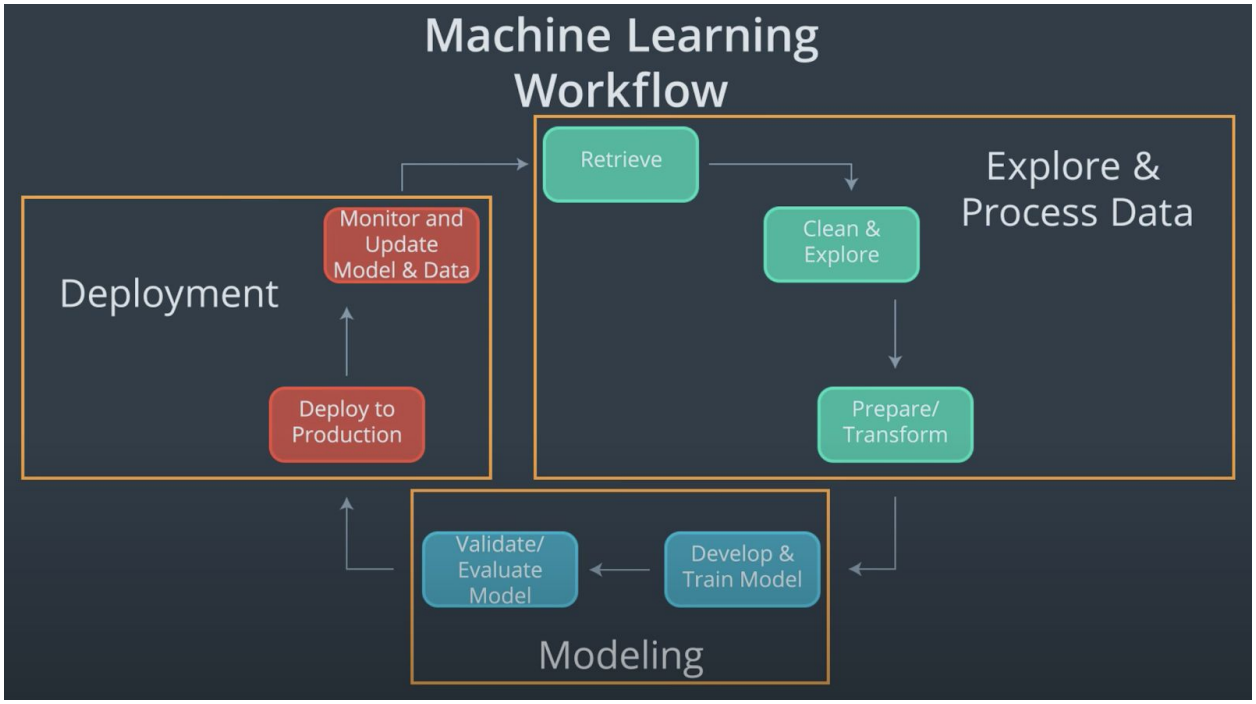
Machine Learning in Production

Machine Learning Workflow



What is Cloud Computing & Why Would We Use It?

Cloud Computing

What is cloud computing?

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

Consider the following example:

> Have you ever had to backup and store important files on your computer? Maybe these files are family photos from your last vacation. You might store these photos on a *flash drive*. These days you have an *alternative option* to store these photos in the ***cloud*** using a *cloud storage provider*, like: Google Drive, Apple's iCloud, or Microsoft's OneDrive.

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.

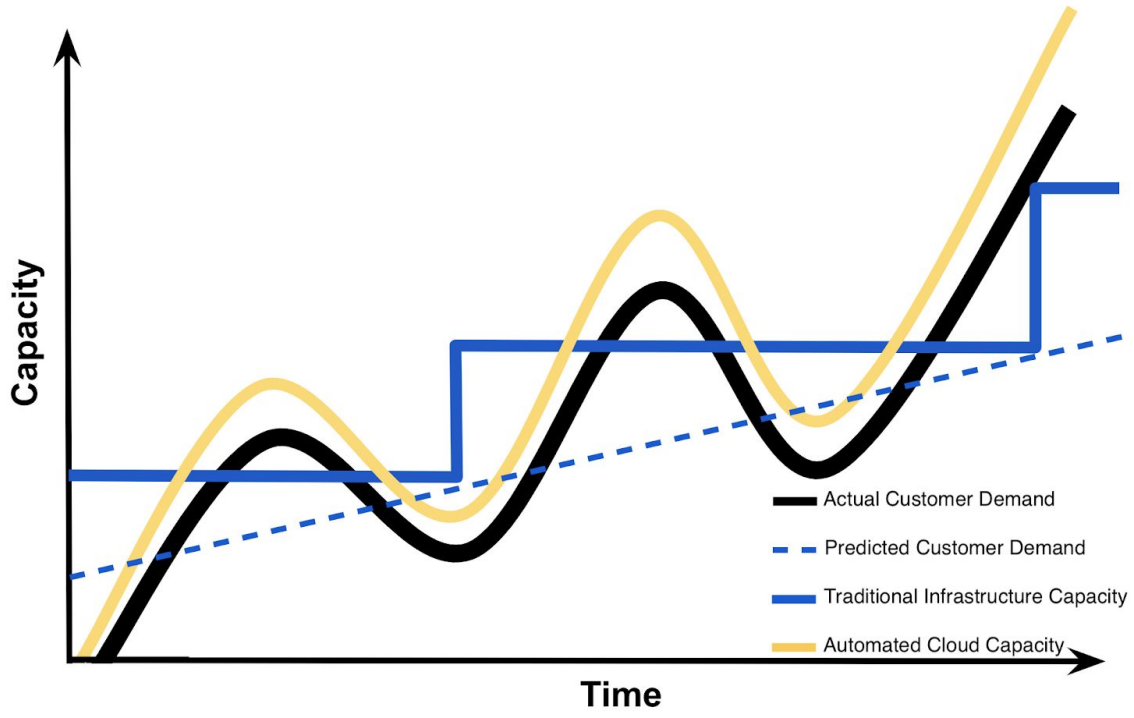
Generally, think of **cloud computing** as using an internet connected device to log into a *cloud computing service*, like ***Google Drive***, to access an *IT resource*, your vacation ***photos***. These *IT resources*, your vacation ***photos***, are stored in the *cloud provider's* ***data center***.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

Why would a business decide to use cloud computing?

Most of the factors related to choosing *cloud computing services*, instead of *developing on-premise IT resources* are related to ***time*** and ***cost***. The ***capacity utilization*** graph *below* shows how *cloud computing* compares to *traditional infrastructure* (on-premise IT resources) in meeting *customer demand*.

Capacity in the graph *below* can be thought of as the *IT resources* like: compute capacity, storage, and networking, that's needed to meet customer demand for a business' products and the ***costs*** associated with those *IT resources*. In our vacation photos example, *customer demand* is for storing and sharing customer photos. The *IT resources* are the required software and hardware that enables photo storage and sharing in the *cloud* or *on-premise* (traditional infrastructure).

Looking at the graph, notice that *traditional infrastructure* doesn't scale when there are spikes in demand, and also leaves excess when preparing for *future* demand. This ability to easily meet unstable, fluctuating *customer demand* illustrates many of the benefits of *cloud computing*.



Summary of Benefits of Risks Associated with Cloud Computing

The ***capacity utilization*** graph above was initially used by cloud providers like Amazon to illustrate the ***benefits*** of cloud computing. Summarized below are the ***benefits*** of cloud computing that are often what *drives* businesses to include cloud services in their IT infrastructure [**1**]. These same ***benefits*** are echoed in those provided by cloud providers Amazon (benefits), Google (benefits), and Microsoft (benefits).

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)

Below we have also summarized the risks associated with cloud computing [**1**]. Cloud providers don't typically highlight the *risks* assumed when using their cloud services like they do with the *benefits*, but cloud providers like: Amazon (security), Google (security), and Microsoft (security) often provide details on security of their cloud services. It's up to the *cloud user* to understand the compliance and legal issues associated with housing data within a *cloud provider's* data center instead of on-premise. The service level agreements (SLA) provided for a cloud service often highlight security responsibilities of the cloud provider and those *assumed* by the cloud user.

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

Paths to Deployment

Deployment to Production

Recall that:

> ***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.

This means that moving from *modeling* to *deployment*, a *model* needs to be provided to those responsible for *deployment*.

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. The Python model is *recorded* into the programming language of the production environment.
2. Model is *coded* in *Predictive Model Markup Language* (PMML) or *Portable Format Analytics* (PFA).

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

Recoding Model into Programming Language of Production Environment

The *first* method which involves recording 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*.

Model is Coded in PMML or PFA

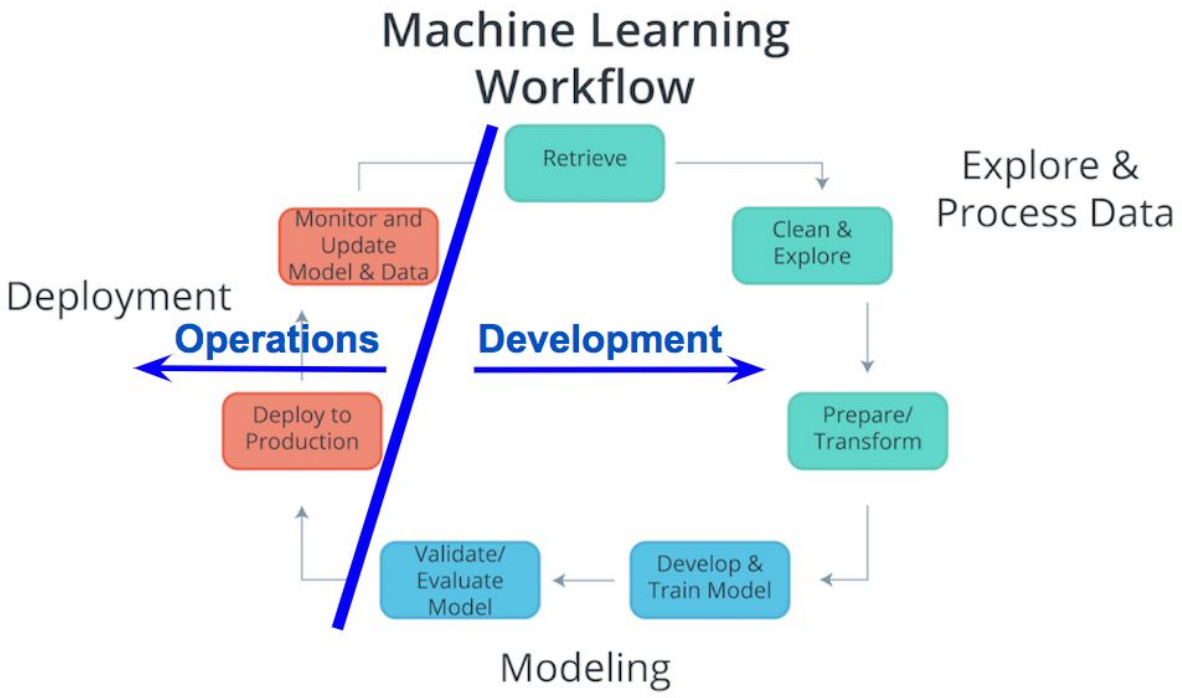
The *second* method is to code the model in Predictive Model Markup Language (PMML) or Portable Format for Analytics (PFA), which are two complementary standards that *simplify* moving predictive models to *deployment* into a *production environment*. The Data Mining Group developed both PMML and PFA to provide vendor-neutral executable model specifications for certain predictive models used by data mining and machine learning. Certain analytic software allows for the direct import of PMML including but *not limited* to IBM SPSS, R, SAS Base & Enterprise Miner, Apache Spark, Teradata Warehouse Miner, and TIBCO Spotfire.

Model is Converted into Format that's used in the Production Environment

The *third* method is to 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, SciKit-Learn, have methods that will *convert* Python models into *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*.

Machine Learning Workflow and DevOps



Machine Learning Workflow and Deployment

Considering the components of the *Machine Learning Workflow*, one can see how *Exploring and Processing Data* is tightly coupled with *Modeling*. The *modeling* **can't** occur without *first* having the *data* the model will be based upon *prepared* for the modeling process.

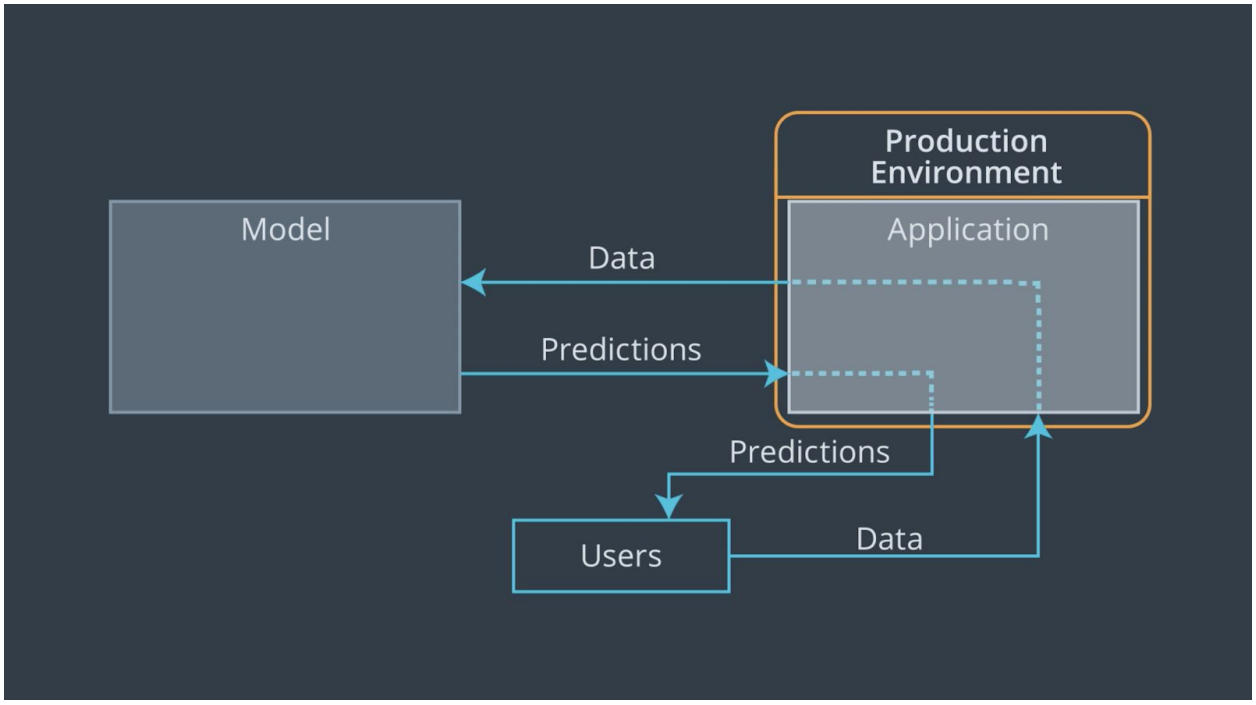
Comparatively *deployment* is **more** tightly coupled with the *production environment* than with *modeling* or *exploring and processing the data*. Therefore, *traditionally* there is a separation between *Deployment* and the *other components* of the machine learning workflow. Specifically looking at the diagram ***above***, the **Process Data and Modeling** are considered ***Development***; whereas, **Deployment** is *typically* considered ***Operations***.

In the past typically, ***development*** was handled by **analysts**; whereas, ***operations*** were handled by **software developers** responsible for the *production environment*. With *recent* developments in *technology* (containers, endpoints, APIs) and the *most* common path of deployment; *this division* between ***development*** and ***operations*** *softens*. The softening of this division enables **analysts** to handle certain aspects of ***deployment*** and enables faster updates to faltering models.

Deployment within Machine Learning Curriculum

Deployment is **not** commonly included in *machine learning curriculum*. This likely is associated with the *analyst's* typical focus on ***Exploring and Processing Data*** and ***Modeling***, and the *software developer's* focusing more on ***Deployment*** and the *production environment*. Advances in cloud services, like SageMaker and ML Engine, and deployment technologies, like Containers and REST APIs, allow for *analysts* to easily take on the responsibilities of ***deployment***.

Production Environments



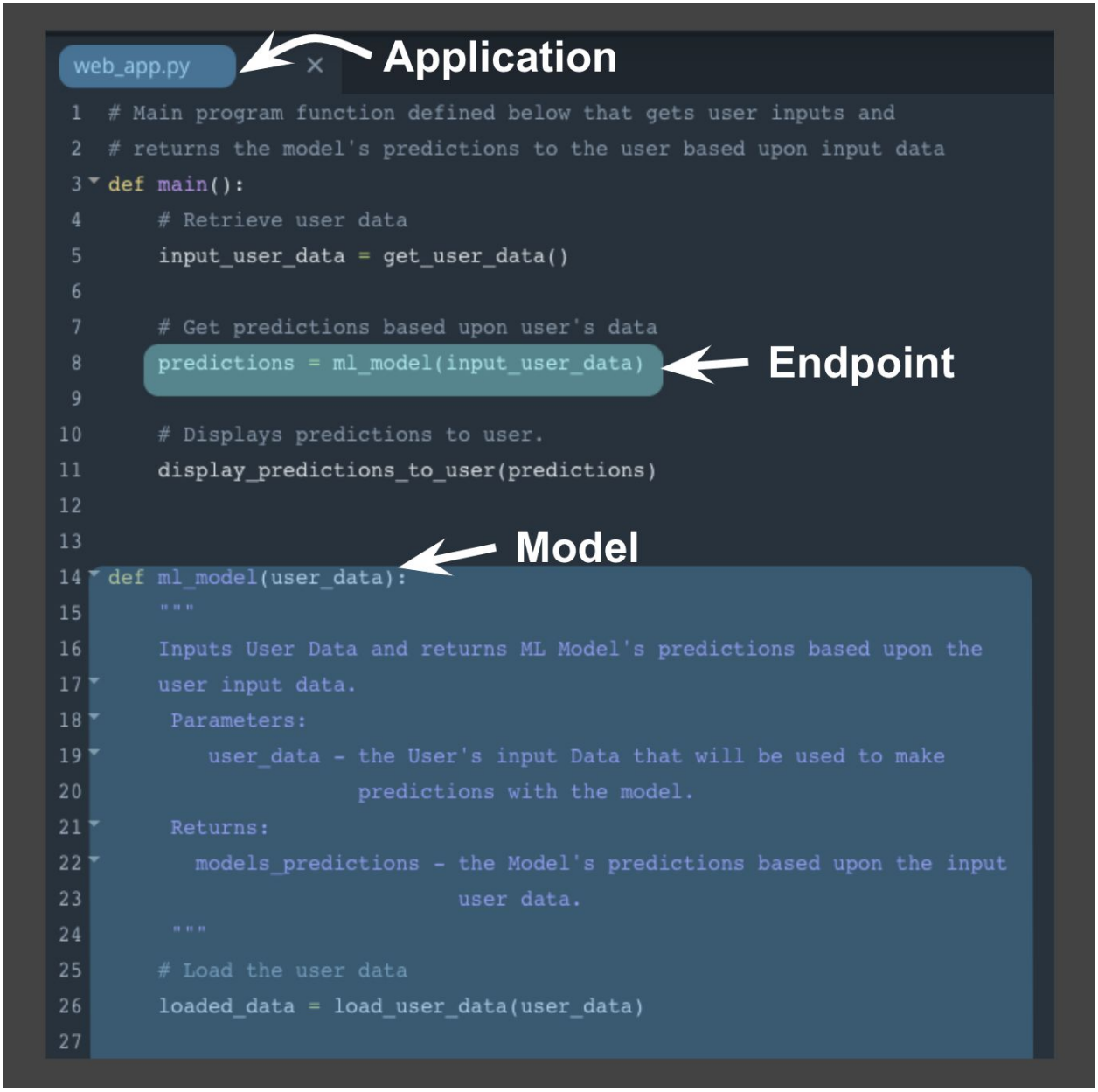
Endpoints & Rest APIs

Production Environment and the Endpoint

When we discussed the *production environment*, the ***endpoint*** was defined as the **interface** to the model. This **interface** (***endpoint***) facilitates an ease of communication between the *model* and the *application*. Specifically, this **interface** (***endpoint***);

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

Model, Application, and Endpoint

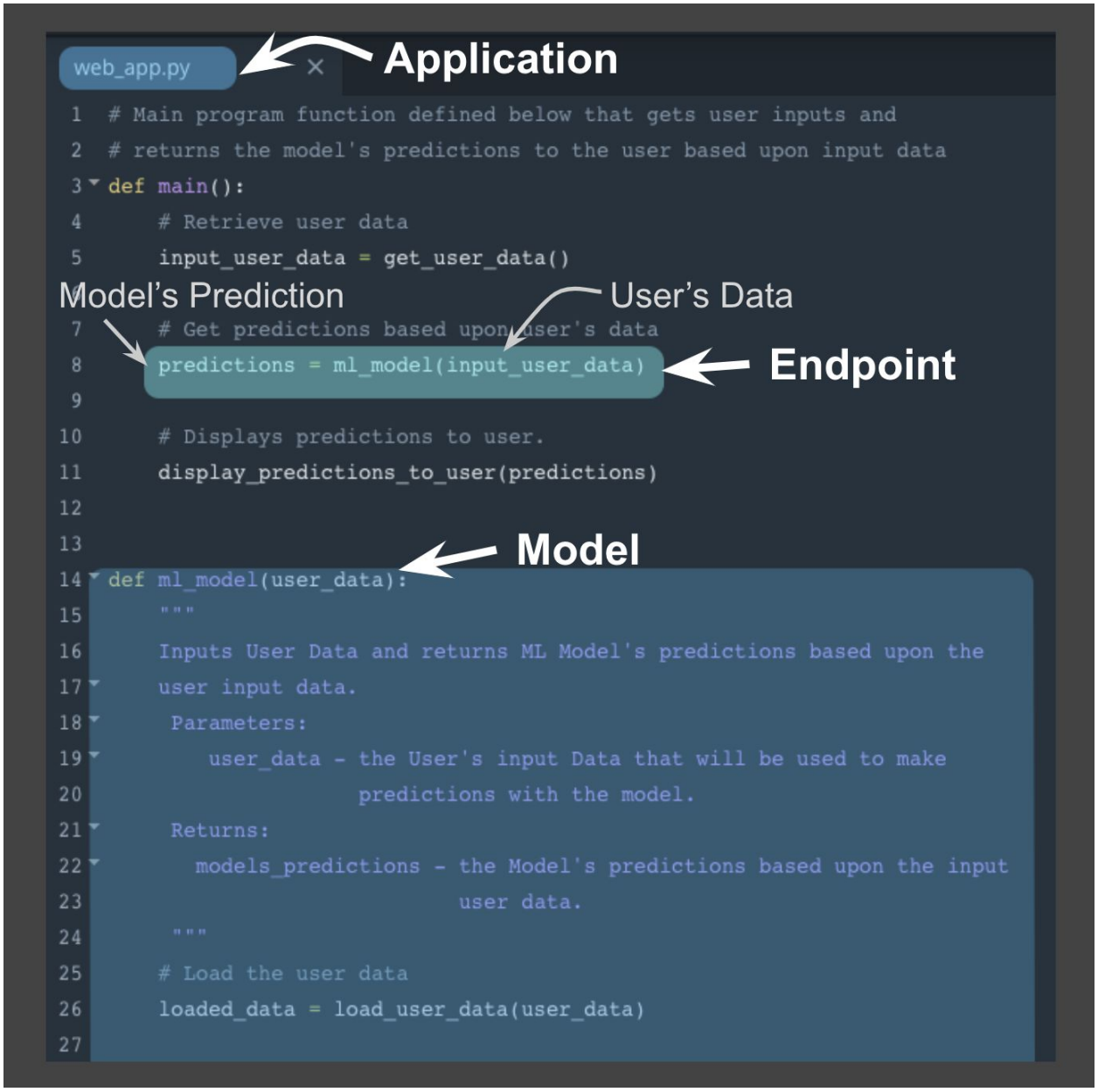


One way to think of the ***endpoint*** that acts as this *interface*, is to think of a *Python program* where:

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

The image **above** depicts the association between a ***Python program*** and the **endpoint**, **model**, and **application**.

- the **endpoint**: *line **8 function call*** to **ml_model**
- the **model**: beginning on *line **14 function definition*** for **ml_model**
- the **application**: ***Python program*** **web_app.py**



Using this example **above** notice the following:

- *Similar* to a ***function call*** the **endpoint** accepts *user data* as the ***input*** and ***returns*** the *model's prediction* based upon this ***input*** through the **endpoint**.
- In the example, the *user data* is the ***input argument*** and the *prediction* is the ***returned value*** from the ***function call***.
- The **application**, here the ***Python program***, displays the *model's prediction* to the *application user*.

This example highlights how the **endpoint** itself is just the ***interface*** between the **model** and the **application**; where this ***interface*** enables users to get *predictions* from the ***deployed*** *model* based on their *user data*.

Endpoint and REST API

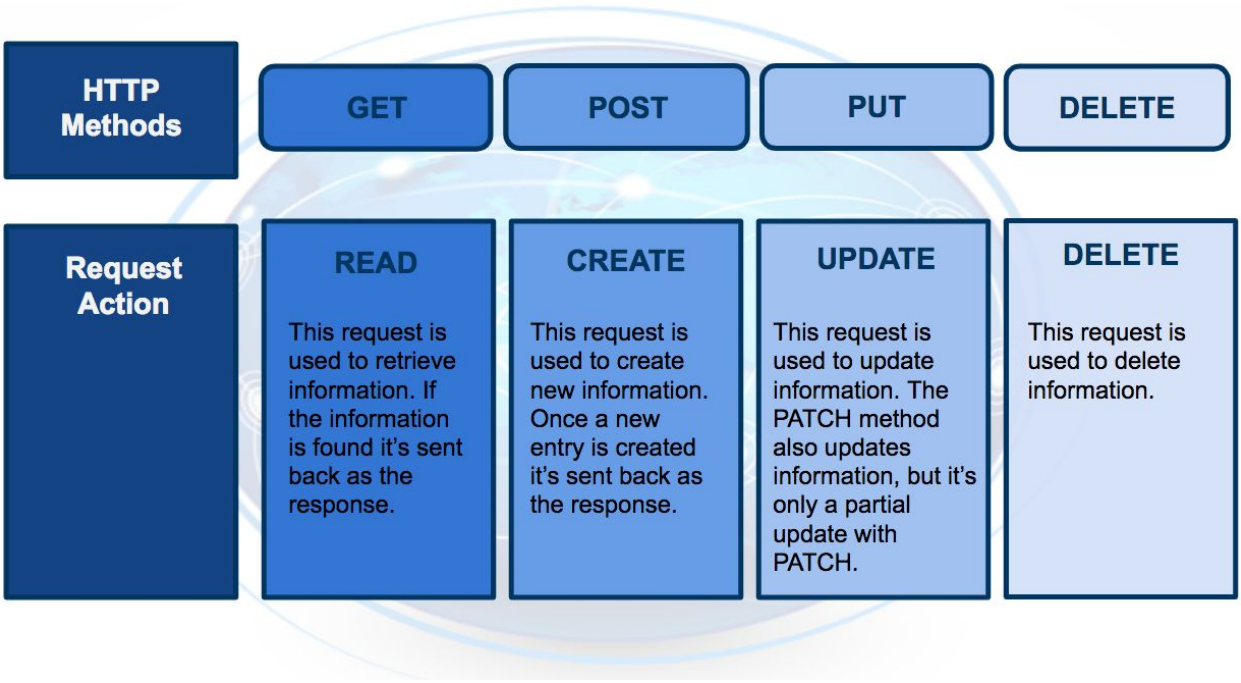
Communication between the **application** and the **model** is done through the **endpoint** (***interface***), where the **endpoint** is an **Application Programming Interface** (**API**).

- 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:

- **Endpoint**
    - This **endpoint** will be in the form of a URL, Uniform Resource Locator, which is commonly known as a web address.
- HTTP Method
    - Below you will find four of the **HTTP methods**, but for purposes of ***deployment*** our **application** will use the ***POST method*** *only*.

- HTTP Headers
    - The **headers** will contain additional information, like the format of the data within the message, that's passed to the *receiving* program.
- 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:

- 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*.
- HTTP Headers
    - The **headers** will contain additional information, like the format of the data within the **message**, that's passed to the receiving program.
- 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**.

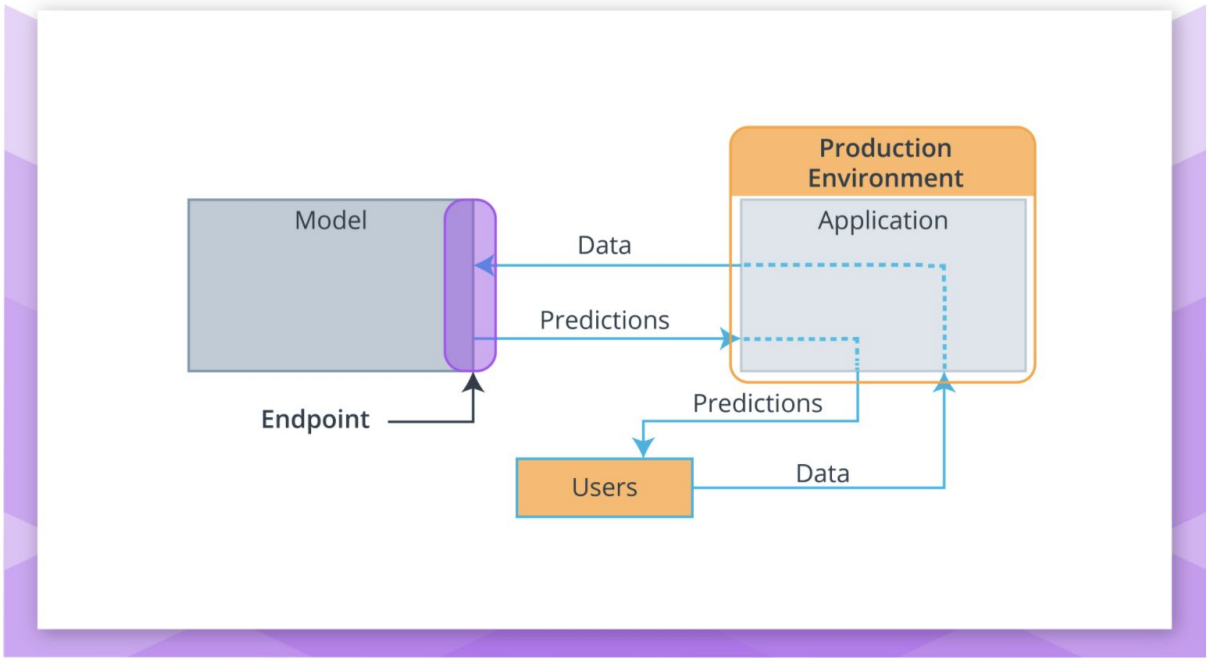
As we learn more about **REST*ful* API**, realize that it's the **application's** responsibility:

- 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.

Notice the following regarding the *information* included in the ***HTTP messages*** sent between **application** and **model**:

- Often *user's data* will need to be in a *CSV* or *JSON* format with a specific *ordering* of the data that's dependent upon the **model** used.
- Often *predictions* will be returned in *CSV* or *JSON* format with a specific *ordering* of the returned *predictions* dependent upon the **model** used.

Containers



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 tool set* works with *any* container regardless of the software the container contains.

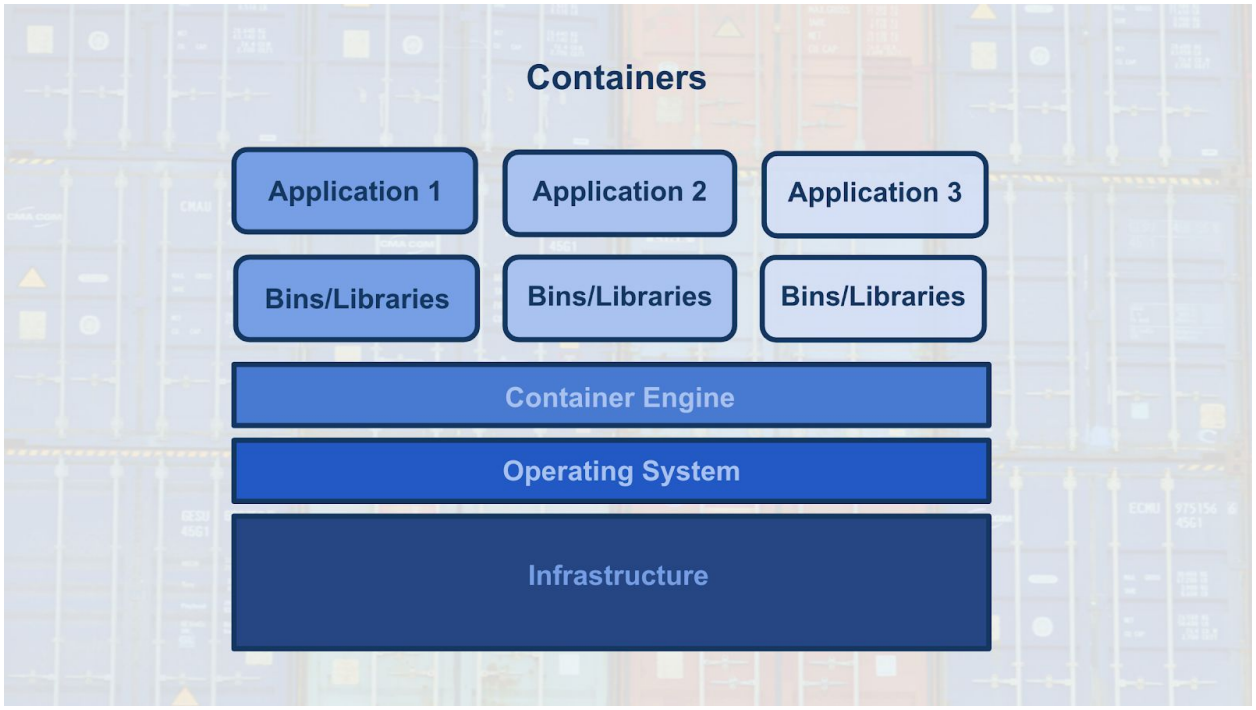
Container Structure

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

- The underlying *computational infrastructure* which 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*:

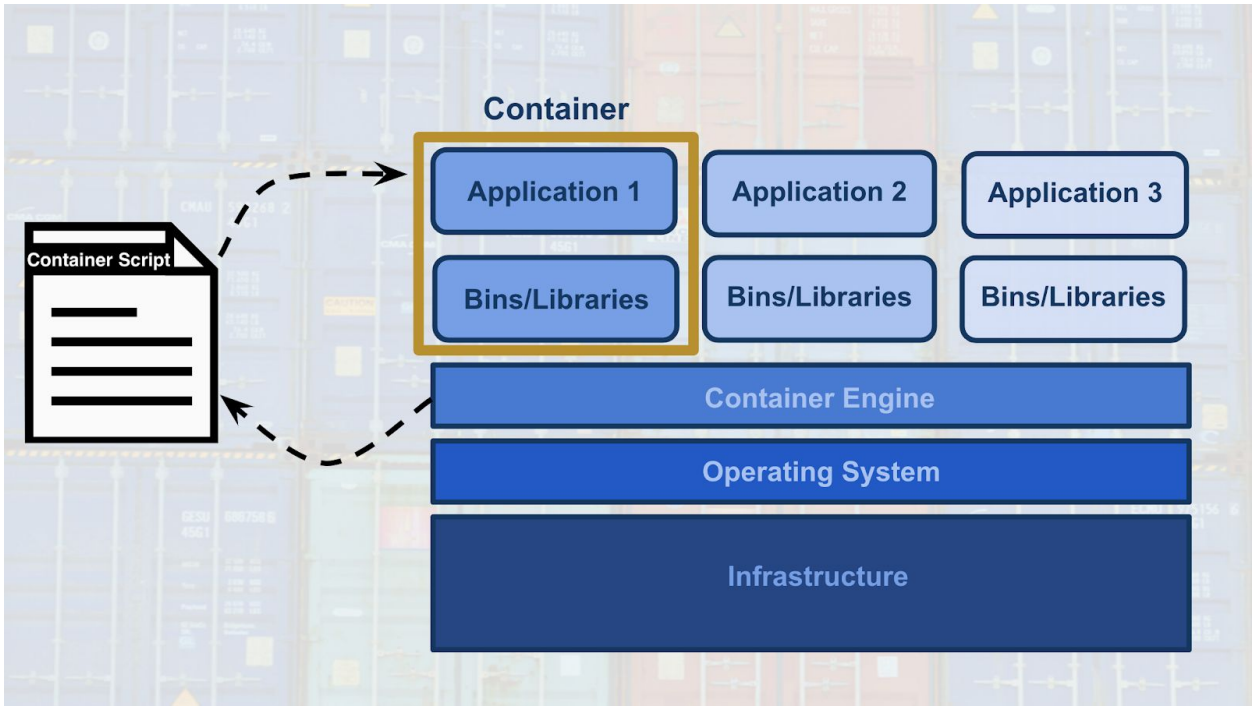
1. Isolates the application, which *increases* security.
2. Requires *only* software needed to run the application, which uses computational resources *more efficiently* and allows for faster application deployment.
3. Makes application creation, replication, deletion, and maintenance easier and the 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 is used to create a *container*; for *Docker* these *container scripts* are referred to as *dockerfiles*.

This is shown with the image below, where the *container engine* uses a *container script* to create a *container* for an application to run within. These *container script files* can be stored in repositories, which provide a simple means to share and replicate *containers*. For *Docker*, the Docker Hub is the official repository for storing and sharing *dockerfiles*. Here's an example of a dockerfile that creates a docker container with Python 3.6 and PyTorch installed.

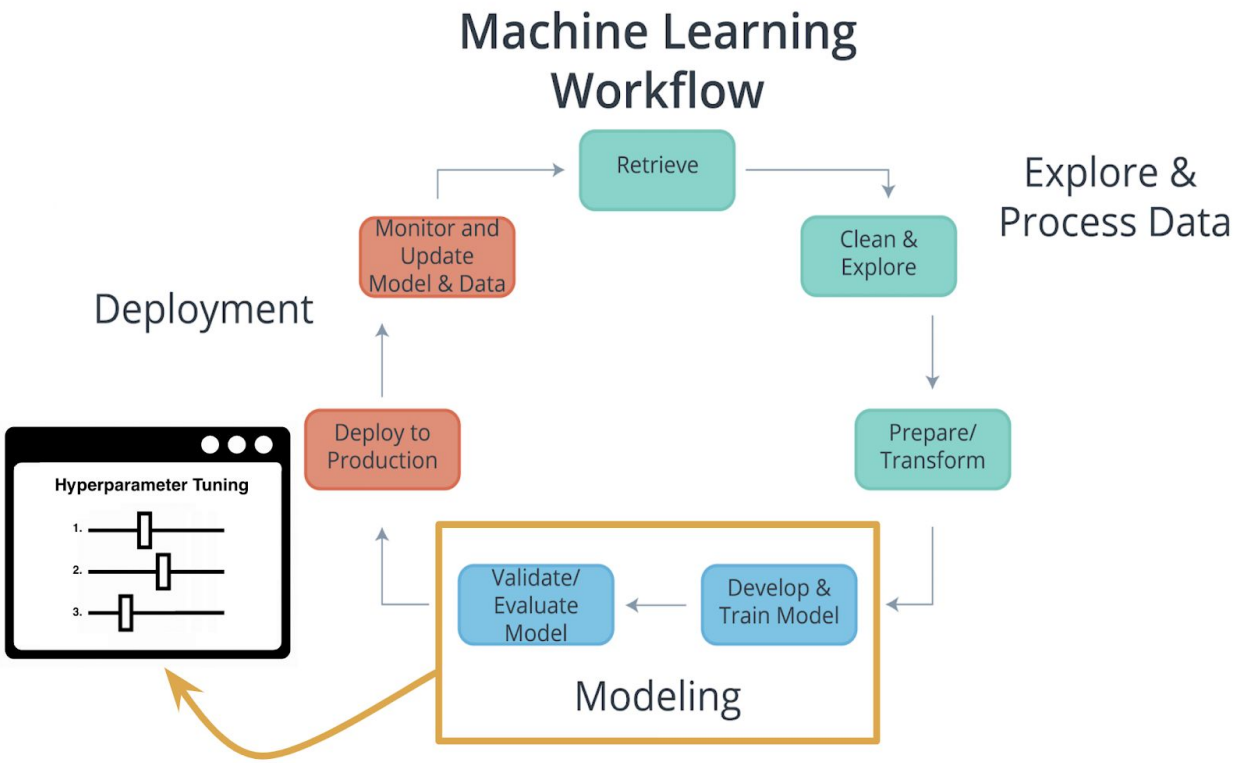


Characteristics of Deployment and Modeling

Recall that:

- 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 this model.

- Also remember that a production environment can be thought of as a web, mobile, or other software application that is currently being used by many people and must respond quickly to those users' requests.



Characteristics of Modeling

Hyperparameters

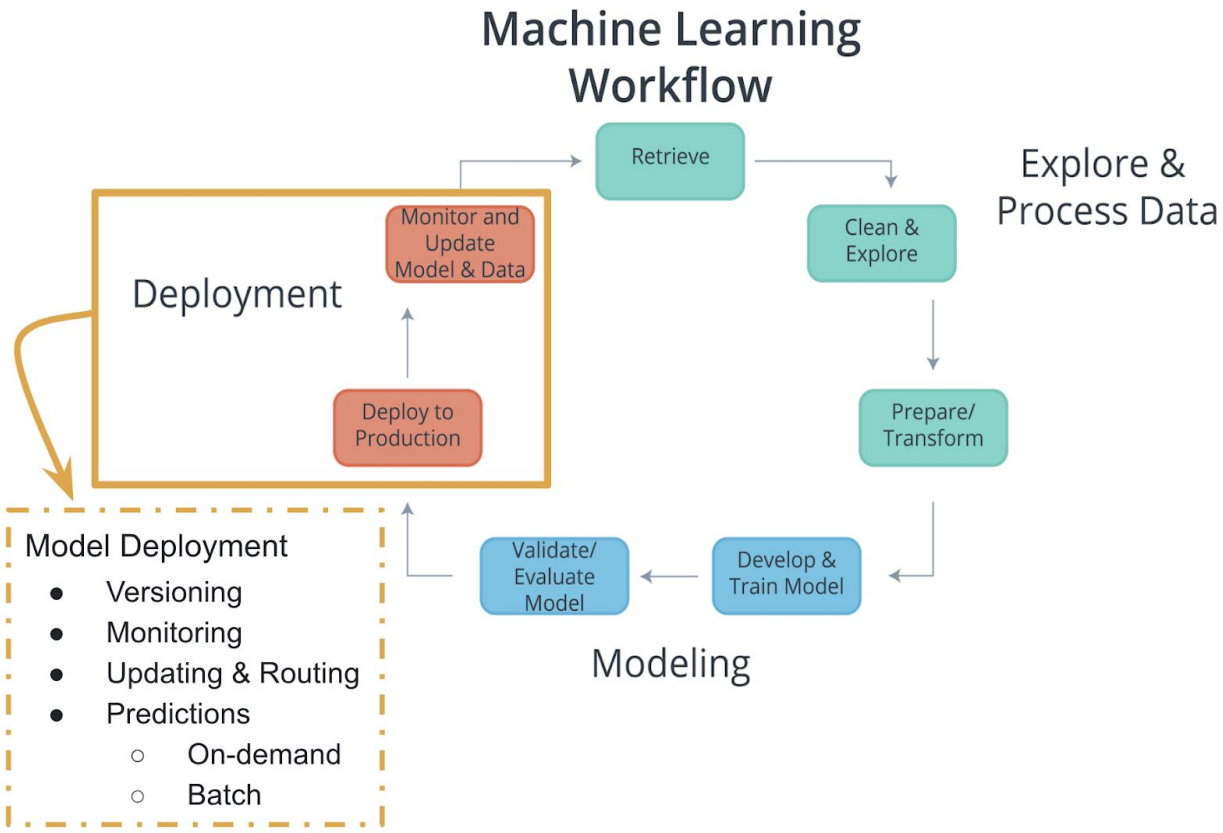
In machine learning, a hyperparameter is a parameter whose value cannot be estimated from the data.

Specifically, a hyperparameter is not directly learned through the estimators; therefore, their 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 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 type of predictions, one expects:

- a low latency of response to each prediction request,
- but allows for the possibility of high variability in request volume.

Predictions are returned in the response from 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 type of predictions, one expects:

- 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 to 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 uses a complex model to predict customer satisfaction across a number of their products and they need these estimates for a weekly report. This would require processing customer data through a batch prediction request on a weekly basis.

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.

We also looked at how we could combine a deployed SageMaker endpoint with Lambda and API Gateway to create our own simple web app.