Machine Learning Modeling Pipelines in Production

In the third course of Machine Learning Engineering for Production Specialization, you will build models for different serving environments; implement tools and techniques to effectively manage your modeling resources and best serve offline and online inference requests; and use analytics tools and performance metrics to address model fairness, explainability issues, and mitigate bottlenecks.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 4: Model Analysis

Contents

Model Performance Analysis

What is next after model training/deployment?

- Is model performing well?
- Is there scope for improvement?
- Can the data change in future?
- Has the data changed since you created your training dataset?

- Successfully training a model and getting it to converge feels good. It often feels like you're done.
- If you're training a model for a class project or a paper that you're writing, then you are done.
- But for production ML, you now need to enter into a new phase of your development which involves a much deeper level of analysis of your model's performance from a few different directions, and that's what we'll be learning about this week.
- Let's start with a little bit of review of model analysis by looking at some of the different ways of analyzing models and determining performance.
- In much of this week, you'll be using the tools of TensorFlow model analysis or **TFMA**, and some related tools and technologies, but first, let's review some of the basics of model analysis.
- Here, it's important to emphasize that you want to look at your model performance, not just on your entire dataset, but on **individual slices** of your data.
- After training and or deployment, you might notice a decay in performance.
- It's natural to inquire about possible ways to improve model performance.
- Additionally, you need to anticipate data changes that you might expect to see in the future and act on those data changes that have occurred since you originally trained your model.
- Choosing the slices that are important to analyze is usually **based on domain knowledge**.
- This will allow you to determine, if there is room for improvement in your model, and perhaps address data changes that have happened since you originally trained your model.
- For example, if your model is designed to predict demand for different kinds of shoes, looking at the performance of your model on individual types of shoes, perhaps different colors or styles, that'll be important, and that will require slicing your data for different types of shoes.

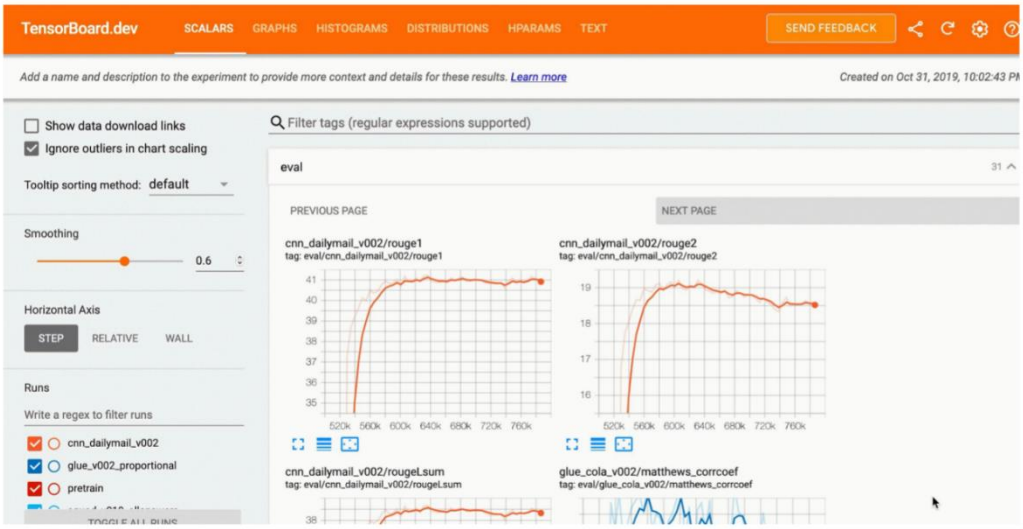
Black box evaluation vs model introspection

- Models can be tested for metrics like accuracy and losses like test error without knowing internal details
- For finer evaluation, models can be inspected part by part

- At a high level, there are two main ways to analyze the performance of your model.
- Black-box evaluation and model introspection.
- In black-box evaluation, you generally don't consider the internal structure of the model.
- You're just interested in quantifying the performance of the model through metrics and losses.
- This is often sufficient within the normal course of development.
- But if you're interested in how the model is working internally, maybe looking for ways to improve it, you can apply various **model introspection methods**.

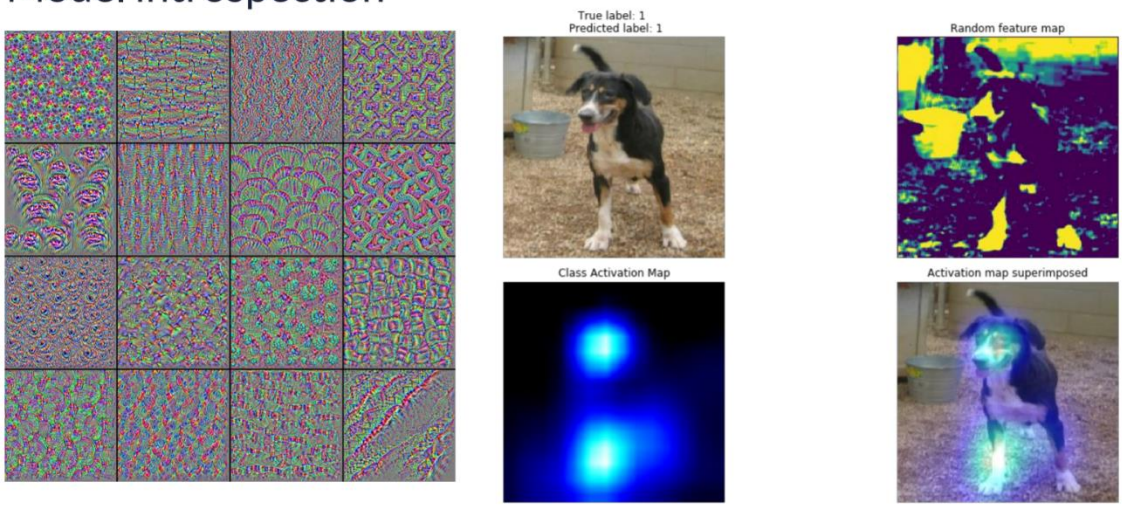
- Model introspection methods are very useful when you're experimenting with new architectures, to understand how data is flowing internally within each layer of the model.
- This can help you adjust and iterate your model architecture to improve performance and efficiency.

Black box evaluation



- TensorBoard is an example of a tool for **black-box evaluation**.
- Using TensorBoard, we can monitor the loss and accuracy of every iteration of the model.
- You can also closely monitor the training process itself.

Model introspection



- In model introspection, the goals change completely.
- You're not just interested in the model's final results. You're **also** interested in the details of each layer.
- On the left side, we have the maximally activated patches of the various filters in the convolutional layers of the model - this is a CNN - when we run through a series of images belonging to a particular class.
- Using those patterns, you can inspect at which layer the model is learning a particular structure of your data.
- On the right side is an example of a **class activation map**.

- Here, you're interested in knowing which parts of the image are primarily responsible for making the desired prediction of that class.
- Using this information, you can try to improve the performance of your model by tuning or including more features related to those highlighted areas.

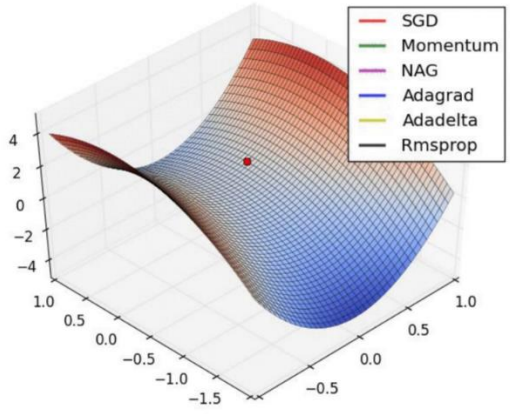
Performance metrics vs optimization objectives

- Performance metrics will vary based on the task like regression, classification, etc.
- Within a type of task, based on the end-goal, your performance metrics may be different
- Performance is measured after a round of optimization

- Machine Learning formulates the problem statement into an objective function
- Learning algorithms find optimum values for each variable to converge into local/global minima

- Next, let's look at the difference between performance metrics and optimization.
- First, **performance** metrics. Based on the problem you're trying to solve, you need to quantify the success of your model using some measurement, and for this, you use various performance metrics.
- Performance metrics will be different for different types of tasks like classification and regression and so forth. You're familiar with these from designing and training models.
- Now let's focus on the **optimization** part. This is your **objective function or cost function or loss function**. People use different names for it.
- When you train your model, you try to minimize the value of this function in order to find an optima, hopefully, a global optima in your loss surface.
- If you check out TensorBoard again, you'll notice options for tracking performance metrics like accuracy and optimization objectives such as loss after each epic of training and validation.

Performance metrics vs optimization objectives



- Let's quickly inspect the **optimization landscape**. This is also referred to as a **loss surface**.
- Here, the losses represented as a function of two parameters.
- Depicted here is a saddle point, where the curvature along different dimensions has different signs. One dimension curves up and another down.
- This is typical of many optimization problems.
- The animation shows trajectories for different optimization algorithms.
- Each of these optimizers traverses the landscape differently, according to their update method, and may take different amounts of time to converge to an optimum value.

Top level aggregate metrics vs slicing

- Most of the time, metrics are calculated on the entire dataset
- Slicing deals with understanding how the model is performing on each subset of data

- When you're evaluating your training performance, you're usually watching the top-level metrics.
- This is to decide whether your model is doing well or not. But it doesn't tell you about how well it does, on individual parts of the data.
- For example, different customers in different stores may experience your model very differently.
- If their experience isn't good, it reflects badly on your model

Introduction to TensorFlow Model Analysis (TFMA)

Why should you slice your data?

Your top-level metrics may hide problems

- Your model may not perform well for particular [customers | products | stores | days of the week | etc.]

Each prediction request is an individual event, maybe an individual customer

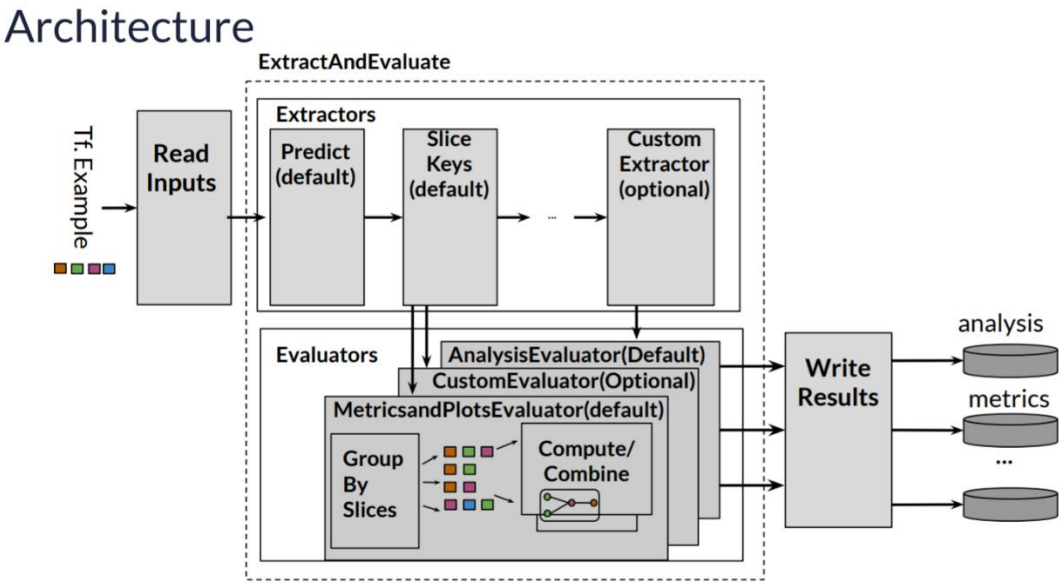
- For example, customers may have a bad experience
- For example, some stores may perform badly

- TensorFlow Model Analysis, or TFMA is a versatile tool for doing deep analysis of your model's performance.
- Your top-level metrics can easily hide problems with particular parts of your data.
- For example, your model may not perform well for particular customers, or particular products or stores or days of the week, or different subsets of your data that makes sense for your domain or problem.
- Thinking about customers who are requesting a prediction from your model, if your model produces a bad prediction, then that customer's experience of your model is bad, regardless of how well it may do in top-level metrics.
- To enable developers to look at their performance of their model at a deeper level, Google created TensorFlow Model Analysis or TFMA.

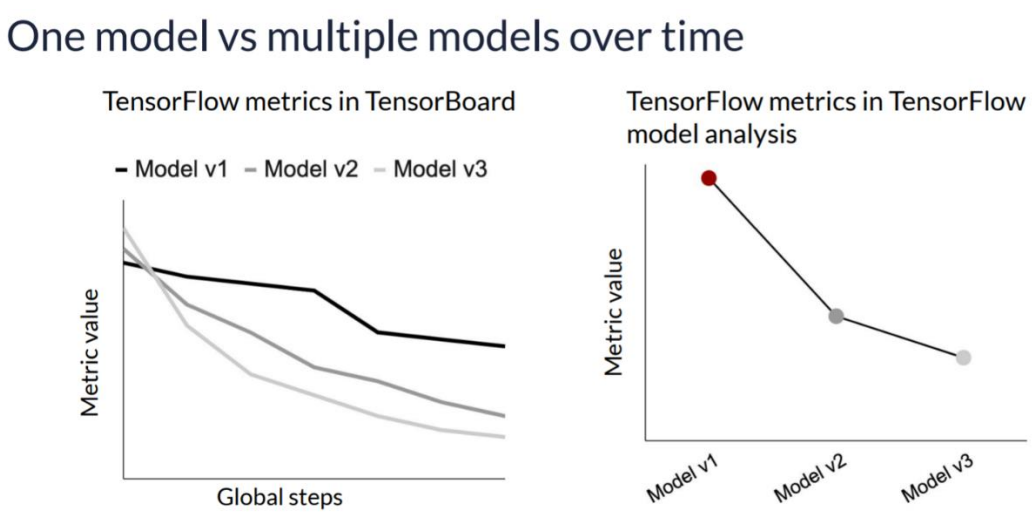
TensorFlow Model Analysis (TFMA)

Scalable framework	Open source library	Ensures models meet required quality thresholds	Used to compute and visualize evaluation metrics	Inspect model's performance against different slices of data

- TFMA is an open source, scalable framework for doing deep analysis of model performance, including analyzing performance on slices of data.
- TFMA is also a key part of a machine-learning pipeline, such as a TFX pipeline, to **perform deep analysis** before you deploy a newly trained version of your model.
- TFMA has built-in capabilities to check that your models meet certain quality standards and to visualize evaluation metrics and inspect performance on different slices of data.
- TFMA can be used by itself or as part of another framework such as TFX.
- Let's take a look at the high level architecture of TFMA.

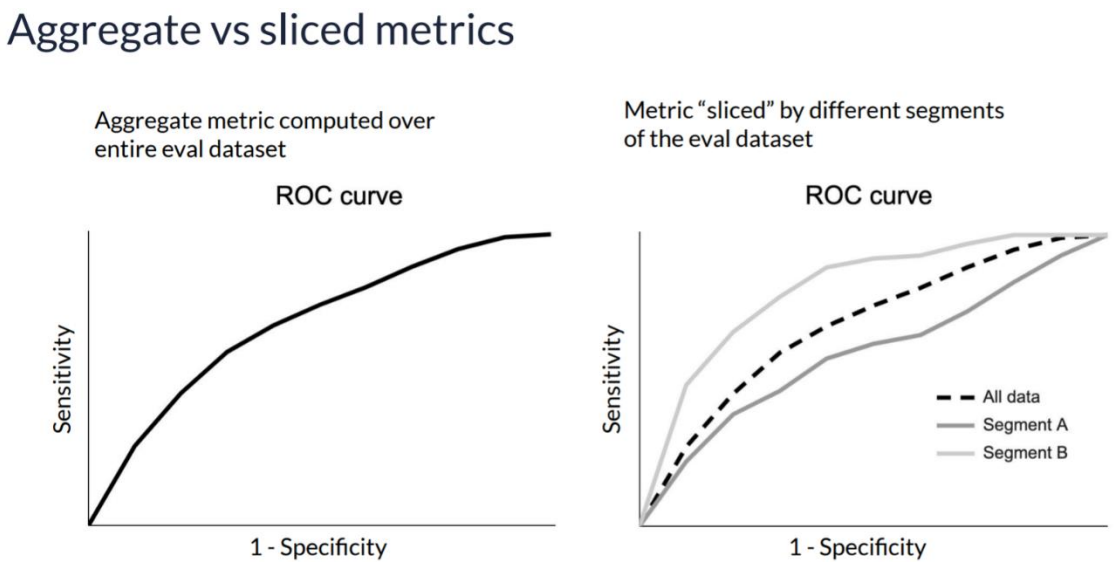


- TFMA pipeline is made up of four main components: Read inputs, extractors, evaluators, and write results.
- The first stage, read input, is made up of a transform that takes raw inputs such as CSV or TF records and so forth, and it converts it into a dictionary format which is understandable by the next stage, extraction.
- Across all the stages, the **output is kept in this dictionary format**, which is at the datatype **tfma.extracts**.
- The next stage, extraction, performs distributed processing using Apache Beam.
- Input extractor and slice key extractor pull slices of the original dataset, which will be used by predict extractor to run predictions on each slice.
- The results are sent to the next stage, again, as a tfma.extracts dictionary.
- The next stage, evaluation, also performs distributed processing using Apache Beam.
- There are several evaluators and you can create custom evaluators as well.
- For example, the MetricsandPlotsEvaluator extracts the required fields from the data to evaluate the performance of the model against the received predictions from the previous stage.
- The final stage, write results, as the name suggests, writes the results to disk.

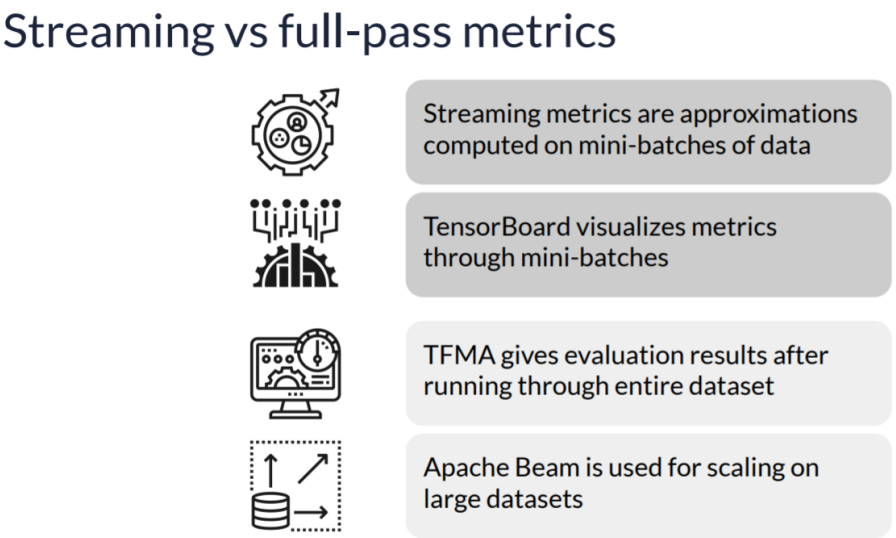


- TensorBoard and TFMA are used in different stages of the development process.
- At a high level, **TensorBoard is used to analyze the training process itself, while TFMA is used to do deep analysis of the finished trained model.**

- TensorBoard is used to inspect the training process of a single model, often as you're monitoring your progress during training.
- It can also be used to visualize the training progress for more than one model with performance for each plotted against their global training steps as their training.
- After training is finished, TFMA allows developers to compare different versions of their trained models.
- While TensorBoard visualizes streaming metrics of multiple models over global training sets, TFMA visualizes the metrics computed for a single model over multiple versions of the exported saved model.

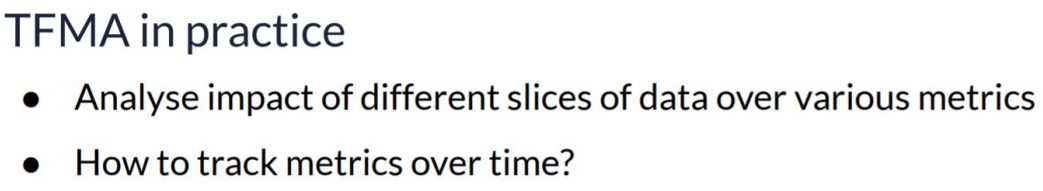


- Most model evaluation results look at aggregate or a top-level metrics on the entire training dataset.
- This aggregation often hides problems with model performance.
- For example, a model may have an acceptable AUC over the entire EVAL dataset, but under-performs on specific slices.
- In general, a model with good performance on average may exhibit failure modes that are not apparent by looking at an aggregate metric.
- Slicing metrics allow you to analyze the performance of a model on a more granular level.
- This functionality enables developers to identify slices where examples may be mislabeled or where the model over or under predicts.
- For example, TFMA could be used to analyze whether a model that predicts the generosity of a taxi tip works equally well for riders that take the taxi during the day versus at night, by slicing the data by the hour.
- In general, **determining which slices are important to analyze requires domain knowledge** about your data and application.



- TensorBoard reports metrics on a mini-batch basis during training.
- They're called **streaming metrics** and are our approximations based on those observed mini-batches.
- TFMA uses Apache Beam to do a **full pass** over the EVAL dataset.
- This not only allows for accurate calculation of metrics, but also scales up to massive evaluation dataset since **Beam pipelines** can be run using distributed processing back ends.
- Note that TFMA may compute the same TensorFlow metrics that are computed by the TensorFlow EVAL worker, just more accurately by doing a full pass over the specified dataset.
- TFMA can also be configured to compute additional metrics that were not identified or not defined in the model.
- Furthermore, if evaluation datasets are sliced to compute metrics for specific segments, each of those segments may only contain a small number of examples.
- To accurately compute metrics, a deterministic full pass over those examples is important.

TFMA in Practice



- Let's take a quick look at how to set up and use TFMA to perform model analysis.
- Over the next set of slides, you'll be learning how TFMA helps you evaluate your model performance in different ways.
- You'll learn how TFMA evaluates your model against different slices of data, and how TFMA keeps track of metrics over time.
- Before getting started, note that this example uses TFMA by itself, outside of a TFX pipeline.
- When you use TFMA in a TFX pipeline, the evaluator component already includes many of the steps performed in this example, so you won't have to do them yourself.
- Instead, you would usually only provide the eval_config to the evaluator component, which we'll see later.

Step 1: Export EvalSavedModel for TFMA

```
import tensorflow as tf
import tensorflow_transform as tft
import tensorflow_model_analysis as tfma

def get_serve_tf_examples_fn(model, tf_transform_output):
  # Return a function that parses a serialized tf.Example and applies TFT

  tf_transform_output = tft.TFTransformOutput(transform_output_dir)
signatures = {
  'serving_default': get_serve_tf_examples_fn(model, tf_transform_output)
                       .get_concrete_function(tf.TensorSpec(...)),
}

model.save(serving_model_dir_path, save_format='tf', signatures=signatures)
```

- As usual, let's begin by importing all the necessary libraries.
- In this case, we need TensorFlow, TensorFlow Transform, and TensorFlow Model Analysis.
- Next, to use TFMA, the first step is to train your model and generate a saved model object, which you will normally do anyway so that you can serve your model.
- The raw input examples need to be pre-processed in the same way during training while performing evaluation using TFMA.
- For example, if a feature was normalized before training, it will also need to be normalized before evaluation.
- If you train your model in a TFX pipeline and include a transform component, then the same pre-processing and feature engineering done during training will already be included in your saved model.
- However, if you did not train in TFX, you will need to apply the same pre-processing manually, which is done here in the *get_serve_tf_examples* function, in this case, using TensorFlow Transform.
- Next, we generate a saved model. In most cases, you'll want to use the serving default signature for your saved model, but you can also explicitly specify a different signature for the model in the model specification configuration.

Step 2: Create EvalConfig

```
# Specify slicing spec
slice_spec = [slicer.SingleSliceSpec(columns=['column_name']), ...]
# Define metrics
metrics = [tf.keras.metrics.Accuracy(name='accuracy'),
           tfma.metrics.MeanPrediction(name='mean_prediction'), ...]
metrics_specs = tfma.metrics.specs_from_metrics(metrics)

eval_config = tfma.EvalConfig(
                    model_specs=[tfma.ModelSpec(label_key=features.LABEL_KEY)],
                    slicing_specs=slice_spec,
                    metrics_specs=metrics_specs, ...)
```

- The next step involves creating an *eval_config* object that encapsulates the requirements for TFMA.

- More concretely, first, you have to **define the slices** of your dataset and the metrics that you want to use for analyzing your model.
- Then you wrap the model, slices, metrics, and other configurations in an eval_config object.
- This will determine what TFMA does when it's run. This is the same eval_config that you would give to the evaluator component in a TFX pipeline.

Step 3: Analyze model

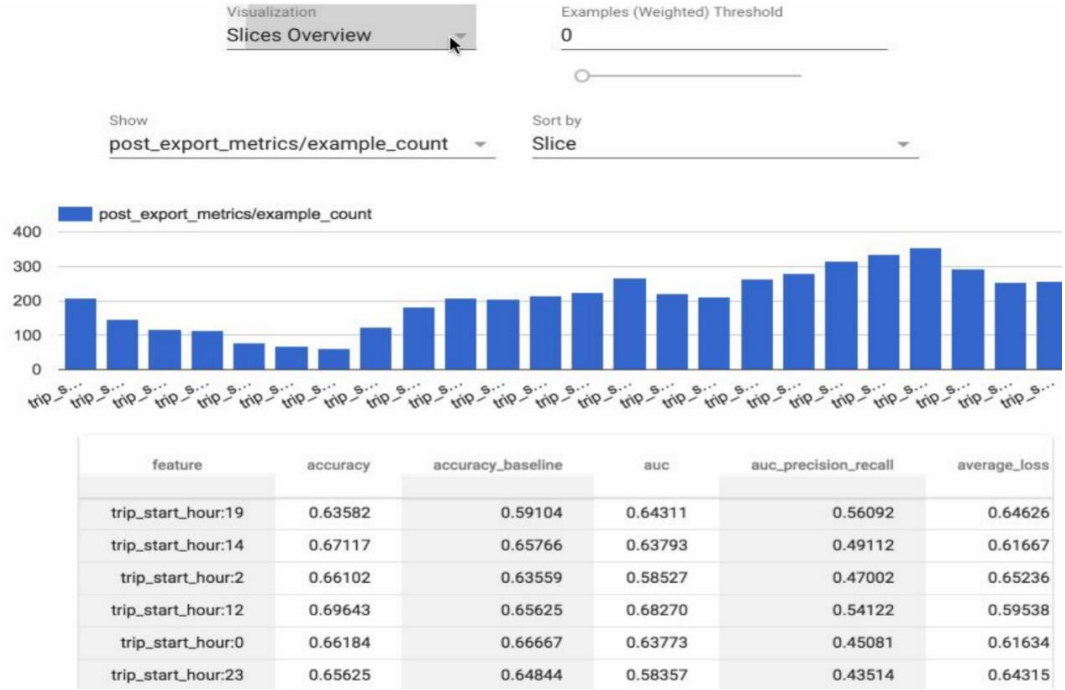
```
# Specify the path to the eval graph and to where the result should be written
eval_model_dir = ...
result_path = ...

eval_shared_model = tfma.default_eval_shared_model(
                            eval_saved_model_path=eval_model_dir,
                            eval_config=eval_config)
# Run TensorFlow Model Analysis
eval_result = tfma.run_model_analysis(eval_shared_model=eval_shared_model,
                                      output_path=result_path,
                                      ...)
```

- With the eval_config object set, you're now ready to start analyzing your model.
- These examples use only one machine to perform the analysis, so using an orchestrator is not necessary.
- Here, the code sets the path to an evaluation graph and a directory to store the analysis.
- The next step is to call the TFMA run_model_analysis API. This will get your model analysis started.
- This API can accept slicing specifications directly as well.

Step 4: Visualizing metrics

```
# render results
tfma.viewer.render_slicing_metrics(result)
```

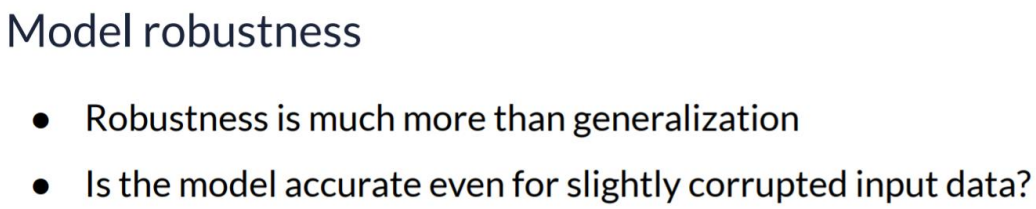
- Finally, let's render the results and visualize the model analysis metrics. TFMA provides interactive widgets that can be run in notebooks, and this is a code that you would include in a notebook cell to generate the TFMA visualization widget.



feature	accuracy	accuracy_baseline	auc	auc_precision_recall	average_loss
trip_start_hour:19	0.63582	0.59104	0.64311	0.56092	0.64626
trip_start_hour:14	0.67117	0.65766	0.63793	0.49112	0.61667
trip_start_hour:2	0.66102	0.63559	0.58527	0.47002	0.65236
trip_start_hour:12	0.69643	0.65625	0.68270	0.54122	0.59538
trip_start_hour:0	0.66184	0.66667	0.63773	0.45081	0.61634
trip_start_hour:23	0.65625	0.64844	0.58357	0.43514	0.64315

- Here's what the visualizations look like.
- This tool enables you to really dig into your model performance and understand how it varies on different slices of data.

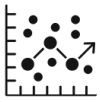
Model Debugging Overview



- Now let's discuss model performance beyond simple metrics and introduce some ways to analyze it and improve it.
- To talk about model debugging, let me first focus on model robustness.
- Checking the robustness of the model is a step **beyond** the simple measurement of model performance or generalization.
- **A model is considered to be robust if its results are consistently accurate, even if one or more of the features change fairly drastically**.
- Of course there are limits to robustness and all models are sensitive to changes in the data.
- But there is a clear difference between a model that changes in gradual, predictable ways as the data changes and a model that suddenly produces wildly different results.

Robustness metrics

Robustness measurement shouldn't take place during training



Split data in to train/val/dev sets

Specific metrics for regression and classification problems

- So how do you measure the robustness of a model?
- The first and foremost thing is you shouldn't be measuring the robustness of a model during its training.
- Also you shouldn't be using the same data set that you used during training.
- So before you start the training process, you should split the data set into train, validation, and test splits.
- You may **use the test split** which is totally unseen by the model even during the validation stage for testing the model robustness.
- Or otherwise, the best choices to generate a variety of new types of data, and we'll discuss some of the methods to generate this later on.
- The **metrics** themselves will be the same types that you use for training depending on the model type. So things like RMSE for regression models and AUC for classification.

Model debugging

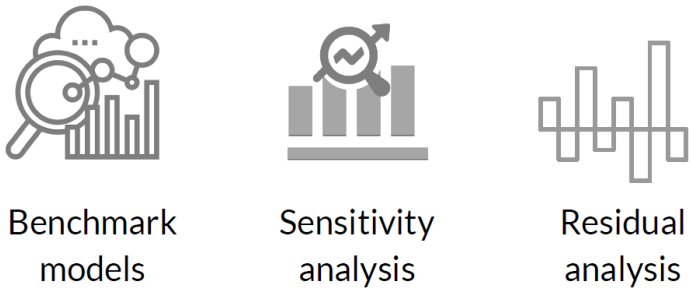
- Deals with detecting and dealing with problems in ML systems
- Applies mainstream software engineering practices to ML models

- Let's take a look at ways to increase model robustness.
- Model debugging is an emerging discipline focusing on finding and fixing problems in models and improving model robustness.
- Model debugging borrows various practices from model risk management, traditional model diagnostics, and software testing.
- Model debugging attempts to test ML models just like code, very similar to how you test in software development.
- Model debugging tries to improve the transparency of models by highlighting how data is flowing inside and thus can prevent harmful social discrimination.
- It probes sophisticated ML response functions and decision boundaries to detect and correct accuracy, fairness, security, and other problems in ML systems.

Model Debugging Objectives

- Model debugging has several objectives. One of the big problems with ML models is that they can be quite opaque and become black boxes.
- Model debugging tries to improve the transparency of models by highlighting how data is flowing inside.
- Another problem is social discrimination. Does your model work poorly for certain groups of people?
- Model debugging also aims to reduce the vulnerability of your model to attacks. For example, certain requests can be made once the model is in production that may be aimed at extracting data out of your model in order to understand how your model has been built.
- This is especially a problem when data has private information and it's been used for training. So was the data anonymized before it was used for training?
- Lastly, with time, your model performance decays as the distribution of incoming data changes.

Model Debugging Techniques



- Three of the most widely used debugging techniques are benchmarking models, sensitivity analysis, and residual analysis. We'll discuss each of these individually.

Benchmark Models

Benchmark models

Simple, trusted and interpretable models solving the same problem

Compare your ML model against these models

Benchmark model is the starting point of ML development

- We'll start with a brief discussion of using benchmarking models, which is always a good model debugging technique.
- A good place to start is by comparing your model to benchmark models, which is a simple and powerful way to check your model's performance.
- Benchmarking models are small, simple models that are used before you start development for baselining your problem.
- They're generally not state-of-the-art, but instead are linear or other simple models with very consistent performance.
- You compare your model to see if it actually is performing better than the simpler benchmarking model as a sanity test.
- If it doesn't, it could be that your model has a problem, or that a simple model accurately models the data and is really all that you need for your application.
- Once a model passes the benchmark test, the benchmark model can serve as a solid debugging tool.
- Even if you're performing better than the benchmark model, you can still evaluate which test samples your model is failing where the benchmark model predicts correctly.
- Then you need to study your model to find out why that's happening.

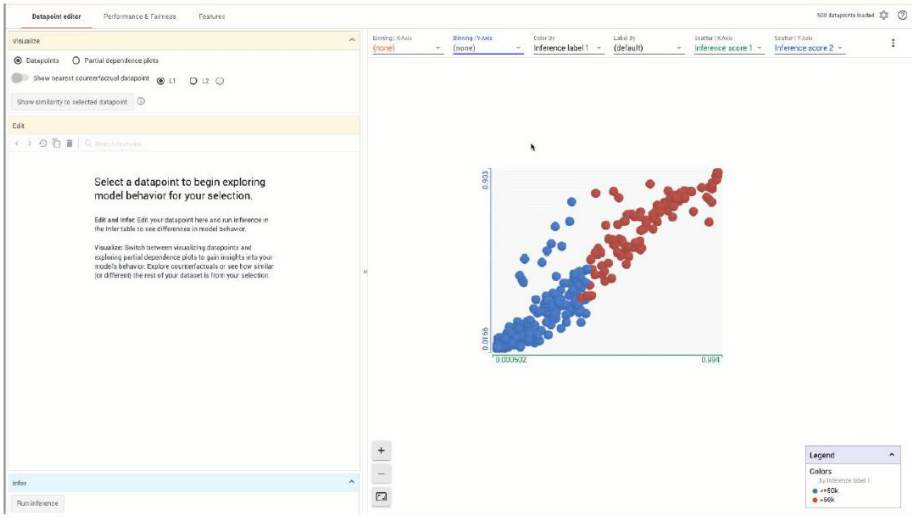
Sensitivity Analysis and Adversarial Attacks

Sensitivity analysis

- Simulate data of your choice and see what your model predicts
- See how model reacts to data which has never been used before

- Sensitivity analysis is an important way to evaluate your model's performance, including its vulnerability to adversarial attacks. Let's see what this is all about.
- Sensitivity analysis helps you understand your model by examining the impact that each **feature** has on your model's prediction.
- In sensitivity analysis you experiment by changing a single features value while holding the other features constant and observe the model results.
- If changing the features value causes the models result to be drastically different, it means that this feature has a big impact on the prediction.
- Usually you're changing the values of the features synthetically according to some distribution or process and ignoring the labels for the data.
- You're really not looking to see if the prediction is correct or not, but instead how much it changes
- Different ways of doing sensitivity analysis use different techniques for changing the feature value.
- Let's explore a few different approaches.

What-If Tool for sensitivity analysis



- One of the more powerful ways of conducting sensitivity analysis is by using the **What-If** tool, which was created by the TensorFlow team.
- The What-If tool allows you to visualize the results of sensitivity analysis to understand and debug your performance.

Random Attacks

- Expose models to high volumes of random input data
- Exploits the unexpected software and math bugs
- Great way to start debugging

- Let's discuss some of the most common approaches for doing sensitivity analysis.
- In random attacks you generate lots of random input data and test the models outputs.
- Random attacks can reveal all kinds of unexpected software and math bugs.
- **If you don't know where to begin debugging an ML system, a random attack is a great place to get started.**

Partial dependence plots

- Visualize the effects of changing one or more variables in your model
- PDPbox and PyCEbox open source packages

- Another approach is to use a partial dependence plot.
- Partial dependence plots show the marginal effect of one or two features and the effect they have on the model results.
- A partial dependence plot can show whether the relationship between the label and a particular feature is linear, monotonic, or more complex.

- For example, when applied to a linear regression model, partial dependence plots always show a linear relationship.
- PDPbox and PyCEbox are open source packages which are available for creating partial dependence plots.

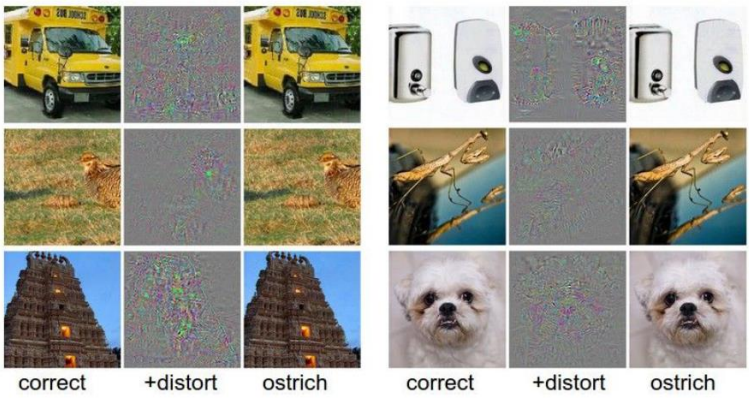
How vulnerable to attacks is your model?

Sensitivity can mean vulnerability

- Attacks are aimed at fooling your model
- Successful attacks could be catastrophic
- Test adversarial examples
- Harden your model

- Now that we've introduced two sensitivity analysis techniques, it's worth asking how vulnerable your model is to attacks.
- Several machine learning problems, including neural networks can be fooled into misclassifying adversarial examples which are formed by making small but carefully designed changes to the data so that the model returns an incorrect answer with high confidence.
- This could have daunting implications. Imagine making a wrong decision on an important question based on only slightly corrupted data.
- Depending on how catastrophic and incorrect result could be for your application, you may need to test your model for vulnerabilities and based on your analysis, harden your model to make it more resilient to attacks.

A Famous Example: Ostrich



- So what do these attacks look like? Well, here's a famous example with two groups of images applying only a tiny distortion to the images in the left columns, results in the images in the right columns, which a model train on image that classifies as an ostrich.

How vulnerable to attacks is your model?

Example:

A self-driving car crashes because black and white stickers applied to a stop sign cause a classifier to interpret it as a Speed Limit 45 sign.

- So how serious of a problem is it? Well, it depends on your application, but let's discuss a few examples
- First, with an autonomous vehicle, it's important to recognize traffic signs, other vehicles, people, etc.
- Unfortunately, we've seen examples of this in real life, but as in this example, if a sign is altered in just the right way, it can fool the model and the results can be catastrophic.

How vulnerable to attacks is your model?

Example:

A spam detector fails to classify an email as spam. The spam mail has been designed to look like a normal email, but is actually phishing.

- Another example is just application quality.
- If your business sells software to detect spam and phishing emails can get through, it reflects badly on your product.

How vulnerable to attacks is your model?

Example:

A machine-learning powered scanner scans suitcases for weapons at an airport. A knife was developed to avoid detection by making the system think it is an umbrella.

- Finally, a somewhat scarier example. As you rely on machine learning for more and more mission critical applications, you need to consider the security implications.
- A suitcase scanner is basically just an object classifier, but if it's vulnerable to attack, the results can be dangerous.

Informational and Behavioral Harms

- Informational Harm: Leakage of information
- Behavioral Harm: Manipulating the behavior of the model

- The 'future of privacy' forum, an industry group that studies privacy and security, suggest that security and privacy harms enabled by machine learning fall into roughly two categories: Informational and behavioral.
- Informational harms relate to the unintended or unanticipated leakage of information
- Behavioral harms, on the other hand, relate to manipulating the behavior of the model itself, impacting the predictions or outcomes of the model.

Informational Harms

- Membership Inference: was this person's data used for training?
- Model Inversion: recreate the training data
- Model Extraction: recreate the model

- Let's look at informational harm
- Membership inference attacks are aimed at inferring whether or not an individual's data was used to train the model based on a sample of the model's output.
- While seemingly complex, studies have shown that these attacks require much less sophistication than is frequently assumed.
- Model inversion attacks use model outputs to recreate the training data.
- In one well known example, researchers were able to reconstruct an image of an individual's face.
- Another study focused on ML systems that use genetic information to recommend dosing of specific medications, and it was able to directly predict individual patient's genetic markers.
- Model extraction attacks use model outputs to recreate the model itself.
- This has been demonstrated against model as a service providers like big ML and Amazon machine learning, and can compromise privacy and security, as well as the intellectual property of the underlying model itself.

Behavioral Harms

- Poisoning: insert malicious data into training data
- Evasion: input data that causes the model to intentionally misclassify that data

- And what about behavioral harms?
- **Model poisoning** attacks occur when an adversary inserts malicious data into training data in order to alter the behavior of the model
- For example, creating an **artificially low insurance premium** for particular individuals.
- Evasion attacks occur when data in an inference request intentionally causes the model to misclassify that data.
- These attacks occur in a range of scenarios and the changes in the data may not be noticeable by humans.
- Our earlier example of an altered stop sign is one example of an evasion attack.

Measuring your vulnerability to attack

Cleverhans:
an open-source Python library to benchmark machine learning systems' vulnerability to adversarial examples

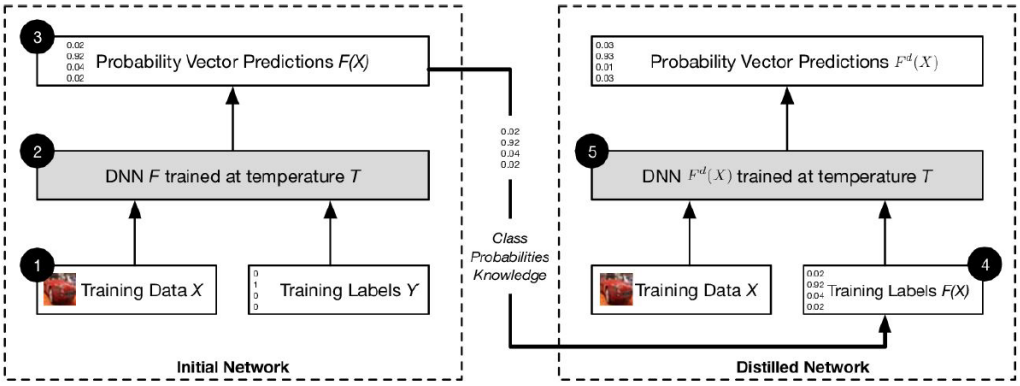
Foolbox:
an open-source Python library that lets you easily run adversarial attacks against machine learning models

- Before hardening your models, you need to have some way of measuring the vulnerability to attack
- Cleverhans is an open source python library that you can use to benchmark your models to measure their vulnerability to adversarial examples.
- To harden your model to adversarial attacks, one approach is **to include sets of adversarial images** in your training data so that the classifier is able to understand the various distributions of noise, and your model learns to recognize the correct class.
- This is known as adversarial training, examples created by tools such as CleverHans can be added to your data set, but doing so limits your ability to use them to measure your models vulnerability, since you are now almost testing with your training data
- Foolbox is another open source python library that lets you easily run adversarial attacks against machine learning models like deep neural networks.
- It's built on top of EagerPy and works natively with models in PyTorch, Tensorflow and Jax.

Adversarial example searches

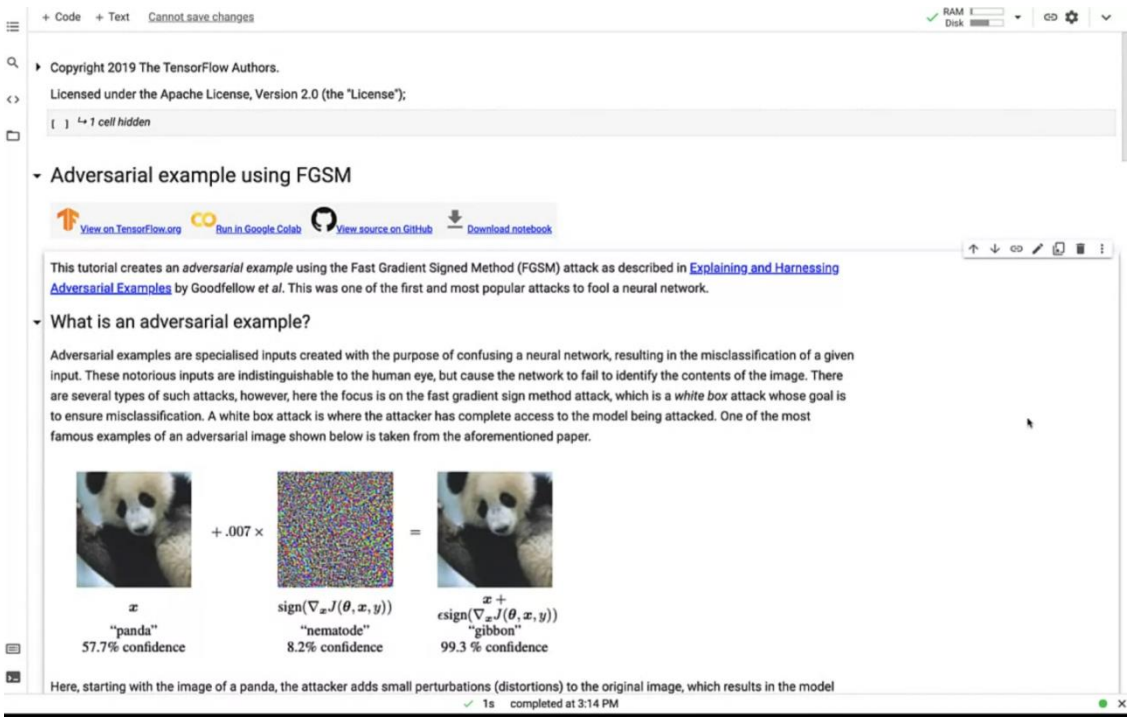
Attempted defenses against adversarial examples

- Defensive distillation



- Unfortunately, **detecting vulnerability is easier than fixing it**.
- This is an emerging field and like many things in security, there is an arms race between attackers and defenders.
- One fairly advanced approach is **defensive distillation**.
- Since it does not use specific adversarial examples, it may provide more general hardening to new attacks.
- As the name suggests, this is **very similar to knowledge distillation training**.
- The goal is to increase model robustness and decrease sensitivity in order to decrease vulnerability to attacks.
- Defensive distillation reduced the effectiveness of a sample creation from 95% to less than 0.5% in one study.
- The main difference between defensive distillation and the original distillation proposed by Hinton and others is that we keep the **same network architecture** to train **both** the original network as well as the distilled network.
- In other words, instead of transferring knowledge between different architectures, the authors proposed using knowledge distillation to improve a model's own resilience to attacks.

Adversarial Attack Demo



This section is based on the notebook here: https://www.coursera.org/learn/machine-learning-modeling-pipelines-in-production/supplement/adTGj/sensitivity-analysis-and-adversarial-attacks#:~:text=Google%20Colab%20notebook

- Let's take a look at an attack on a model using an adversarial example, which uses the fast gradient sign method, which was described by Goodfellow and others in a kind of a landmark paper.
- So what is an adversarial example? Adversarial examples are specialized inputs created with the purpose of confusing and neural network, resulting in the misclassification of a given input.
- These notorious inputs are indistinguishable to the human eye, but cause the model to fail to identify the contents of the image.
- There are several types of such attacks. However, here the focus is on the **fast gradient signed method** attack, which is a white box attack whose goal is to ensure misclassification.
- A white box attack is where the attacker has complete access to the model being attacked.
- One of the most famous examples of an adversarial image is shown here, and it's taken from that paper.
- We have a panda on the left that the model predicts with fairly high confidence, some noise is added, which is fairly opaque to a human.
- And the result is that the model now thinks that that panda is a gibbon with a very high confidence.
- What we will do is just walk through this example and show really how easy this kind of attack is.
- So we're running into CoLab here, we'll start by doing some imports of Tensorflow and matplotlib, and then we'll load a pre-trained MobileNetV2 and the ImageNet class names.
- And then, we have this helper function here to do pre-processing, and then, we'll take a look at the original image, which in this case is a Labrador Retriever.
- We'll create an adversarial image using the fast gradient signed method, and we can visualize the perturbations that that method makes to give you an idea.
- So looking at that, most people wouldn't expect it to have really any change in the model based on applying that, but as we'll see it does.

- So one of the things we do is we're going to try different values for **epsilon**, which is one of the parameters of the method.
- And we'll see how different values of epsilon have different predictions that the model makes.
- We have a helper method there to display the image, and then here's some different values of epsilon and the resulting model predictions.
- So there's our original Labrador retriever confidences.
- With the input and then with epsilon 0.01, the model now thinks it's a Saluki which is a thin dog.
- And then with a little higher value of epsilon, it thinks it's a Weimaraner

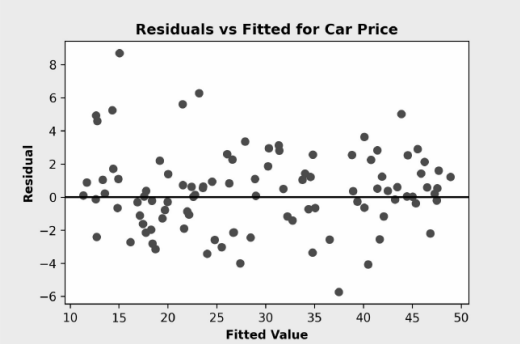
Residual Analysis

Residual analysis

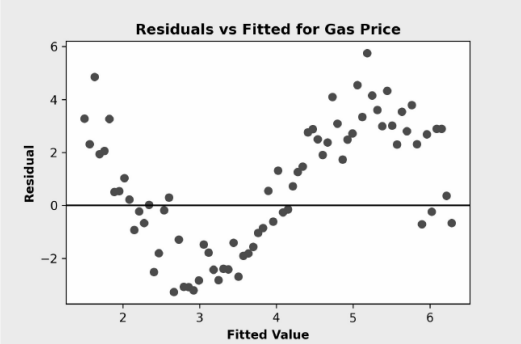
- Measures the difference between model's predictions and ground truth
- Randomly distributed errors are good
- Correlated or systematic errors show that a model can be improved

- Now let's discuss residual analysis, which is the last model debugging technique that will cover in this course.
- Another debugging technique is Residual Analysis.
- In most cases this can be used for regression models, since it requires measuring the distance between predictions and the ground truth.
- However, it requires having **ground truth** values for comparison, which can be **difficult** in many online or real time scenarios.
- In general, you want the residuals to follow a random distribution.
- If you find correlation between residuals, it's usually a sign that your model can be improved.

Residual analysis



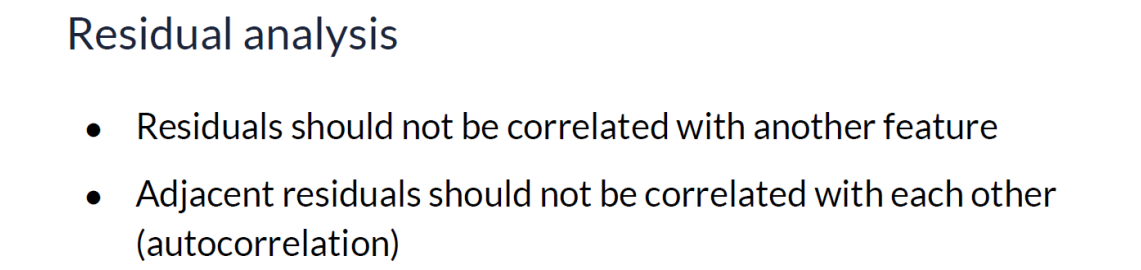
Random = Good



Systematic = Bad

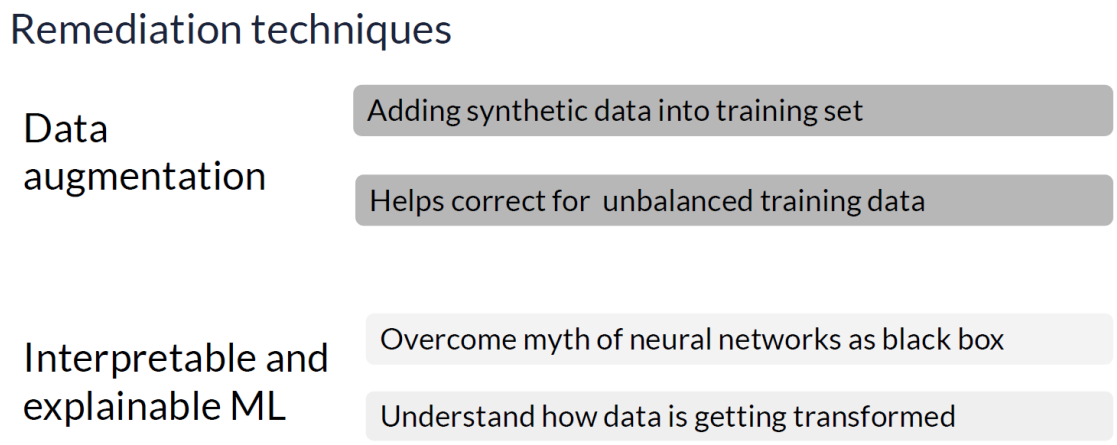
- If you've trained your aggression model, then you're probably familiar with metrics like root mean square error or RMSE, and similar metrics
- Residual analysis uses those kinds of metrics but it primarily looks at the distribution of the errors, often using residual plots to visually inspect the distribution of residuals.

- If your model is well trained and has captured the predictive information in the data, then the residuals should be **randomly distributed**.
- However, **if you see systematic or correlated residuals then it tells you that there is predictive information that the model has not captured**. You can then look for ways to improve your model.



- So what should you aim for when performing Residual Analysis?
- The residuals should not be correlated with another feature that was available but was left out of the feature vector.
- If you can predict the residuals with another feature, that feature should be included in the feature vector.
- This requires checking the unused features for correlation with the residuals.
- Also, adjacent residuals should not be correlated with each other, in other words, they should not be autocorrelated.
- If you can use one residual to predict the next residual, then there's some predictive information that is not being captured by the model.
- Often, but not always, you can see this visually in a residuals plot
- Ordering can be important for understanding this.
- For example, if a residual is more likely to be followed by another residual that has the same sign, adjacent residuals are positively correlated.
- Performing a Durbin Watson test is also useful for detecting autocorrelation.

Model Remediation



- So far we've discussed ways to analyze model robustness but we haven't talked about ways to **improve** it.
- Let's do that now with a discussion of model remediation.
- So what can you do to **improve robustness**?

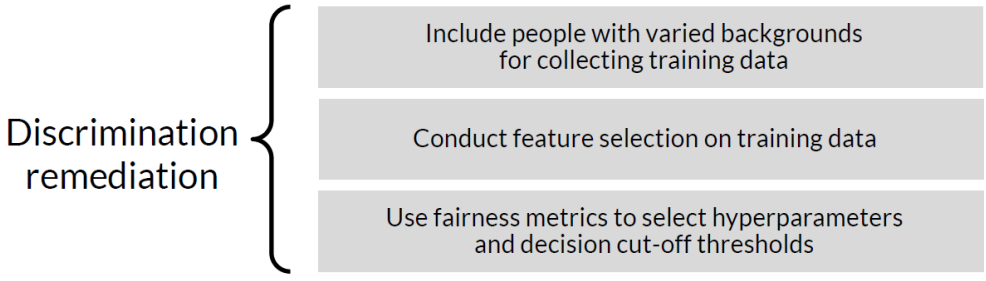
- You should make sure that your training data accurately mirrors a request that you'll receive for your trained model.
- But data augmentation can also help your model generalize, which typically reduces sensitivity.
- You can **generate data in many ways**, including generative techniques, interpretive techniques or simply adding noise to your data.
- Data augmentation is also a great way to help correct for unbalanced data.
- Understanding the inner workings of your model can also be important.
- Often more complex models can be black boxes and we sometimes don't make much effort to understand what is happening internally.
- However, there are tools and techniques which can help with model interpretability and this can also help with improving model robustness.
- There are also model architectures which are more easily interpreted, including **tree-based models** as well as neural network models which are specifically designed for interpretability.

Remediation techniques

- Model editing:
  - Applies to decision trees
  - Manual tweaks to adapt your use case
- Model assertions:
  - Implement business rules that override model predictions

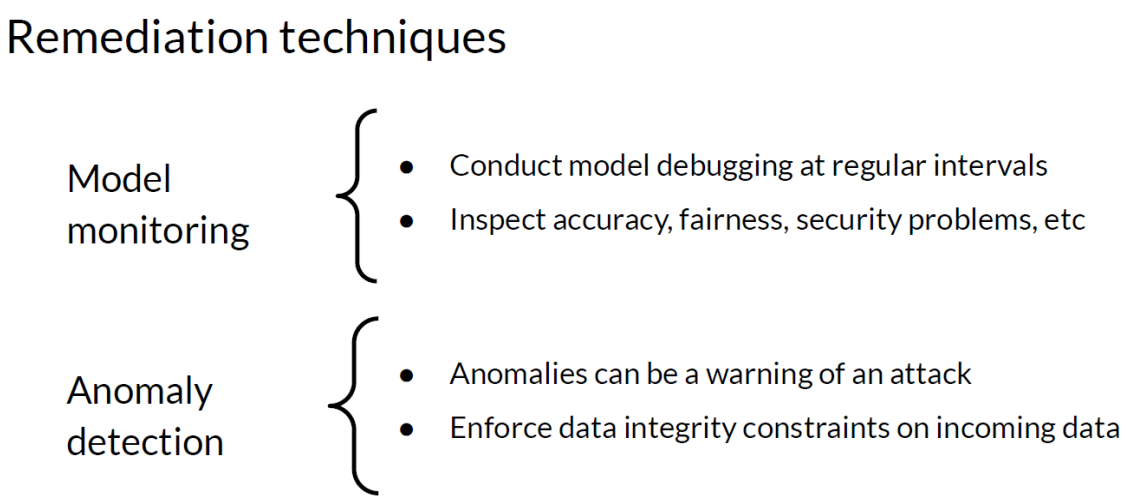
- Another remediation technique is model editing.
- Some models, such as decision trees, are so directly interpretable that the learned parameters can be understood easily.
- If you find that something is going wrong, you can tweak the model to improve its performance and robustness.
- Another technique is model assertions.
- Model assertions **apply business rules** or simple sanity checking to your models results and either alter or bypass the results before delivering them.
- For example, if predicting someone's age numbers should never be negative, or if predicting a credit limit, then it should never be more than a maximum amount.

Remediation techniques



- Next let's look at how model bias can be reduced or eliminated. This is **discrimination remediation**.

- The best solution for this is to have a diverse data set which is representative of the people who will be using your model.
- It also helps to have people on the development team from **diverse backgrounds** with expertise in ethics, privacy, social sciences and other related disciplines.
- Careful feature selection, including sampling and reweighting rows to minimize discrimination in training data, can also be helpful
- When training, you should consider fairness metrics when selecting hyperparameters and decision cut-off thresholds.
- This requires using a tool like **fairness indicators** to measure fairness.
- This may also involve training fair models directly by learning fair representations (LFR) and adversarial de-biasing using tools such as **AIF360**, or using dual objective functions that consider both accuracy and fairness metrics
- For prediction post-processing, changing model predictions after training, **reject option** classification tools such as AIF 360 or Themis ML can also help to reduce unwanted bias
- Tools like Google model remediation library can help with improving model fairness as well



- Model debugging is not something that you only do during development.
- This requires **constant monitoring** of your model.
- The accuracy, fairness, or security characteristics of MM models will change during the lifetime of a model as the data and the world changes, and as new attacks emerge.
- You should build monitoring into your process so that your models are checked for **accuracy, fairness, and security** problems on a regular basis.
- Strange anomalous input and prediction values are always worrisome in ML, and can be indicative of an attack,
- Luckily anomalous inputs and predictions can be caught and corrected in real time using a variety of tools and techniques, including **data integrity constraints on input streams, statistical process control methodologies on inputs and predictions, anomaly detection through auto encoders and isolation forest, and also by comparing your model's prediction to benchmark model predictions**

Fairness indicators

- Open source library to compute fairness metrics
- Easily scales across dataset of any size
- Built in top of TFMA

- In this section, we'll focus on how to make models fair and look at using fairness indicator libraries to assess fairness.
- Remember that in addition to serving your community well, focusing on fairness helps you serve different types of customers or situations well.
- In addition to analyzing and improving your model performance, you want to introduce **checks and controls** to ensure that your **model behaves fairly in different scenarios**.
- Accounting for fairness and reducing bias towards any group of people is an important part of that.
- You need to make sure that your model is not causing harm to the people who use it.
- Fairness Indicators is an open-source library built by the TensorFlow team to easily compute commonly identified fairness metrics for binary and multiclass classifiers.
- Fairness indicators scales well, and has been built on top of the TensorFlow Model Analysis framework.

What does fairness indicators do?

- Compute commonly-identified fairness metrics for classification models
- Compare model performance across subgroups to other models
- No remediation tools provided

- What are the main features of the fairness indicators library?
- Fairness indicators is the library that enables easy measurement of commonly identified fairness metrics for binary and multiclass classifiers.
- With the fairness indicators tool suite, you can also compare model performance across subgroups to a baseline or to other models.
- This includes using confidence intervals to surface statistically significant disparities and performing evaluation over multiple thresholds.
- Fairness indicators is primarily a tool for measuring fairness, **not** for doing remediation to improve fairness.

Evaluate at individual slices

- Overall metrics can hide poor performance for certain parts of data
- Some metrics may fare well over others

- Looking at slices of data is actually quite informative when you're trying to mitigate bias and check for fairness.
- When measuring fairness, it's important to identify slices of data which are sensitive to fairness and measure your model's performance on those slices.
- **If you only measure fairness using the entire dataset, it can easily hide fairness problems** with particular groups of people.
- It's also important to consider and select the right metrics to measure for your dataset and users, because otherwise, you may measure the wrong thing and be unaware of problems.
- This is often based on **domain knowledge**.

Aspects to consider

- Establish context and different user types
- Seek domain experts help
- Use data slicing widely and wisely

- Keep in mind that measuring fairness is only one part of evaluating a broader user experience.
- Start by thinking about the different contexts through which a user may experience your application.
- **Who** are the different types of users for your application? **Who** else may be affected by the experience?
- It's important to remember that human societies are extremely complex.
- Understanding people and their social identities, social structures, and cultural systems are each huge fields of open research.
- Whenever possible, we recommend talking to appropriate domain experts, which may include social scientists, sociolinguists, or cultural anthropologists as well as members of the communities that will be using your application.
- A good rule of thumb is to slice for **as many groups of data as possible**.
- Pay special attention to slices of data that deal with **sensitive characteristics such as race, ethnicity, gender, nationality, incomes, sexual orientation, and disability status**.
- Ideally, you should be working with labeled data, but if not, then you can apply statistics to look at the distributions of the outcomes with some assumptions around any expected differences.

General guidelines

- Compute performance metrics at all slices of data
- Evaluate your metrics across multiple thresholds
- If decision margin is small, report in more detail

- Let's look at some important guidelines to avoid common pitfalls when working with fairness indicators.
- When you're just getting started with fairness indicators, you should conduct various fairness tests on **all** the available slices of data.
- Next, you should evaluate the fairness metrics across **multiple thresholds** to understand how the threshold can affect the performance of different groups.
- Finally, for predictions which don't have a good margin of separation from their decision boundaries, you should consider reporting the rate at which the label is predicted.

Measuring Fairness

Positive rate / Negative rate

- Percentage data points classified as positive/negative
- Independent of ground truth
- Use case: having equal final percentages of groups is important

- Now let's look at the various fairness metrics which are available in fairness indicators.
- This includes the positive-negative rate, the ratios of true and false positives, accuracy, and area under the curve. We will start with a quick overview of these metrics.
- Let's first consider the basic positive and negative rates.
- The basic positive and negative rates show the percentage of data points that are classified as positive or negative. These are independent of ground truth labels.
- These metrics help with understanding demographic parity, the equality of outcomes, which should be equal across subgroups.
- This applies to use cases where having equal percentages of outcomes for different groups is important.
- Let's talk about the true positive rate, also known as TPR, and the false negative rate, also referred to as FNR.

True positive rate (TPR) / False negative rate (FNR)

- TPR: percentage of positive data points that are correctly labeled positive
- FNR: percentage of positive data points that are incorrectly labeled negative
- Measures equality of opportunity, when the **positive class** should be equal across subgroups
- Use case: where it is important that same percent of qualified candidates are rated positive in each group

- The true positive rate, measures the percentage of positive data points as labeled in the ground truth that are correctly predicted positive.
- Similarly, the false negative rate measures the percentage of positive data points that are incorrectly predicted negative.
- This **metric relates to the equality of opportunity for the positive class when it should be equal across subgroups.**
- This often applies to use cases where it's important that the same percentage of qualified candidates are rated positively in each group, such as for **loan applications or school admissions**.

True negative rate (TNR) / False positive rate (FPR)

- TNR: percentage of negative data points that are correctly labeled negative
- FPR: percentage of negative data points that are incorrectly labeled positive
- Measures equality of opportunity, when the **negative class** should be equal across subgroup
- Use case: where misclassifying something as positive are more concerning than classifying the positives

- Now, let's talk about the true negative rate, also known as TNR, and the false positive rate, also referred to as FPR.
- The true negative rate measures the percentage of negative data points has labeled in the ground truths that are correctly predicted negative.
- Similarly, the false positive rate is the percentage of negative data points that are incorrectly predicted positive.
- This metric relates to equality of opportunity when the negative class should be equal across subgroups.
- This applies to use cases where **misclassifying something is positive is more concerning than classifying the positives.**
- This is most common in abuse cases where positives often lead to negative actions.
- These are also important for facial analysis techniques such as **face detection** or face attributes.

Accuracy & Area under the curve (AUC)

- Accuracy: The percentage of data points that are correctly labeled
- AUC: The percentage of data points that are correctly labeled when each class is given equal weight independent of number of samples
- Metrics related to predictive parity
- Use case: when precision is critical

- The last set of fairness metrics will go through our accuracy and area under the curve, also known as AUC.
- Accuracy is the percentage of data points that are correctly labeled
- **AUC is the percentage of data points that are correctly labeled when each class is given equal weight independent of the number of samples.**
- Both of these metrics relate to **predictive parity** when equal across subgroups.
- This applies to use cases where the **precision of the task is critical**, but not necessarily in a given direction, such as face identification or face clustering.

Tips

Unfair skews if there is a gap in a metric between two groups

Good fairness indicators doesn't always mean the model is fair

Continuous evaluation throughout development and deployment

Conduct adversarial testing for rare, malicious examples

- Here's some thoughts to consider. A significant difference in a metric between two groups can be a sign that your model may have fairness issues. You should interpret your results according to your use case.
- Next, achieving equality across groups with fairness indicators doesn't guarantee that your model is fair.
- Systems are highly complex and achieving equality on one or even all of the provided metrics can't guarantee fairness.
- Fairness evaluation should be run throughout the development process and post-launch as well.
- Just like improving your product is an ongoing process and subject to adjustment based on user and market feedback, making your product fair and equitable requires ongoing attention.
- As different aspects of the model change, such as training data, inputs from other models or the design itself, fairness metrics are likely to change.
- Lastly, adversarial testing should be performed for rare and malicious examples.

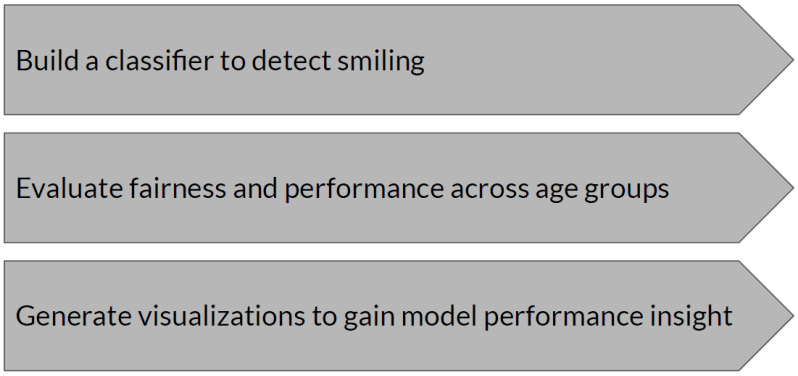
- Fairness evaluations aren't meant to replace adversarial testing, but to provide an **additional defense** against rare targeted examples.
- This is crucial as these examples probably will not be included in training or evaluation data.

About the CelebA dataset

- 200K celebrity images
- Each image has 40 attribute annotations
- Each image has 5 landmark locations
- Assumption on smiling attribute

- Now let's look at a concrete example of using Fairness Indicators.
- Let's consider the CelebA dataset which contains 200,000 images of celebrities.
- In addition to an image of a celebrity, each example has 40 attribute annotations.
- Some of these attributes are hair type, facial features, fashion accessories, etc.
- Every image also has five facial landmark locations, which mark the positions of the eyes, nose, and mouth.
- It's customary to assume that during the labeling of this dataset, the smiling attribute was determined through a pleasing, kind, or amusing type of expression on the subject's face.

Fairness indicators in practice



- So you'll practice using fairness indicators to explore the smile attribute in this dataset.
- First, you start by building a classification model to identify whether the person in the image is smiling or not.
- Once the model is ready, you evaluate model performance across **various age groups** using fairness indicators.
- Finally, you plot a few of the metrics to understand how well our model is performing.

Why do models need to be monitored?

- Training data is a snapshot of the world at a point in time
- Many types of data change over time, some quickly
- ML Models do not get better with age
- As model performance degrades, you want an early warning

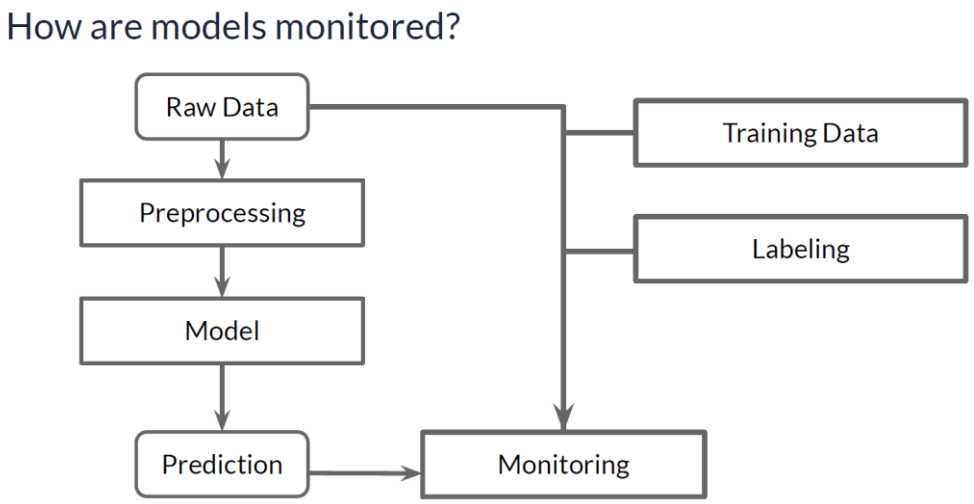
- We've mentioned a few times that keeping a healthy model running and operational requires continuous evaluation and monitoring. Let's take a quick look at these important topics.
- Let's look at some of the ways of monitoring your model once it has been deployed to production.
- When you train your model, you use the training data that's available at that time.
- The training data represents a snapshot of the world at the time that it was collected and labeled.
- But the world changes, and for many domains, the data changes too.
- Sometime later when your model is being used to generate predictions, it may or may not know enough about the current state of the world to make accurate predictions.
- For example, if your model to predict movie sales was trained on data collected in the 90s, it might predict that customers would buy VHS tapes.
- Is that still a good prediction today? My guess is no.
- When the model goes bad, your application and your customers will suffer.
- Before it becomes a fire drill to collect new training data and fix the model, you want an early warning that your model performance is changing.
- Continuously monitoring and evaluating your data and your model performance will give that early warning.

Data drift and shift

- Concept drift: loss of prediction quality
- Concept Emergence: new type of data distribution
- Types of dataset shift:
  - covariate shift
  - prior probability shift

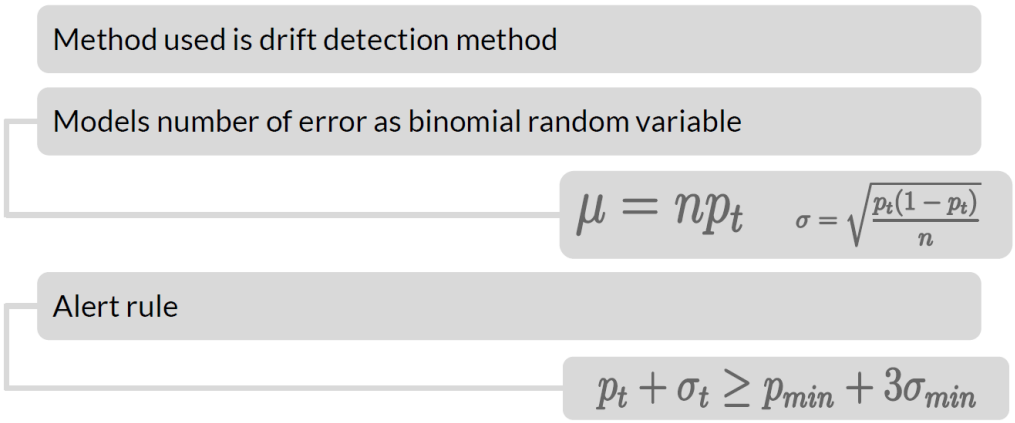
- Let's take a look at data drift and shift.
- The most **extreme** form of this is **concept** drift, which is a **change in the relationship between your input data and your labels.**
- Concept drift can happen even when the rest of your data doesn't change.

- For example, the exact same customer who bought red boots six months ago may instead want white sneakers today, if the fashions have changed.
- The input to your model might be exactly the same, depending on which features you capture, but the correct prediction has changed.
- There's also the idea of an **emerging concept (aka concept emergence)**. An emerging concept refers to new patterns in the data distribution that weren't previously present in your dataset.
- This can happen in several ways: labels may have become obsolete and new labels may need to be added, as in our VHS example earlier.
- Based on the type of distribution change, the **dataset shift** can be classified into two types.
- The first is covariate shift. In **covariate shift**, the **distribution of your input data** changes, but the conditional probability of your output over the input remains the same. The distribution of your labels doesn't change.
- **Prior probability shift is basically the opposite of covariate shift**.
- The distribution of your labels changes, but your input data stays the same.
- Concept drift can be thought of as a type of prior probability shift.



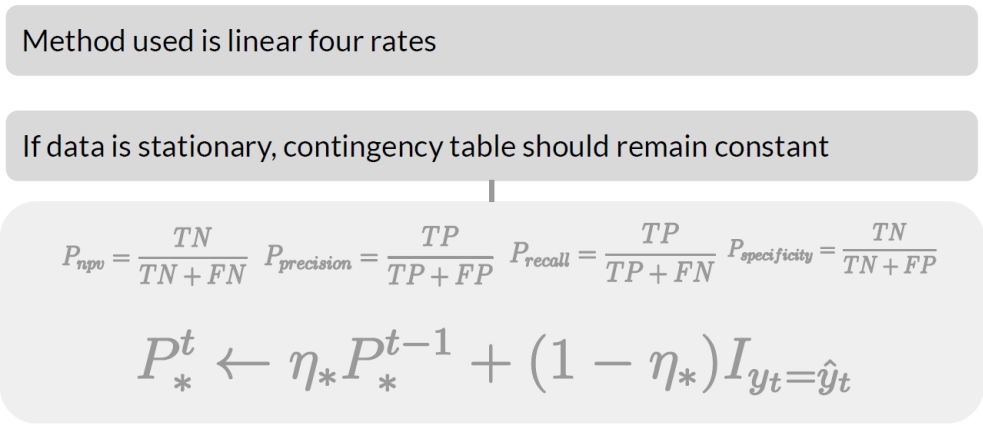
- Now let's think about how to monitor your data and model performance.
- First, think of your trained model in a production setting. Let's start with the raw data that you'll use to make a prediction.
- Before making a prediction, you probably need to do some preprocessing of the raw data, and then you use your model to generate a prediction given the input data.
- Now, let's add monitoring and evaluation.
- First, let's monitor the raw data being received in production and compare it to the data that you trained your model with.
- This lets you look for **changes in the input data over time or covariate shift**.
- Next, let's add monitoring for the predictions which you're generating.
- This enables you to detect **prior probability shift**.
- Finally, let's add a labeling process which enables the detection of concept drift.
- Note, that in many cases, there will be a significant delay in generating labels for a sample of the new input data.
- Now that you've added monitoring, how do you look for drift and shift?
- There are both supervised and unsupervised methods, based on whether or not you have labels for the incoming stream of data and the kind of drift or shift that you're looking for. Let's discuss each of these next.

Statistical process control



- Our first **supervised** technique is **statistical process control**.
- Statistical process control has been primarily used in manufacturing for quality control since the 1920s.
- It uses statistical methods to monitor and control process, which in the case of your deployed model is the incoming stream of raw data for prediction requests. This is useful to detect **drift**.
- It assumes that the stream of data will be stationary, which it may or may not be depending on your application, and that the errors follow a **binomial** distribution.
- It analyzes the rate of errors and since it's supervised, it **requires labels** for incoming stream of data.
- Essentially, this method triggers a drift alert if the parameters of the distribution go beyond a certain threshold rule.

Sequential analysis



- The second **supervised** technique is sequential analysis.
- In sequential analysis, we use a method called linear four rates.
- The basic idea is that **if data is stationary, the contingency table should remain constant**.
- The contingency table, in this case, corresponds to the truth table for a classifier that you're probably familiar with: true positive, false positive, false negative, and true negative.
- You use those to calculate **the four rates**: net predictive value, precision, recall, and specificity.
- If the model is predicting correctly, these four values should continue to **remain constant**.
- This analysis is tedious as it requires re-computing of all the metrics each time we get a new sample
- In order to overcome this problem, we can apply the **incremental update rule** (see formula in slide)

Error distribution monitoring

Method used is Adaptive Windowing (ADWIN)

Calculate mean error rate at every window of data

Size of window adapts, becoming shorter when data is not stationary

$$|\mu_0 - \mu_1| > \Theta_{Hoeffding}$$

- The last **supervised** technique to review is error distribution monitoring.
- The method of choice here is known as **adaptive windowing (ADWIN)**.
- In this method, you divide the incoming data into windows, the size of which adapts to the data, then you calculate **the mean error rate** at every window of data.
- Finally, let's calculate the absolute difference of the mean error rate at every successive window and compare it with a threshold based on the Hoeffding bound.
- **The Hoeffding bound is used for testing the difference between the means of two populations**.

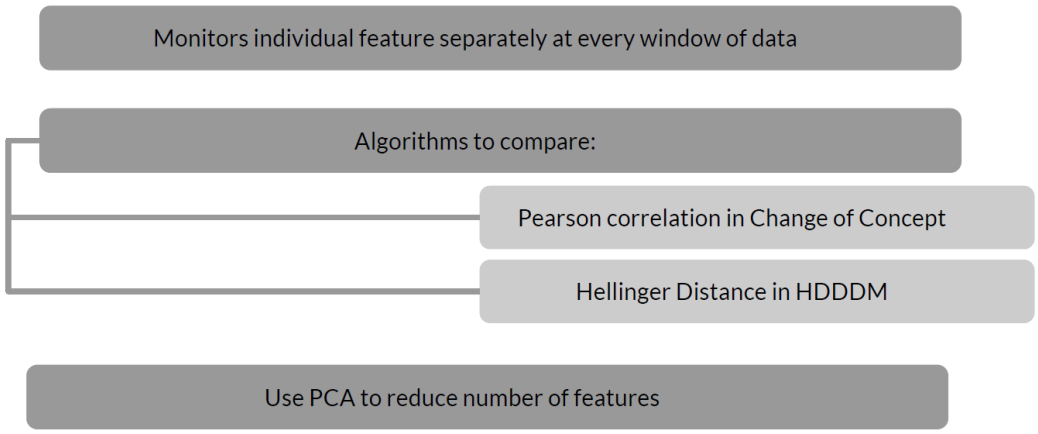
Clustering/novelty detection

- Assign data to known cluster or detect emerging concept
- Multiple algorithms available: OLINDDA, MINAS, ECSMiner, and GC3
- Susceptible to curse of dimensionality
- Does not detect population level changes

- Now, let's move on to **unsupervised** techniques.
- **The main problem with supervised techniques is that you need to have labels**, and generating labels can be expensive and slow.
- In unsupervised techniques, you don't need labels.
- Note that you can also use unsupervised techniques in addition to supervised techniques when you do have labels.
- Let's start with clustering or novelty detection.
- In this method, you cluster the incoming data into one of the known classes.
- **If you see that the features of the new data are far away from the features of known classes, then you know that you're seeing an emerging concept**.
- Based on the type of clustering you choose, there are multiple algorithms available.
- We've listed a few, including OLINDDA, MINAS, ECSMiner, and GC3.
- But for this course, we won't be discussing the details of these algorithms.
- While the visualization and ease of working with clustering work well with low dimensional data, the curse of dimensionality kicks in once the number of dimensions grow significantly.

- Eventually, these methods start getting inefficient, but you can use dimensionality reduction techniques like PCA and NMF to help make it manageable.
- However, this is the **only method** that helps you in detecting emerging concepts.
- The downside of this method is that the text only **cluster-based drift, not population-based changes**.

Feature distribution monitoring



- The next unsupervised technique is feature distribution monitoring.
- In feature distribution monitoring, we monitor each feature of the dataset separately.
- You split the incoming dataset into uniformly size windows, and then compare the individual features against each window of data.
- There are multiple algorithms available to do the comparison, and here are two of the most popular ones.
- The first is **Pearson correlation**, which is using the change of concept technique.
- There's also the **Hellinger distance**, which is used in the Hellinger Distance Drift Detection Method or HDDDM.
- The Hellinger distance is used to quantify the similarity between two probability distributions
- Similar to the case of novelty detection, if the curse of dimensionality kicks in, you can make use of dimensionality reduction techniques like PCA and NMF to reduce the number of features.
- The downside of this method is that it is not able to detect population drift since it only looks at individual features.

Model-dependent monitoring

- Concentrate efforts near decision margin in latent space
- One algorithm is Margin Density Drift Detection (MD3)
- Area in latent space where classifiers have low confidence matter more
- Reduces false alarm rate effectively

- The last unsupervised technique to cover is model-dependent monitoring.
- This method monitors the space near the decision boundaries or margins in the latent feature space of your model.

- One of the algorithms used is **Margin Density Drift Detection** or MD3.
- Space near the margins where the model has **low confidence** matters more than in other places, and this method looks for incoming data that falls into the margins.
- A change in the number of samples in the margin, the **margin density**, indicates drift.
- This method is very good at reducing the false alarm rate.

Google Cloud AI Continuous Evaluation

- Leverages AI Platform Prediction and Data Labeling services
- Deploy your model to AI Platform Prediction with model version
- Create evaluation job
- Input and output are saved in BigQuery table
- Run evaluation job on few of these samples
- View the evaluation metrics on Google Cloud console

- There are many ways to evaluate different aspects of model performance, looking at both the incoming stream of requests and the predictions generated by the model.
- Cloud hosting providers, including Google, offer services which can be used for continuous evaluation of your data and your model.
- One such service is Microsoft Azure Machine Learning DataSense which focuses on data drift.
- Another is Amazon SageMaker **Model Monitor** which focuses on concept drift.
- Here, let's take a look at Google Cloud AI **Continuous Evaluation** which also focuses primarily on concept drift, as an example of the services which are available.
- Google Cloud AI Continuous Evaluation regularly samples prediction input and output from trained models that you've deployed to AI Platform Prediction.
- AI Platform Data Labeling service then assigns human reviewers to provide ground truth labels for a sample of the prediction requests that you receive.
- Or alternatively you can provide your own ground truth labels using a different technique.
- We discussed several labeling techniques previously.
- The data labeling service then compares your model's predictions with the ground truth labels to provide continual feedback on how your model is performing over time.
- You start by deploying a trained model to AI Platform Prediction as a model version, then you can create an evaluation job for the model version.
- As your model serves online predictions, the input and output for some of these predictions is saved in a BigQuery table. You can customize how much data gets sampled.
- Intermittently, the evaluation job runs, generating evaluation metrics which you can view in the Google Cloud console.

How often should you retrain?

- Depends on the rate of change
- If possible, automate the management of detecting model drift and triggering model retraining

- How often should you retrain your model?
- There's no fixed answer to this question. It largely depends on the data and the world.
- The rate of change of the data and the rate of change of the world that you are modeling will determine how often you need to retrain your model to adapt to change.
- Of course, you can retrain your model whenever you wish, including when you've made improvements to your model's design.
- Retraining too often is okay, but it can result in higher costs for compute resources.
- Not retraining often enough can lead to degraded model performance.
- Ideally, you should monitor your data and model well enough to be able to use your evaluation results to **trigger retraining automatically**.
- If you can't do that or haven't reached the level of maturity in your deployment, then you can also just retrain on a **schedule**.

- For example, imagine that you're selling sporting goods and using your model to predict sales so that you can order inventory.
- In the winter, your model is performing well and your inventory ordering is nearly perfect.
- Your customers are buying winter clothes and winter sports equipment like skis.
- However, as spring approaches, you start to see changes in customer behavior.
- The same customers that we're buying skis before start to buy tennis rackets.
- But your model is still predicting skis and you're in danger of ordering too many.
- Because you had continuous monitoring in place, you catch the problem early and retrain your model with new data from recent customer interactions and sales.
- Now, your inventory is back to nearly perfect again.

References

- TensorBoard
- TFMA
- TFMA architecture
- Explaining and Harnessing Adversarial Examples
- PDPbox
- PyCEbox
- Fairness
- Learning fair representations
- Fairness-aware Machine Learning library (Themis ML)
- AI 360 open source model fairness library
- Model remediation
- Model cards
- Instrumentation, Observability & Monitoring of Machine Learning Models
- Monitoring Machine Learning Models in Production - A Comprehensive Guide
- Concept Drift detection for Unsupervised Learning
- Google Cloud
- Amazon SageMaker
- Microsoft Azure