Model Severing

Szu-Chi Chung

Department of Applied Mathematics, National Sun Yat-sen University

Ship an inference model

- A data science project doesn’t end when you arrive at a Colab notebook that can save a trained model. You rarely put in production the exact same Python model object that you manipulated during training
- First, you may want to export your model to something other than Python:
  - Your production environment may not support Python at all - for instance, if it’s a mobile app or an embedded system
  - If the rest of the app isn’t in Python (it could be in JavaScript, C++, etc.), the use of Python to serve a model may induce significant overhead
- Second, since your production model will only be used to output predictions, rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint

Ship an inference model

- Once you have trained a model, you can easily use it in any Python code
  - But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API).
  - This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions

Ship an inference model

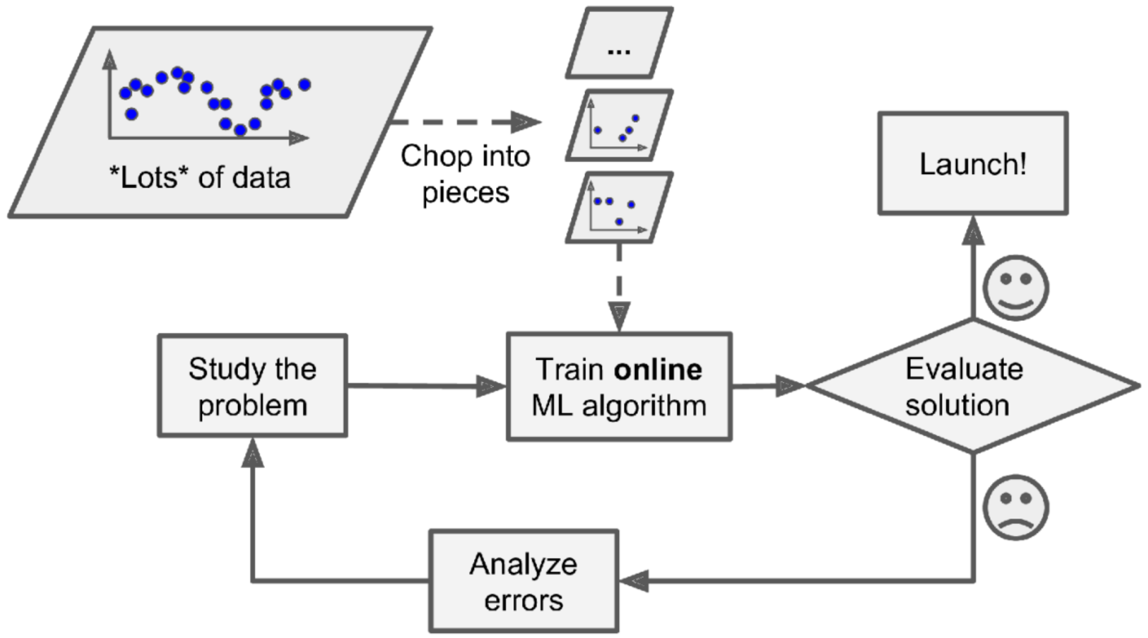
- It’s usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production
  - This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires
  - However, we can skip this step if we retrieve our features from the same source location for both training and serving, ie. from a feature store

Deploy the model!

- Back to the imagination that you're starting your own data science consulting shop. You put up a fancy website, you notify your network. The projects start rolling in:
  1. A personalized photo search engine for a picture-sharing social network—type in "wedding" and retrieve all the pictures you took at weddings
  2. Flagging spam and offensive text content among the posts of a chat app
  3. Building a music recommendation system for users of an online radio
  4. Detecting credit card fraud for an e-commerce website
  5. Predicting display ad click-through rate to decide which ad to serve to a given user at a given time
  6. Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line

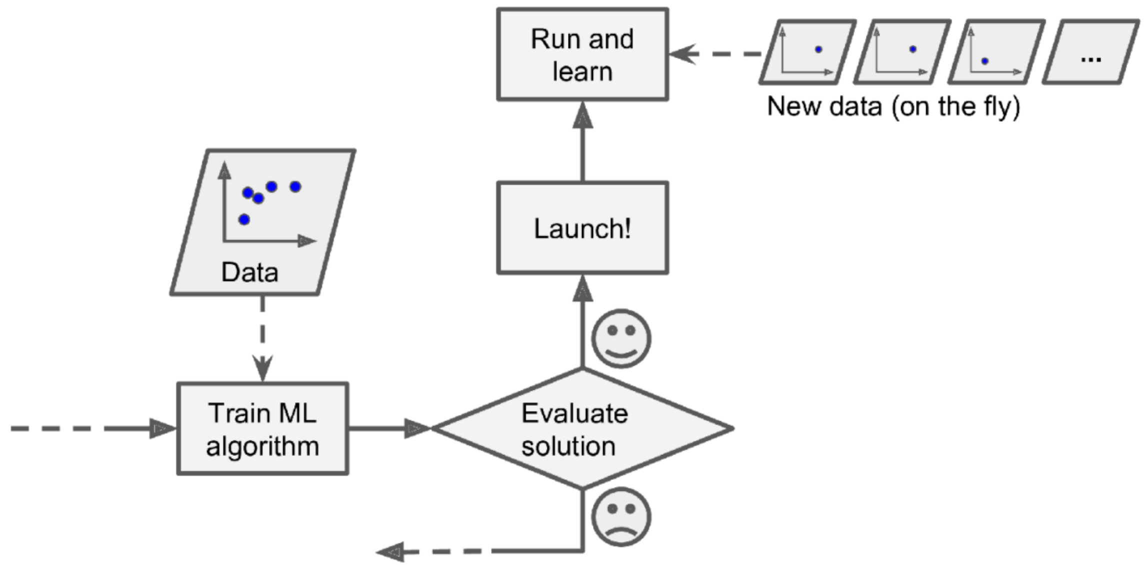
Batch and online learning

- Batch and Online Learning
  - In *online learning*, you train the system incrementally by feeding it data sequentially, either individually or *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly
  - It can receive data as a continuous flow and need to adapt to change rapidly
    - It is also a good option if you have limited computing resources
  - One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*



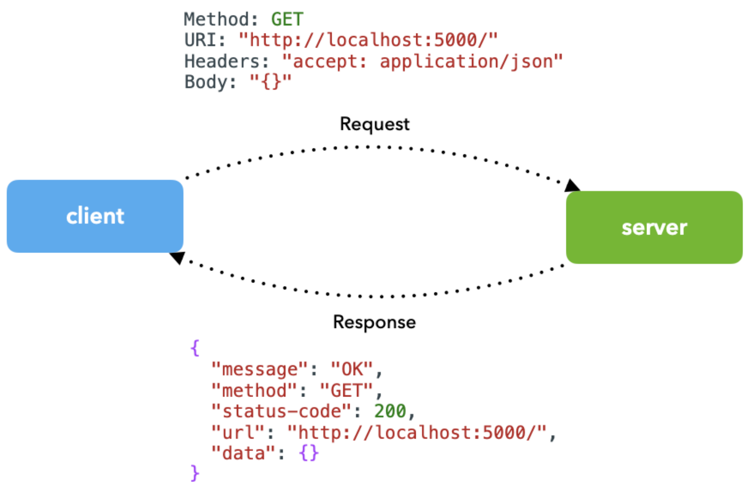
Batch and online learning

- Batch and Online Learning
  - If bad data is fed to the system, the system's performance will gradually decline
  - For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk you need to monitor your system closely and promptly switch learning off



Deploying a model as a REST API

- This is perhaps the common way to turn a model into a product: install model environment (Like scikit-learn, xgboost or Tensorflow) on a server or cloud instance, and query the model's predictions via a REST API
  - You could build your own serving app using something like Flask or FastAPI (or any other Python web development library)
  - TensorFlow also has its own library for shipping models as APIs, called TensorFlow Serving. With TensorFlow Serving, you can deploy a Keras model in minutes



```
Method: GET
URI: "http://localhost:5000/"
Headers: "accept: application/json"
Body: "{}"
```

```
{
  "message": "OK",
  "method": "GET",
  "status-code": 200,
  "url": "http://localhost:5000/",
  "data": {}
}
```

Deploying a model as a REST API

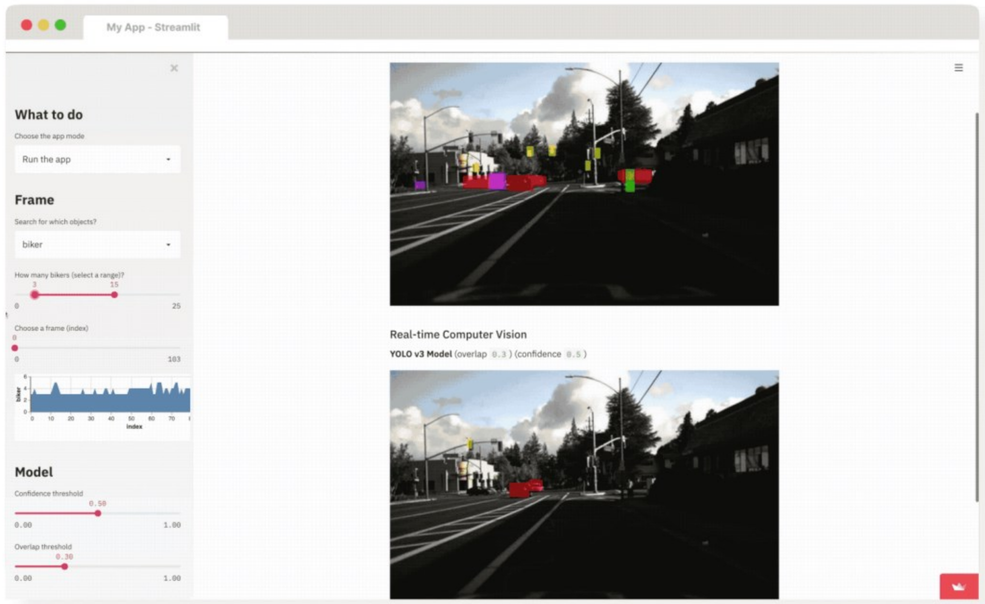
- You should use this deployment setup when
  - The application that will consume the model's prediction will have reliable *access to the internet* (obviously)
    - For instance, if your application is a mobile app, serving predictions from a remote API means that the application won't be usable in airplane mode or in a low-connectivity environment
  - The application does *not have strict latency requirements*: the request, inference, and answer round trip will typically take around 500 ms
  - The input data sent for inference is *not highly sensitive*: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer)
- For instance, the image search engine project, the music recommender system, and the credit card fraud detection project are all good fits for serving via a REST API

Deploying a model as a REST API

- The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies
  - However, it is based on JSON, which is text-based and fairly verbose
    - It is inefficient in terms of serialization/deserialization time and payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth. You may try other protocol like gRPC instead

Deploying a model in the browser

- Deep learning is often used in browser-based or desktop-based JavaScript applications
  - While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in having the model run directly in the browser, on the user's computer (utilizing GPU resources if they're available).



What to do

Choose the app mode: Run the app

Frame

Search for which objects?: biker

How many bikers (select a range)? 3 – 15 (0 to 25)

Choose a frame (index): 0 (0 to 103)

Model

Confidence threshold: 0.50 (0.00 to 1.00)

Overlap threshold: 0.30 (0.00 to 1.00)

Real-time Computer Vision

YOLO v3 Model (overlap `0.3`) (confidence `0.5`)

Deploying a model in the browser

- Use this setup when
  - You want to *offload compute* to the end user, which can dramatically reduce server costs
  - You need your app to keep working without connectivity, after the model has been downloaded and cached
    - When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable
  - Your application has strict latency constraints. While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 *ms* of network round trip
  - The input data needs to stay on the end user's computer or phone. For instance, in spam detection project, the web version and the desktop version of the chat app (implemented as a cross-platform app written in Java-Script) should use a locally run model

Deploying a model in the browser

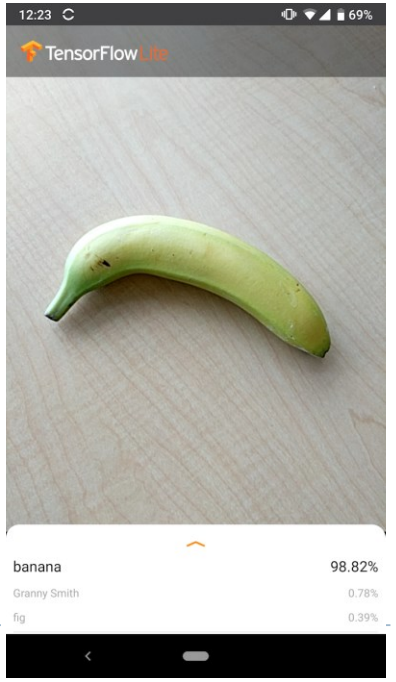
- You should only go with this option if your model is small enough that it won’t hog the CPU, GPU, or RAM of your user’s laptop or smartphone
  - Since the entire model will be downloaded to the user’s device, you should make sure that nothing about the model needs to stay confidential. Be mindful of the fact that, given a trained model, it is usually possible to recover some information about the training data: better not to make your trained model public if it was trained on sensitive data
  - To deploy a model in JavaScript, the TensorFlow ecosystem includes TensorFlow.js, a JavaScript library for deep learning that implements almost all of the Keras API as well as many lower-level TensorFlow APIs. You can easily import a saved Keras model into TensorFlow.js to query it as part of your browser-based JavaScript app or your desktop Electron app

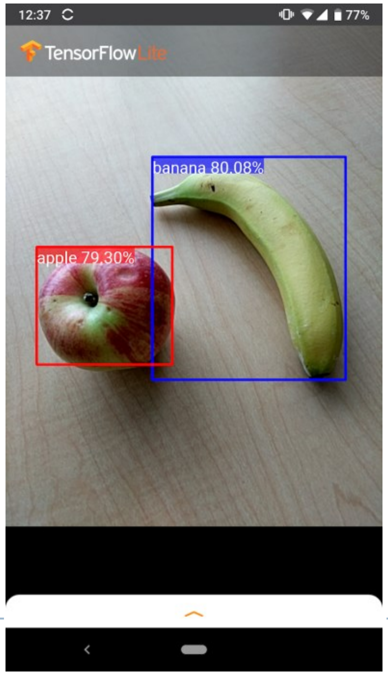
The infrastructure

- An important question when deploying a model as a REST API or webapp is whether you want to host the code on your own, or whether you want to use a fully managed thirdparty cloud service
  - For instance, Google Cloud AI Platform lets you simply upload your model to Google Cloud Storage (GCS), and it gives you an API endpoint to query it. It takes care of many practical details such as batching predictions, load balancing, and scaling
  - Use container orchestration via Kubernetes for managed deployment, scaling, etc. There are also fully-managed solutions, such as SageMaker, BentoML, etc
  - Use serverless options such as AWS Lambda, Google Cloud Functions, etc

Deploying a model on a device

- Sometimes, you may need your model to live on the same device that runs the application that uses it - maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device
  - You may have seen a camera capable of automatically detecting people and faces in the scenes you pointed it at: that was probably a small deep learning model running directly on the camera





Deploying a model on a device

- You should use this setup when
  - Your model has *strict latency constraints* or needs to run in a low-connectivity environment. If you're building an immersive augmented reality application, querying a remote server is not a viable option
  - Your model can be made *sufficiently small* that it can run under the memory and power constraints of the target device. You can use the TensorFlow Model Optimization Toolkit to help with this
  - Getting the highest possible accuracy isn't mission critical for your task. There is always a trade-off between *runtime efficiency and accuracy*, so memory and power constraints often require you to ship a model that isn't quite as good as the best model you could run
  - The input data is *strictly sensitive* and thus shouldn't be decryptable on a remote server

Deploying a model on a device

- For instance
  - Spam detection model for chat app will need to run on the end user's smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model. Likewise, the bad-cookie detection model has strict latency constraints and will need to run at the factory
- To deploy a Keras model on a smartphone or embedded device, your go-to solution is TensorFlow Lite
  - It's a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers. It includes a converter that can straightforwardly turn your Keras model into the TensorFlow Lite format

Monitor your model in the wild

- You’ve exported an inference model, you’ve integrated it into your application, and you’ve done a dry run on production data - the model behaved exactly as you expected
  - You’ve written unit tests as well as logging and status-monitoring code. Now it’s time to press the big red button and deploy to production
  - Even this is not the end. Once you’ve deployed a model, you need to keep monitoring its behavior, its performance on new data, its interaction with the rest of the application, and its eventual impact on business metrics

Monitor your model in the wild

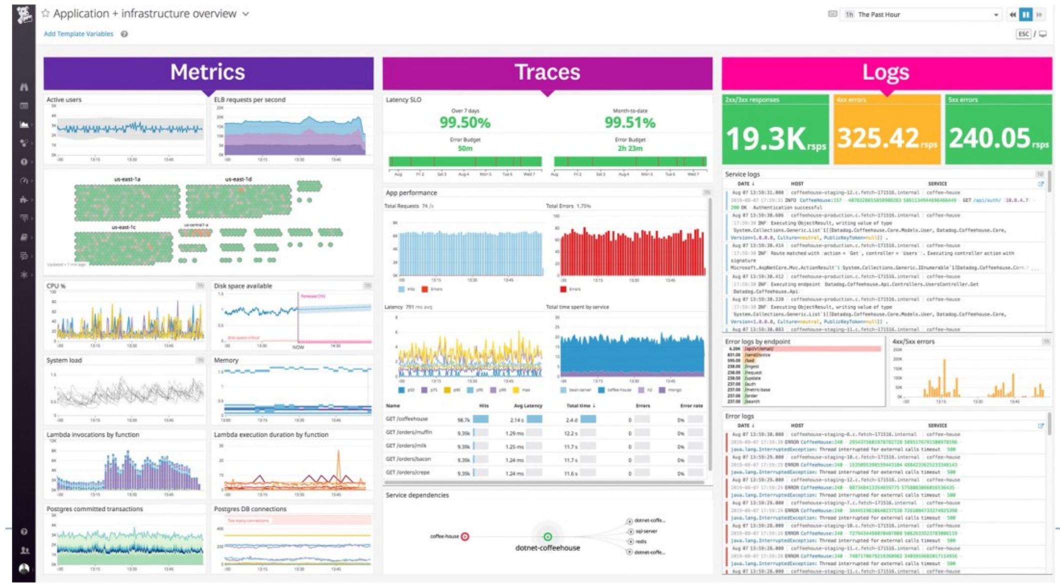
- For instance
  - Is user engagement in your online radio up or down after deploying the new music recommender system?
  - Has the average ad click-through rate increased after switching to the new click-through-rate prediction model?
  - If possible, do a regular manual audit of the model's predictions on production data. It's generally possible to reuse the same infrastructure as for data annotation: send some fraction of the production data to be manually annotated, and compare the model's predictions to the new annotations. For instance, you should definitely do this for the image search engine and the bad-cookie flagging system
  - When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system)

Monitor your model in the wild

- Machine learning model is subject to natural performance degradation over time, as well as unintended behavior, since the data exposed to the model will be different from what it has been trained on. This isn't something we should be trying to avoid but rather understand and mitigate as much as possible
  - We'll understand the short comings from attempting to capture performance degradation in order to motivate the need for drift detection.
- Testing and monitoring share a lot of similarities, such as ensuring that certain expectations around data completeness, distributions, schema adherence, etc. are met
  - However, a key distinction is that monitoring involves comparing live, streaming data distributions from production to fixed/sliding reference distributions from training data

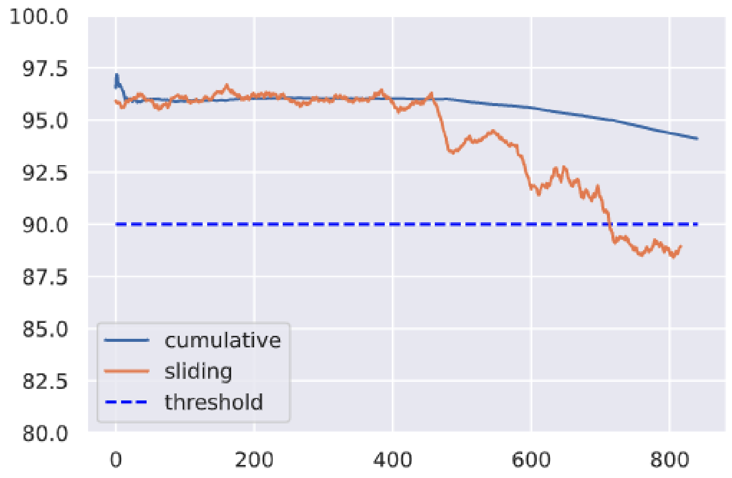
System health

- The first step is to ensure that the actual system is up and running as it should
  - This can include metrics specific to service requests such as latency, throughput, error rates, etc. as well as infrastructure utilization such as CPU/GPU utilization, memory, etc
  - Fortunately, most cloud providers and even orchestration layers will provide this insight into our system's health for free through a dashboard. In the event we don't, we can easily use Grafana, Datadog, etc. to ingest system performance metrics from logs to create a customized dashboard and set alerts



Performance

- Monitoring the system's health won't be enough to capture the underlying issues with our model. So, naturally, the next layer of metrics to monitor involves the model's performance
  - It's usually never enough to just analyze the cumulative performance metrics across the entire span of time since the model has been deployed
  - Instead, we should also inspect performance *across a period of time that's significant for our application (ex. daily).* These sliding metrics might be more indicative and we might be able to identify issues faster by not obscuring them with historical data



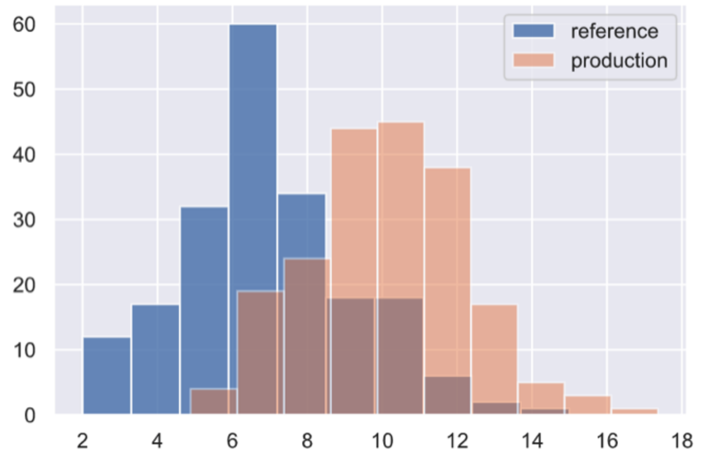
Drift

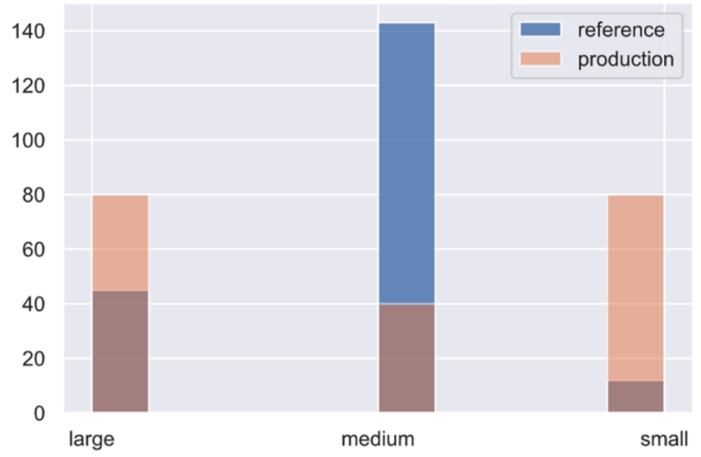
- We need to first understand the different types of issues that can cause our model's performance to decay (model drift). The best way to do this is to look at all the moving pieces of what we're trying to model and how each one can experience drift

		Drift
X	inputs (features)	Data drift → $P(X) \neq P_{ref}(X)$
y	outputs (ground-truth)	Target drift → $P(y) \neq P_{ref}(y)$
$p(y\|X)$	actual relationship between and X and y	Concept drift → $P(y\|X) \neq P_{ref}(y\|X)$

Data drift

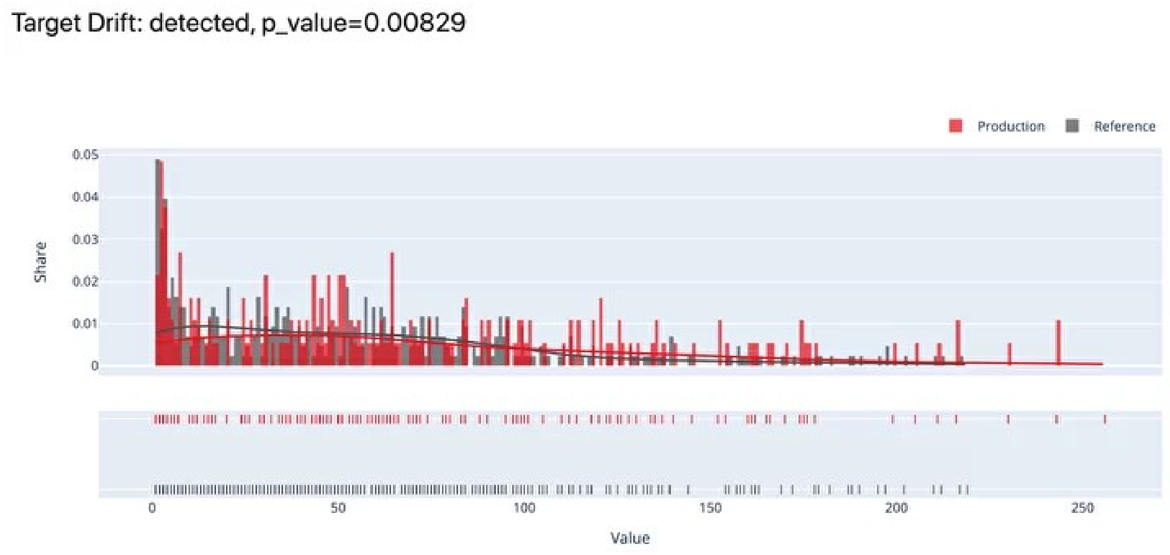
- Data drift, also known as feature drift or covariate shift, occurs when the distribution of the production data is different from the training data
  - The model is not equipped to deal with this drift in the feature space and so, it's predictions may not be reliable. The actual cause of drift can be attributed to natural changes in the real-world but also to systemic issues such as missing data, pipeline errors, schema changes, etc. It's important to inspect the drifted data and trace it back along it's pipeline to identify when and where the drift was introduced





Target drift

- Besides just the input data changing, as with data drift, we can also experience drift in our outcomes
  - This can be a shift in the distributions but also the removal or addition of new classes with categorical tasks. Though retraining can mitigate the performance decay caused target drift, it can often be avoided with proper inter-pipeline communication about new classes, schema changes, etc

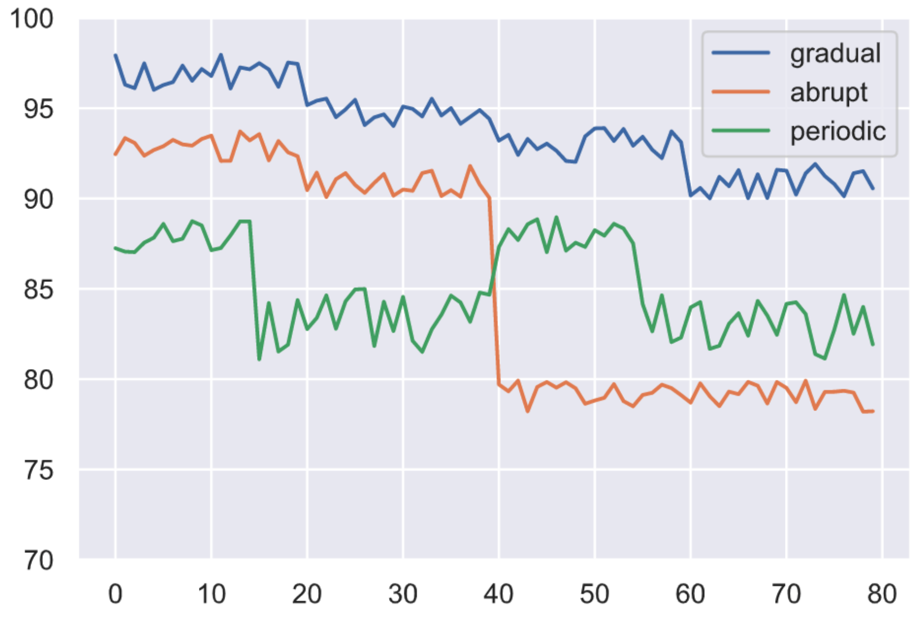


Concept drift

- Lastly, no model lasts forever. You’ve already learned about *concept drift*: over time, the characteristics of your production data will change, gradually degrading the performance and relevance of your model
  - The lifespan of your music recommender system will be counted in weeks. For the credit card fraud detection systems, it will be days. A couple of years in the best case for the image search engine. As soon as your model has launched, you should be getting ready to train the next generation that will replace it
    - Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
    - Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to collecting samples that seem to be difficult for your current model to classify - such samples are the most likely to help improve performance

Concept drift

- Besides the input and output data drifting, we can have the actual relationship between them drift as well
  - This concept drift renders our model ineffective because the patterns it learned to map between the original inputs and outputs are no longer relevant. Concept drift can be something that occurs in various patterns



Locating drift

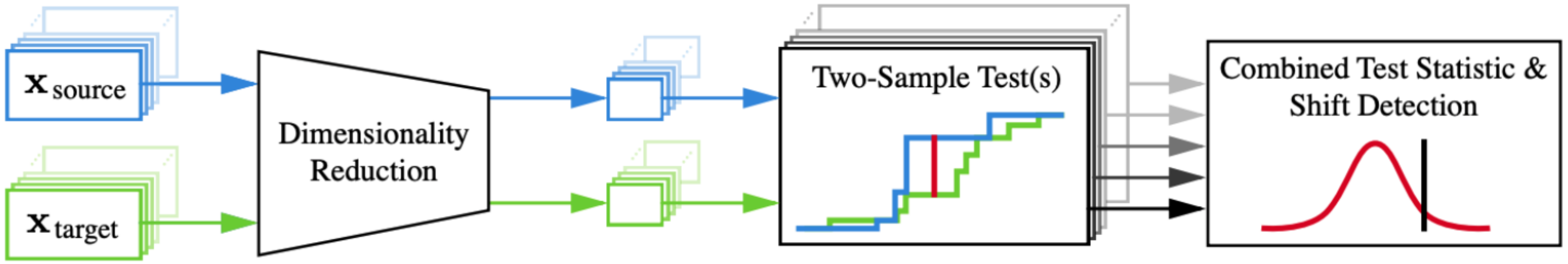
- Now that we've identified the different types of drift, we need to locate and how often to measure it. Here are the constraints we need to consider:
  - Reference window: the set of points to compare production data distributions with to identify drift
  - Target window: the set of points to compare with the reference window to determine if drift has occurred
- Since we're dealing with online drift detection (ie. detecting drift in live production data as opposed to past batch data), we can employ either a fixed or sliding window approach to identify our set of points for comparison
  - Typically, the reference window is a fixed, recent subset of the training data while the target window slides over time

Measuring drift

- Once we have the window of points we wish to compare, we need to know how to compare them
  - Expectations - The first line of measurement can be rule-based such as validating expectations around missing values, data types, value ranges, etc
  - Univariate - Once we've validated our rule-based expectations, we need to quantitatively measure drift. Traditionally, in order to compare two different sets of points to see if they come from the same distribution, we use two-sample hypothesis testing on the distance measured by a test
  - Multivariate - Measuring drift is fairly straightforward for univariate data but difficult for multivariate data

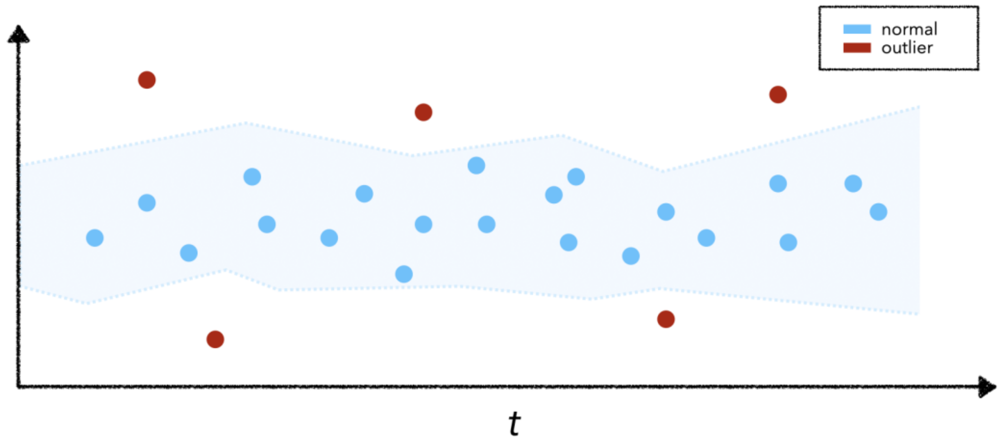
Measuring drift

- Univariate
  - Kolmogorov-Smirnov (KS) test or Chi-squared test
- Multivariate
  - Dimensionality reduction – PCA or Autoencdoer
  - Two-sample tests
    - Maximum Mean Discrepancy (MMD)
    - Kolmogorov-Smirnov (KS) Test + Bonferroni Correction


$\mathbf{x}_{\text{source}}$
$\mathbf{x}_{\text{target}}$
Dimensionality Reduction
Two-Sample Test(s)
Combined Test Statistic & Shift Detection

Outlier

- With drift, we're comparing a window of production data with reference data as opposed to looking at any one specific data point. While each individual point may not be an anomaly or outlier, the group of points may cause a drift
  - It's not very easy to detect outliers because it's hard to constitute the criteria for an outlier. Therefore the outlier detection task is typically unsupervised and requires a stochastic streaming algorithm to identify potential outliers
  - Typically, outlier detection algorithms fit to the training set to understand what normal data looks like and then we can use a threshold to predict outliers



Conclusion

- This concludes the universal workflow of machine learning—that's a lot of things to keep in mind. It takes time and experience to become an expert, but don't worry, you're already a lot wiser than you were a few chapters ago. You are now familiar with the big picture—the entire spectrum of what machine learning projects entail. Always keep in mind the big picture!

References

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition Chapter 1 and 19

[2] Deep learning with Python, 2nd Edition Chapter 9

[3] https://madewithml.com/courses/mlops/monitoring/

[4] https://evidentlyai.com/blog/tutorial-1-model-analytics-in-production

Appendix

Resources and libraries

- Lectures
  - https://github.com/microsoft/ML-For-Beginners/blob/main/3-Web-App/1-Web-App/README.md
  - https://github.com/microsoft/Data-Science-For-Beginners/blob/main/5-Data-Science-In-Cloud/17-Introduction/README.md
  - https://github.com/microsoft/Data-Science-For-Beginners/blob/main/6-Data-Science-In-Wild/20-Real-World-Examples/README.md
  - https://madewithml.com/courses/mlops/dashboard/
- CI/CD workflows
  - https://madewithml.com/courses/mlops/cicd/
- Pipelines
  - https://madewithml.com/courses/mlops/pipelines/
- Continual learning
  - https://madewithml.com/courses/mlops/continual-learning/

Resources and libraries

- Testing
  - https://madewithml.com/courses/mlops/testing/
- Deployment
  - https://github.com/bentoml/BentoML
  - Tensorflow serving
  - TensorFlow.js
  - TensorFlow Lite
  - Streamlit
  - Gradio
  - Flask
  - FastAPI

Resources and libraries

- Monitoring and testing
  - https://github.com/deepchecks/deepchecks
  - https://github.com/evidentlyai/evidently
  - https://github.com/SeldonIO/alibi-detect
  - https://github.com/scikit-multiflow/scikit-multiflow
  - https://greatexpectations.io/

Inference model optimization

- Optimizing your model for inference is especially important when deploying in an environment with strict constraints on available power and memory or for applications with low latency requirements
  - You should always seek to optimize your model. There are two popular optimization techniques you can apply:
    - *Weight pruning*—Not every coefficient in a weight tensor contributes equally to the predictions. It's possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. This reduces the memory and compute footprint of your model, at a small cost in performance metrics. By deciding how much pruning you want to apply, you are in control of the trade-off between size and accuracy
    - *Weight quantization*—Deep learning models are trained with single-precision floating-point (float32) weights. However, it's possible to *quantize* weights to 8-bit signed integers (int8) to get an inference-only model that's a quarter the size but remains near the accuracy of the original model