Chapter 7. Model Analysis and Validation

At this point in our machine learning pipeline, we have checked the statistics of our data, we have transformed our data into the correct features, and we have trained our model. Surely now it's time to put the model into production? In our opinion, there should be two extra steps before you move on to deploy your model: analyzing your model's performance in-depth and checking that it will be an improvement on any model that's already in production. We show where these steps fit into the pipeline in Figure 7-1.

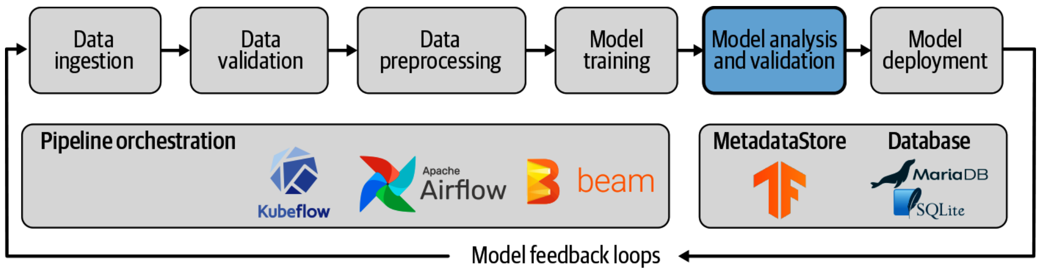


Figure 7-1. Model analysis and validation as part of ML pipelines

While we're training a model, we're monitoring its performance on an evaluation set during training, and we're also trying out a variety of hyperparameters to get peak performance. But it's common to only use one metric during training, and often this metric is accuracy.

When we're building a machine learning pipeline, we're often trying to answer a complex business question or trying to model a complex real-world system. One single metric is often not enough to tell us whether our model will answer that question. This is particularly true if our dataset is imbalanced or if some of our model's decisions have higher consequences than others.

In addition, a single metric that averages performance over an entire evaluation set can hide a lot of important details. If your model is dealing with data that is about people, does everyone who interacts with the model get the same experience? Does your model perform better for female users than male users? Are users from Japan seeing poorer results than users from the US? These differences can be both commercially damaging and cause harm to real people. If your model is doing object detection for an autonomous vehicle, does it work acceptably in all lighting conditions? Using one metric for your whole training set can hide important edge and corner cases. It's essential to be able to monitor metrics across different slices of your dataset.

It's also extremely important to monitor your metrics through time—before deployment, after deployment, and while in production. Even if your model is static, the data that comes into the pipeline will change through time, often causing a decline in performance.

In this chapter we'll introduce the next package from the TensorFlow ecosystem: TensorFlow Model Analysis (TFMA), which has all these capabilities. We'll show how you can get detailed metrics of your model's performance, slice your data to get metrics for different groups, and take a deeper dive into model fairness with Fairness Indicators and the What-If Tool. We'll then explain how you can go beyond analysis and start to explain the predictions your model is making.

We'll also describe the final step before deploying your new model: validating that the model is an improvement on any previous version. It's important that any new model deployed into production represents a step forward so that any other service depending on this model is improved in turn. If the new model is not an improvement in some way, it is not worth the effort of deploying.

How to Analyze Your Model

Our model analysis process starts with our choice of metrics. As we discussed previously, our choice is extremely important to the success of our machine learning pipeline. It's good practice to pick multiple metrics that make sense for our business problem because one single metric may hide important details. In this section, we will review some of the most important metrics for both classification and regression problems.

Classification Metrics

To calculate many classification metrics, it's necessary to first count the number of true/false positive examples and true/false negative examples in your evaluation set. Taking any one class in our labels as an example:

True positives

Training examples that belong to this class and are correctly labelled as this class by the classifier. For example, if the true label is `1`, and the predicted label is `1`, the example would be a true positive.

False positives

Training examples that do not belong to this class and are incorrectly labelled as this class by the classifier. For example, if the true label is `0`, and the predicted label is `1`, the example would be a false positive.

True negatives

Training examples that do not belong to this class and are correctly labelled as not in this class by the classifier. For example, if the true label is `0`, and the predicted label is `0`, the example would be a true negative.

False negatives

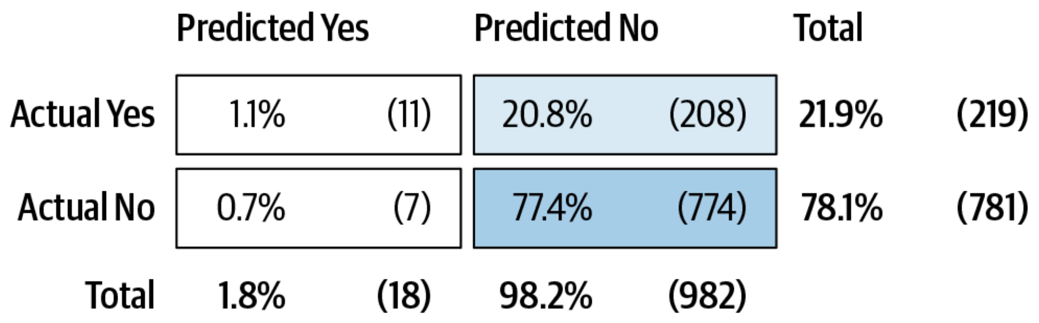
Training examples that belong to this class and are incorrectly labelled as not in this class by the classifier. For example, if the true label is `1`, and the predicted label is `0`, the example would be a false negative.

These basic metrics are all commonly shown in Table 7-1.

Table 7-1. Confusion matrix

	Predicted `1`	**Predicted `0`**
True value `1`	True positives	False negatives
True value `0`	False positives	True negatives

If we calculate all these metrics for the model from our example project, we get the results shown in Figure 7-2.



	Predicted Yes		Predicted No		Total	
Actual Yes	1.1%	(11)	20.8%	(208)	21.9%	(219)
Actual No	0.7%	(7)	77.4%	(774)	78.1%	(781)
Total	1.8%	(18)	98.2%	(982)		

Figure 7-2. Confusion matrix for our example project

We'll see that these counts are particularly useful when we talk about model fairness later in this chapter. There are several other metrics for comparing models that combine these counts into a single number:

Accuracy

Accuracy is defined as *(true positives + true negatives)/total examples*, or the proportion of examples that were classified correctly. This is an appropriate metric to use for a dataset where the positive and negative classes are equally balanced, but it can be misleading if the dataset is imbalanced.

Precision

Precision is defined as *true positives/(true negatives + false positives)*, or the proportion of examples predicted to be in the positive class that were classified correctly. So if a classifier has high precision, most of the examples it predicts as belonging to the positive class will indeed belong to the positive class.

Recall

Recall is defined as *true positives/(true positives + false negatives)*, or the proportion of examples where the ground truth is positive that the classifier correctly identified. So if a classifier has high recall, it will correctly identify most of the examples that are truly in the positive class.

Another way to generate a single number that describes a model's performance is the AUC (area under the curve). The "curve" here is the receiver operating characteristic (ROC), which plots the true positive rate (TPR) against the false positive rate (FPR).

The TPR is another name for *recall*, and it is defined as:

$$\text{true positive rate} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

The FPR is defined as:

$$\text{false positive rate} = \frac{\text{false positives}}{\text{false positives} + \text{true negatives}}$$

The ROC is generated by calculating the TPR and FPR at all classification thresholds. The *classification threshold* is the probability cutoff for assigning examples to the positive or negative class, usually 0.5. Figure 7-3 shows the ROC and the AUC for our example project. For a random predictor, the ROC would be a straight line from the origin to [1,1] that follows the *x* axis. As the ROC moves further away from the *x* axis toward the upper left of the plot, the model improves and the AUC increases. AUC is another useful metric that can be plotted in TFMA.

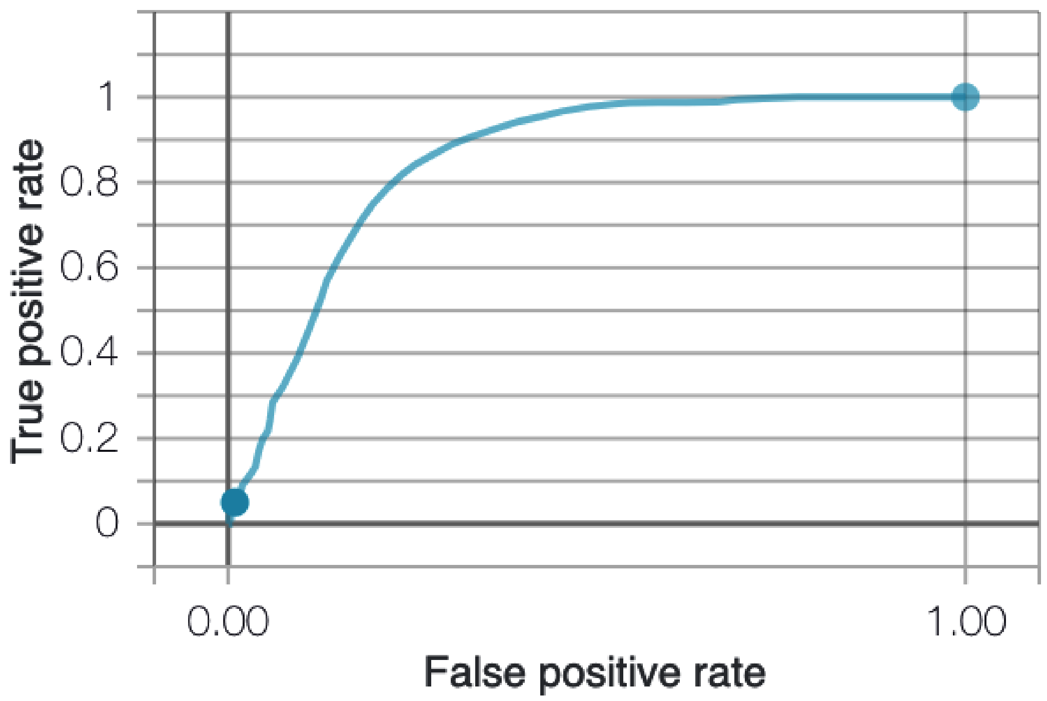


Figure 7-3. ROC for our example project

Regression Metrics

In a regression problem, the model predicts some numerical value for each training example, and this is compared with the actual value. Common regression metrics we can use in TFMA include:

Mean absolute error (MAE)

MAE is defined as:

$$\text{MAE} = \frac{1}{n}\sum\left|y - \hat{y}\right|$$

where n is the number of training examples, y is the true value, and $\hat{y}$ is the predicted value. For each training example, the absolute difference is calculated between the predicted value and the true value. In other words, the MAE is the average error produced by the model.

Mean absolute percentage error (MAPE)

MAPE is defined as:

$$\text{MAPE}= \frac{1}{n}\sum\left|\frac{y-\hat{y}}{y}\right| \times 100\%$$

As the name implies, this metric gives the percentage error for all examples. This is particularly useful for spotting when the model makes systematic errors.

Mean squared error (MSE)

MSE is defined as:

$$\text{MSE}= \frac{1}{n}\sum\left(y-\hat{y}\right)^2$$

This is similar to the MAE, except the $y - \hat{y}$ term is squared. This makes the effect of outliers on the overall error much greater.

Once you have chosen the metrics that are appropriate for your business problem, the next step is to include them in your machine learning pipeline. You can do this using TFMA, which we will describe in the next section.

TensorFlow Model Analysis

TFMA gives us an easy way to get more detailed metrics than just those used during model training. It lets us visualize metrics as time series across model versions, and it gives us the ability to view metrics on slices of a dataset. It also scales easily to large evaluation sets thanks to Apache Beam.

In a TFX pipeline, TFMA calculates metrics based on the saved model that is exported by the `Trainer` component, which is exactly the one that will be deployed. Thus, it avoids any confusion between different model versions. During model training, if you are using TensorBoard you will only get approximate metrics extrapolated from measurements on mini-batches, but TFMA calculates metrics over the whole evaluation set. This is particularly relevant for large evaluation sets.

Analyzing a Single Model in TFMA

In this section, we'll look at how to use TFMA as a standalone package. TFMA is installed as follows:

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

It takes a saved model and an evaluation dataset as input. In this example, we'll assume a Keras model is saved in `SavedModel` format and an evaluation dataset is available in the TFRecord file format.

First, the `SavedModel` must be converted to an `EvalSharedModel`:

```
import tensorflow_model_analysis as tfma

eval_shared_model = tfma.default_eval_shared_model(
    eval_saved_model_path=_MODEL_DIR,
    tags=[tf.saved_model.SERVING])
```

Next, we provide an `EvalConfig`. In this step, we tell TFMA what our label is, provide any specifications for slicing the model by one of the features, and stipulate all the metrics we want TFMA to calculate and display:

```
eval_config=tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='consumer_disputed')],
    slicing_specs=[tfma.SlicingSpec()],
    metrics_specs=[
        tfma.MetricsSpec(metrics=[
            tfma.MetricConfig(class_name='BinaryAccuracy'),
            tfma.MetricConfig(class_name='ExampleCount'),
            tfma.MetricConfig(class_name='FalsePositives'),
            tfma.MetricConfig(class_name='TruePositives'),
            tfma.MetricConfig(class_name='FalseNegatives'),
            tfma.MetricConfig(class_name='TrueNegatives')
        ])
    ]
)
```

ANALYZING TFLITE MODELS

We can also analyze TFLite models in TFMA. In this case, the model type must be passed to the `ModelSpec`:

```
eval_config = tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='my_label',
                                model_type=tfma.TF_LITE)],
    ...
)
```

We discuss TFLite in more detail in "TFLite".

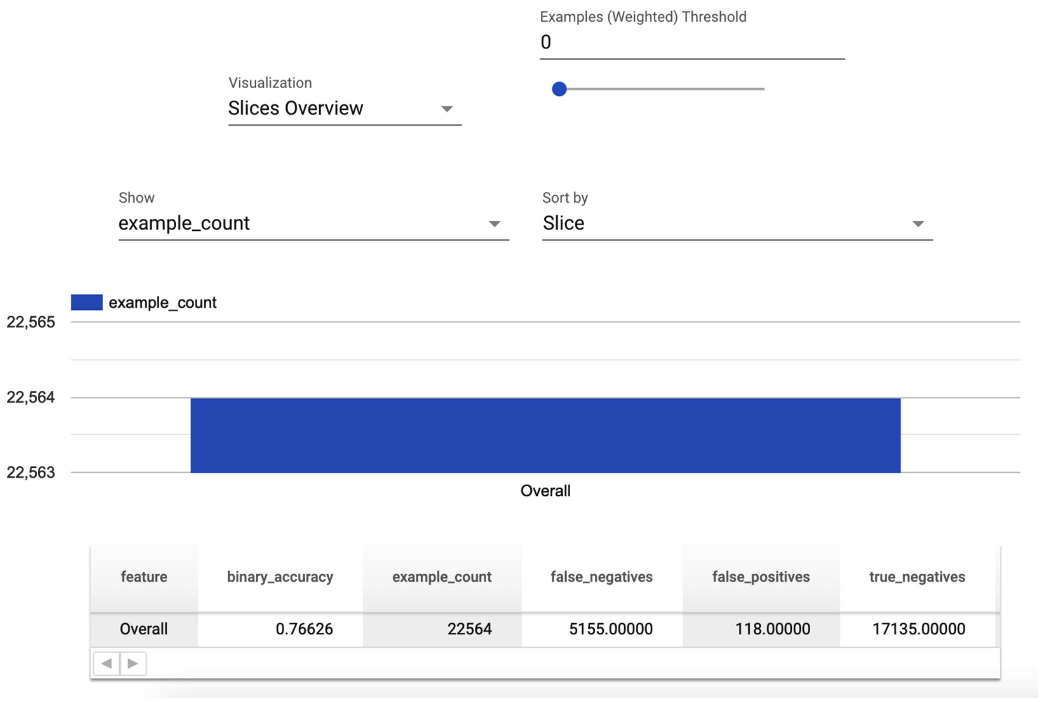
Then, run the model analysis step:

```
eval_result = tfma.run_model_analysis(
    eval_shared_model=eval_shared_model,
    eval_config=eval_config,
    data_location=_EVAL_DATA_FILE,
    output_path=_EVAL_RESULT_LOCATION,
    file_format='tfrecords')
```

And view the results in a Jupyter Notebook:

```
tfma.view.render_slicing_metrics(eval_result)
```

Even though we want to view the overall metrics, we still call `render_slicing_metrics`. The slice in this context is the *overall slice*, which is the entire dataset. The result is shown in Figure 7-4.



feature	binary_accuracy	example_count	false_negatives	false_positives	true_negatives
Overall	0.76626	22564	5155.00000	118.00000	17135.00000

Figure 7-4. TFMA notebook visualization for overall metrics

USING TFMA IN A JUPYTER NOTEBOOK

TFMA works as previously described in a Google Colab notebook. But a few extra steps are required to view the visualizations in a standalone Jupyter Notebook. Install and enable the TFMA notebook extension with:

```
$ jupyter nbextension enable --py widgetsnbextension
$ jupyter nbextension install --py \
    --symlink tensorflow_model_analysis
$ jupyter nbextension enable --py tensorflow_model_analysis
```

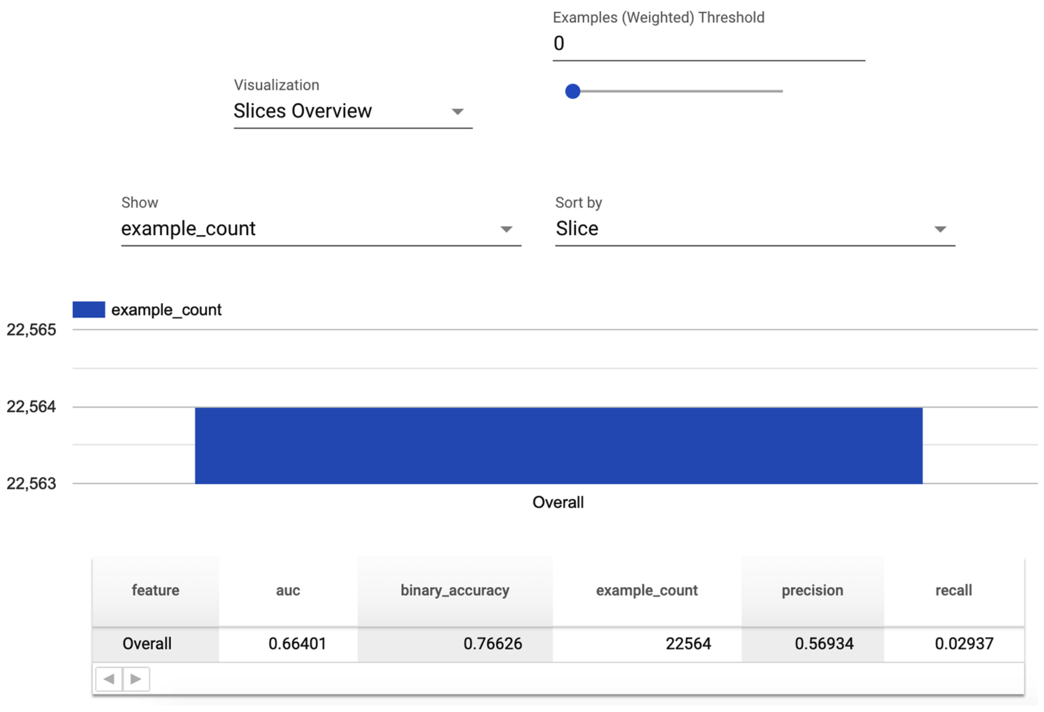
Append the flag `--sys_prefix` to each of these commands if you are running them in a Python virtual environment. The `widgetsnbextension`, `ipywidgets`, and `jupyter_nbextensions_configurator` packages may also require installation or upgrading.

At the time of writing, TFMA visualizations are not available in Jupyter Lab, only in Jupyter Notebook.

All the metrics we described in "How to Analyze Your Model" can be displayed in TFMA by providing them in the `metrics_specs` argument to the `EvalConfig`:

```
metrics_specs=[
    tfma.MetricsSpec(metrics=[
        tfma.MetricConfig(class_name='BinaryAccuracy'),
        tfma.MetricConfig(class_name='AUC'),
        tfma.MetricConfig(class_name='ExampleCount'),
        tfma.MetricConfig(class_name='Precision'),
        tfma.MetricConfig(class_name='Recall')
    ])
]
```

The results are shown in Figure 7-5.



feature	auc	binary_accuracy	example_count	precision	recall
Overall	0.66401	0.76626	22564	0.56934	0.02937

Figure 7-5. TFMA notebook visualization for other metrics

Analyzing Multiple Models in TFMA

We can also use TFMA to compare our metrics across multiple models. For example, these may be the same model trained on different datasets, or two models with different hyperparameters trained on the same dataset.

For the models we compare, we first need to generate an `eval_result` similar to the preceding code examples. We need to ensure we specify an `output_path` location to save the models. We use the same `EvalConfig` for both models so that we can calculate the same metrics:

```
eval_shared_model_2 = tfma.default_eval_shared_model(
    eval_saved_model_path=_EVAL_MODEL_DIR, tags=[tf.saved_model.SERVING])

eval_result_2 = tfma.run_model_analysis(
    eval_shared_model=eval_shared_model_2,
    eval_config=eval_config,
    data_location=_EVAL_DATA_FILE,
    output_path=_EVAL_RESULT_LOCATION_2,
    file_format='tfrecords')
```

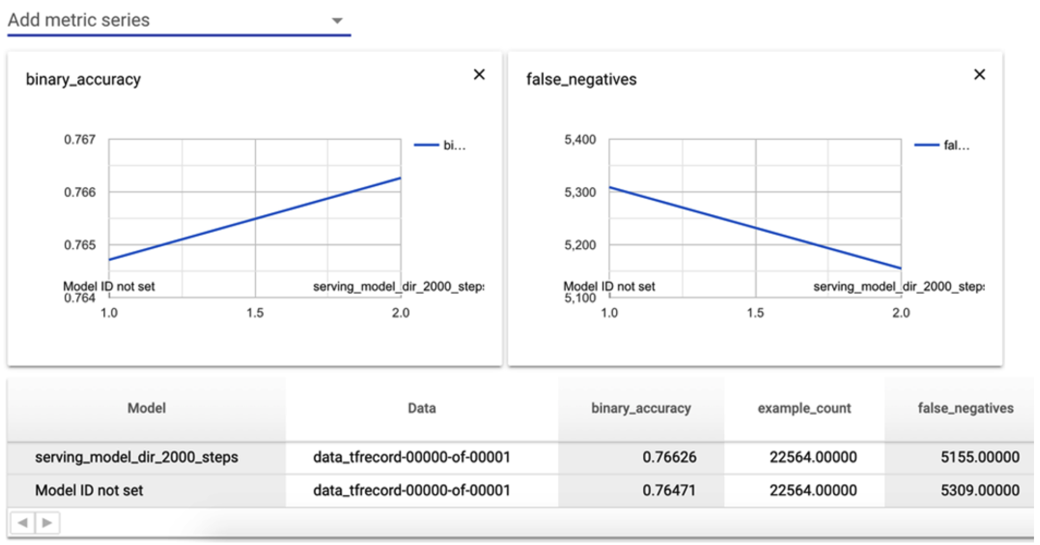
Then, we load them using the following code:

```
eval_results_from_disk = tfma.load_eval_results(
    [_EVAL_RESULT_LOCATION, _EVAL_RESULT_LOCATION_2],
    tfma.constants.MODEL_CENTRIC_MODE)
```

And we can visualize them using:

```
tfma.view.render_time_series(eval_results_from_disk, slices[0])
```

The result is shown in Figure 7-6.



Model	Data	binary_accuracy	example_count	false_negatives
serving_model_dir_2000_steps	data_tfrecord-00000-of-00001	0.76626	22564.00000	5155.00000
Model ID not set	data_tfrecord-00000-of-00001	0.76471	22564.00000	5309.00000

Figure 7-6. TFMA visualization comparing two models

The key thing to note here is that for both classification and regression models in TFMA it is possible to view many metrics at once, rather than being restricted to one or two during training. This helps to prevent surprising behavior once the model is deployed.

We can also slice the evaluation data based on features of the dataset, for example, to get the accuracy by product in our demo project. We'll describe how to do this in the following section.

Model Analysis for Fairness

All the data that we use to train a model is biased in some way: the real world is an incredibly complex place, and it's impossible to take a sample of data that adequately captures all this complexity. In Chapter 4, we looked at bias in data on the way into our pipeline, and in this chapter we'll look at whether the model's predictions are fair.

FAIRNESS AND BIAS

The terms "fairness" and "bias" are often used interchangeably to refer to whether different groups of people experience different performance from a machine learning model. Here, we'll use the term "fairness" to avoid confusion with data bias, which we discussed in Chapter 4.

To analyze whether our model is fair, we need to identify when some groups of people get a different experience than others in a problematic way. For example, a group of people could be people who don't pay back loans. If our model is trying to predict who should be extended credit, this group of people should have a different experience than others. An example of the type of problem we want to avoid is when the only people who are incorrectly turned down for loans are of a certain race.

A high-profile example of groups getting different experiences from a model is the COMPAS algorithm that predicts recidivism risk. As reported by Propublica, the algorithm's error rate was roughly the same for black and white defendants. However, it was especially likely to incorrectly predict that black defendants would be future criminals, at roughly twice the rate of the same incorrect prediction for white defendants.

We should try to recognize such problems before our models are deployed to production. To start with, it's useful to define numerically what we mean by *fairness*. Here are several example methods for classification problems:

Demographic parity

> Decisions are made by the model at the same rate for all groups. For example, men and women would be approved for a loan at the same rate.

Equal opportunity

The error rate in the opportunity-giving class is the same for all groups. Depending on how the problem is set up, this can be the positive class or the negative class. For example, of the people who can pay back a loan, men and women would be approved for a loan at the same rate.

Equal accuracy

Some metrics such as accuracy, precision, or AUC are equal for all groups. For example, a facial recognition system should be as accurate for dark-skinned women as it is for light-skinned men.

Equal accuracy can sometimes be misleading, as in the preceding COMPAS example. In that example, the accuracy was equal for both groups, but the consequences were much higher for one group. It's important to consider the directions of errors that have the highest consequences for your model.

DEFINITIONS OF FAIRNESS

There is no one definition of fairness that is appropriate for all machine learning projects. You will need to explore what is best for your specific business problem, taking into account potential harms and benefits to the users of your model. More guidance is provided in the book *Fairness in Machine Learning* by Solon Barocas et al., this article by Martin Wattenberg et al. from Google, and the Fairness Indicators documentation by Ben Hutchinson et al.

The groups we're referring to can be different types of customers, product users from different countries, or people of different gender and ethnicity. In US law, there is the concept of *protected groups*, where individuals are protected from discrimination based on the groups of gender, race, age, disability, color, creed, national origin, religion, and genetic information. And these groups intersect: you may need to check that your model does not discriminate against multiple combinations of groups.

GROUPS ARE A SIMPLIFICATION

Groups of people are never clean-cut in the real world. Everyone has their own complex story: someone may have changed their religion or their gender during their lifetime. Someone may belong to multiple races or multiple nationalities. Look for these edge cases and give people ways to tell you if they have a poor experience with your model.

Even if you are not using these groups as features in your model, this does not mean that your model is fair. Many other features, such as location, might be strongly correlated with one of these protected groups. For example, if you use a US zip code as a feature, this is highly correlated with race. You can check for these problems by slicing your data for one of the protected groups, as we describe in the following section, even if it is not a feature that you have used to train your model.

Fairness is not an easy topic to deal with, and it leads us into many ethical questions that may be complex or controversial. However, there are several projects that can help us analyze our models from a fairness perspective, and we'll describe how you can use them in the next few sections. This kind of analysis can give you an ethical and a commercial advantage by giving everyone a consistent experience. It may even be a chance to correct underlying unfairness in the system that you are modeling—for example, analyzing a recruiting tool at Amazon revealed an underlying disadvantage experienced by female candidates.

In the next sections, we will describe how to use three projects for evaluating fairness in TensorFlow: TFMA, Fairness Indicators, and the What-If Tool.

Slicing Model Predictions in TFMA

The first step in evaluating your machine learning model for fairness is slicing your model's predictions by the groups you are interested in—for example, gender, race, or country. These slices can be generated by TFMA or the Fairness Indicators tools.

To slice data in TFMA, a slicing column must be provided as a `SliceSpec`. In this example, we'll slice on the *Product* feature:

```
slices = [tfma.slicer.SingleSliceSpec(),
          tfma.slicer.SingleSliceSpec(columns=['Product'])]
```

`SingleSliceSpec` with no specified arguments returns the entire dataset.

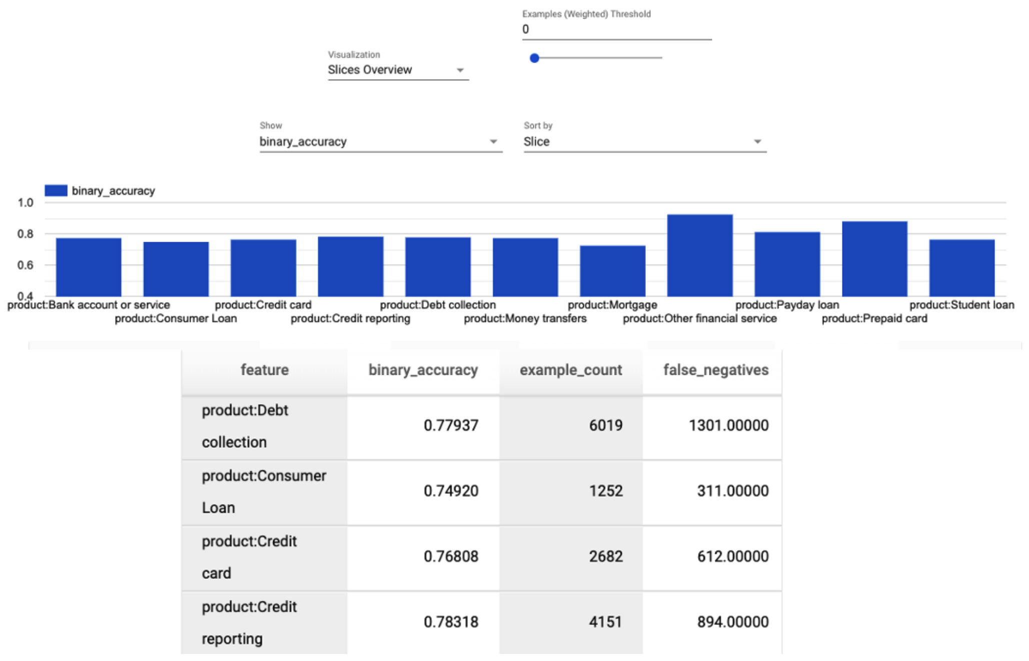
Next, run the model analysis step with the slices specified:

```
eval_result = tfma.run_model_analysis(
    eval_shared_model=eval_shared_model,
    eval_config=eval_config_viz,
    data_location=_EVAL_DATA_FILE,
    output_path=_EVAL_RESULT_LOCATION,
    file_format='tfrecords',
    slice_spec = slices)
```

And view the results, as shown in Figure 7-7:

```
tfma.view.render_slicing_metrics(eval_result, slicing_spec=slices[1])
```

If we want to consider Demographic parity, as defined previously, then we need to check whether the same proportion of people in each group are in the positive class. We can check this by looking at the TPR and the TNR for each group.



Examples (Weighted) Threshold: 0

Visualization: Slices Overview

Show: binary_accuracy

Sort by: Slice

feature	binary_accuracy	example_count	false_negatives
product:Debt collection	0.77937	6019	1301.00000
product:Consumer Loan	0.74920	1252	311.00000
product:Credit card	0.76808	2682	612.00000
product:Credit reporting	0.78318	4151	894.00000

Figure 7-7. TFMA slicing visualization

CONSIDER WHICH CLASS IS BENEFICIAL

We're assuming that there is some choice made by the model that is beneficial to a person, and we're assuming that this is the positive class. If the positive class is not beneficial and the negative class is harmful to a person, we should consider the true negative rate and the false positive rate instead.

For equal opportunity, we can check the FPR for each group. For more details on this, the Fairness Indicators project has useful advice.

Checking Decision Thresholds with Fairness Indicators

Fairness Indicators is another extremely useful tool for model analysis. It has some overlapping capabilities with TFMA, but one particularly useful feature of it is the ability to view metrics sliced on features at a variety of decision thresholds. As we discussed previously, the decision threshold is the probability score at which we draw the boundary between classes for a classification model. This lets us check if our model is fair to groups at different decision thresholds.

There are several ways to access the Fairness Indicators tool, but the simplest way to use it as a standalone library is via TensorBoard. We also mention how to load it as part of a TFX pipeline in "Evaluator Component". We install the TensorBoard Fairness Indicators plug-in via:

```
$ pip install tensorboard_plugin_fairness_indicators
```

Next, we use TFMA to evaluate the model and ask it to calculate metrics for a set of decision thresholds we supply. This is supplied to TFMA in the `metrics_spec` argument for the `EvalConfig`, along with any other metrics we wish to calculate:

```
eval_config=tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='consumer_disputed')],
    slicing_specs=[tfma.SlicingSpec(),
                   tfma.SlicingSpec(feature_keys=['product'])],
    metrics_specs=[
        tfma.MetricsSpec(metrics=[
            tfma.MetricConfig(class_name='FairnessIndicators',
                              config='{"thresholds":[0.25, 0.5, 0.75]}')
            ])
        ]
    )
```

Then run the model analysis step as before via `tfma.run_model_analysis`.

Next, write the TFMA evaluation result to a log directory so that it can be picked up by TensorBoard:

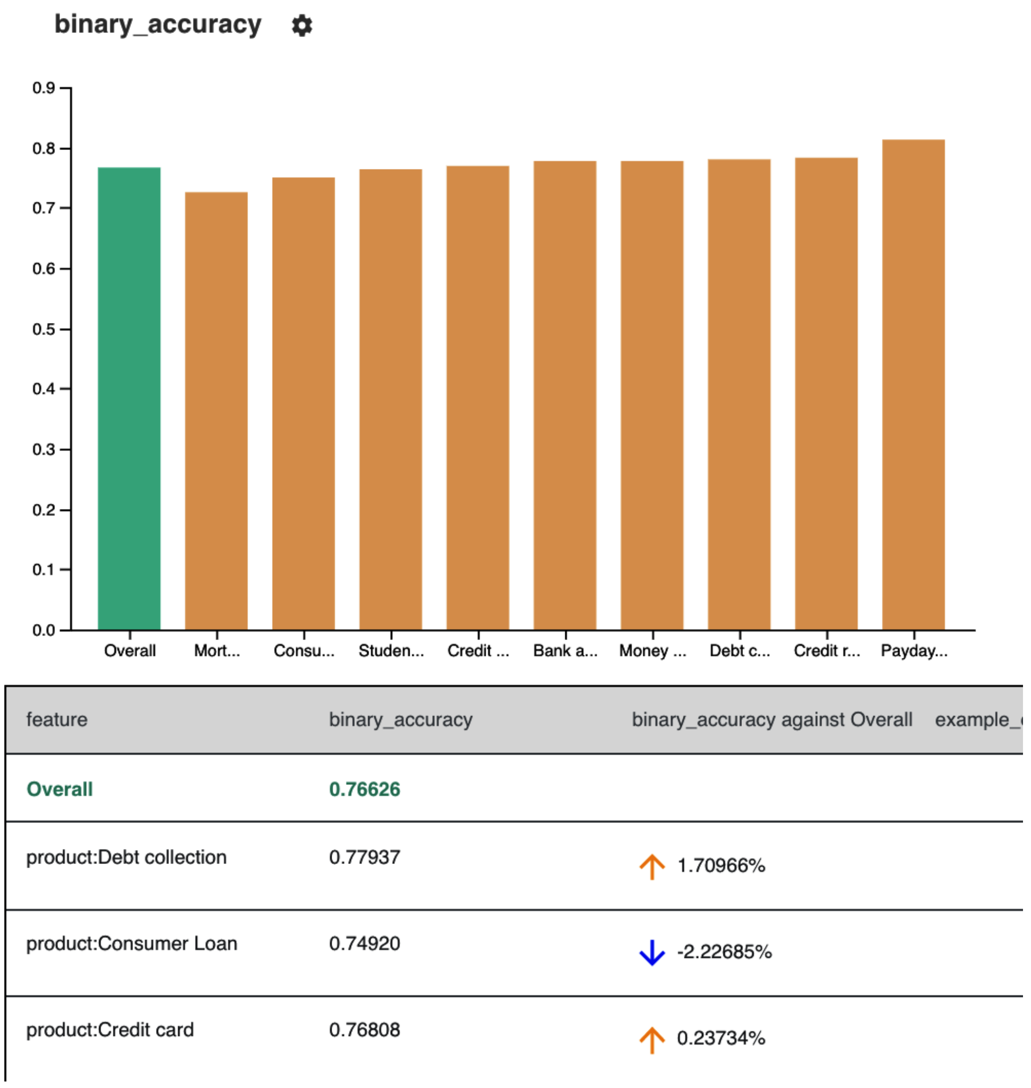
```
from tensorboard_plugin_fairness_indicators import summary_v2

writer = tf.summary.create_file_writer('./fairness_indicator_logs')
with writer.as_default():
    summary_v2.FairnessIndicators('./eval_result_fairness', step=1)
writer.close()
```

And load the result in TensorBoard to a Jupyter Notebook:

```
%load_ext tensorboard
%tensorboard --logdir=./fairness_indicator_logs
```

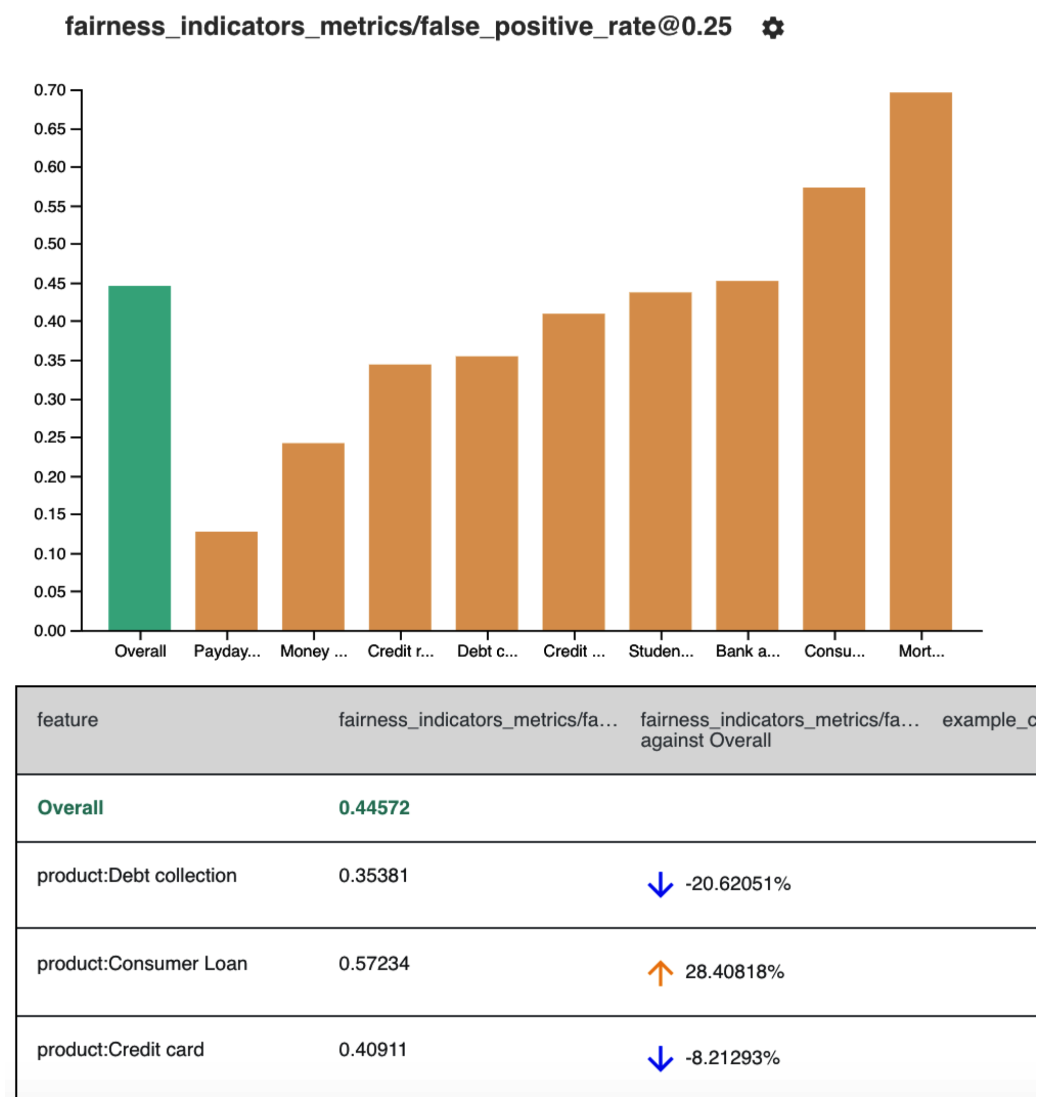
The Fairness Indicators tool highlights variations from the overall metric value, as shown in Figure 7-8.



feature	binary_accuracy	binary_accuracy against Overall	example_
Overall	**0.76626**		
product:Debt collection	0.77937	↑ 1.70966%	
product:Consumer Loan	0.74920	↓ -2.22685%	
product:Credit card	0.76808	↑ 0.23734%	

Figure 7-8. Fairness Indicators slicing visualization

And for our example project, Figure 7-9 shows more extreme differences between groups when the decision threshold is reduced to 0.25.



feature	fairness_indicators_metrics/fa…	fairness_indicators_metrics/fa… against Overall	example_c
Overall	**0.44572**		
product:Debt collection	0.35381	↓ -20.62051%	
product:Consumer Loan	0.57234	↑ 28.40818%	
product:Credit card	0.40911	↓ -8.21293%	

Figure 7-9. Fairness Indicators threshold visualization

In addition to exploring fairness considerations in the overall model, we might want to look at individual data points to see how individual users are affected by our model. Fortunately, there is another tool in the TensorFlow ecosystem to help us with this: the What-If Tool.

Going Deeper with the What-If Tool

After looking at slices of our dataset with TFMA and Fairness Indicators, we can go into greater detail using another project from Google: the What-If Tool (WIT). This lets us generate some very useful visualizations and also investigate individual data points.

There are a number of ways to use the WIT with your model and data. It can be used to analyze a model that has already been deployed with TensorFlow Serving via TensorBoard, or a model that is running on GCP. It can also be used directly with an Estimator model. But for our example project, the most straightforward way to use it is to write a *custom predic-*

tion function that takes in a list of training examples and returns the model's predictions for these examples. This way, we can load the visualizations in a standalone Jupyter Notebook.

We can install the WIT with:

```
$ pip install witwidget
```

Next, we create a `TFRecordDataset` to load the data file. We sample 1,000 training examples and convert it to a list of `TFExamples`. The visualizations in the What-If Tool work well with this number of training examples, but they get harder to understand with a larger sample:

```
eval_data = tf.data.TFRecordDataset(_EVAL_DATA_FILE)
subset = eval_data.take(1000)
eval_examples = [tf.train.Example.FromString(d.numpy()) for d in subset]
```

Next, we load the model and define a prediction function that takes in the list of `TFExamples` and returns the model's predictions:

```
model = tf.saved_model.load(export_dir=_MODEL_DIR)
predict_fn = model.signatures['serving_default']

def predict(test_examples):
    test_examples = tf.constant([example.SerializeToString() for example in exampl
    preds = predict_fn(examples=test_examples)
    return preds['outputs'].numpy()
```

Then we configure the WIT using:

```
from witwidget.notebook.visualization import WitConfigBuilder

config_builder = WitConfigBuilder(eval_examples).set_custom_predict_fn(predict)
```

And we can view it in a notebook using:

```
from witwidget.notebook.visualization import WitWidget

WitWidget(config_builder)
```

This will give us the visualization in Figure 7-10.

Figure 7-10. WIT front page

USING THE WIT IN A JUPYTER NOTEBOOK

As with TFMA, a few extra steps are required to run the WIT in a standalone notebook. Install and enable the WIT notebook extension with:

```
$ jupyter nbextension install --py --symlink \
    --sys-prefix witwidget

$ jupyter nbextension enable witwidget --py --sys-prefix
```

Append the flag `--sys_prefix` to each of these commands if you are running them in a Python virtual environment.

There are many functions included in the WIT, and we will describe some of the most useful here. Full documentation is provided at the WIT project home page.

The WIT provides *counterfactuals*, which for any individual training example show its nearest neighbor from a different classification. All the features are as similar as possible, but the model's prediction for the counterfactual is the other class. This helps us see how each feature impacts the model's prediction for the particular training example. If we see that changing a demographic feature (race, gender, etc.) changes the model's prediction to the other class, this is a warning sign that the model may not be fair to different groups.

We can explore this feature further by editing a selected example in the browser. Then we can rerun the inference and see what effect this has on the predictions made for the specific example. This can be used to explore demographic features for fairness or any other features to see what happens if they are changed.

Counterfactuals can also be used as explanations for the model's behavior. But note that there may be many possible counterfactuals for each data point that are close to being the nearest neighbor and also that there are likely to be complex interactions between features. So counterfactuals themselves should not be presented as if they completely explain the model's behavior.

Another particularly useful feature of the WIT is partial dependence plots (PDPs). These show us how each feature impacts the predictions of the model, for example, whether an increase in a numeric feature changes the class prediction probability. The PDP shows us the shape of this dependence: whether it is linear, monotonic, or more complex. PDPs can also be generated for categorical features, as shown in Figure 7-11. Again, if the model's predictions show a dependence on a demographic feature, this may be a warning that your model's predictions are unfair.

Figure 7-11. WIT PDPs

A more advanced feature, for which we won't dive into detail here, is to optimize decision thresholds for fairness strategies. This is provided as a page in the WIT, and it's possible to automatically set decision thresholds based on a chosen strategy, as shown in Figure 7-12.

Figure 7-12. WIT decision thresholds

All of the tools we describe in this section on model fairness can also be used to interrogate any model even if it does not have the potential to harm users. They can be used to get a much better understanding of a model's behavior before it is deployed and can help avoid surprises when it reaches the real world. This is an area of active research, and new tools are released frequently. One interesting development is *constrained optimization* for model fairness, in which instead of just optimizing for one metric, models can be optimized by taking into consideration other constraints, such as equal accuracy. An experimental library exists for this in TensorFlow.

Model Explainability

Discussing fairness and using the WIT naturally leads us to discussing how we can not only describe the performance of our model but also explain what is going on inside it. We mentioned this briefly in the previous section on fairness, but we'll expand on it a little more here.

Model *explainability* seeks to explain why the predictions made by a model turn out that way. This is in contrast to *analysis*, which describes the model's performance with respect to various metrics. Explainability for machine learning is a big topic, with a lot of active research on the subject happening right now. It isn't something we can automate as part of our pipeline because, by definition, the explanations need to be shown to people. We will just give you a brief overview, and for more details we recommend the ebook *Interpretable Machine Learning* by Christoph Molnar and this whitepaper from Google Cloud.

There are a few possible reasons for seeking to explain your model's predictions:

- Helping a data scientist debug problems with their model
- Building trust in the model
- Auditing models
- Explaining model predictions to users

The techniques we discuss further on are helpful in all these use cases.

Predictions from simpler models are much easier to explain than predictions from complex models. Linear regression, logistic regression, and single decision trees are relatively easy to interpret. We can view the weights for each feature and know the exact contribution of the feature. For these models, looking at the entire model provides an explanation because their architecture makes them interpretable by design, so that a human can understand the entire thing. For example, the coefficients from a linear regression model give an explanation that is understandable with no further processing required.

It is more difficult to explain random forests and other ensemble models, and deep neural networks are the hardest of all to explain. This is due to the enormous number of parameters and connections in a neural network, which results in extremely complex interactions between features. If your model's predictions have high consequences and you require explanations, we recommend you choose models that are easier to explain. You can find more details on how and when to use explanations in the paper "Explainable Machine Learning in Deployment" by Umang Bhatt et al.

LOCAL AND GLOBAL EXPLANATIONS

We can divide ML explainability methods into two broad groups: local and global explanations. *Local explanations* seek to explain why a model made a particular prediction for one single data point. *Global explanations* seek to explain how a model works overall, measured using a large set of data points. We will introduce techniques for both of these in the next section.

In the next section, we will introduce some techniques for generating explanations from your model.

Generating Explanations with the WIT

In "Going Deeper with the What-If Tool", we described how we could use the WIT to help with our model fairness questions. But the WIT is also useful for explaining our models—in particular, using counterfactuals and PDPs, as we noted. Counterfactuals provide us with local explanations, but PDPs can provide either local or global explanations. We showed an example of global PDPs previously in Figure 7-11, and now we'll consider local PDPs, as shown in Figure 7-13.

Figure 7-13. WIT local PDPs

PDPs show the change in prediction results (the *inference score*) for different valid values of a feature. There is no change in the inference score across the `company` feature, showing that the predictions for this data point don't depend on the value of this feature. But for the `company_response` feature, there is a change in the inference score, which shows that the model prediction has some dependence on the value of the feature.

ASSUMPTIONS WITH PDPS

PDPs contain an important assumption: all features are independent of each other. For most datasets, especially those that are complex enough to need a neural network model to make accurate predictions, this is not a good assumption. These plots should be approached with caution: they can give you an indication of what your model is doing, but they do not give you a complete explanation.

If your model is deployed using Google Cloud's AI Platform, you can see *feature attributions* in the WIT. For a single data example, feature attributions provide positive or negative scores for each feature for each feature, which indicate the effect and magnitude of the feature's contributions to a model's prediction. They can also be aggregated to provide global explanations of the feature's importance in your model. The feature attributions are based on *Shapley values*, which are described in the following section. Shapley values do not assume that your features are independent, so unlike PDPs, they are useful if your features are correlated with each other. At the time of writing, feature attributions were only available for models trained using TensorFlow 1.x.

Other Explainability Techniques

LIME, or local interpretable model-agnostic explanations, is another method for producing local explanations. It treats a model as a black box and generates new data points around the point that we would like to get an explanation for. LIME then obtains predictions from the model for these new data points and trains a simple model using these points. The weights for this simple model give us the explanations.

The SHAP, or Shapley Additive Explanations, library provides both global and local explanations using Shapley values. These are computationally expensive to calculate, so the SHAP library contains implementations that speed up calculations or calculate approximations for boosted trees and deep neural networks. This library is a great way to show a feature's importance for your models.

SHAPLEY VALUES

Shapley values are useful for both local and global explanations. This concept is an algorithm borrowed from game theory that distributes gains and losses across each player in a cooperative game for some outcome of the game. In a machine learning context, each feature is a "player," and the Shapley values can be obtained as follows:

1. Get all the possible subsets of features that don't contain feature F.
2. Compute the effect on the model's predictions of adding F to all the subsets.
3. Combine these effects to get the importance of feature F.

These are all relative to some *baseline*. For our example project, we could pose this as "how much was a prediction driven by the fact that the `company_response` was `Closed with explanation` instead of `Closed with monetary relief`." The value `Closed with monetary relief` is our baseline.

We would also like to mention model cards, a framework for reporting on machine learning models. These are a formal way of sharing facts and limitations about machine learning models. We include these here because even though they don't explain why the model makes its predictions, they are extremely valuable for building trust in a model. A model card should include the following pieces of information:

- Benchmarked performance of the model on public datasets, including performance sliced across demographic features
- Limitations of the model, for example, disclosing if a decrease in image quality would produce a less accurate result from an image classification model
- Any trade-offs made by the model, for example, explaining if a larger image would require longer processing time

Model cards are extremely useful for communicating about models in high-stakes situations, and they encourage data scientists and machine learning engineers to document the use cases and limitations of the models they build.

LIMITATIONS OF EXPLANATIONS

We recommend that you proceed with caution when tackling model explainability. These techniques may give you a warm and happy feeling that you understand what your model is doing, but it may actually be doing something very complex that is impossible to explain. This is especially the case with deep learning models.

It just isn't feasible to represent all the complexity of the millions of weights that make up a deep neural network in a way that is human readable. In situations where model decisions have high real-world consequences, we recommend building the simplest models possible with features that are easy to explain.

Analysis and Validation in TFX

Up to this point in this chapter, we've focused on model analysis with a human in the loop. These tools are extremely useful for monitoring our models and checking that they are behaving in the way we want. But in an automated machine learning pipeline, we want the pipeline to run smoothly and alert us to problems. There are several components in TFX that handle this part of the pipeline: the `Resolver`, the `Evaluator`, and the `Pusher`. Together, these components check the model's performance on an evaluation dataset and send it to a serving location if it improves the previous model.

TFX uses the concept of *blessing* to describe the gating process for deciding whether or not to deploy a model for serving. If a model improves the previous model, based on a threshold we define, it is blessed and can move forward to the next step.

ResolverNode

A `Resolver` component is required if we want to compare a new model against a previous model. `ResolverNodes` are generic components that query the metadata store. In this case, we use the `latest_blessed_model_resolver`. It checks for the last blessed model and returns it as a baseline so it can be passed on to the `Evaluator` component with the new candidate model. The `Resolver` is not needed if we don't want to validate our model against a threshold of some metric, but we highly recommend this step. If you don't validate the new model, it will automatically get pushed to the serving directory, even if its performance is worse than that of the previous model. On the first run of the `Evaluator` when there is no blessed model, the `Evaluator` automatically blesses the model.

In an interactive context, we can run the `Resolver` component as follows:

```
from tfx.components import ResolverNode
from tfx.dsl.experimental import latest_blessed_model_resolver
from tfx.types import Channel
from tfx.types.standard_artifacts import Model
from tfx.types.standard_artifacts import ModelBlessing

model_resolver = ResolverNode(
    instance_name='latest_blessed_model_resolver',
    resolver_class=latest_blessed_model_resolver.LatestBlessedModelResolver,
    model=Channel(type=Model),
    model_blessing=Channel(type=ModelBlessing)
)
context.run(model_resolver)
```

Evaluator Component

The `Evaluator` component uses the TFMA library to evaluate a model's predictions on a validation dataset. It takes as input data from the ExampleGen component, the trained model from the `Trainer` component, and an `EvalConfig` for TFMA (the same as when using TFMA as a standalone library).

First, we define the `EvalConfig`:

```
import tensorflow_model_analysis as tfma

eval_config=tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='consumer_disputed')],
    slicing_specs=[tfma.SlicingSpec(),
                   tfma.SlicingSpec(feature_keys=['product'])],
    metrics_specs=[
        tfma.MetricsSpec(metrics=[
            tfma.MetricConfig(class_name='BinaryAccuracy'),
            tfma.MetricConfig(class_name='ExampleCount'),
            tfma.MetricConfig(class_name='AUC')
        ])
    ]
)
```

Then we run the `Evaluator` component:

```
from tfx.components import Evaluator

evaluator = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer.outputs['model'],
    baseline_model=model_resolver.outputs['model'],
    eval_config=eval_config
)
context.run(evaluator)
```

We can also show the TFMA visualization with:

```
eval_result = evaluator.outputs['evaluation'].get()[0].uri
tfma_result = tfma.load_eval_result(eval_result)
```

And we can load Fairness Indicators with:

```
tfma.addons.fairness.view.widget_view.render_fairness_indicator(tfma_result)
```

Validation in the Evaluator Component

The `Evaluator` component also carries out validation, in which it checks whether the candidate model we have just trained is an improvement on a baseline model (such as the model that is currently in production). It obtains predictions from both models on an evaluation dataset and compares a performance metric (e.g., model accuracy) from both models. If the new model is an improvement on the previous model, the new model receives a "blessing" artifact. Currently it is only possible to calculate the metric on the whole evaluation set, not on slices.

To carry out validation, we need to set a threshold in the `EvalConfig`:

```
eval_config=tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='consumer_disputed')],
    slicing_specs=[tfma.SlicingSpec(),
                   tfma.SlicingSpec(feature_keys=['product'])],
    metrics_specs=[
        tfma.MetricsSpec(
            metrics=[
                tfma.MetricConfig(class_name='BinaryAccuracy'),
                tfma.MetricConfig(class_name='ExampleCount'),
                tfma.MetricConfig(class_name='AUC')
            ],
            thresholds={
                'AUC':
                    tfma.config.MetricThreshold(
                        value_threshold=tfma.GenericValueThreshold(
                            lower_bound={'value': 0.65}),
                        change_threshold=tfma.GenericChangeThreshold(
                            direction=tfma.MetricDirection.HIGHER_IS_BETTER,
                            absolute={'value': 0.01}
                        )
                    )
            }
        )
    ]
)
```

In this example, we state that the AUC must be over 0.65, and we want the model to be validated if its AUC is at least 0.01 higher than that of the baseline model. Any other metric can be added in place of AUC, but note that the metric you add must also be included in the list of `metrics` in the `MetricsSpec`.

We can check the results with:

```
eval_result = evaluator.outputs['evaluation'].get()[0].uri
print(tfma.load_validation_result(eval_result))
```

If the validation check passes, it will return the following result:

```
validation_ok: true
```

TFX Pusher Component

The `Pusher` component is a small but important part of our pipeline. It takes as input a saved model, the output of the `Evaluator` component, and a file path for the location our models will be stored for serving. It then checks whether the `Evaluator` has blessed the model (i.e., the model is an improvement on the previous version, and it is above any thresholds we have set). If it has been blessed, the `Pusher` pushes the model to the serving file path.

The `Pusher` component is provided with the model `Evaluator` outputs and the serving destination:

```
from tfx.components import Pusher
from tfx.proto import pusher_pb2

_serving_model_dir = "serving_model_dir"

pusher = Pusher(
    model=trainer.outputs['model'],
    model_blessing=evaluator.outputs['blessing'],
    push_destination=pusher_pb2.PushDestination(
        filesystem=pusher_pb2.PushDestination.Filesystem(
            base_directory=_serving_model_dir)))
context.run(pusher)
```

Once the new model has been pushed to the serving directory, it can then be picked up by TensorFlow Serving—see the next chapter for more details on this.

Summary

In this chapter, we saw how to analyze a model's performance in much greater detail than during model training, and we started thinking about how to make a model's performance fair. We also discussed the process for checking that a model's performance is an improvement on the previously deployed model. We also introduced explainability for machine learning and gave a brief overview of some of the techniques in this area.

We must advise caution here, though: just because you've analyzed your model's performance in detail with Fairness Indicators, this doesn't guarantee that your model is fair or ethically sound. It's important to keep monitoring your model once it is in production and provide ways for your users to let you know if they feel that the model’s predictions are unjust. This is especially important when the stakes are high and the model’s decisions have the potential to cause large real-world impacts to users.

Now that our model has been analyzed and validated, it’s time to move on to the crucial next step in the pipeline: serving the model! The next two chapters will tell you all you need to know about this important step.