Azure Machine Learning

Flight Delay ML Demo

With Power BI

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2021 Microsoft. All rights reserved.

# Introduction

Delays are often considered as an inevitable part of airline travel. Their impact ranges from a minor irritant through to tangible financial loss through missed connections & work opportunities. For a large business, this can result in a material impact to their bottom line as well as having a negative impact on the morale of their employees.

Today, we’ll examine a solution to this challenge by using data to predict the risk of delays and thus provide an organization the opportunity to optimize their travel. We’ll see how the Azure Platform, and in particular Azure Machine Learning, can be used to ingest & prepare data, train & deploy a model, and operationalize an end-to-end solution to this ever-challenging problem.

For the purposes of our analysis, a flight delay will be defined as a flight that arrives more than fifteen minutes late.

# Datasets

### Primary

These datasets are used as part of the main story flow.

[Flight Delay](http://stat-computing.org/dataexpo/2009/the-data.html), with supplementary resources:

* [Airports](http://stat-computing.org/dataexpo/2009/airports.csv)
* [Carriers](http://stat-computing.org/dataexpo/2009/carriers.csv)
* [Airplanes](http://stat-computing.org/dataexpo/2009/plane-data.csv)

[Daymet Weather](https://azure.microsoft.com/en-us/services/open-datasets/catalog/daymet/) (via Azure Open Datasets)

[Bird Object Detection](https://github.com/olgaliak/detection-amlworkbench/tree/master/assets)

### Secondary

These datasets complement the main story flow to show *brief* examples of features not well highlighted by the primary datasets.

[Shap Census Dataset](https://github.com/slundberg/shap)

[Boat/Car Bounding Box](http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html) & [MIT Indoor 5 Multi-class](http://web.mit.edu/torralba/www/indoor.html)

# Setup

Jupyter Notebook tasks will take place in Azure ML compute instances. Ensure you provision one in your workspace before starting.

Setup VS Code locally:

1. From the command prompt, change to the “$/notebooks/flight-delay-classicalml-local” folder
2. From the command prompt, start Visual Studio Code: “code .”
3. Open flight-delay-classicalml-local.ipynb and scroll down to “Data exploration and visualization”

Open Power BI Desktop (and ensure it’s the latest version):

1. Open $/powerbi/Flight Delay Report 2020.pbix

**Note:** Before starting with the following script please make sure that the notebook in scope has been executed successfully end to end. This will allow you to review output or execute specific cells during the demo.

# 

# Demo

### 0. Preparing data

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Open Azure ML portal to show the datastore (datalakestoragegene2). | Let’s start in Azure ML. For data scientists, the first task is typically to source the data needed for the experiment. The Data Lake has been attached to their Azure ML workspace, allowing the data scientist to seamlessly access the work of their data engineering colleagues. |
|  | Click on “Datasets”.  Click on “flightdelayweather\_ds”.  Scroll down to preview data.  Click on “Consume”.  Click on “Explore”. | While our input data starts life as plain files, the reality is that it’s much more: a dataset needs to be managed, versioned, & controlled like other assets in an ML pipeline. Datasets provide this functionality. We can register the output of our data preparation efforts as a dataset and then it becomes a managed - and shareable - resource.  As you can see, the dataset is versioned, provides quick start instructions, and even enables users to preview rows inline. |
|
|  | Click on “Profile”. | For this dataset, we’ve already created a profile. This helps us understand our data at a glance, with summary statistics and distribution visualizations. |
|  | Open Azure Open Datasets.  Search for “weather”.  Click “Daymet”.  Click “Data access”. | In many scenarios, public datasets are an essential complement to corporate data. Azure Open Datasets make it easy to find and integrate these into your ML workflow.  For example, weather data might be a useful supplement to this machine learning problem. We can find several different weather datasets and even see how to access the data via a downloadable Jupyter Notebook. |

### 1. AutoML UX

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Start in Power BI Online and show the “Analysis” tab.  Optionally drag the month slider and mouse over states. | Let’s first consider this problem through the eyes of a citizen data scientist. With the data prepared by our colleagues, the first step will be to explore and identify whether there might be predictive value. With Power BI, getting insights is quick and easy. |
|  | Switch to the “Key Drivers” report tab.  Show “Key influencers”.  Click on the sample question in the Q+A panel. | We can explore our new dataset using natural language query. Through the use of standard business English, I can quickly visualize the answers to questions like the proportion of flights that will be delayed, the carriers most likely to suffer delays, and most commonly delayed destinations.  There are some complex patterns here, with delays being driven by time, weather, location, and more. Sounds like an ideal problem for machine learning! |
|  | Switch to Power BI Online.  Expand the workspace in the left nav and click on “Flight Delay Dataflow”.  Click on “Apply ML model” (the brain icon).  Click on “Add a machine learning model”. | For some users, the mechanics of machine learning - like algorithms or hyperparameter tuning - might be beyond their current skill set. Automated ML helps automate the model building process and is available to users in a number of different ways depending on their skillset.  Here in Power BI, a simplified version of the entire model creation experience is available. By combining Power BI dataflows with automated ML technology from Azure ML, we can generate a predictive model without leaving the Power BI web experience. |
|  | Click “Next”.  Select “true” as the target outcome, enter the label for true outcomes as “Delayed”, and enter the label for false outcomes as “On time”.  Click “Next”.  Click “Next”.  [Optional] Enter a name for the model and click “Save and train”. | All I need to do is identify the column to predict. The system is intelligent enough to identify the ML problem type - classification - and handle the rest, including feature encoding and algorithm selection. |
|  | Switch to the Azure ML portal.  Click “Automated ML” in the left nav.  Click “New Automated ML run”.  Select the “flightdelayweather\_ds” and click “Next”.  Select “flight-delay-exp” as the experiment, “ArrDelay15” as the target column, “cpucluster” as the compute cluster, and then click “Next”. | Alternatively, for users that want to leverage the same technology as part of a managed MLOps cycle, it's also accessible through the Azure ML portal.  Here I can directly pick up the dataset generated by our last notebook and select the model target. Like Power BI, the system has automatically identified the problem type. |
|  | Click “View featurization settings”.  Click “Cancel”.  Mouse over “Enable deep learning”. | Automated ML has correctly detected the features within my dataset and will intelligently transform them before training starts. As a more sophisticated user than when I was in Power BI, I can override this behavior with custom types and imputation approaches. For text inputs, we can even apply deep learning techniques, like BERT, to improve featurization for inputs like text. |
|
|  | Click “Time series forecasting”.  Click “View additional configuration settings”.  Expand “Additional forecasting settings”.  Open the “Country or region for holidays”.  Click “Cancel” and then “Cancel” again. | In some cases, it may be useful to pull in supplementary datasets. For example, we can easily pull in holiday datasets to enrich the training dataset with a key predictive feature.  Presenter note: it’s recommended that you use a pre-trained and pre-deployed model for the next piece as per the setup document. Alternatively, you can start training this model rather than cancelling. |
|  | Switch to Power BI Desktop and the “Predictions” tab. | Power BI is well known for its powerful analytics capability over historical data. But through its connectivity to Azure ML, we can easily pull in predictive web services. This capability allows me to mash together my knowledge of future travel bookings and identify risks of delays using ML (as we have in the lower right corner).  Let’s see how this works. |
|  | Click “Transform Data”.  Select the “predictions” data set.  Click “Azure Machine Learning”.  Click on “flight-delay-designer”.  Click “Apply”.  Scroll to the right of the grid and click on the icon at the right end of the new column.  Unselect all except “Scored Labels” and “Scored Probabilities”.  Click “Cancel”. | Power BI’s self-service data prep technology, Dataflows, can complete many different data wrangling tasks through a visual interface. As part of a data flow, Power BI users can seamlessly add ML services to run predictions over their data. |
|  | Switch to the “Predictions” tab.  Hold down the control key and then click on the arrow. | Once applied, I can pull the data I want from the response and visualize it. I can see a clear opportunity to avoid flights with high delay risk and keep my travel on schedule. |
|  | Switch back to Power BI Online.  Click the second arrow on the “Predictions” tab to show operations data. | With Power BI, I can easily combine this data for an overview of my operations. We can see the input data – like the carrier – as well as the output predictions. We can also pull in data from AppInsights to monitor service response times and result codes. |

### 2. Designer

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | In Azure ML, click on “Designer”. | Now that we’ve established the potential for ML, our citizen data scientists might want to bring their domain knowledge to build a more sophisticated model. As beginners, they’ll need a visual composition and no-code environment. Let’s see how the Azure ML platform supports us in building a flight delay prediction model.  Designer, the next iteration of Azure ML Studio, retains all of the key visual composition capabilities but leverages the common data (through datasets), experiment, training, deployment (through AKS), and management (through IAM) platform |
|  | Open “Flight Delay Experiment” (or whatever you named your experiment during setup). | There is a rich set of components to support the citizen data scientist, from data preparation through the model evaluation. |
|  | Expand “Data Transformation”. | To get started, we used out of the box data transformation capabilities to clean, select, and transform the data. As you can see, there are many out of the box data wrangling components I can use to prepare my data for ML. |
|  | Click on “Filter Based Feature Selection”. | We can also use techniques like Pearson Correlation and Chi-Squared analysis to identify the most predictive features to use in training our ML model. |
|  | [Optional – illustrate how to build a model]  From “Machine Learning Algorithms > Classification”, drag in “Two-Class Logistic Regression”.  From “Model Training”, drag in “Train Model”.  From “Model Scoring & Evaluation”, drag in “Score Model”.  Connect the new nodes in a similar fashion to the existing model.  Click on the “Train Model” node and select the column to train on (“ArrDelay15”).  Mouse over the existing nodes and call out the reuse icon. | On the left, we have a model configured for training. Let’s add another one.  First I select the type of model that suits my ML problem. Then I add training & scoring tasks and connect these into the existing experiment. The only configuration required is to specify which column needs to be the focus on the training. |
|  | From “Model Training”, show “Tune Model Hyperparameters” .  From “Model Scoring & Evaluation”, show “Cross Validate Model”.  Expand “Python Language” and “R Language”. | Azure ML supports the citizen data scientist as their capabilities grow. While we didn’t require them today, advanced automation to improve models like automated hyperparameter tuning and model cross validation is easily added. In addition, Python & R snippets can be easily added to an Azure ML workflow. |
|  | Right-click “Summarize Data > Visualize > Result\_dataset” and then close.  Right-click “Score Model > Visualize > Scored dataset” and then close.  Right-click “Evaluate Model > Visualize > Evaluation results”.  Click “Precision/Recall” and then close.  Click on the “Real-time inference pipeline” tab. | I ran this experiment earlier. If I run it again, Designer is intelligent enough to identify that it doesn’t need to re-run the entire graph; just the nodes that are new.  As you can see, each node provides visual feedback to the implementer. From understanding the input data, through to scoring the model, and comparing results, Azure ML allows for a code-free experience for the beginner.  In this case we can see the second model we trained outperforms the first, so let’s deploy it to a web service. This is as easy as clicking “Create inference pipeline” to generate a predictive experiment.  The required components to build a predictive pipeline are then extracted. And then we just click “Deploy”. |
|  | Click on “Endpoints”.  Click on “flight-delay-designer”.  Switch to the “Test” tab.  Click on “Test”. | Let’s test it out! Here are all the models running as services on my Azure Kubernetes cluster. Some of these are from Designer, others from mechanisms we’ll look at shortly.  I’ll add in details for a flight to Little Rock from Detroit. And in just a moment, we’ll get our results back and see it’s predicting a very low chance of delay. Our web service is now available for any RESTful client. |

### 3. Notebooks

1. AutoML & Responsible ML & ML Pipelines

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Click “Notebooks”.  Click “flight-delay.ipynb” under “notebooks/flight-delay-automl”.  [Optional] Click “Jupyter” > “Edit in Jupyter…”. | Now that we’ve examined this problem through the eyes of a citizen data scientist, let’s see how Azure ML supports an emerging data scientist. Someone who is comfortable with Python and Jupyter, but perhaps isn’t an ML expert (or lacks the time to become one).  Automated ML is an ideal fit here too. Let’s see this in action. |
| Graphical user interface, application, Word  Description automatically generated | Scroll through to “Power BI”. | The work analyzing the problem space completed by our citizen data scientist colleagues can be easily included at the start of our analysis through the Power BI package for Jupyter. This accelerates my understanding of the key features in play, meaning I don’t waste time re-creating visualizations. |
|  | Scroll through to “Load Data from Azure Dataset Registry”.  Scroll through to “Create AML Compute Cluster”. | As with our data notebook, the first step is to connect to the Data Lake attached to the Azure ML workspace and fetch the dataset.  Before starting with Automated ML, we need to define where this job will run. In this case, we’re using a remote target. This cluster, which we also used in Designer and in the Automated ML UI experience, can use a range of compute types and will spin down to zero when not in use. |
|  | Scroll through to “Instantiate an Automated ML config”.  Run the Automated ML widget cell. | Configuring Automated ML in code is much like configuring it through the portal. I can pass the managed datasets used for training, the metric for optimization, the problem type, and customize the types of models trained (which, in this case, I’m focused on ONNX-compatible models). I’ve also enabled model explanability, which we’ll return to shortly.  I also have a number of dials to manage my training cost. I can enable early stopping based on performance, configure the number iterations, timeouts, core usage, and more.  Importantly, we can just leave string values like carrier, origin, and destination. Automated ML can apply the correct encoding techniques before starting training. |
|  | Scroll through to the widget.  Click on the ensemble model (top performer), show each tab, and then close the dialog. | Let’s take a look at an experiment I ran earlier. We can quickly review the performance of each model and the various algorithms used by Automated ML. You’ll also see some algorithms repeated as Automated ML reruns promising algorithms and works to optimize their training parameters.  We can also drill in and easily visualize metrics and the underlying transformation pipeline for individual models. |
|  | Scroll through to “Display Automated ML Run Details”. | If we look a little closer, we can see that the best model is actually an open source, Scikit-compatible LightGBM classifier.  We can also see that, before it was trained, Automated ML applied intelligent data guardrails to identify dataset issues, including common traps like class imbalance.  Once the data was validated, Automated ML applied data transformations depending on the type of data - numeric or categorical - as well as other factors like the number of distinct inputs. |
|
|
|
|
|
|  | Scroll through to “Feature Engineering”. | Automated ML also has optional model explainability. With a single flag, it will generate explainability data for every model it creates. Here we can see the key drivers behind our best classifier, including month & precipitation. These model explanations are stored alongside the experiment in the cloud for future reference. |
|  | Scroll through to “InterpretML”. | Azure ML also integrates with the InterpretML open source toolkit, allowing us to bring our existing models and analyses or create new ones. In addition we can visualize these results using the community widget directly from within the notebook. |
|
|  | Scroll through to “Fairlearn”. | Fairlearn, another open source toolkit, helps us further improve our models by identifying unfair outcomes for specific groups of people. While no features in our airline dataset stand out as particularly sensitive, we could run the toolkit and focus on the airline feature. As you can see, there isn’t a specific bias in accuracy or outcomes across this particular feature. |
|  | Scroll through to “Register Model”.  Scroll through to “Deployment”. | Now that our experimentation is concluded, and a suitable model identified for deployment, let’s add it to our model registry. In addition, to the raw model itself, I’ve attached a Markdown-based datasheet to document the background of the model and its intended usage. Furthermore, we can attach other assets to the model, like our Fairlearn analysis, so these are available for our peers to review now and in the future.  And now I’m ready to deploy my model. I have a variety of deployment options available from the SDK, including Managed Endpoints and Kubernetes. I can easily define my model requirements and push to a managed endpoint or create or connect to an existing Kubernetes cluster and then push my model. Regardless of where it deploys, I can still manage it from within Azure ML. |
|
|
|  | Scroll through to “Publish the pipeline”. | Finally, we know model building isn’t a one time effort. It’s typically repeated to improve or maintain performance over time. To support this process, Azure ML pipelines can be created to automate the steps in the retraining process. In this case, I’ve gone a step further and used the many models sample code to build a pipeline that builds per-airport models, rather than trying to build a nationwide solution. The process is just like training a single model: I register my datasets and configure my Automated ML settings. But instead of submitting one-off experiment, I create a pipeline I can trigger whenever I need new models to be produced. |
|
|
|

1. Classical ML with VS Code Deployment

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From your local computer, launch “VS Code”.  Click “flight-delay-classicalml-local.ipynb” under “notebooks/flight-delay-classicalml-local”.  Scroll through to “Connect to workspace”. | While we’ve focused on a cloud-first methodology so far, Azure ML also enables on premises and hybrid organizations. Let’s switch to VS Code running on my local laptop to see this in action.  The first step is to connect to our Azure ML Workspace. This will enable us to have access to all of the resources deployed within the workspace, including data, compute, and experiment tracking. Alternatively, I can run locally and disconnected too. |
|  | Scroll through to “Load Data”. | Instead of pulling data from the Data Lake attached to my Azure ML environment, the SDK works equally well with local files. It allows us to complete all of the same data exploration, plotting, and manipulation tasks as we completed online. |
|  | Scroll through to “Machine Learning”.  Expand the “Azure” resource explorer.  Expand your Azure ML workspace. | But what about actual machine learning? Let’s take a look at a simple scikit-learn model. When training locally, we no longer need to create a remote compute target or a script to fetch data. In the event the ML problem warrants remote training down the line, VS Code makes this seamless.  We can create & manipulate remote resources directly from our IDE. And only a few code tweaks are required to cloud-enable our experiment. |
|  | Scroll through to “Register Model”. | Now that our experimentation is concluded, and a suitable model identified for deployment, let’s add it to our model registry. Optionally, in addition to the raw model itself, I could attach a Markdown-based datasheet to document the background of the model and its intended usage. Furthermore, we can attach other assets to the model, like a Fairlearn analysis, so these are available for our peers to review now and in the future.  Now that the model is registered, it’s almost time to deploy. Before that, let’s prepare our scoring code. With scikit-learn models, I could leverage Azure ML’s no-code deployment option; however, in this case I want full control over the scoring code. |
|  | Scroll through to “Deployment”. | Now I’m ready to deploy my model. I have a variety of deployment options available from the SDK, including managed endpoints and Kubernetes. Deploying to managed endpoints only requires me to define the runtime requirements of the model and the compute is managed by AML. |
|  | Scroll through to “Deploy the model to Kubernetes” | Alternatively, I can easily create or connect to an existing cluster and then push my model. While the model is now running on a multi-container cluster, I can still manage it from within Azure ML. |
|  | Scroll through to “Connect to the deployed web service” | Let’s see it in action! Let me send some requests to the REST endpoint and collect the results. |
|  | Scroll through to “Present scoring service predictions” | Regardless of whether our model is created on premises or in the cloud (or even using another stack), we can manage the resulting model file with Azure ML through the SDK, VS Code, or via the web. |

1. Deep Learning & Labelling

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From Azure ML, click “Notebooks”.  Click “flight-delay-tf.ipynb” under “notebooks/flight-delay-dl”.  Scroll through to “Create and prepare training data”. | Next, let’s consider our final data scientist persona: the pro. Here, we’ll see how the power of the cloud through Azure ML can amplify their existing skills and complement their existing framework investments. To explore this, we’ll consider a related question: how can we combat delays more directly? We know airspace incursions can wreak havoc on plane movements so let’s consider how AI could be used to identify birds and other objects in controlled airspace. |
|  | Switch to the Azure ML portal.  Click on “Data Labeling”.  Click on the label link icon next to “Runway\_Safety”.  Click “Tasks”.  Click “Runway\_Safety” in the top nav bar.  Click “Data” > “Labeled data”. | Image data or otherwise, often one of the biggest challenges in machine learning is accurate data. Azure ML Data Labeling helps orchestrate teams to turn raw data into trainable data.  First, let’s look at the labeling experience for labelers. This is secured through the same permission mechanism as all Azure ML resources. If we switch back the project view, we can monitor progress and examine freshly labeled data and optionally reject. Once complete, I can export the results for use in training.  By using automatic ML models, the system will learn from the initial batch of manually labeled images and start pre-labeling images, considerably accelerating the labeling process for your team. |
|  | Switch back to the notebook.  Scroll through to “Augmented Dataset Sample”. | Before we start building a model, we can use libraries like Keras to augment our training dataset with synthetic imagery. Keras, like many DL frameworks, is preinstalled on Azure ML instances. |
|  | Scroll through to “Create auto-scaling AML Compute GPU cluster”.  Open a command prompt.  Enter “ssh theadmin@<ip-address> -p 50000” and provide the password “Password123!” when prompted.  Run the command “sudo docker ps”.  Run the command “nvidia-smi”. | Creating an experiment run in Azure ML is similar regardless of whether you’re using Automated ML or a deep learning framework. In this case, rather than configuring a CPU-based cluster, we’ll select a node type that’s GPU enabled.  And this training platform isn’t just a black box: while Azure ML automates provisioning and experiment deployment, we have full access to monitor and debug our jobs. This cluster happens to be active, so we can SSH directly in and run commands to monitor things like GPU usage.  In addition to traditional CPU & GPU training, Azure also supports specialized hardware like FGPAs for inferencing. |
|
|  | Return to the Jupyter Notebook.  Scroll through to the training script preview. | Now our compute platform is ready, we can proceed. Let’s upload our training data to our datastore as we did in previous experiments.  We can then configure our Tensorflow experiment. Like Automated ML, Tensorflow has a first class entry point in the Azure ML SDK with simplified deployment of common dependencies, transparent connectivity to datastores, and support for advanced capabilities like configuring multi-node training through parameter servers or MPI/Horovod. In this case, we’ve taken advantage of another advanced feature: the ability to provide a custom Docker container image for use with training. This enables a range of scenarios, including adding support for proprietary or emerging ML frameworks.  Finally: let’s inspect our training script. There’s nothing specific to Azure ML at all: despite all the capabilities we’ve mentioned, it’s all copied from the Tensorflow object detection model library. This makes adding the power of the cloud to existing workloads easy.  Azure ML also supports common TF tools like TensorBoard, as well as TF model development through automated hyperparameter tuning. By specifying the target parameters and the optimization metric, Azure ML can handle intelligently exploring the search space to find the best model. |
|
|
|
|  | Scroll through to “Register model”. | Like Automated ML, we can monitor our training run directly from inside the notebook. It’s also easy to fetch detailed logs to debug issues (along with the direct SSH access). But, because this is built on standard Tensorflow & Docker, the debugging tools you’re familiar today with will continue to work. |
|  | Scroll through to the end of the notebook. | Once training is complete, turning the model into a service follows the same process as our Automated ML experience. We create or attach to a host cluster and push the model out. And now we can see it in action: here’s a test image where we recognize the most prominent twenty birds. |

### 4. MLOps

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From Azure ML, click “Notebooks”.  Click “flight-delay-datadrift.ipynb” under “notebooks/flight-delay-mlops/flight-delay-datadrift”.  Scroll through to the “Connect to Workspace” cell. | Sourcing data and building models is only the first step in the journey towards operationalizing machine learning. MLOps, or machine learning operations, is focused on automating and accelerating the machine learning lifecycle. Let’s take a look at how Azure ML supports us in this process.  The first step is to connect to our Azure ML Workspace. This will enable us to have access to all of the resources deployed within the workspace.  One of the most important signals for model obsoletion is when trends in the data change over time. To proactively identify when this happens, we can set up a Data Drift monitor. |
|  | Scroll through to the “Create AML Compute Cluster” cell. | Before starting with Data Drift, we need to define where this job will run. In this case, we’re using a remote target. This cluster, which we also used in Designer and in the Automated ML UI experience, can use a range of compute types and will spin down to zero when not in use.  To detect data drift over time, we first need to establish a baseline. Each record in our dataset needs to have a timestamp so the system can correctly allocate the data to the appropriate period. Our training dataset doesn’t have this in a single field so can add this using standard pandas capabilities before proceeding. |
|  | Scroll through to “Create Data Drift Monitor”. | With our dataset registered, we can now attach a monitor and schedule it to run weekly. We can also focus it to run on specific features within the dataset. In our case, we’ll exclude the weather data as this drift over time is already expected. |
|  | Scroll through to “Analyze historical data and backfill”. | Given we already have a substantial amount of data, we can request Azure ML uses the first three months of the year as a baseline and then backfill the drift information for the next three months. This means we can analyze drift within existing data sets, as well as into the future. As we can see from the built-in visualizations, there is inherent drift over time within our existing dataset. |
|  | Scroll through to “Register Model”. | Now that our dataset is managed, let’s turn our attention to our models and how Azure ML can help us improve our operational efficiency.  Let’s take a suitable model from a recent run and add it to our model registry. In addition, to the raw model itself, I’ve attached a Markdown-based datasheet to document the background of the model and its intended usage for our peers to review. |
|  | Scroll through to “Trace back to model run”. | Speaking of collaboration, it’s important to retain a clear lineage. In addition to Azure ML providing a persistent store for the experiment itself and its results, the run captures the Git commit hash and repo details associated with the code used to submit the run. This allows tie back into version controlled scripts & notebooks.  And once the model is registered in the Azure ML model registry, it’s not only version controlled but also retains a link back to the experiment and the training dataset. |
|  | Scroll through to “Get ONNX model”.  Switch to the C# app.  Optionally run the app.  Switch back to the notebook. | Models built by Azure ML can be easily ported to a number of different package formats, making them available wherever needed.  Depending on the use case, it might be preferable to use an optimized model format like ONNX. With native support in Automated ML, it’s as easy as requesting the ONNX version of the model from the service. We can then run it with any ONNX-compatible runtime, like this C# app. |
|  | Scroll through to “Register OSS Model”. | In other use cases, the underlying open source version of the model might be preferable. Here, we can inspect the underlying OSS model and execute it using the OSS runtime, making it easy to deploy from edge to cloud. We can even publish it back to our registry as a standalone, OSS-based model, like a model trained externally. It can then be managed going forward. |
|  | Scroll through to “Docker Package”. | In other use cases, we might prefer to host the model as a Docker container. When deploying to Azure, Docker containers are created transparently. We can explicitly trigger this step and then take the resulting image and deploy it freely anywhere with a container runtime, from multi-cloud, to on premises, or the edge.  Edge offerings like Azure IoT Edge and Databox Edge are examples of Docker-enabled edge targets for deployment. |
|  | Scroll through to “Register scoring\_explainer as model”. | Before we deploy our model to Azure, let’s prepare it for explainability at runtime. By creating an explainer using the out-of-the-box Automated ML capabilities, we can then upload it to the model registry and deploy it alongside our predictive model. This will allow our service to return not only a prediction, but also an explanation for every request. |
|  | Scroll through to “Create Scoring File”.  Scroll through to “Profile model”. | With our model and explainer, we can now start the deployment process. In simple scenarios, I can just click the “Deploy” button next to the model; however, in this case, I’m going to prepare my own scoring code. This allows me to configure how the explanations are returned and add code to monitor incoming inference data and the associated predictions to gain insight on usage.  Before I deploy, I can also use Azure ML to profile the model and the associated scoring script. This will allow me to understand the resources required for the model to run in production. |
|  | Scroll through to “Deployment”.  Scroll through to “Connect to the deployed webservice”.  Scroll through to “Interpretability at inference time”. | Now I’m ready to deploy my model. I have a variety of deployment options available from the SDK, including managed endpoints and Kubernetes. I can easily define my model requirements and push to a managed endpoint or can easily create or connect to an existing cluster and then push my model. While the model is now running on a multi-container cluster, I can still manage it from within Azure ML.  You’ll also notice that I’ve enabled AppInsights on my deployment. This enables me to monitor usage and capture diagnostic output.  Let’s see it in action! Let me send some requests to the REST endpoint and collect the results.  In addition to reviewing the results, I can also dive into explainability for a specific result and see the key influencing features for the prediction. |
|  | Scroll through to “Decrypting Service Response”. | In this scenario, standard encryption technologies, like encryption in transit and at rest, are sufficient. But in others, especially where personal information is present, this requires a level of trust that might be difficult to establish. Support for homomorphic encryption is now in preview and allows encrypted data to be inferenced on.  The process is very similar: we create our scoring script, albeit with a specialist model runtime, and deploy it to our Kubernetes cluster. We then post encrypted data to our service and receive an encrypted prediction, which we can then decrypt on the client (and nowhere else). |

### 5. MLOps with MLflow

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From Azure ML, click “Notebooks”.  Click “flight-delay-mlflow.ipynb” under “notebooks/flight-delay-mlops/flight-delay-ml-flow”.  Scroll through to “Training script”. | Azure ML not only brings great support for building models with open source technologies, but also supports managing the MLOps lifecycle through open source packages like MLflow. Rather than using the Azure ML SDK directly, data scientists can use MLflow instead, with the knowledge it can be targeted to record experiment and deploy models to a range of services including Azure ML. |
|  | Scroll through to “Connect to Workspace”. | The first step is to connect to our Azure ML Workspace. This will enable us to have access to all of the resources deployed within the workspace. We’ll use this to fetch the data we need for training. |
| See chapter 4 for notes on the Data Drift section of the notebook. | | |
|  | Scroll through to “ML Training”.  Scroll back to the “Training Script”. | Here I’m using scikit-learn to build a classification model.  If I dig into the training script, you’ll see that we train within an MLflow run, log metrics with MLflow, and even register the model with MLflow; however, by setting the tracking URI to my Azure ML workspace, all of this information is stored with AML. I get the best of both worlds: the tools I’m familiar with and the power of Azure ML. |
|  | From Azure ML, click “Experiments”.  Select the “flight\_delay\_with\_mlflow” experiment.  Click the first run.  Showcase the logged metrics. | With our training complete, let’s switch over to the Azure ML portal. From the experiments page, we can see the MLflow experiment we just ran, including the logged metrics. Without changing my MLflow-based workflow, I’ve been able to surface my results with my colleagues. |
|  | Click “Images” tab.  Showcase the logged ConfusionMatrix.png. | MLflow integration with Azure ML supports more than simple numbers or messages. It also supports adding rich visualizations, like this confusion matrix we built with matplotlib. |
|  | From Azure ML, click “Notebooks”.  Scroll through to “Deploy to Azure Container Instance (ACI)”. | Now I’m ready to deploy my model. I have a variety of deployment options available from the SDK, including Azure Kubernetes Service and Azure Container Instances.  Again, I can use the MLflow SDK I’m familiar with to fetch a reference to the model stored in Azure ML and then deploy it as a web service (using ACI in this case). |
|  | From Azure ML, click “Endpoints”.  Showcase the deployed endpoint: fd-mlflow-aci | Let’s see it in our Azure ML workspace. From the endpoints page, we can see the deployed endpoint. If we drill into it, we can learn more about where it’s hosted, how to connect to it, and the model in use. Our operations team can now manage the health of this service from here in Azure ML, even though our data scientists continue to work in MLflow. |
|  | From Azure ML, click “Notebooks”.  Scroll through to “Traceability”. | In addition to Azure ML providing a persistent store for the experiment itself and its results, we can also tag the model with the dataset used during training and a datasheet providing our colleagues with detailed background information. |
|  | Scroll through to “Projects”. | The MLOps lifecycle is more than operationalizing our model once. It’s about building repeatable processes to ensure continuous improvement. In the MLflow community, this is enabled through MLflow projects. These define key training steps to enable automation.  In this simple project file, we specify our training script, our dependencies, and our experiment name. |
|  | Click “train.py” under “notebooks/flight-delay-mlops/flight-delay-mlflow”. | If we look at the training script, it’s just a variation of the script we reviewed in the last notebook. It leverages MLflow, connected to Azure ML, to build a new version of the model and push it to ACI. |
|  | Showcase “set\_tracking\_uri” to the Azure ML Workspace.  Showcase “set\_experiment” to the Azure ML Workspace.  Showcase “project.run” | Now we can use the MLflow SDK to run the project. We update the tracking URI, so the results are shared with our Azure ML workspace, specify an experiment name, and then invoke it. In this case, it’ll run locally; however, by specifying a compute cluster name in Azure ML, it can take advantage of remote training. |
|  | From Azure ML, click “Experiments”.  Select the “fs\_with\_mlflow\_proj” experiment. | With our project run complete, we can observe the results directly from Azure ML. Alternatively, I can even start the MLflow web server and use their interface to review the same data if I prefer. |

### 6. Hybrid ML with Azure Arc

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From the Azure Portal, navigate to the Resource Group holding your Azure Arc cluster.  Find the Kubernetes - Arc Enabled instance. | In some scenarios, training needs to happen on premises. Whether it be for compliance or security reasons, or even simply because hardware is available, Azure ML enables these scenarios through Azure Arc. By registering your on premises Kubernetes cluster with Azure Arc, Azure ML can then take advantage of the resource for model training.  Here we can see our Arc-enabled cluster. For our demo, it’s actually an AKS cluster; however, it could be any of the supported Kubernetes distributions, including other public clouds. |
|  | From Azure ML, click “Compute”.  Click “Attached Computes”.  Showcase the attached AKS compute instance. | From my Azure ML workspace, I can see the Azure Arc cluster appear as an attached compute offering. It’s managed through the same portal as my other compute, including notebook instances and managed, cloud-based training compute. |
|  | From Azure ML, click “Notebooks”.  Click “flight-delay-arc.ipynb” under “notebooks/flight-delay-arc”.  Scroll through to “Connect to workspace”. | Let’s see it in action. The first step is to connect to our Azure ML workspace from our notebook. This will enable us to have access to all of the resources deployed within the workspace, including the Azure Arc cluster. |
|  | Scroll through to “Fetch Azure Arc Attached Compute”. | In this notebook, we’ll use scikit-learn, a popular open source ML package. We could equally train with Automated ML (part of Azure ML), TensorFlow, or any other framework you might need.  Before starting with scikit-learn, we need to define where this job will run. In this case, we’re using an attached compute target from the Azure Arc enabled cluster. |
|  | Scroll through to “Create an Experiment”. | Next, we’ll create an experiment in Azure ML. This will record the results of our training run. Despite the actual work happening on premises, the results will be available for us to review from the cloud. |
|  | Scroll through to “Write Conda Dependencies file”. | We’ll also explicitly define our dependencies. Our training script creates some simple visualizations to log as outputs so we need to add some additional libraries to the training image. |
|  | Switch to “flight-delay-train.py”. | If we switch over to the training script, we can see it’s using standard sklearn libraries and references to the Azure ML run just like a script in the cloud would. You’ll also note we’re pulling data from a local location: this means that your training can occur without needing to migrate your data sources to the cloud! |
|  | Switch back to the notebook.  Scroll through to “Create Script Configuration”. | Finally, we specify our training script, dependencies, and specify our Azure Arc cluster as the training target. Compared to a normal training run in the cloud, the only difference is that we’re pointing to an Azure Arc compute cluster. |
|  | Scroll through to “Show Run Details”. | We can now submit and monitor the execution of the experiment using the standard Azure ML monitor widget. It gives us full visibility into what’s happening on premises, including logging output.  We can also drill in and easily visualize metrics and the underlying transformation pipeline for individual models. With Azure ML and Azure Arc, on premises training is just as easy and powerful as training in the cloud. |

### 7. Security & Enterprise Readiness

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Switch to Azure Portal.  Click on “Resource Groups” and then click on the resource group you created for the workspace security demo.  Click on the “Machine Learning” resource.  Click on “Launch Studio”.  Click on “Datasets”.  Click on “Experiments”. | For some of our customers, security & compliance constrains means the end-to-end machine learning experience needs to occur within their private, internal network. In other words, no resources can be accessible or addressable from the public internet. Fortunately, Azure ML, with the private endpoint capabilities available across the Azure product portfolio, can be deployed into a customer’s private virtual network to complete the end-to-end machine learning lifecycle.  Let’s start by trying to access our Azure ML workspace from an open internet connection. We see that the workspace is unable to even access resources such as datasets, notebooks, and experiments. In this case, the failures are a good sign! |
|
|
|  | Switch to Azure Portal.  Click on “Resource Groups” and then click on your created resource group for the workspace security.  Click on the “Virtual Machine” resource.  Click on “Connect”.  Click on “Bastion”  Enter the username and password for the virtual machine.  Open an Edge browser in the virtual machine.  Navigate to the Azure ML Portal.  Click on “Datasets”.  Click on “Experiments”. | Let’s instead try and access the same Azure ML workspace from inside the organization’s virtual network. In this case, we’ll use a bastion host and connect via the Azure Bastion service; however, this could just as easily be a machine connected to the corporate network and joined to Azure through a site-to-site VPN.  Our bastion host sits on the virtual network and, like Azure ML, has an internal IP address. We can now connect to the Azure ML workspace and gain access to the datasets, notebooks, experiments, and more. And all within the private network. |
|  | Click “Compute”.  Click “Compute Clusters”.  Click “cpu-cluster”.  Click “Nodes”. | Let’s take a look at the managed compute. These autoscaling clusters accelerate model training and work just as well within a private network. As you can see, the nodes in this cluster have private IP addresses so will be able to securely interact with Azure ML and your secure data sets. |
|  | Click “Notebooks”.  Click “flight-delay-automl-private.ipynb” under “notebooks/flight-delay-automl-private”.  Scroll to “Connect to Workspace” | Let’s see this environment in action. We’ll open a notebook on a compute instance that’s also running in our private network. The first step is to connect to our Azure ML workspace. This will enable us to have access to all of the resources deployed within the workspace. Connecting to our workspace in a private network from a private compute instance is no different to connecting in a public scenario. |
|  | Scroll to “Fetch existing AML Compute Cluster” | Before starting with Automated ML, we need to define where this job will run. In this case, we’ll use the cluster we looked at earlier. |
|  | Scroll to “Instantiate an Automated ML Config” | Configuring Automated ML in code is much like configuring it through the portal. I can pass the managed datasets – also stored within the private network – used for training, the metric for optimization, the problem type, and customize the types of models trained.  I also have a number of dials to manage my training cost. I can enable early stopping based on performance, configure the number iterations, timeouts, core usage, and more.  Importantly, we can just leave string values like carrier, origin, and destination. Automated ML can apply the correct encoding techniques before starting training. |
|  | Scroll to “Display Automated ML Run Details” | Let’s take a look at an experiment I ran earlier. Datasets were fetched and results, including models, published back to Azure ML without leaving the private network.  From the widget, we can quickly review the performance of each model and the various algorithms used by Automated ML. You’ll also see some algorithms repeated as Automated ML reruns promising algorithms and works to optimize their training parameters.  We can also drill in and easily visualize metrics and the underlying transformation pipeline for individual models. |
|  | Scroll to “Register Model” | Now that our experimentation is concluded, and a suitable model identified for deployment, let’s add it to our model registry. |
|  | Scroll to “Create/connect to the Kubernetes compute cluster”. | We’re now ready to deploy our model. Azure ML supports a number of different deployment options, including Azure Kubernetes Service. Like our training, our new AKS cluster used for inferencing will be deployed within our private network. |
|  | Scroll to “Deploy the model to AKS” | Next, I prepare my scoring code and define runtime requirements. With this complete, I can now deploy the model as a real-time inferencing service on AKS. I’ll even configure my training compute cluster to be used to build the Docker container that hosts the real-time service, ensuring no aspect of my solution leaves the network.  While the model is now running on a multi-container cluster, I can still manage it from within Azure ML like any other service. |
|  | Scroll to “Connect to the deployed web service”. | Let’s see it in action! Let me send some requests to the REST endpoint and collect the results. |
|  | Scroll to the “Scoring Endpoint” cell. | Before we inspect the results, we are able to see that the deployed AKS service has a private IP address. It’s only accessible to clients on the private network. |
|  | Scroll to “Present scoring service predictions”. | Despite operating within a virtual network, Azure ML still provides the same end-to-end experimentation and model operationalize capabilities. Combined with support for Azure Arc on premises training, Azure ML is ready for your most sensitive workloads. |

### 8. Responsible ML

1. Differential Privacy with SmartNoise

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | From Azure ML, click “Notebooks”.  Click “get-data.ipynb” under “notebooks/responsible-ml”.  Show the dataset visualizations. | As we look to build a new model, the first thing we need is data. It is always good to understand your data and its distributions, so that you can catch any unwarranted skews or missing values etc.  For this, you can query the data to see some aggregated values and visualize them for any anomalies.  In our case, we’re going to build a model to predict whether an applicant should or shouldn’t be approved for credit. The data has all the right characteristics: it includes features like age, sex, and marital status and provides a binary label (i.e. approved or not). |

|  |  |  |
| --- | --- | --- |
|  | Scroll down to the advanced attack cell.  Run cells to render results & plot. | While our dataset provides many useful features for model building, it also contains highly sensitive information about our customers. While it may be possible to prevent the retrieval of individual records at the data store layer, it's been shown that with a sufficient number of queries, almost the entire dataset can be reconstructed.  Here we can see some aggregate income statistics about my dataset. But with a little bit of knowledge about specific data points, we can use an off the shelf SAT solver to reconstruct the dataset that lines up with the summary information.  Within moments, we can extract the incomes of individuals despite being unable to query them directly. By comparing this to the real dataset, we can see it’s not difficult to unmask a substantial number of individuals. |
|  | Scroll down to the Differential Privacy graphic. | To protect the privacy of individuals, we can use Differential Privacy. It adds a small amount of statistical noise to each result set to obfuscate the contribution of individual datapoints.  The noise is significant enough to protect the privacy of an individual, but still small enough that it will not materially impact the accuracy of the answers.  Then, the amount of information revealed from each query is calculated and deducted from an overall privacy-loss budget, which will stop additional queries when personal privacy may be compromised. |
|  | Scroll down to the DP-protected version of the attack.  Run the cells to render results & plot. | Let’s see this approach in action by adding the Differential Privacy toolkit to manage our dataset.  Speaker note: highlight the cells and explain where you analyze the dataset using DP Toolkit and define the epsilon (budget).  By adding Differential Privacy, we’ve quickly protected the privacy of our customers during the ML model building process. |
|  | Scroll down to the DP private vs. non-private charts. | So now that we’ve proven the effectiveness of Differential Privacy, let’s see how we can still complete our work. Let’s compare the real (non-private) and the masked (private) means across a range of features in our dataset. |
|  | Scroll down to the Privatize data cell. | One way to use Differential Privacy in machine learning is to privatize an entire dataset before making it available to data scientists.  By using it to intelligently add statistical noise, I can generate a dataset that retains the overall patterns and trends required by machine learning but ensures the privacy of individuals is protected. When we hand this dataset off to our data science team, we can be confident we aren’t putting our users’ private data at risk.  Now I have a differentially private dataset for machine learning. Let’s see what we can do with it. |

1. Fairness, Explainability, and Error Analysis

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Switch to the “build-model.ipynb” notebook.  Show the “Fetch Differentially Private data” and “Create experiment” cells. | Here I’m using scikit-learn to build a logistic regression model. Because Differential Privacy privatized the entire dataset, the training code is entirely standard. The full set of OSS algorithms are available to me.  And while I’m using the Azure ML SDK to manage and track my experimental runs, I could alternatively use MLflow. It’s built-in integration with Azure ML means I get the best of both worlds: the tools I’m familiar with and the power of Azure ML. |
|  | Scroll down to the Fairlearn animation.  Show/Execute the fairlearn dashboard cell. | The next step is to build confidence in the decisions our new model makes. In our scenario, making the wrong call can have dire consequences, for both the individual concerned and from a regulatory perspective.  It’s critical for all machine learning models that they don’t result in negative impacts for groups of people, such as those defined in terms of race, gender, age, or disability status.  With Fairlearn, I can easily calculate a wide variety of fairness metrics like demographic parity or parity in accuracy rate, over my model before releasing it into production. |
|  | Scroll down to the first Fairlearn dashboard.  Click ‘SEX’ and then ‘Accuracy’. | Within the dashboard, I can quickly identify that the accuracy rates for both male and female applicants are good.  However, the disparity in predictions is significant. The model is showing a significant bias towards approving credit for one class over the other.  This isn’t necessarily an issue - it could be that the distribution of the people in each class applying for loans are very different. But it is definitely a strong signal that further investigation is required. |
|  | Scroll down to the explainability animation. | To understand the model in more detail, we can use InterpretML and its set of blackbox explainers to generate global and local explanations of the decisions it makes. |
|  | Scroll down to the new Explainability Dashboard.  Click on open in a new tab.  Add a cohort for Women (SEX = 0).  Add a cohort for Men (SEX = 1).  Show disparity with feature importance between cohorts.  Perform ‘What-if’ analysis by changing a rejected data point with SEX as a primary factor to 1 instead of 0. | From my notebook, I can retrieve the explainability results and visualize them. It allows me to identify key features, like sex, that most strongly influence the model results at a global level. I can dive even deeper and examine the impact of each feature on individual records.  Speaker note: demo the new dashboard and show the feature importance of Sex for WOMEN as a cohort, and do what-if analysis, to showcase it is indeed an indication of unfairness. |
|  | Scroll down to the Error Analysis cell. | Another aspect of building trusted models is understanding how they are likely to operate in practice. While broad claims like “this model is X% accurate” are common, it’s less understood performance may not be uniform across all subgroups of data.  With Error Analysis, a widget built into the Responsible AI package, we can take the global explanation from InterpretML explore how failures are distributed for our model. It helps us to identify cohorts of data with higher levels of error than the benchmark rate and then debug and explore them to gain the insight required to improve our model. |
|  | Scroll down to the mitigate cell. | Now that we know that this model is indeed not optimal because of its unfairness, how do we mitigate it? The answer is again FairLearn.  In addition to tools for fairness assessment, FairLearn also pro provides a range of mitigations to reduce disparity during - or after - training.  Here I’ll apply one such mitigation technique, the GridSearch algorithm, and build a set of new models. |
|  | Scroll down to the second Fairlearn dashboard.  Click on the mitigated model. | With the operation complete, I can load the new models into the Fairlearn dashboard.  As the dashboard shows us, there is a clear trade-off between disparity and accuracy. If we select a balanced model, we’ll see a slight drop in accuracy but a substantial decrease in disparity between the two classes.  While the right balance will be determined by your scenarios, Fairlearn and InterpretML provide the necessary tools to identify and address bias to ensure machine learning works for everyone. |
|  | Scroll to the cell registering all models and dashboard to Azure ML.  Switch to the Azure ML workspace and show experiment results, explainability and FairLearn dashboard, and model registration. | While I’ve been working in my notebook for this demo, MLflow and the Azure ML SDK have ensured my progress is visible and persisted for my entire team to review, both now and in the future. Capturing every artefact of the model building process is another key step towards trustworthy, repeatable, and responsible machine learning.  Speaker note: show the Azure ML workspace and highlight that the experiment, explanations and fairness metrics are all available for review in the workspace too. |

1. Homomorphic Encryption

|  |  |  |
| --- | --- | --- |
| **Demo** | | |
| Screenshot | Steps | Notes |
|  | Switch to the “make-predictions.ipynb” notebook.  Scroll down to the decrypted diagram. | With the model complete, it’s now time to release it to production while ensuring confidentiality even while inferencing against it.  In a traditional system, data is encrypted for transport between services but needs to be decrypted for work to be performed. This means every tier needs to be trusted and meet compliance standards.  With homomorphic encryption, however, we can work on encrypted data without needing to decrypt it first. The outputs are similarly encrypted, meaning only the caller can understand the results. |
|  | Scroll down to the scoring script. | With Microsoft SEAL and the encrypted-inferencing library, we can build a scoring service that can use a standard model to make an encrypted prediction over encrypted data.  Speaker note: highlight global server definition and highlight the make prediction snippets. |
|  | Scroll down to the deployment section. | And because it builds on top of existing Azure ML inferencing components, deploying an encrypted inferencing service is no different to any other service. Today I’ll show you how to use managed endpoints as well as Azure Container Instances, but it could just as easily be a Kubernetes cluster. |
|  | Scroll down to the remote inference cell.  Run the cells.  Show the cell below with raw data needed to inference against the model.  Show the encryption and its resulting that will be sent for inferencing.  Show the decryption of result cell. | Let’s see it in action. Before I make the web service call, I encrypt the row. The service then returns an encrypted result. At no time can the service see the unencrypted prediction. Once we decrypt it, we can see the application for credit was successful. This technology allows you to further protect personal data by shrinking the trust boundary present in your applications.  This is an example of a truly secure ML model. |