 2D space.



Slightly noisy, purple (top) and yellow (bottom) "moon" data.

Building and training a custom model is presented as an alternative to something like a LinearLearner, which is great in many cases but may fail for data that is not easily separable. As you follow along with this lesson, you should work in the referenced SageMaker notebooks. We will present a solution to you, but please try to work on a solution of your own, when prompted. Much of the value in this experience will come from experimenting with the code, **in your own way**.

To open this notebook:

* *Navigate to your SageMaker notebook instance, in the [SageMaker console](https://console.aws.amazon.com/sagemaker/" \t "_blank), which has been linked to the main [Github exercise repository](https://github.com/udacity/ML_SageMaker_Studies" \t "_blank)*
* *Activate the notebook instance (if it is in a "Stopped" state), and open it via Jupyter*
* *Click on the exercise notebook in the Moon\_Data directory.*

You may also directly view the exercise and solution notebooks via the repository at the following links:

* [Exercise notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Moon_Data/Moon_Classification_Exercise.ipynb)
* [Solution notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Moon_Data/Moon_Classification_Solution.ipynb)

In this particular case, you will also find a directory source and source\_solution for further reference.

**The solutions are meant to be consulted if you are stuck or want to check your work.**

#### Notebook Outline

We'll go over the following steps to complete the notebook.

* Upload data to S3
* Define a PyTorch neural network for binary classification
* Write a custom **training script**
* Train and evaluate the custom model

#### Later: Delete Resources

At the end of this exercise, and intermittently, you will be reminded to delete your endpoints and resources so that you do not incur any extra processing or storage fees!

## Moon Data & Custom Models

<https://www.youtube.com/watch?time_continue=82&v=vb5ojq8Jw7k&feature=emb_logo>

## Upload Data to S3

<https://www.youtube.com/watch?time_continue=3&v=Mz08Bac6h2Y&feature=emb_logo>

## Exercise: Custom PyTorch Classifier

<https://www.youtube.com/watch?v=kiZ22MJWSFU&feature=emb_logo>

## Solution: Simple Neural Network

<https://www.youtube.com/watch?time_continue=8&v=FINTJpz1Yx0&feature=emb_logo>

## Exercise: Training Script

<https://www.youtube.com/watch?time_continue=1&v=1cbvRmKvQIg&feature=emb_logo>

## Solution: Complete Training Script

<https://www.youtube.com/watch?v=xmrB3sqbeTU&feature=emb_logo>

## Custom SKLearn Model

### Defining a Custom Model

To define a custom model, you need to have the **model** itself and the following two scripts:

* A **training** script that defines how the model will accept input data, and train. This script should also save the trained model parameters.
* A **predict** script that defines how a trained model produces an output and in what format.

#### PyTorch

In PyTorch, you have the option of defining a neural network of your own design. These models do not come with any built-in predict scripts and so you have to write one yourself.

#### SKLearn

The scikit-learn library, on the other hand, has many pre-defined models that come with train and predict functions attached!

You can define custom SKLearn models in a very similar way that you do PyTorch models only you typically **only** have to define the training script. You can use the default predict function.

## Exercise: Create a PyTorchModel & Endpoint

## Create and Deploy a Trained Model

Before you can deploy a custom PyTorch model, you have to take one more step: creating a PyTorchModel. In earlier exercises, you could see that a call to .deploy() created **both** a model and an endpoint, but for PyTorch models, these steps have to be separate. PyTorch models do not automatically come with .predict() functions attached (as many Amazon and Scikit-learn models do, for example) and you may have noticed that you've been given a predict.py file in the source directory. This file is responsible for loading a trained model and applying it to passed in, numpy data.

Now, when we created a PyTorch **estimator**, we specified where the training script, train.py was located. And we'll have to do something very similar here, but for a PyTorch **model** and the predict.py file.

### EXERCISE: Instantiate a PyTorchModel

You can create a PyTorchModel from your trained, estimator attributes. This model is responsible for knowing how to execute a specific predict.py script. And this model is what you'll deploy to create an endpoint.

#### Model Parameters

To instantiate a PyTorchModel, ([documentation, here](https://sagemaker.readthedocs.io/en/stable/sagemaker.pytorch.html#pytorch-model)) you pass in the same arguments as your PyTorch estimator, with a few additions/modifications:

* **model\_data**: The trained model.tar.gz file created by your estimator, which can be accessed as estimator.model\_data.
* **entry\_point**: This time, this is the path to the Python script SageMaker runs for prediction rather than training, predict.py.

In the exercise notebook, you've been given the following code to fill in:

%%time

*# importing PyTorchModel*

**from** sagemaker.pytorch **import** PyTorchModel

*# Create a model from the trained estimator data*

*# And point to the prediction script*

model = **None**

### EXERCISE: Deploy the trained model

Deploy your model to create a predictor. We'll use this to make predictions on our test data and evaluate the model.

%%time

*# deploy and create a predictor*

predictor = **None**

### Evaluate your Model

After deploying your model, you have been given some code to evaluate its performance according to a variety of metrics!

*Remember to****delete your predictor endpoint***after*you've finished evaluating the model.*

Try to complete these steps on your own, and I'll go over one solution, next!

## Solution: PyTorchModel & Evaluation

<https://www.youtube.com/watch?time_continue=2&v=qZTyQqo9FWM&feature=emb_logo>

## Clean up Resources

It is good practice to always clean up and delete any resources that you are no longer using. That is, after you complete an exercise, and you are done with predictions and data analysis, you should get rid of any:

* Data source in S3 that you are no longer using
* Endpoint configuration files that you no longer need
* Endpoints that you will no longer use
* CloudWatch logs that are no longer useful

### Deleting Endpoints

In the notebook, we have usually included code to delete your endpoints after creating some predictions, for example:

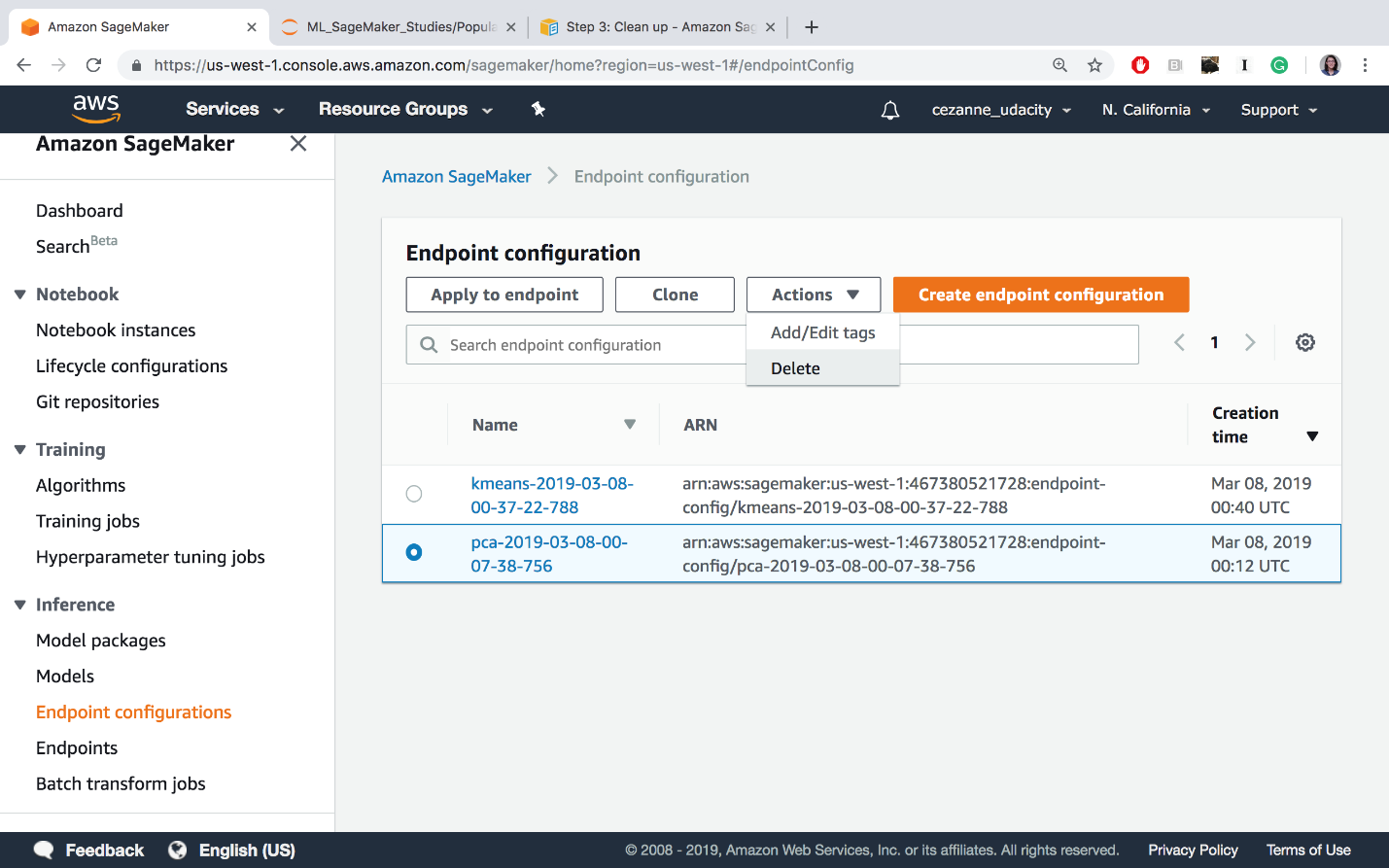
*# delete predictor endpoint*

session.delete\_endpoint(predictor.endpoint)

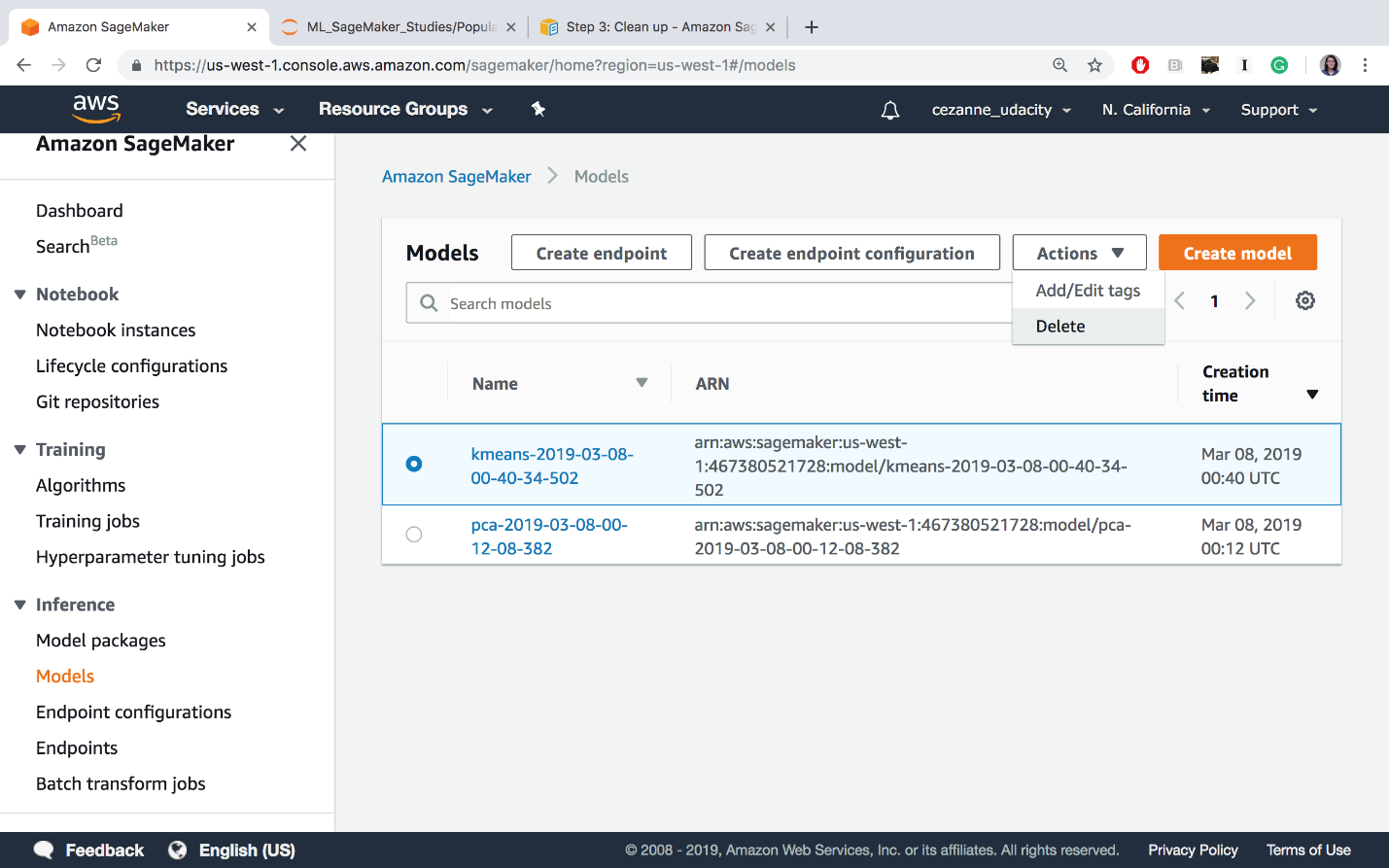
### Thorough Clean up

You can find a link for instructions on cleaning up all your resources, [in this documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html) and I will go over some of these details, next.

* Open the Amazon SageMaker console at <https://console.aws.amazon.com/sagemaker/> and delete the following resources:
* The endpoint configuration.
* The model.

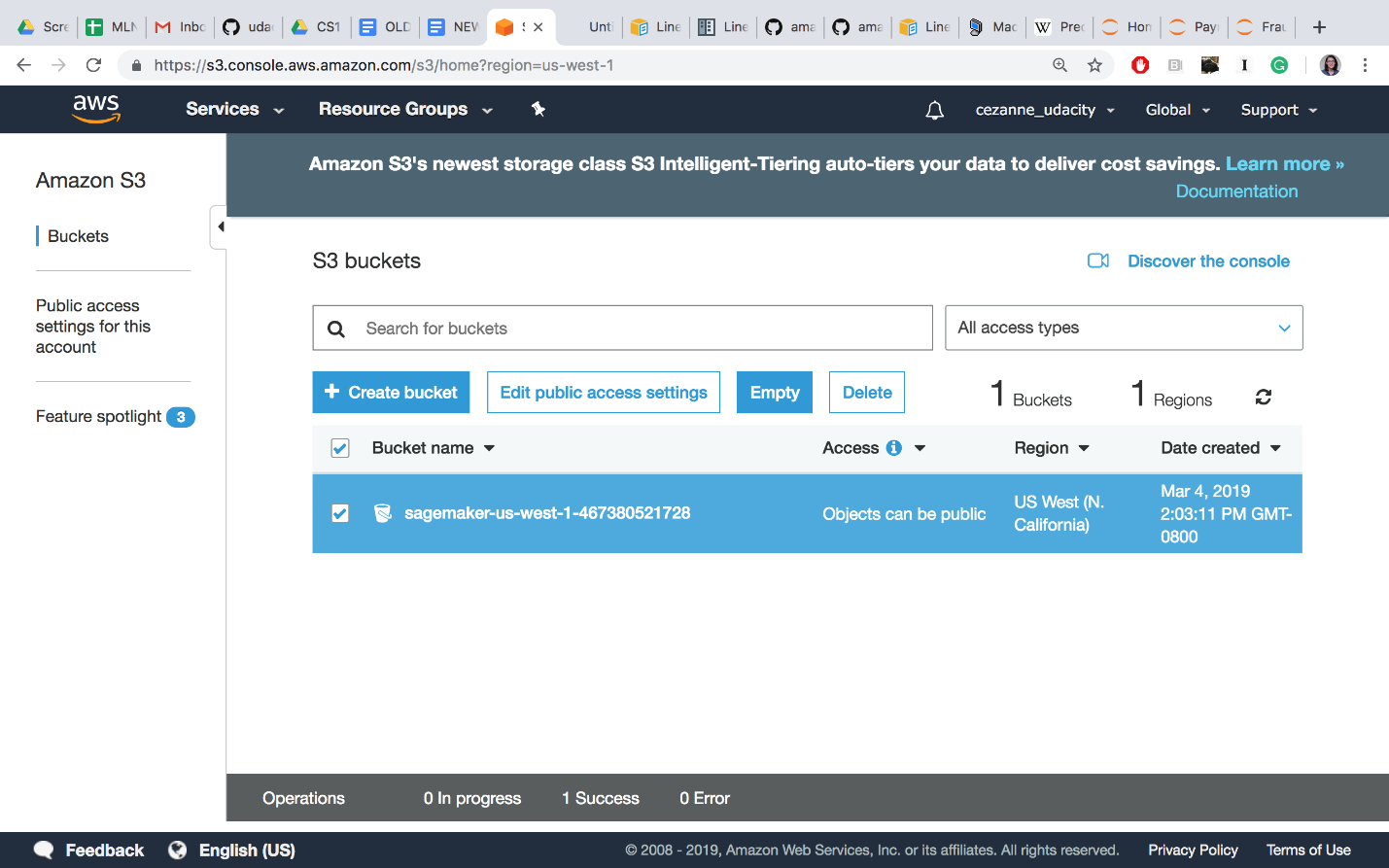


Delete endpoint config files.



Deleting models

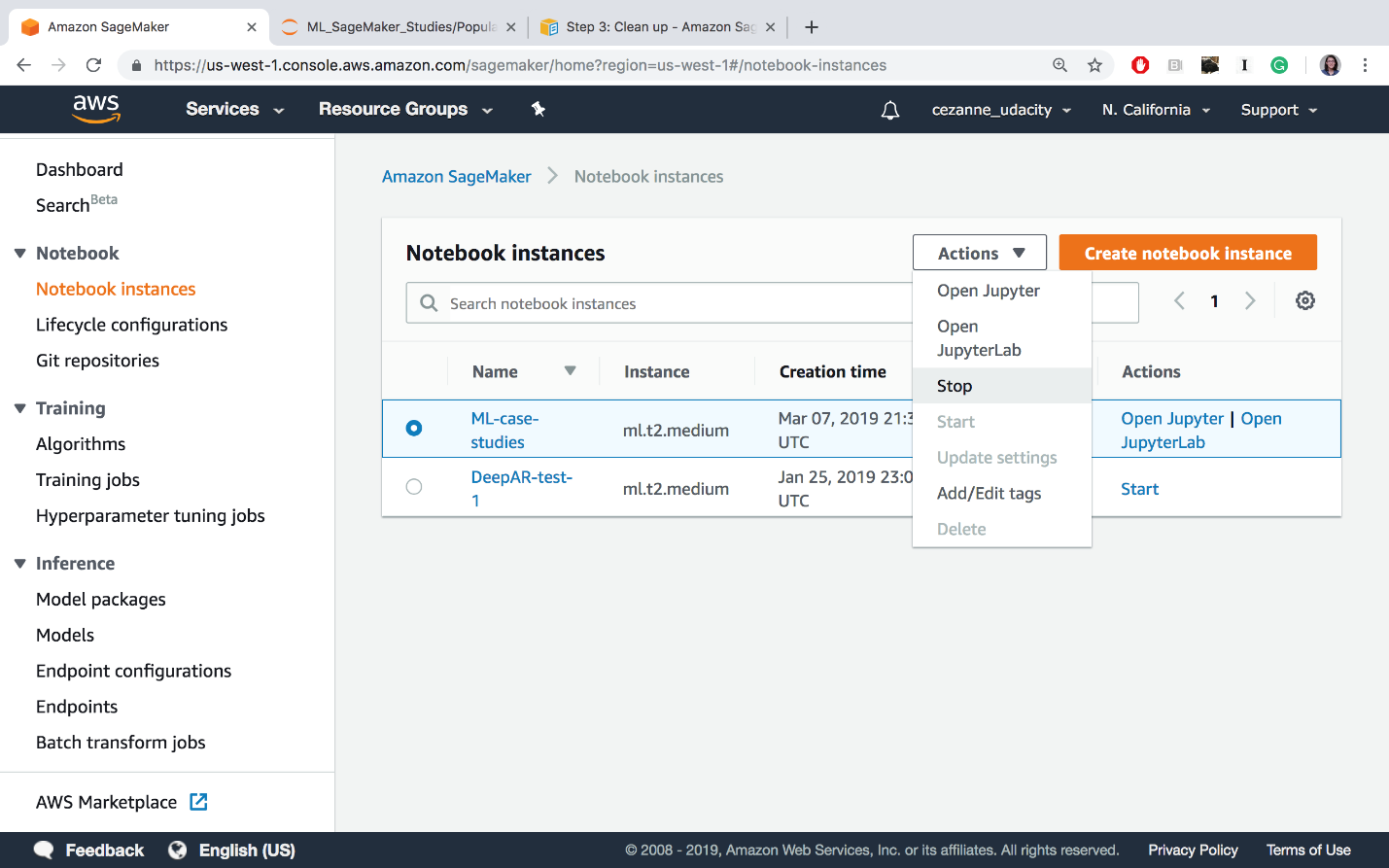
* Open the Amazon S3 console at <https://console.aws.amazon.com/s3/> and delete or empty the bucket that you created for storing model artifacts and the training dataset.



Delete or empty your S3 bucket (empty is recommended until the end of the course, when you should delete this bucket entirely)

* Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/> and delete all of the log groups that have names starting with /aws/sagemaker/.

At the end of this course, you may also choose to delete the entire notebook instance and IAM Role, but you may keep these as is, for now. In between lessons, if you are taking a break, you may want to **Stop** your notebook and pause it from continuously running.



Stopping the ML-case-studies notebook

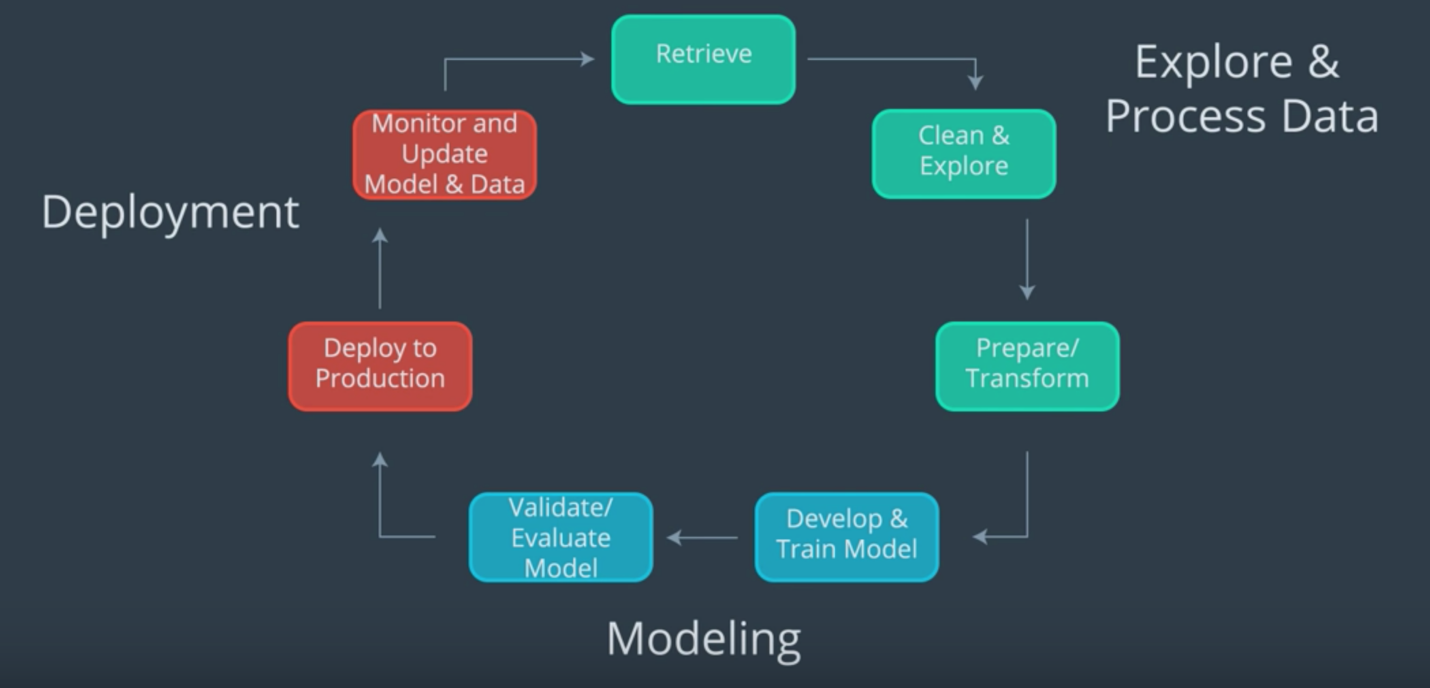
Cleaning up resources at the end of an exercise or lesson is a great practice to get into!

### IMPORTANT

**To avoid incurring additional charges, it is suggested that you DELETE any unused notebooks and data resources on S3 and CloudWatch.**

### Summary of Skills

You've really learned a lot at this point in the course! You should be familiar with each part of the machine learning workflow from data loading and processing to model training and deployment.



Machine learning workflow.

By studying specific case studies, you should have a better idea of what kinds of machine learning models may be applied to different scenarios.

* Essentially, you want to choose an **unsupervised** or **supervised** model based on the given data
* Often you'll want to process and transform your given data to extract the most relevant and comparable **features** for a given task
* Then you'll want to refine your choice of model and tune it based on constraints given by a business problem or the data itself
* You also looked at different ways to visualize what your trained models have learned and different ways to measure success through a variety of **model evaluation metrics**

In this last example, you saw how to define and train a custom model of your own design, by specifying the model architecture and a **training script**. Now, you are ready to move on to the final project!

# Lesson 5: Time-Series Forecasting

## Time-Series Forecasting

<https://www.youtube.com/watch?time_continue=3&v=U8k2Fl2zgJ8&feature=emb_logo>

## Forecasting Energy Consumption, Notebook

<https://www.youtube.com/watch?time_continue=5&v=OZJu6or8Fl0&feature=emb_logo>

## Pre-Notebook: Time-Series Forecasting

Next, you'll approach the task of time-series forecasting. You'll be taking a look at household energy consumption data, originally taken from [Kaggle](https://www.kaggle.com/uciml/electric-power-consumption-data-set). As you follow along with this lesson, you should work in the referenced SageMaker notebooks. We will present a solution to you, but please try to work on a solution of your own, when prompted. Much of the value in this experience will come from experimenting with the code, **in your own way**.

To open this notebook:

* *Navigate to your SageMaker notebook instance, in the [SageMaker console](https://console.aws.amazon.com/sagemaker/" \t "_blank), which has been linked to the main [Github exercise repository](https://github.com/udacity/ML_SageMaker_Studies" \t "_blank)*
* *Activate the notebook instance (if it is in a "Stopped" state), and open it via Jupyter*
* *Click on the exercise notebook in the Time\_Series\_Forecasting directory.*

You may also directly view the exercise and solution notebooks via the repository at the following links:

* [Exercise notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Time_Series_Forecasting/Energy_Consumption_Exercise.ipynb)
* [Solution notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Time_Series_Forecasting/Energy_Consumption_Solution.ipynb)

**The solution notebook is meant to be consulted if you are stuck or want to check your work.**

### Notebook Outline

We'll go over the following steps to complete the notebook.

* Load in and explore household energy consumption data
* Clean the data and transform it to prepare for training a model
* Format the data into JSON Lines
* Train a DeepAR model on defined context and prediction data points
* Evaluate the model by comparing known and predicted consumption values

### Later: Delete Resources

At the end of this exercise, and intermittently, you will be reminded to delete your endpoints and resources so that you do not incur any extra processing or storage fees!

## Processing Energy Data

<https://www.youtube.com/watch?time_continue=3&v=zxnoYK4sYgk&feature=emb_logo>

## Exercise: Creating Time Series

<https://www.youtube.com/watch?time_continue=8&v=KMzVAmoa66k&feature=emb_logo>

## Solution: Split Data

### Splitting in Time

We'll evaluate our model on a test set of data. For machine learning tasks like classification, we typically create train/test data by randomly splitting examples into different sets. For forecasting it's important to do this train/test split in **time** rather than a random split of all data points.

#### Training Time Series

In general, we can create training data by taking each of our complete time series and leaving off the last prediction\_length data points to create corresponding, training time series.

In code this looks like this:

**def** **create\_training\_series**(complete\_time\_series, prediction\_length):

'''Given a complete list of time series data, create training time series.

:param complete\_time\_series: A list of all complete time series.

:param prediction\_length: The number of points we want to predict.

:return: A list of training time series.

'''

*# get training series*

time\_series\_training = []

**for** ts **in** complete\_time\_series:

*# truncate trailing `prediction\_length` pts*

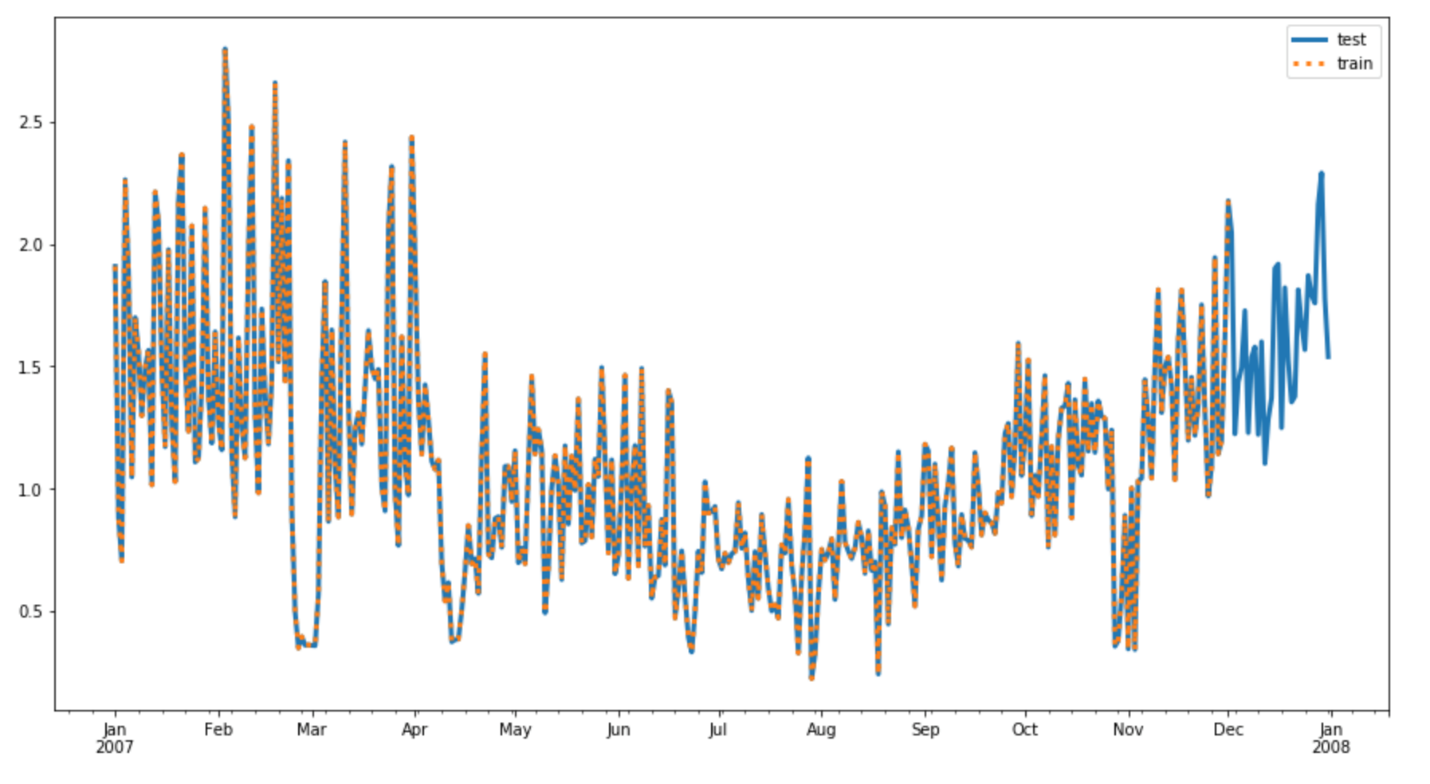
time\_series\_training.append(ts[:-prediction\_length])

**return** time\_series\_training

DeepAR will train on the provided data looking at different intervals that are context\_length number of points as input and the next prediction\_length number of points as output. It selects the context from the given, truncated training data, which is why it is important to leave off the last prediction\_length points.

#### Training and Test Series

We can visualize what these series look like, by plotting the train/test series on the same axis. We should see that the test series contains all of our data in a year, and a training series contains all but the last prediction\_length points. Below are train/test series for 2007.



Test series and train series (truncated, in orange).

## Exercise: Convert to JSON

<https://www.youtube.com/watch?time_continue=1&v=YyxfrVQcM1E&feature=emb_logo>

## Solution: Formatting JSON Lines & DeepAR Estimator

<https://www.youtube.com/watch?time_continue=33&v=1Wx-LK9TVWY&feature=emb_logo>

## Exercise: DeepAR Estimator

**Instantiating a DeepAR Estimator**

Some estimators have specific, SageMaker constructors, but not all. Instead, you can create a base Estimator and pass in the specific **image** (or container) that holds a specific model. The container for the DeepAR model can be gotten as follows:

**from** sagemaker.amazon.amazon\_estimator **import** get\_image\_uri

image\_name = get\_image\_uri(boto3.Session().region\_name, *# get the region*

'forecasting-deepar') *# specify image*

Now that you have the correct image, you can instantiate an Estimator. You're given the following, in-notebook exercise.

### EXERCISE: Instantiate an Estimator

A generic Estimator will be defined by the usual constructor arguments and an image\_name.

You can take a look at the [estimator source code](https://github.com/aws/sagemaker-python-sdk/blob/master/src/sagemaker/estimator.py#L601) to view specifics.

If you complete this task, you can move on to setting DeepAR's model and training hyperparameters and call .fit() to start a training job! You're encouraged to keep going, deploying and evaluating the model on your owm; you are welcome to consult the solution videos to see if your answer matches mine.

## Solution: Complete Estimator & Hyperparameters

<https://www.youtube.com/watch?v=ah7muNBc3dI&feature=emb_logo>

## Making Predictions

<https://www.youtube.com/watch?v=BKOYIfgjsq8&feature=emb_logo>

## Exercise: Predicting the Future

Recall that we did not give our model any data about 2010, but let's see if it can predict the energy consumption given **no target**, only a known start date!

### EXERCISE: Format a request for a "future" prediction

Your task is to create a formatted input to send to the deployed predictor passing in my usual parameters for "configuration". The "instances" will, in this case, just be one instance, defined by the following:

* **start**: The start time will be time stamp that you specify. To predict the first 30 days of 2010, start on Jan. 1st, '2010-01-01'.
* **target**: The target will be an empty list because this year has no, complete associated time series; we specifically withheld that information from our model, for testing purposes. For example:

{"start": start\_time, "target": []} *# empty target*

You'll see the following code to complete in the main exercise notebook. Complete the instances and see if you can generate some future predictions. **Also, remember to delete your model endpoint when you are done making predictions and evaluating your model.**

*# Starting my prediction at the beginning of 2010*

start\_date = '2010-01-01'

timestamp = '00:00:00'

*# formatting start\_date*

start\_time = start\_date +' '+ timestamp

*# formatting request\_data*

*## TODO: fill in instances information*

request\_data = {"instances": [{"start": **None**, "target": **None**}],

"configuration": {"num\_samples": 50,

"output\_types": ["quantiles"],

"quantiles": ['0.1', '0.5', '0.9']}

}

json\_input = json.dumps(request\_data).encode('utf-8')

print('Requesting prediction for '+start\_time)

## Solution: Predicting the Future

<https://www.youtube.com/watch?time_continue=1&v=HT5xKDOgHYw&feature=emb_logo>

# Project: Plagiarism Detector

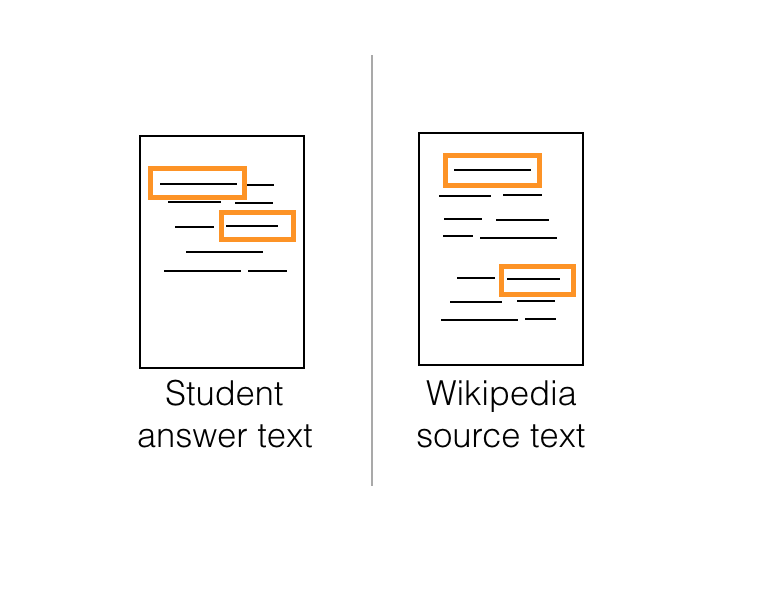
Ref: <https://github.com/tm1611/Machine-Learning-Engineer-Nanodegree/blob/master/03%20-%20ML_SageMaker_Studies/04%20-%20Project_Plagiarism_Detection/02%20-%20Plagiarism_Feature_Engineering.ipynb>

## Project Overview

### Plagiarism Detection Project

In this project, you will be tasked with building a plagiarism detector that examines a text file and performs binary classification; labeling that file as either plagiarized or not, depending on how similar that text file is to a provided source text. Detecting plagiarism is an active area of research; the task is non-trivial and the differences between paraphrased answers and original work are often not so obvious.

Later in this lesson, you'll find a link to all of the relevant project files.



### Defining Features

One of the ways you might go about detecting plagiarism, is by computing **similarity features** that measure how similar a given text file is as compared to an original source text. You can develop as many features as you want and are required to define a couple as outlined in [this paper](https://s3.amazonaws.com/video.udacity-data.com/topher/2019/January/5c412841_developing-a-corpus-of-plagiarised-short-answers/developing-a-corpus-of-plagiarised-short-answers.pdf) (which is also linked in the Lesson Resources tab. In this paper, researchers created features called **containment** and **longest common subsequence**.

In the next few sections, which explain how these features are calculated, I'll refer to a submitted text file (the one we want to label as plagiarized or not) as a **Student Answer Text** and an original, wikipedia source file (that we want to compare that answer to) as the **Wikipedia Source Text**.

You'll be defining a few different similarity features to compare the two texts. Once you've extracted relevant features, it will be up to you to explore different classification models and decide on a model that gives you the best performance on a test dataset.

## Containment

One of your first tasks will be to create **containment** features that first look at a whole body of text (and count up the occurrences of words in several text files) and then compare a submitted and source text, relative to the traits of the whole body of text.

<https://www.youtube.com/watch?time_continue=5&v=FwmT_7fICn0&feature=emb_logo>

### Calculating containment

You can calculate n-gram counts using count vectorization, and then follow the formula for containment:

\frac{{count(\text{n-gram})}\_{A} \cap count (\text{n-gram})\_{S}}{count(\text{n-gram})\_{A}}*count*(n-gram)*A*​*count*(n-gram)*A*​∩*count*(n-gram)*S*​​

If the two texts have no n-grams in common, the containment will be 0, but if all their n-grams intersect then the containment will be 1. Intuitively, you can see how having longer n-gram's in common, might be an indication of cut-and-paste plagiarism.

## Longest Common Subsequence

<https://www.youtube.com/watch?time_continue=4&v=yxXXwBKeYvU&feature=emb_logo>

## Dynamic Programming

<https://www.youtube.com/watch?time_continue=18&v=vAwu-sW9GJE&feature=emb_logo>

## Project Files & Evaluation

**Plagiarism Detection**

### Project Overview

In this project, you will be tasked with building a plagiarism detector that examines a text file and performs binary classification; labeling that file as either plagiarized or not, depending on how similar the text file is to a provided source text.

This project will be broken down into three main notebooks:

**Notebook 1: Data Exploration**

* Load in the corpus of plagiarism text data.
* Explore the existing data features and the data distribution.
* This first notebook is **not** required in your final project submission.

**Notebook 2: Feature Engineering**

* Clean and pre-process the text data.
* Define features for comparing the similarity of an answer text and a source text, and extract similarity features.
* Select "good" features, by analyzing the correlations between different features.
* Create train/test .csv files that hold the relevant features and class labels for train/test data points.

**Notebook 3: Train and Deploy Your Model in SageMaker**

* Upload your train/test feature data to S3.
* Define a binary classification model and a training script.
* Train your model and deploy it using SageMaker.
* Evaluate your deployed classifier.

### Getting the Project Materials

You have been given the starting notebooks in a Github repository, linked below.

Since this project uses SageMaker, it is suggested that you create a new SageMaker notebook instance using your [**AWS console**](https://console.aws.amazon.com/) and link it to the Github repository [**https://github.com/udacity/ML\_SageMaker\_Studies**](https://github.com/udacity/ML_SageMaker_Studies).

**The project files are in the Project\_Plagiarism\_Detection directory.**

You should complete each exercise and question; your project will be evaluated against [**this rubric**](https://review.udacity.com/#!/rubrics/2516/view).

### Project Evaluation

You will be graded on your implementation of a plagiarism detector as well as complete answers to any questions in the project notebook. You'll submit a **zip file** or Github repo that includes complete notebooks, with all cells executed, and you'll be graded according to the project rubric.

### Exploring the Data

Before starting the project, you are given the option to explore the plagiarism data you'll be working with, in the **next workspace**