Introduction to Deployment

What is cloud computing?

You can think of **cloud computing** as transforming an *IT product* into an *IT service*.

Cloud computing can simply be thought of as transforming an *Information Technology (IT) product* into a *service*. With our vacation photos example, we transformed storing photos on an *IT product*, the ***flash drive***; into storing them *using a service*, like ***Google Drive***.

Using a *cloud storage service* provides the ***benefits*** of making it *easier to access* and *share* your vacation photos because you no longer need the flash drive. You'll only need a device with an internet connection to *access* your photos and to grant permission to *others to access* your photos.

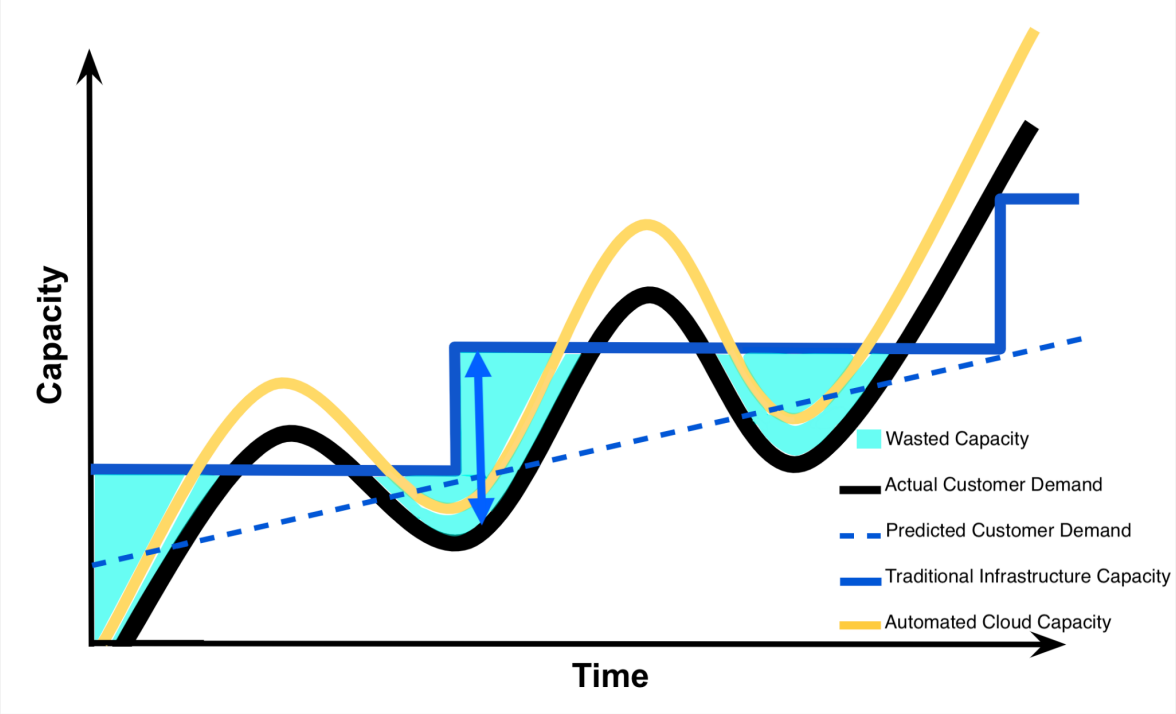
Generally, think of **cloud computing** as using any internet-connected device to log into a *cloud computing service*, like ***Google Drive***, to access an *IT resource*, your vacation ***photos***. These *IT resources*, your vacation ***photos***, are stored in the *cloud provider's* ***data center***. Besides cloud storage, other cloud services include cloud applications, databases, virtual machines, and other services like **SageMaker**.

Benefits

1. Reduced Investments and Proportional Costs (providing cost reduction)
2. Increased Scalability (providing simplified capacity planning)
3. Increased Availability and Reliability (providing organizational agility)

Risks

1. (Potential) Increase in Security Vulnerabilities
2. Reduced Operational Governance Control (over cloud resources)
3. Limited Portability Between Cloud Providers
4. Multi-regional Compliance and Legal Issues



Notes:

- All algorithms used within the machine learning workflow are *similar* for both cloud and on-premise computing. The *only* real difference may be in the user interface and libraries that will be used to execute the machine learning workflow.

Deployment to Production

Deployment to production can simply be thought of as a method that integrates a machine learning model into an existing production environment so that the model can be used to make decisions or predictions based upon data input into the model.

Paths to Deployment

There are *three* primary methods used to transfer a model from the ***modeling*** *component* to the ***deployment*** *component* of the machine learning workflow. We will be discussing them in *order* of ***least*** to ***most*** commonly used. The ***third*** *method* that's *most* similar to what's used for *deployment* within ***Amazon's SageMaker***.

Paths to *Deployment*:

1. Python model is *recoded* into the programming language of the production environment.

    involves recoding the Python model into the language of the production environment, often Java or C++. This method is *rarely used anymore* because it takes time to recode, test, and validate the model that provides the *same* predictions as the *original*.

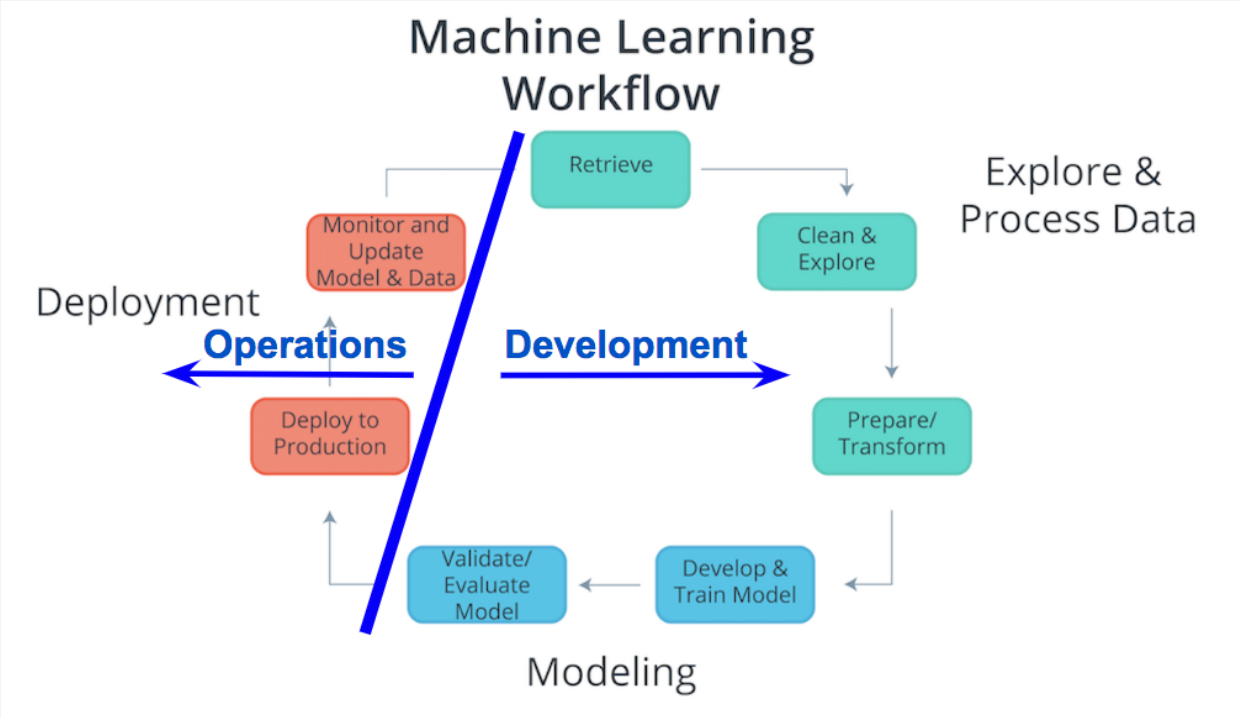
2. Model is *coded* in *Predictive Model Markup Language* (PMML) or *Portable Format Analytics* (PFA).

    codes the model in Predictive Model Markup Language (PMML) or Portable Format for Analytics (PFA), which are two complementary standards that simplify moving predictive models to *deployment* into a *production environment*.

3. Python model is *converted* into a format that can be used in the production environment.

    build a Python model and ***use*** *libraries* and *methods* that *convert* the model into ***code*** that can be used in the *production environment*. Specifically, *most* popular machine learning software frameworks, like PyTorch, TensorFlow, and SciKit-Learn, have methods that will *convert* Python models into an *intermediate standard format*, like ONNX (Open Neural Network Exchange format). This *intermediate standard format* then can be ***converted*** into the software native to the *production environment*.

    - This is the *easiest* and *fastest* way ***to move*** a Python model from *modeling* directly to *deployment*.
    - Moving forward this is *typically* the way *models* are ***moved*** into the *production environment*.
    - Technologies like *containers*, *endpoints*, and *APIs* (Application Programming Interfaces) also help ***ease*** the ***work*** required for *deploying* a model into the *production environment*.



Production Environment and the Endpoint

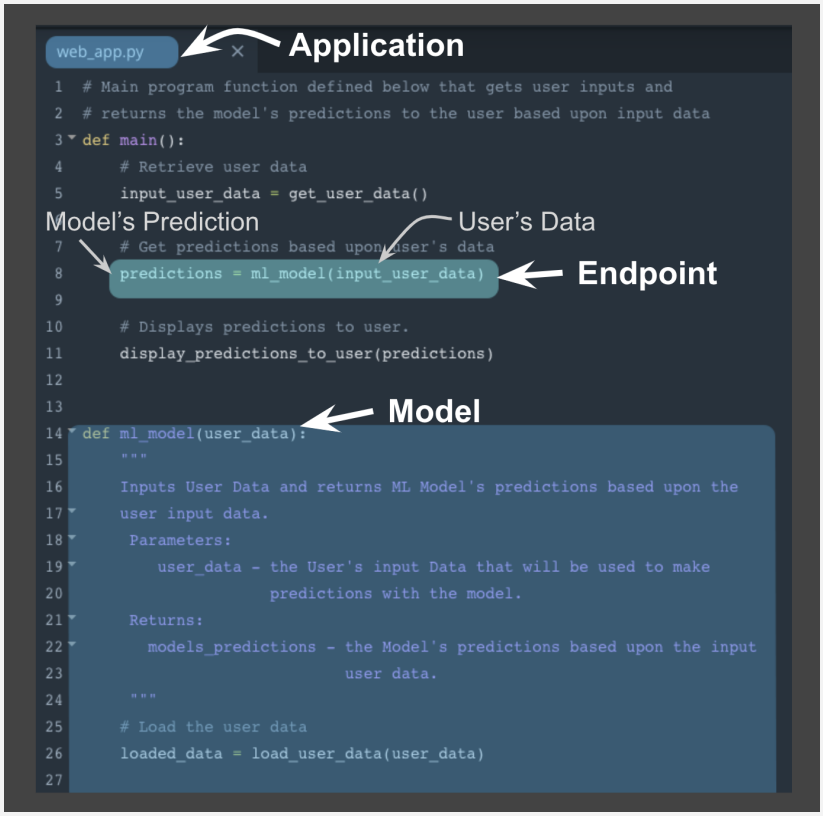
The ***endpoint*** was defined as the **interface** to the model. This **interface** (***endpoint***) facilitates ease of communication between the *model* and the *application*. Specifically, this **interface** (***endpoint***)

An **endpoint**, is a URL that allows an application and a model to speak to one another.

- Allows the *application* to send ***user data*** to the *model* and
- Receives ***predictions*** back from the *model* based upon that ***user data***.

One way to think of the ***endpoint*** that acts as this *interface*, is to think of a *Python program* where:

- the **endpoint** itself is like a ***function call***
- the ***function*** itself would be the **model** and
- the ***Python program*** is the **application**.



Endpoint and REST API

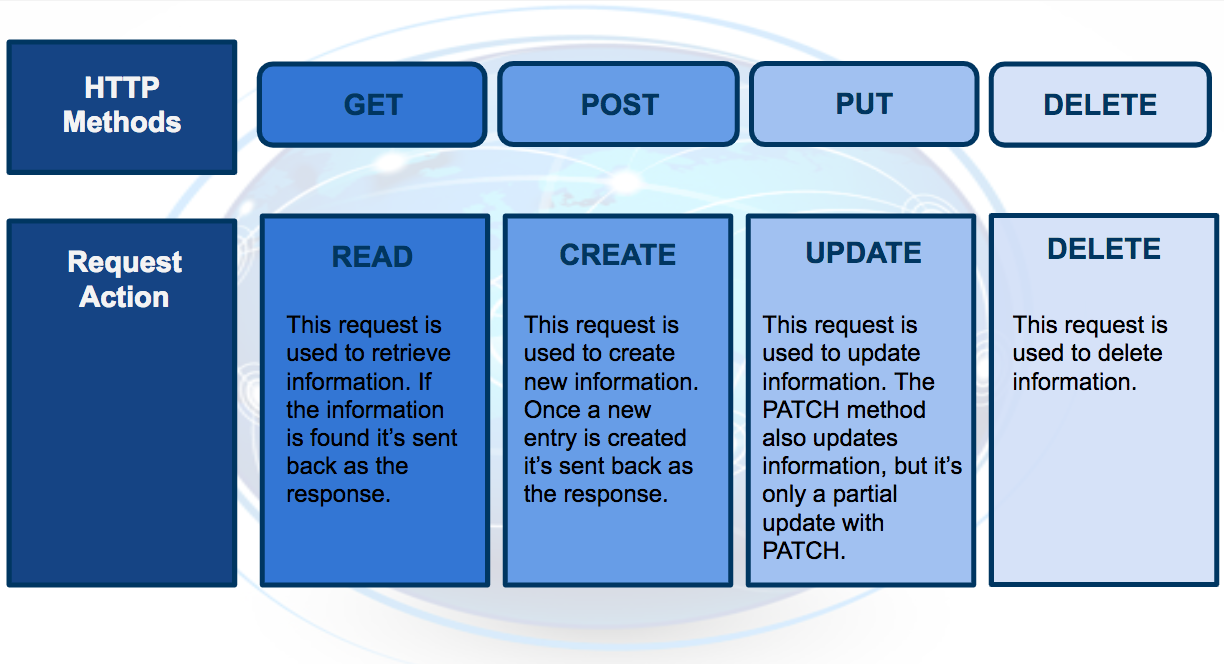
Communication between the **application** and the **model** is done through the **endpoint** (***interface***), where the **endpoint** is an **Application Programming Interface** (**API**).

- An *easy* way to think of an **API**, is as a *set of rules* that *enable* programs, here the **application** and the **model**, to *communicate* with each other.
- In this case, our **API** uses a **RE**presentational **S**tate **T**ransfer, **REST**, architecture that provides a framework for the *set of rules* and *constraints* that must be adhered to for *communication* between programs.
- This **REST API** is one that uses *HTTP requests* and *responses* to enable communication between the **application** and the **model** through the **endpoint** (***interface***).
- Noting that ***both*** the **HTTP request** and **HTTP response** are *communications* sent between the **application** and **model**.

The **HTTP request** that's sent from your **application** to your **model** is composed of *four* parts:

1. **Endpoint:** This **endpoint** will be in the form of a URL, Uniform Resource Locator, which is commonly known as a web address.

2. HTTP Method: Below you will find four of the **HTTP methods**, but for purposes of ***deployment*** our **application** will use the ***POST method*** *only*.
3. HTTP Headers: The **headers** will contain additional information, like the format of the data within the message, that's passed to the *receiving* program.
4. Message (Data or Body): The final part is the **message** (data or body); for ***deployment*** will contain the *user's data* which is input into the **model**.



The **HTTP response** sent from your model to your application is composed of *three* parts:

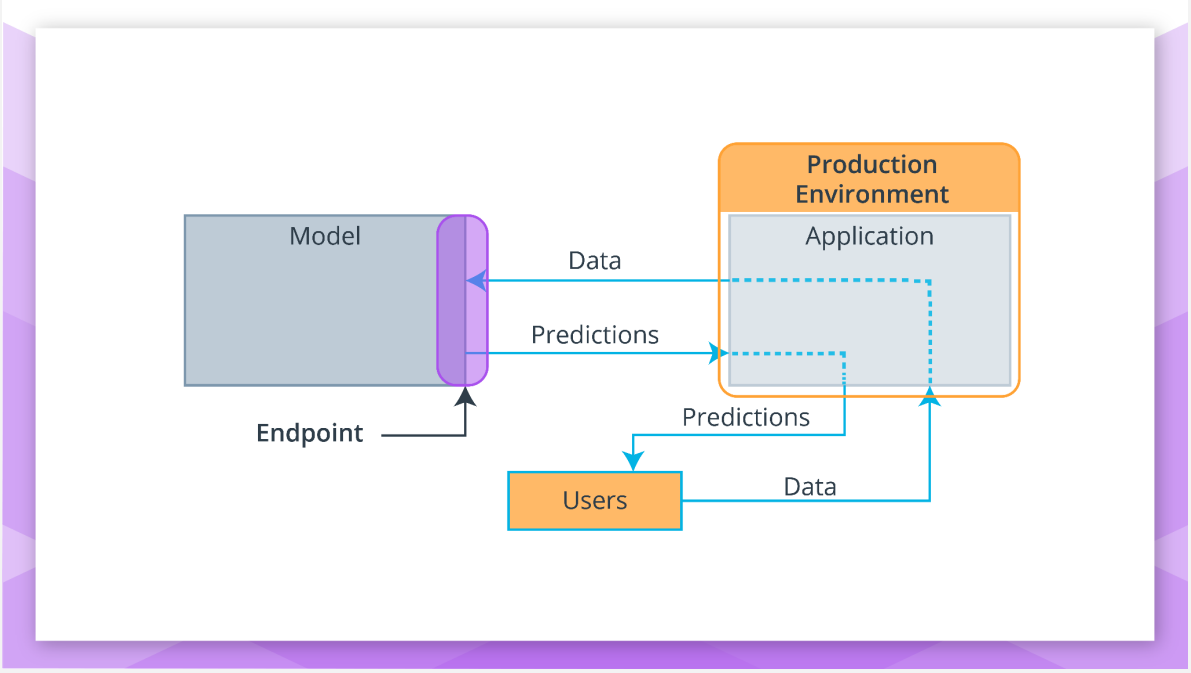
1. HTTP Status Code: If the model successfully received and processed the *user's data* that was sent in the **message**, the status code should start with a ***2***, like *200*.
2. HTTP Headers: The **headers** will contain additional information, like the format of the data within the **message**, that's passed to the receiving program.
3. Message (Data or Body): What's returned as the *data* within the **message** is the *prediction* that's provided by the **model**.

This *prediction* is then presented to the *application user* through the **application**. The **endpoint** is the ***interface*** that *enables communication* between the **application** and the **model** using a **REST API**.

As we learn more about **REST*ful* API**, realize that it's the **application's** responsibility:

- To format the *user's data* in a way that can be easily put into the **HTTP request *message*** and ***used*** by the **model**.
- To translate the *predictions* from the **HTTP response *message*** in a way that's easy for the *application user's* to understand.

Model, Application, and Containers



When we discussed the *production environment*, it was composed of two primary programs, the **model** and the **application**, that *communicate* with each other through the **endpoint** (***interface***).

- The **model** is simply the *Python model* that's created, trained, and evaluated in the ***Modeling*** component of the *machine learning workflow*.
- The **application** is simply a *web* or *software application* that *enables* the application users to use the *model* to retrieve *predictions*.

Both the **model** and the **application** require a *computing environment* so that they can be run and available for use. One way to *create* and *maintain* these *computing environments* is through the use of ***containers***.

- Specifically, the **model** and the **application** can each be run in a ***container*** *computing environment*. The ***containers*** are created using a ***script*** that contains instructions on which software packages, libraries, and other computing attributes are needed in order to run a *software application*, in our case either the **model** or the **application**.

Containers Defined

- A ***container*** can be thought of as a *standardized collection/bundle of software* that is to be *used* for the specific purpose of *running an application*.

As stated **above *container*** *technology* is *used to create* the **model** and **application** *computational environments* associated with ***deployment*** in machine learning. A common **container** software is *Docker*. Due to its popularity sometimes *Docker* is used synonymously with **containers**.

Containers Explained

Often to first explain the concept of ***containers***, people tend to use the analogy of how Docker *containers* are similar to shipping containers.

- Shipping containers can contain a wide variety of products, from food to computers to cars.
- The structure of a shipping container provides the ability for it to hold *different types* of products while making it *easy* to track, load, unload, and transport products worldwide within a shipping container.

Similarly *Docker* **containers**:

- Can *contain **all*** types of *different* software.
- The structure of a *Docker* **container** enables the **container** to be *created*, *saved*, *used* and *deleted* through a set of *common tools*.
- The *common toolset* works with ***any*** **container** regardless of the software the **container** contains.

Container Structure

The image **below** shows the basic structure of a **container**, you have:

- The underlying *computational infrastructure* can be a cloud provider's data center, an on-premise data center, or even someone's local computer.
- Next, you have an *operating system* running on this computational infrastructure, this could be the operating system on your local computer.
- Next, there's the *container engine*, this could be *Docker* software running on your local computer. The *container engine* software enables one to create, save, use, and delete containers; for our example, it could be *Docker* running on a local computer.
- The final two layers make up the composition of the *containers*.
    - The first layer of the container is the *libraries* and *binaries* required to launch, run, and maintain the *next* layer, the *application* layer.
- The image **below** shows *three* containers running *three* different applications.

This *architecture* of **containers** provides the following *advantages*:

1. Isolates the application, which *increases* security.

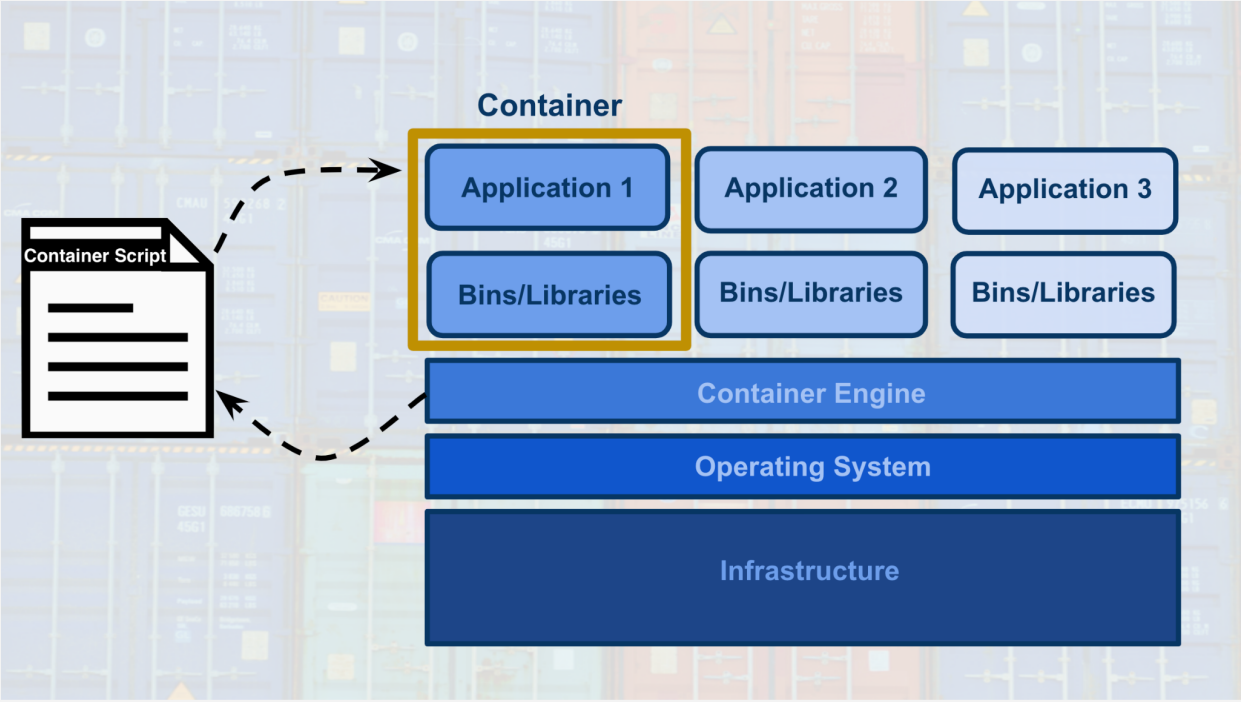
2. Requires *only* software needed to run the application, which uses computational resources *more efficiently* and allows for faster application deployment.

3. Makes application creation, replication, deletion, and maintenance easier and the same across all applications that are deployed using containers.

4. Provides a more simple and secure way to replicate, save, and share containers.

As indicated by the ***fourth*** *advantage* of using ***containers***, a ***container*** *script file* is used to create a ***container***.

- This *text script file* can easily be shared with others and provides a simple method to *replicate* a particular ***container***.
- This ***container*** *script* is simply the *instructions (algorithm)* that are used to create a ***container***; for *Docker,* these ***container*** *scripts* are referred to as *docker files*.



Characteristics of Modeling & Deployment

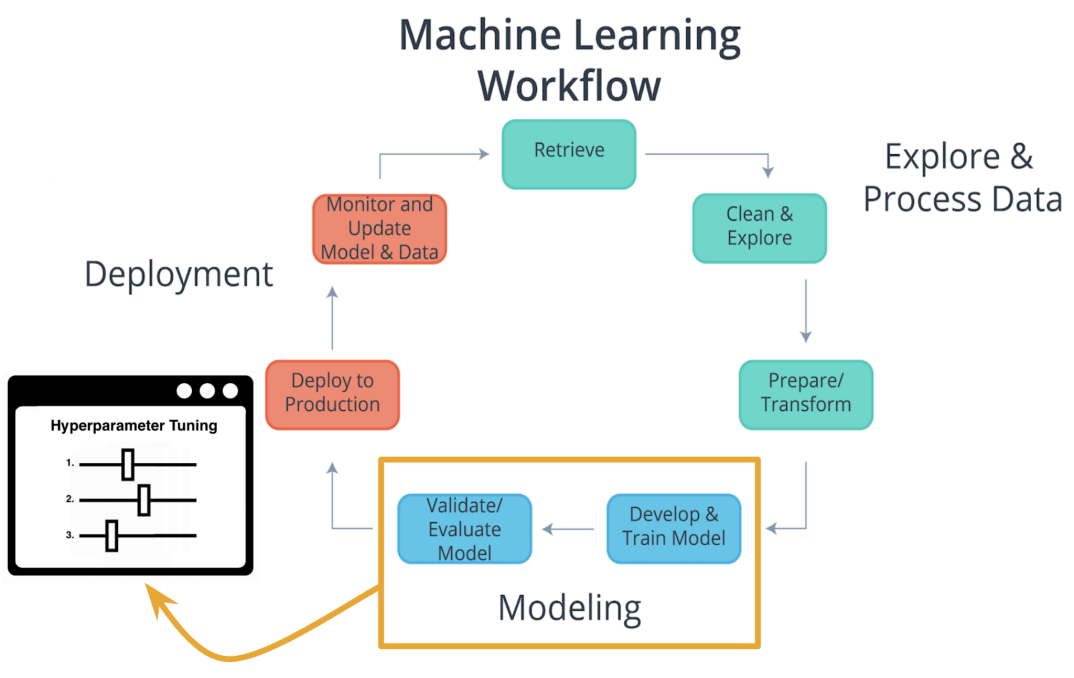
Characteristics of Modeling

Hyperparameters

In machine learning, a **hyperparameter** is a parameter whose value *cannot* be estimated from the data.

- Specifically, a **hyperparameter** is *not directly* learned through the estimators; therefore, its value must be *set* by the model developer.
- This means that **hyperparameter** *tuning* for optimization is an ***important part*** of *model training*.
- Often cloud platform machine learning services provide methods that allow for **automatic hyperparameter** *tuning* for use with model training.

If the machine learning platform fails to offer an *automatic* **hyperparameter** option, one option is to use methods from the scikit-learn Python library for **hyperparameter** *tuning*. Scikit-learn is a free machine learning Python library that includes *methods* that help with **hyperparameter** *tuning*.



Characteristics of Deployment

Model Versioning

One characteristic of deployment is the **version** of the model that is to be deployed.

- Besides saving the **model version** as a part of a *model's metadata* in a database, the *deployment platform* should allow one to indicate a deployed **model's version**.

This will make it easier to maintain, monitor, and update the *deployed model*.

Model Monitoring

Another characteristic of deployment is the ability to easily **monitor** your deployed models.

- Once a model is deployed you will want to make certain it continues to meet its performance metrics; otherwise, the application may need to be updated with a *better-performing* model.

Model Updating and Routing

The ability to easily **update** your deployed model is another characteristic of deployment.

- If a deployed model is *failing* to meet its performance metrics, it's likely you will need to **update** this model.

If there's been a *fundamental change* in the *data* that's being input into the model for predictions; you'll want to **collect** this *input data* to be used to **update** the model.

- The *deployment platform* should support **routing** *differing* proportions of *user requests* to the deployed models; to allow *comparison* of performance between the deployed model *variants*.

Routing in this way allows for a test of a model *performance* as *compared* to other model *variants*.

Model Predictions

Another characteristic of deployment is the *type* of **predictions** provided by your deployed model. There are *two common* types of **predictions**:

- **On-demand predictions**
- **Batch predictions**

On-Demand Predictions

- **On-demand predictions** might also be *called*:
    - online,
    - real-time, or
    - synchronous predictions
- With these types of predictions, one *expects*:
    - *low latency* of response to each prediction request,
    - but allows for the possibility of *high variability* in request volume.
- Predictions are returned in response to the request. Often these requests and responses are done through an API using JSON or XML formatted strings.
- Each prediction request from the user can contain *one* or *many* requests for predictions. Noting that *many* is limited based upon the *size* of the data sent as the request.

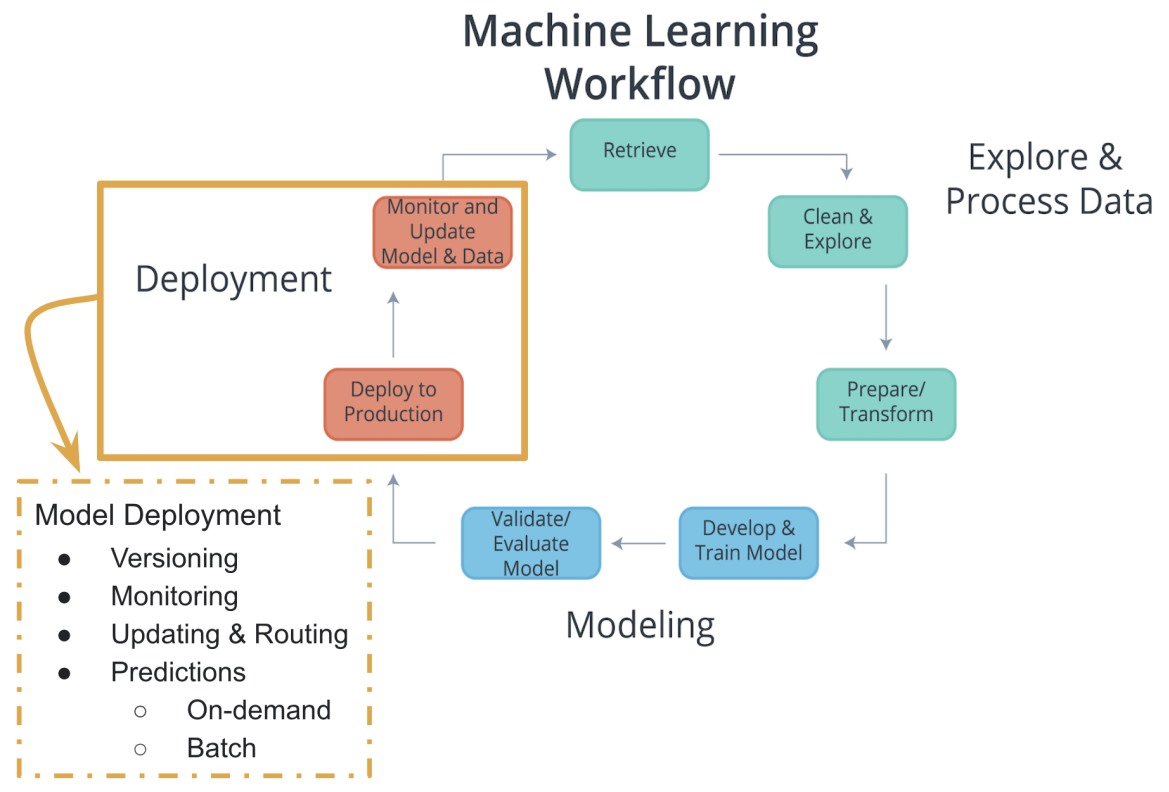
Common cloud platforms' **on-demand prediction** request size limits can range from *1.5(ML Engine)* to *5 Megabytes (SageMaker)*.

On-demand predictions are commonly used to provide customers, users, or employees with real-time, online responses based upon a deployed model. Thinking back on our *magic eight-ball web application* example, users of our web application *would be* making **on-demand prediction** requests.

Batch Predictions

- **Batch predictions** might also be *called*:
    - asynchronous, or
    - batch-based predictions.
- With these types of predictions, one *expects*:
    - *a high volume* of requests with more *periodic submissions*
    - so *latency* won't be an issue.
- Each batch request will point to a specifically *formatted data file* of requests and will return the predictions to a file. Cloud services ***require*** these files will be *stored* in the cloud provider's cloud.
- Cloud services typically have *limits* to how much data they can process with each batch request based upon *limits* they impose on the *size of file* you can store in their cloud storage service. For example, *Amazon's SageMaker* limits batch prediction requests to the size limit they enforce on an object in their S3 storage service.

Batch predictions are *commonly* used to help make *business decisions*. For example, imagine a business using a complex model to predict customer satisfaction across a number of their products and need these *estimates* for a *weekly* report. This would require processing customer data through a **batch prediction** request on a *weekly basis*.



Building a Model using SageMaker

What is AWS Sagemaker?

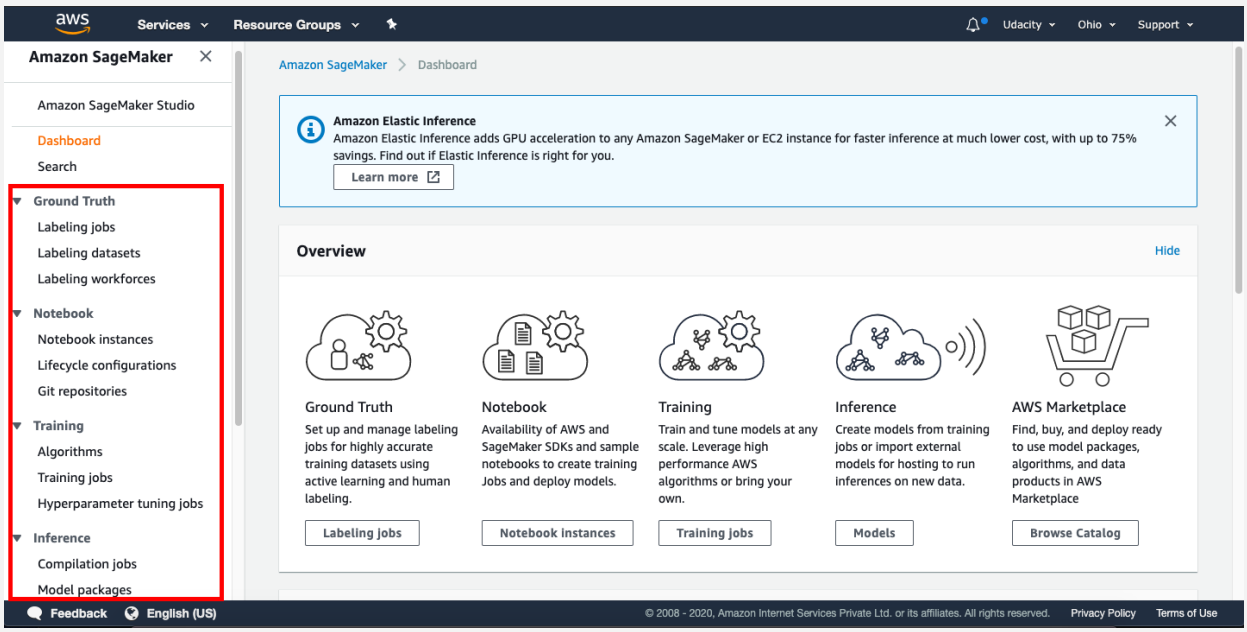
AWS (or Amazon) SageMaker is a *fully managed* service that provides the ability to build, train, tune, deploy, and manage large-scale machine learning (ML) models quickly. Sagemaker provides tools to make each of the following steps simpler:

1. Explore and process data
- Retrieve
- Clean and explore
- Prepare and transform
1. Modeling
- Develop and train the model
- Validate and evaluate the model
1. Deployment
- Deploy to production
- Monitor, and update model & data

The Amazon Sagemaker provides the following tools:

- Ground Truth - To label the jobs, datasets, and workforces
- Notebook - To create Jupyter notebook instances, configure the lifecycle of the notebooks, and attache Git repositories
- Training - To choose an ML algorithm, define the training jobs, and tune the hyperparameter
- Inference - To compile and configure the trained models, and endpoints for deployments

The snapshot of the *Sagemaker Dashboard* below shows the tools mentioned above.



Why is SageMaker a "fully managed" service?

SageMaker helps to reduce the complexity of building, training, and deploying your ML models by offering all these steps on a single platform.

SageMaker supports building the ML models with *modularity*, which means you can reuse a model that you have already built earlier in other projects.

SageMaker Instances

SageMaker instances are the dedicated VMs that are optimized to fit different machine learning (ML) use cases. The supported instance types, names, and pricing in SageMaker are different than that of EC2. Refer the following links to have better insight:

- Amazon SageMaker ML Instance Types - *See that an instance type is characterized by a combination of CPU, memory, GPU, GPU memory, and networking capacity.*
- Amazon EC2 Instance Types - To have you know the difference in naming and combinations of CPU, memory, storage, and networking capacity.

Supported Instance Types and Availability Zones

Amazon SageMaker offers a variety of instance types. Interestingly, *the type of SageMaker instances that are supported varies with AWS Regions and Availability Zones*.

- First, you need to check the List of the AWS Regions that support Amazon SageMaker.
- Next, you can check the various available Amazon SageMaker ML Instance Types, again.

Note

- Sagemaker quotas, also referred to as limits, are very tricky. Every AWS user does not get the default quotas for SageMaker instances, which is why the last column shows a range, e.g., 0 - 20. The **Default Quota** depends on the instance type, the task you want to run (see table above), and also the region in which the Sagemaker service is requested. Refer to this document having a caveat that new accounts may not always get the default limits.
- We recommend you **shut down every resource** (e.g., SageMaker instances, or any other hosted service) on the AWS cloud immediately after the usage; otherwise, you will be billed even if the resources are not in actual use.
- Even if you are in the middle of the project and need to step away, **PLEASE SHUT DOWN YOUR SAGEMAKER INSTANCE**. You can re-instantiate later.
- 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.
- Recently, SageMaker has added a line in the setup code to link directly to a Github repository and it's recommended that you use that setup!
-

AWS Service Utilization Quota (Limits)

You need to understand the way AWS imposes ***utilization quotas*** (limits) on almost all of its services. *Quotas*, also referred to as *limits*, are the maximum number of resources of a particular service that you can create in your AWS account.

- AWS provides default quotas, **for each AWS service.**
- Importantly, **each quota is region-specific.**
- There are three ways to **view your quotas**, as mentioned here:

    1. Service Endpoints and Quotas,
    2. Service Quotas console, and
    3. AWS CLI commands - list-service-quotas and list-aws-default-service-quotas
- In general, there are three ways to **increase the quotas**:
    1. Using *Amazon Service Quotas* service - This service consolidates your account-specific values for quotas across all AWS services for improved manageability. Service Quotas is available at no additional charge. You can directly try logging into the Service Quotas console here.
    2. Using *AWS Support Center* - You can create a case for support from AWS.
    3. AWS CLI commands - request-service-quota-increas

SageMaker Sessions & Execution Roles

SageMaker has some unique objects and terminology that will become more familiar over time. There are a few objects that you'll see come up, over and over again:

- **Session** - A session is a special *object* that allows you to do things like manage data in S3 and create and train any machine learning models; you can read more about the functions that can be called on a session, in this documentation. The upload_data function should be close to the top of the list! You'll also see functions like train, tune, and create_model all of which we'll go over in more detail, later.
- **Role** - Sometimes called the *execution role*, this is the IAM role that you created when you created your notebook instance. The role basically defines how data that your notebook uses/creates will be stored. You can even try printing out the role with print(role) to see the details of this creation.

Training Jobs

A training job is used to train a specific estimator.

When you request a training job to be executed you need to provide a few items:

1. A location on S3 where your training (and possibly validation) data is stored,
2. A location on S3 where the resulting model will be stored (this data is called the model artifacts),
3. A location of a docker container (certainly this is the case if using a built-in algorithm) to be used for training
4. A description of the compute instance that should be used.

Once you provide all of this information, SageMaker will execute the necessary instance (CPU or GPU), load up the necessary docker container and execute it, passing in the location of the training data. Then when the container has finished training the model, the *model artifacts* are packaged up and stored on S3.

You can see a high-level (which we've just walked through) example of training a KMeans estimator, in this documentation. This high-level example defines a KMeans estimator, and uses .fit() to train that model. Later, we'll show you a low-level model, in which you have to specify many more details about the training job.

Estimators

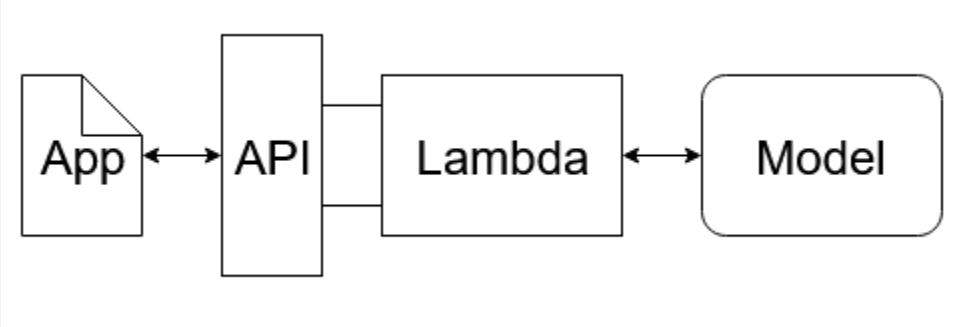
You can read the documentation on estimators for more information about this object. Essentially, the Estimator is an object that specifies some details about how a model will be trained. It gives you the ability to create and deploy a model.

Transformer

You can read more about the transform and wait for functions, in the transformer documentation. In this case, the transformer is used to create a transform job and **evaluate a trained model**. The transform function takes in the location of some test data, and some information about how that test data is formatted.

Deploying and Using a Model

How to use a Deployed Model



What this means is that when someone uses our web app, the following will occur.

1. To begin with, a user will type out a review and enter it into our web app.
2. Then, our web app will send that review to an endpoint that we created using API Gateway. This endpoint will be constructed so that anyone (including our web app) can use it.
3. API Gateway will forward the data on to the Lambda function
4. Once the Lambda function receives the user's review, it will process that review by tokenizing it and then creating a bag of words encoding of the result. After that, it will send the processed review off to our deployed model.

5. Once the deployed model performs inference on the processed review, the resulting sentiment will be returned back to the Lambda function.
6. Our Lambda function will then return the sentiment result back to our web app using the endpoint that was constructed using API Gateway.

Lambda Function

In general, a Lambda function is an example of a 'Function as a Service'. It lets you perform actions in response to certain events, called triggers. Essentially, you get to describe some events that you care about, and when those events occur, your code is executed.

For example, you could set up a trigger so that whenever data is uploaded to a particular S3 bucket, a Lambda function is executed to process that data and insert it into a database somewhere.

One of the big advantages to Lambda functions is that since the amount of code that can be contained in a Lambda function is relatively small, you are only charged for the *number* of executions.

In our case, the Lambda function we are creating is meant to process user input and interact with our deployed model. Also, the trigger that we will be using is the endpoint that we will create using API Gateway.

Create a Lambda Function

The steps to create a lambda function are outlined in the notebook and here, for convenience.

Setting up a Lambda function

The first thing we are going to do is set up a Lambda function. This Lambda function will be executed whenever our public API has data sent to it. When it is executed it will receive the data, perform any sort of processing that is required, send the data (the review) to the SageMaker endpoint we've created, and then return the result.

Part A: Create an IAM Role for the Lambda function

Since we want the Lambda function to call a SageMaker endpoint, we need to make sure that it has permission to do so. To do this, we will construct a role that we can later give the Lambda function.

Part B: Create a Lambda function

Now it is time to actually create the Lambda function. Remember from earlier that in order to process the user-provided input and send it to our endpoint we need to gather two pieces of information:

1. The name of the endpoint, and
2. the vocabulary object.

We will copy these pieces of information to our Lambda function, after we create it.

Updating a Model

SageMaker Retrospective

In this module, we looked at various features offered by Amazon's SageMaker service. These features include the following.

- **Notebook Instances** provide a convenient place to process and explore data in addition to making it very easy to interact with the rest of SageMaker's features.
- **Training Jobs** allow us to create *model artifacts* by fitting various machine learning models to data.
- **Hyperparameter Tuning** allows us to create multiple training jobs each with different hyperparameters in order to find the hyperparameters that work best for a given problem.
- **Models** are essentially a combination of *model artifacts* formed during a training job and an associated docker container (code) that is used to perform inference.
- **Endpoint Configurations** act as blueprints for endpoints. They describe what sort of resources should be used when an endpoint is constructed along with which models should be used and, if multiple models are to be used, how the incoming data should be split up among the various models.
- **Endpoints** are the actual HTTP URLs that are created by SageMaker and which have properties specified by their associated endpoint configurations. **Have you shut down your endpoints?**
- **Batch Transform** is the method by which you can perform inference on a whole bunch of data at once. In contrast, setting up an endpoint allows you to perform inference on small amounts of data by sending it to the endpoint bit by bit.

In addition to the features provided by SageMaker we used three other Amazon services.

In particular, we used **S3** as a central repository in which to store our data. This included test/training/validation data as well as model artifacts that we created during training.

SageMaker Documentation

- **Developer Documentation** can be found here: https://docs.aws.amazon.com/sagemaker/latest/dg/

- **Python SDK Documentation** (also known as the high-level approach) can be found here: https://sagemaker.readthedocs.io/en/latest/
- **Python SDK Code** can be found on GitHub here: https://github.com/aws/sagemaker-python-sdk

---

Population Segmentation

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 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!

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.

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 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.

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 from 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.

Payment Fraud Detection

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) overall positives, and will be 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.

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.