Chapter 8. Model Analysis

Successfully training a model and getting it to converge feels good. It often feels like you're done, and if you're training it for a class project or a paper that you're writing, you kind of *are* done. But for production ML, after the training is finished you need to enter a new phase of your development that involves a much deeper level of analysis of your model's performance, from a few different directions. That's what this chapter is about.

Analyzing Model Performance

After training and/or deployment, you might notice a decay in the performance of your model. In addition to determining how to improve your model's performance, you'll need to anticipate changes in your data that you might expect to see in the future, which are generally very domain dependent, and react to the changes that occurred since you originally trained your model.

Both of these tasks require analyzing the performance of your model. In this section, we'll review some basics of model analysis. When conducting model analysis, you'll want to look at model performance not just on your entire dataset, but also on smaller chunks of data that are "sliced" by interesting features. Looking at slices gives you a much better

understanding of the variance of individual predictions than what you'd get by looking at your entire dataset.

Choosing the slices that are important to analyze is usually based on domain knowledge. Though slicing on any feature used by your model can provide insights, doing so may produce too many slices to manageably analyze. Moreover, it can be useful to slice on attributes that are not directly used by the model. For example, an image classifier whose only feature is image bytes may benefit from being sliced by metadata related to the version of label generation logic used for each image.

Ultimately, model analysis comes down to finding the smallest number of slices that will help you understand the relevant behavior of your model, which often requires knowledge about your domain and your dataset. This will allow you to determine whether there is room for improvement in your model across slices. 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, will be important, and knowing this will largely be a result of knowing about the domain.

At a high level, there are two main ways to analyze the performance of your model: black-box evaluation and model introspection. You can also analyze the performance metrics and the optimization objectives to glean important

insights regarding your model’s performance. Let’s take a look at each of these in turn.

Black-Box Evaluation

In black-box evaluation, you generally don’t consider the internal structure of the model. You are just interested in quantifying the performance of the model through metrics and losses. This is often sufficient within the normal course of development.

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

Performance Metrics and Optimization Objectives

Next, let’s look at the difference between performance metrics and optimization.

First, performance metrics. Based on the problem you’re solving, 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, regression, and so on. These are the metrics that you use when you design and train a model.

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 to find an optimal point, hopefully a global optimum, in your loss surface. If you look at TensorBoard again, you’ll notice options for tracking performance metrics such as accuracy, and optimization objectives such as the loss, after each epoch of training and validation.

Advanced Model Analysis

When you’re evaluating your training performance you’re usually watching your top-level metrics, which are aggregated over your entire dataset. You do this to decide whether your model is doing well or not. But this doesn’t tell you how well your model is doing on individual parts of the data. For that, you need more advanced analysis and debugging techniques. We’ll take a look at several analysis techniques in the following subsections. We’ll discuss advanced model debugging techniques later in the chapter.

TensorFlow Model Analysis

Your top-level metrics can easily hide problems with particular parts of your data. For example, your model may not perform well for certain customers, products, stores, days of the week, or subsets of your data that make sense for your domain or problem. For example, say your customers

are requesting a prediction from your model. If your model produces a bad prediction, your customers' experience will be bad—regardless of how well the model may perform in top-level metrics.

TensorFlow Model Analysis (TFMA) is an open source scalable framework for doing deep analysis of model performance, including analyzing performance on slices of data. TFMA is also used as a key part of TensorFlow Extended (TFX) pipelines to perform deep analysis before you deploy a newly trained version of a model. For most of this chapter, we'll be using TFMA as well as some related tools and technologies. TFMA supports black-box evaluation and is a versatile tool for doing deep analysis of your model's performance.

TFMA has built-in capabilities to check that your models meet your quality standards, visualize evaluation metrics, and inspect performance based on different data slices. TFMA can be used by itself or as part of another framework such as TFX. Figure 8-1 shows the high-level architecture of TFMA.

The TFMA pipeline consists of four main stages: read inputs, extract, evaluate, and write results. During the read inputs stage, a transform takes raw input (CSV, TFRecords, etc.) and converts it into a dictionary format that is understandable by the extractors. Across all the stages, the output is kept in this dictionary format, which is of the data type `tfma.Extracts`.

In the next stage, extraction, distributed processing is performed using Apache Beam. `InputExtractor` and `SliceKeyExtractor` form slices of the original dataset, which will be used by `PredictExtractor` to run predictions on each slice. The results are sent to the evaluators, again as a `tfma.Extracts` dictionary.

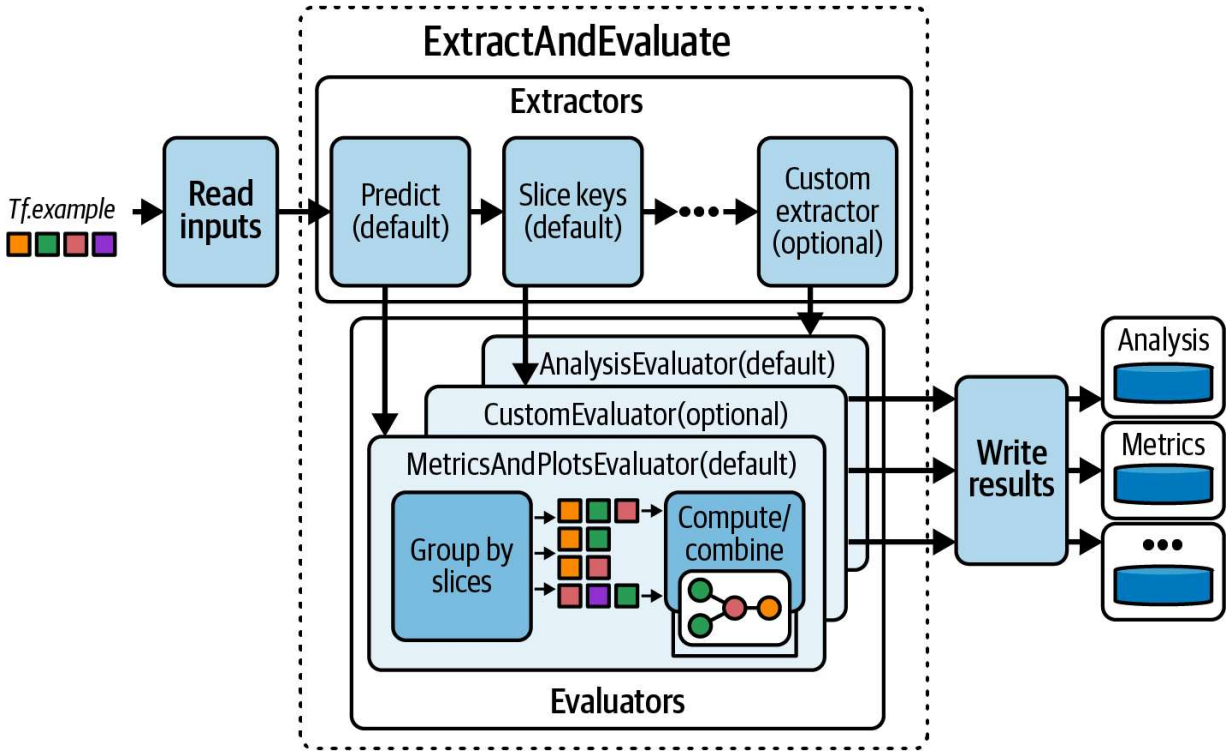

Figure 8-1. TFMA architecture

During the evaluation stage, distributed processing is again performed 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 predictions received from the previous stage.

In the final stage, the results are written 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 also used to inspect the training progress of a single model, often as you're monitoring your progress during training. Additionally, it can be used to visualize the training progress for more than one model, with performance for each model plotted against its global training steps during training.

After training has finished, TFMA allows developers to compare different versions of their trained models, as shown in Figure 8-2. While TensorBoard visualizes streaming metrics of multiple models over global training steps, TFMA visualizes metrics computed for a single model over multiple versions of the exported `SavedModel`.

Basic model evaluation results look at aggregate or top-level metrics on the entire training dataset. This aggregation often hides problems with model performance. For example, a model may have an acceptable area under the curve (AUC) over the entire eval dataset, but it may underperform on specific slices. In general, a model with good performance "on average"

may still exhibit failure modes that are not apparent by looking at an aggregate metric.

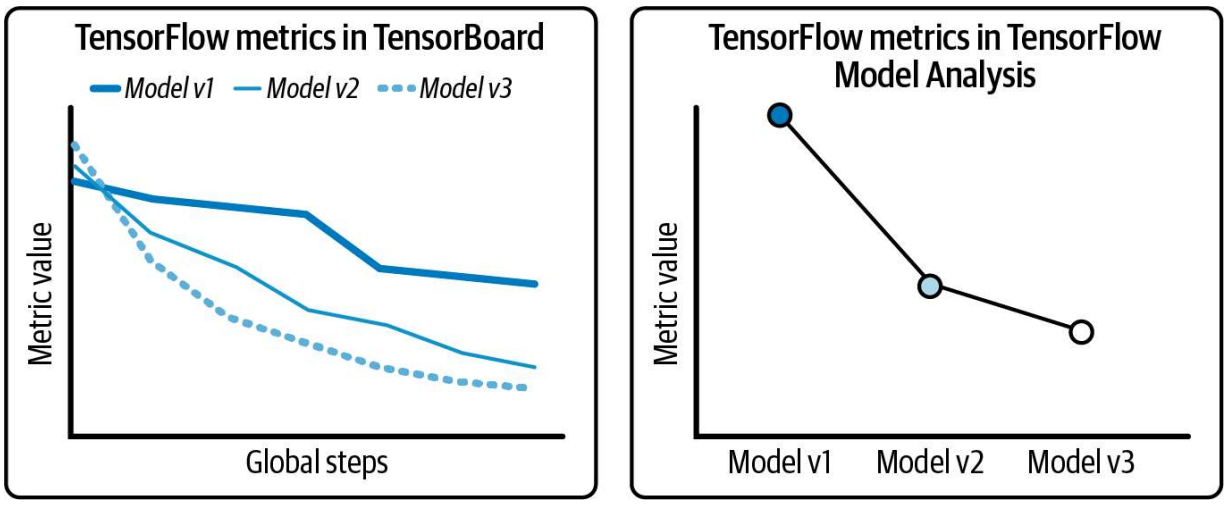

Figure 8-2. Metrics in TensorBoard and TFMA

Slicing metrics allows 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 who take the taxi during the day versus at night, by slicing the data by the hour.

TensorBoard computes metrics on a mini-batch basis during training. These metrics are called *streaming metrics* and they're approximations based on the observed mini batches.

TFMA uses Apache Beam to do a full pass over the eval dataset. This not only allows for more accurate calculation of metrics, but also scales up to

massive evaluation datasets, since Beam pipelines can be run using distributed processing backends. Note that TFMA computes the same TensorFlow metrics that are computed by the TensorFlow eval worker; it just does so more accurately by doing a full pass over the specified dataset. TFMA can also be configured to compute additional metrics that were 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 compute accurate metrics, a deterministic full pass over those examples is important.

TFMA is a highly versatile model evaluation tool that goes beyond evaluating TensorFlow models. For example, recent versions include support for non-TensorFlow models, such as PyTorch and scikit-learn models. Furthermore, TFMA now integrates with TF Transform as it can perform transformations of feature labels. Let's take a look at how TFMA works.

The following example demonstrates how to use TFMA through the standalone TFMA library. However, note that TFMA is often used in combination with a TFX pipeline. In Chapters 20 and 21, we'll discuss TFX pipelines and show you how you can use TFMA in the context of an entire ML pipeline.

To get started, you need to install TFMA via `pip`. If you have installed TFX already, TFMA was installed as one of its dependencies:

```
pip install tensorflow-model-analysis
```

Next, import TFMA to the shortcut `tfma`:

```
import tensorflow_model_analysis as tfma
```

TFMA's model analysis is configured through a protocol buffer configuration. If you haven't used a protocol buffer, no worries. Google provides a method called `text_format.Parse` to convert text configurations to the required protocol buffer format:

```
from google.protobuf import text_format
eval_config = text_format.Parse(
    """
    <TFMA configuration>
    """, tfma.EvalConfig())
```

TFMA configurations take three different inputs:

`model_specs`

: Specifications that define all the details regarding the model and its inference

`metric_specs`

Specifications that define which metrics to use for model evaluation

`slicing_specs`

Specifications that define whether the metrics should be applied to a specific slice of the data

The slicing specifications are especially helpful if you want to compare the model across a specific model input feature (e.g., compare model accuracy across different user age groups). That way, you can spot whether a model is underperforming in a specific feature subset; something you couldn't spot from averages across the feature.

The `metric_specs` input defines the metrics and thresholds if you want to compare the model against baseline models (e.g., your current production model). TFMA will generate a model "blessing" signaling that the new model version performs better in terms of the metrics and is "blessed" for production use cases (we'll come back to model blessings in Chapter 20 when we introduce model pipelines):

```
model_specs {
  name: "candidate"
  label_key: "output_feature"
}
```

```
    model_specs {
      name: "baseline"
      label_key: "output_feature"
      is_baseline: true
    }
```

In our basic example, we define one metric, `BinaryAccuracy`. Our demo model will be blessed as production ready if two conditions are met —the overall accuracy needs to exceed 0.9, and the new model version needs to perform at least as well as the baseline model):

```
    metrics_specs {
      metrics {
        class_name: "BinaryAccuracy"
        threshold {
          value_threshold {
            lower_bound { value: 0.9 }
          }
          change_threshold {
            direction: HIGHER_IS_BETTER
            absolute { value: -1e-10 }
          }
        }
      }
    }
```

You can add one or more metrics to the metric spec. TFMA supports all standard metrics as well as Keras metrics, but you can also write your own custom metric functions. A small list of available metrics besides the mentioned `BinaryAccuracy` includes the following:

- `BinaryCrossEntropy`
- `AUC`
- `AUCPrecisionRecall`
- `Precision`
- `Recall`

Furthermore, you can generate plots from metrics with metrics such as `CalibrationPlot` and `ConfusionMatrixPlot`.

Lastly, we need to define our slicing specifications. If you don't want to slice the data, you can leave the specifications blank. In this case, the metrics will be generated against the entire dataset:

```
  slicing_specs {}
```

If you want to slice the data, you can define the name of the input feature as follows:

```
  slicing_specs {
    feature_keys: ["input_feature_a", "input_feat
  }
```

With the evaluation configuration in place, let's define our model setups:

```
eval_config = text_format.Parse(
    """
    <TFMA configuration>
    """, tfma.EvalConfig())
MODELS_DIR = "..."
candidate_model_path = os.path.join(MODELS_DIR,
candidate_model = tfma.default_eval_shared_model(
    model_name=tfma.CANDIDATE_KEY,
    eval_saved_model_path=candidate_model_path,
    eval_config=eval_config)
baseline_model_path = os.path.join(MODELS_DIR, '
tfma.default_eval_shared_model(
    model_name=tfma.BASELINE_KEY,
    eval_saved_model_path=baseline_model_path,
    eval_config=eval_config),
```

We can now kick off a model evaluation by running the function `run_model_analysis`. This function loads the test dataset from TFRecords, generates metrics for the candidate and baseline models, and outputs the validation results to `OUTPUT_DIR`:

```
BASE_DIR = "..."
tfrecord_file = tfrecord_file = os.path.join(BASE
OUTPUT_DIR = "..."
validation_output_path = os.path.join(OUTPUT_DIR,
eval_result = tfma.run_model_analysis(
    [candidate_model_path, baseline_model_path],
    eval_config=eval_config,
    data_location=tfrecord_file,
    output_path=validation_output_path)
```

TFMA provides a number of tools to inspect and visualize the evaluation results. For example, you can inspect the results here:

```
import tensorflow_model_analysis.experimental.dat
from IPython.display import display
dfs = tfma_dataframe.metrics_as_dataframes(
  tfma.load_metrics(validation_output_path))
display(dfs.double_value.head())
```

You can also plot metrics as follows:

```
tfma.view.render_plot(
    eval_result,
    tfma.SlicingSpec(feature_values={'input_featu
```

The Learning Interpretability Tool

The Learning Interpretability Tool (LIT) is an advanced set of tools that are integrated into a cohesive visual interface. LIT includes a wide range of analytical tools for a variety of modeling types, including text, image, and tabular data. It's especially useful for language model analysis, including large language models (LLMs), and the already extensive list of supported analytical techniques is growing as the field moves forward. The supported techniques include:

- Token-based salience, including LIME and integrated gradients
- Sequence salience
- Salience clustering
- Aggregate analysis
- Testing with Concept Activation Vectors (TCAV)
- Counterfactual analysis

Check out the LIT documentation and examples. Figure 8-3 shows an example of the visual interface, which is highly configurable.

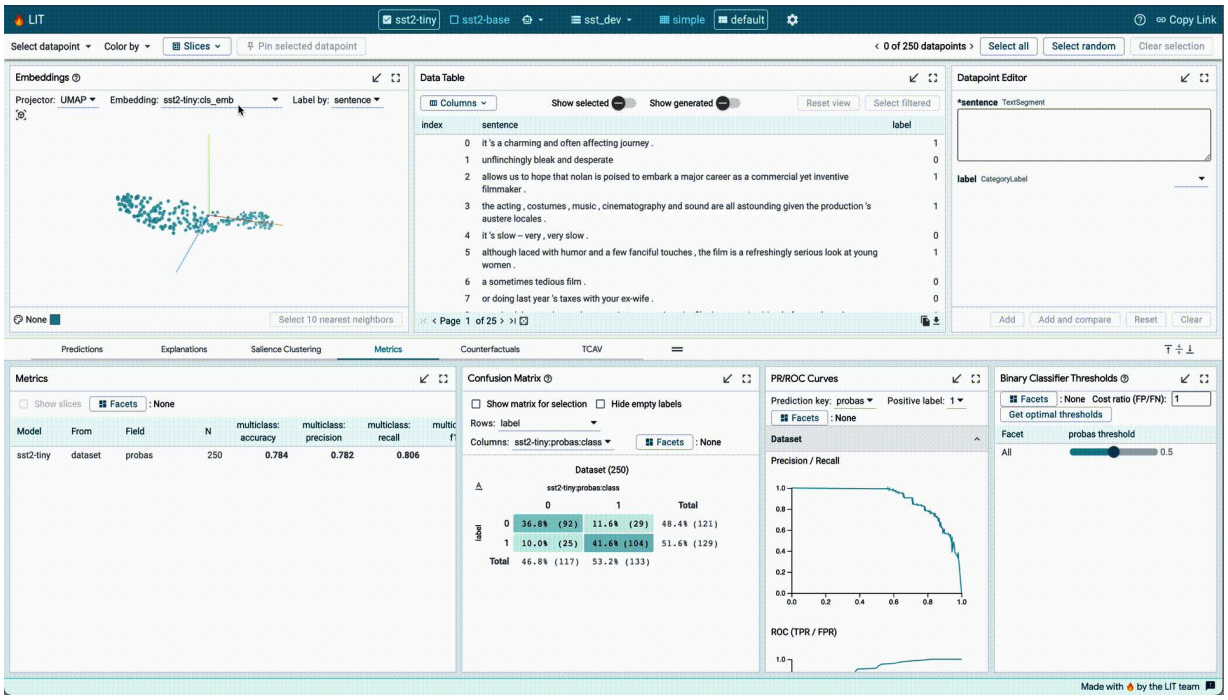

Figure 8-3. The Learning Interpretability Tool interface

Advanced Model Debugging

At some point in your journey toward production ML, you'll need to measure model performance beyond simple metrics and become familiar with ways to analyze it and improve it. Before discussing model debugging, let's 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.

So, how do you measure the robustness of a model?

The first and most important thing to note is that you shouldn't be measuring the robustness of a model during training, since that would require you to either introduce data outside of your training set or attempt to measure robustness with your training set. Also, you shouldn't be using the same dataset you used during training, since by definition, robustness only applies to data that the model was not trained with.

As you probably already know, before you start the training process, you should split the dataset into train, validation, and test splits. You can use the test split, which is totally unseen by the model, even during the validation stage, for testing model robustness. Otherwise, the best choice is to generate a variety of new types of data, and we'll discuss some of the methods to do this in "Sensitivity Analysis". The metrics themselves will be the same types you use for training, depending on the model type; for example, root mean square error (RMSE) for regression models and AUC for classification.

It’s important to note that in this discussion, when we refer to “model debugging” we’re not talking about fixing code errors that might throw exceptions. Instead, model debugging in the context of this discussion is an emerging discipline focused 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 like code in a way that’s very similar to how you would test them in software development. It probes sophisticated ML response functions and decision boundaries to detect and correct accuracy, fairness, security, and other problems in ML systems. We’ll discuss this more in a bit.

Model debugging has several objectives. For example, 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 the model. Another problem with ML models is social discrimination; that is, 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, once the model is in production, certain requests may be aimed at extracting data out of your model in order to understand how it was built. This is especially a problem when data with private information has been used for training. Was the training data anonymized before it was used?

Lastly, with time, your model’s performance will decay as the distribution of the incoming data changes.

Three of the most widely used debugging tools are benchmark models, sensitivity analysis, and residual analysis. We’ll discuss each of these individually.

Benchmark Models

Benchmark models are small, simple models that you use to baseline your problem before you start development. They are generally not state of the art, but instead are linear or other simple models with very consistent, predictable performance.

You compare your model to see whether it is performing better than the simpler benchmark model as a kind of sanity test. If it isn’t, it could be that your model has a problem or that a simple model accurately models the data and is really all you need for your application.

Even after the model you’re testing performs better than the benchmark model, you can continue to use the benchmark model for debugging. For example, you can still evaluate which test samples your model is failing but the benchmark model predicts correctly. Then, you need to study your model to find out why that’s happening.

Sensitivity Analysis

Sensitivity analysis helps you understand your model by examining the impact that each feature has on the model's prediction. Tools such as LIT can help you visualize, explore, and understand your model's sensitivity.

In sensitivity analysis, you experiment by changing a single feature's value while holding the other features constant, and then you observe the model's results. If changing the feature's value causes the model's results to be drastically different, it means this feature has a big impact on the prediction.

Usually you are changing the values of the feature synthetically according to some distribution or process, and you're ignoring the labels for the data. You're not really looking to see whether 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.

Random attacks

With random attacks, you test the model's response to random input data or data that has been randomly altered. By looking at how the model responds to such data, you can identify potential weaknesses and areas for further investigation and debugging. In general, 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

Another tool in model debugging and understanding are partial dependence plots, which show the marginal effect of key features on model predictions. PDPbox and PyCEbox are open source packages that are available for creating partial dependence plots.

Vulnerability to attacks

How vulnerable is your model to attacks? Several ML models, 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, depending on how your model is being used.

Imagine making a wrong decision on an important question, based on only slightly corrupted data. Depending on how catastrophic an 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. What do these attacks look like? Figure 8-4 shows a famous example, with two groups of images.

Applying only the tiny distortions (center columns) to the images in the left columns of Figure 8-4 results in the images in the right columns, which a model trained on ImageNet classifies as an ostrich.

How serious a problem is this? Thinking that a school bus is an ostrich might seem harmless, but it depends on your application. Let's discuss a few examples. First, with an autonomous vehicle, it's important to recognize traffic signs, other vehicles, and people. But as the stop sign in Figure 8-5 shows, if a sign is altered in just the right way, it can fool the model, and the results could be catastrophic.

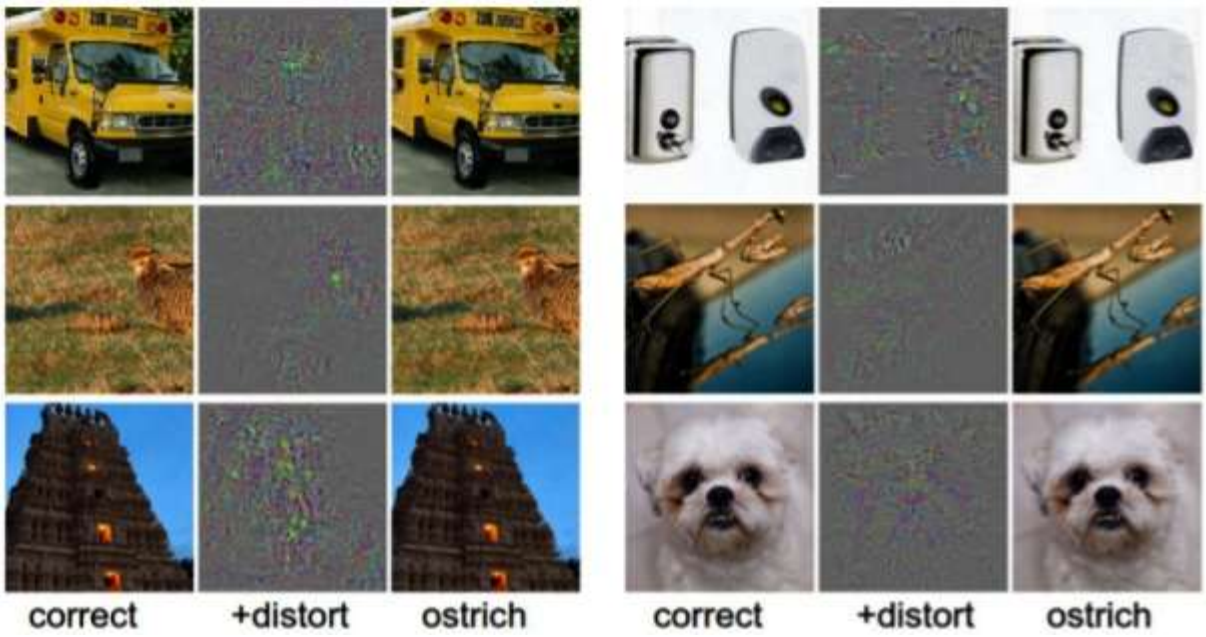

Figure 8-4. Attacks against image models (source: Szegedy et al., 2014)

Figure 8-5. An attack against an autonomous vehicle model (source: Eykholt et al., 2018)

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

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

The Future of Privacy Forum, an industry group that studies privacy and security, suggests that security and privacy harms, enabled by ML, fall into roughly two categories:

> *Informational harms[, which] relate to the unintended or unanticipated leakage of information[, and] Behavioral harms, [which] relate to manipulating the behavior of the model itself, impacting the predictions or outcomes of the model.*
>
> —"Warning Signs: The Future of Privacy and Security in an Age of Machine Learning," Sept 2019

Membership inference attacks are a type of informational harm 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 membership inference attacks are seemingly complex, studies have shown that these attacks require much less technical sophistication than is frequently assumed.

Model inversion attacks, another type of informational harm, use model outputs to re-create 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 used genetic information to recommend dosing of specific medications, was able to directly predict individual patients' genetic markers.

Meanwhile, model extraction attacks use model outputs to re-create the model itself. This has been demonstrated against ML-as-a-service providers such as BigML and Amazon Machine Learning, and it can compromise privacy and security as well as the intellectual property of the underlying model itself.

Examples of behavioral harms include model poisoning attacks and evasion attacks. Model poisoning attacks occur when an adversary inserts malicious data into training data in order to alter the behavior of the model. An example is 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 model vulnerability

Before hardening your models you need to have some way to measure their 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 how to recognize the correct class. This is known as *adversarial training*. Examples created by tools such as CleverHans can be added to your dataset, but doing so limits your ability to use the tools to measure your model's 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 ML models such as deep neural networks. It is built on top of EagerPy and works natively with models in PyTorch, TensorFlow, and JAX.

Hardening your models

Unfortunately, detecting vulnerability is easier than fixing it. This is an emerging field, and like many things in security, there is an arms race occurring between attackers and defenders. One fairly advanced approach is

defensive distillation. Since this approach does not use specific adversarial examples, it may provide more general hardening to new attacks.

As the name suggests, defensive distillation training 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 sample creation from 95% to less than 0.5% in one study. Instead of transferring knowledge among different architectures, as is done with the distillation discussed in Chapter 6, the authors of this study propose keeping the same model architecture and using knowledge distillation to harden the model against attacks. In other words, instead of transferring knowledge among different architectures, the authors propose to use knowledge distillation to improve a model's own resilience to attacks.

Residual Analysis

Alongside benchmark models and sensitivity analysis, residual analysis is another valuable debugging technique. Residuals measure the difference between the model's predictions and the ground truth. In most cases, residual analysis is used for regression models. 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, as shown in Figure 8-6. If you find a correlation between residuals, it is usually a sign that your model can be improved.

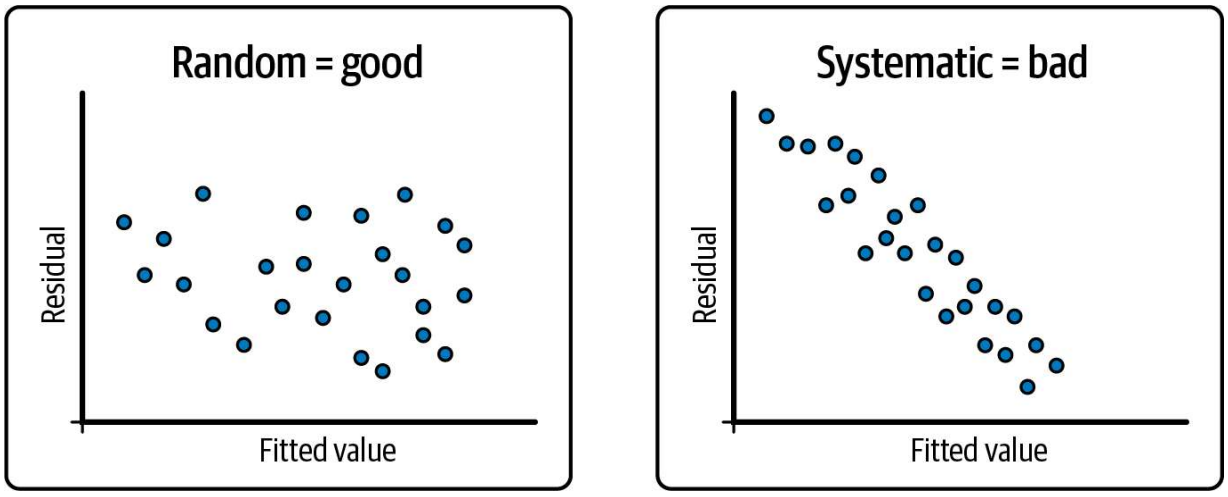

Figure 8-6. Residual analysis

So, what should you aim for when performing residual analysis?

First, 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, there is 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 discussed ways to improve it. What can you do to improve model robustness?

First, you should make sure your training data accurately mirrors the requests you will receive for your trained model. However, data augmentation can also help your model generalize, which typically reduces sensitivity. You can generate data in many ways, including generative techniques, interpolative methods, or simply adding noise to your data. Data augmentation is also a great way to help correct for imbalanced 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 that can help with model interpretability, and this can help with improving model robustness. There are also model

architectures that are more easily interpreted, including tree-based models, as well as neural network models that are specifically designed for interpretability.

Two additional remediation techniques are model editing and model assertions. Some models, such as decision trees, are so directly interpretable that the learned parameters can be understood easily. With model editing, if you find that something is going wrong, you can tweak the model to improve its performance and robustness.

With model assertions, you can apply business rules or simple sanity checks to your model's results and either alter or bypass the results before delivering them. For example, if you're predicting someone's age, the number should never be negative, and if you're predicting a credit limit, the number should never be more than a maximum amount.

Now that you understand ways you can improve model robustness, let's look at how you can reduce or eliminate model bias, a concept known as *discrimination remediation*.

Discrimination Remediation

The best solution for model bias is to have a diverse dataset that represents the people who will be using your model. It also helps to have people on the development team from diverse backgrounds and areas of expertise relevant

to identifying and addressing potential discrimination. Careful feature selection, including sampling and reweighting rows to minimize discrimination in training data, can also be helpful.

When training, you should consider using a tool such as the Fairness Indicators library (discussed in the next section) or AI Fairness 360 (AIF360) toolkit to gather fairness metrics for your model. You can also apply bias mitigation techniques to your data and models and consider building fairness into your learning algorithm or objective function itself. Tools such as the TensorFlow Model Remediation Library and AIF360 toolkit can help.

Fairness

In this section, we'll focus on how to make models fair and look at using the Fairness Indicators library 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's performance, you should introduce checks and controls to ensure that your model behaves fairly in different scenarios. Accounting for fairness and reducing bias toward any group of people is an important part of that. You need to make sure 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 was built on top of the TFMA framework. With the Fairness Indicators suite of tools, 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 evaluating fairness, not for doing remediation to improve fairness.

Looking at slices of data is actually quite informative when you're trying to mitigate bias and check for fairness. When evaluating fairness, it's important to identify slices of data that are sensitive to fairness and to evaluate your model's performance on those slices. Only evaluating fairness using the entire dataset can easily hide fairness problems with particular groups of people. That makes it important for you to consider which slices will be sensitive to fairness issues, often based on your domain knowledge.

It's also important to consider and select the right metrics to evaluate for your dataset and users, because otherwise, you may evaluate the wrong things and be unaware of problems. This is also often based on domain knowledge.

Keep in mind that evaluating 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, which you can do by asking yourself the following questions:

- 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, and cultural anthropologists, as well as with members of the communities that will be using your application. You will probably not get answers unless you ask questions.

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, income, sexual orientation, and disability status. Ideally, you should be working with labeled data, but if not, you can apply statistics to look at the distributions of the outcomes with some assumptions around any expected differences.

In general, 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 that don't have a good margin of separation from their decision boundaries, you should consider reporting the rate at which the label is predicted.

Fairness Evaluation

The measurements for fairness might not be immediately obvious, but fortunately various fairness metrics are available in Fairness Indicators. These metrics include the positive/negative rate, accuracy, and AUC.

A confusion matrix can help visualize the basic components of these metrics, as shown in Figure 8-7.

		Actual	
		Positive	Negative
Predicted	Positive	True positive (TP)	False positive (FP)
	Negative	False negative (FN)	True negative (TN)

Figure 8-7. Confusion matrix

Let's first consider the basic positive and negative rates. These rates show the percentage of data points that are classified as positive or negative, and they are independent of ground truth labels. These metrics help with

understanding demographic parity as well as equality of outcomes, which should be equal across subgroups. This applies to use cases in which having equal percentages of outcomes for different groups is important.

True/false positive/negative rates

The true positive rate (TPR) measures the percentage of positive data points, as labeled in the ground truth, that are correctly predicted to be positive (i.e., TP / (TP + FN)). Similarly, the false negative rate (FNR) measures the percentage of positive data points that are incorrectly predicted to be negative (i.e., FN / (TP + FN)). This metric may often relate to equality of opportunity for the positive class, when it should be equal across subgroups. This often applies to use cases in which it is important that the same percentage of qualified candidates are rated positively in each group, such as for loan applications or school admissions.

Similarly, the true negative rate (TNR) measures the percentage of negative data points, as labeled in the ground truth, that are correctly predicted to be negative (i.e., TN / (FP + TN)). The false positive rate (FPR) is the percentage of negative data points that are incorrectly predicted to be positive (i.e., FP / (FP + TN)). This metric often relates to equality of opportunity for the negative class, when it should be equal across subgroups. This often applies to use cases in which misclassifying something as 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 technologies such as face detection or face attributes.

Accuracy and AUC

The last set of fairness metrics we will discuss are accuracy and area under the curve, or 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 in which the precision of the task is critical, but not necessarily in a given direction, such as face identification or face clustering.

Fairness Considerations

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. However, 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 evaluations should be run throughout the development process and after 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 rather 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.

Continuous Evaluation and Monitoring

It’s important to consider ways to monitor your model once it has been deployed to production. When you train your model you use the training data that is available at that time. That training data represents a snapshot of the world at the time the data 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 a model to predict movie sales was trained on data collected in the 1990s, it might predict that customers would buy VHS tapes. Is that still a good prediction today? Our guess is no.

When your 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 help give you that early warning. Once your monitoring shows that you have issues that need to be fixed, retraining your model is usually necessary. Chapter 16 discusses model monitoring and drift detection, as well as model retraining.

Conclusion

In this chapter, we introduced strategies to analyze your model's performance and tools that can be used to evaluate your models. We also introduced ways to measure model fairness and how to continuously evaluate your models. We explored some advanced techniques for model analysis and model remediation, both of which are important for detecting and fixing problems with your models. We also examined different kinds of attacks and discussed how model sensitivity can both be a problem by itself and make your models more susceptible to attack. These considerations are important in production settings, where customers and your business can be harmed by models that misbehave.