Course status: completed

1. Create machine learning models

Introduction

Unsurprisingly, the role of a data scientist primarily involves exploring and analyzing data.

The results of an analysis might form the basis of a report or a machine learning model, but it all begins with data, with Python being the most popular programming language for data scientists.

After decades of open-source development, Python provides extensive functionality with powerful statistical and numerical libraries:

- NumPy and Pandas simplify analyzing and manipulating data
- Matplotlib provides attractive data visualizations
- Scikit-learn offers simple and effective predictive data analysis
- TensorFlow and PyTorch supply machine learning and deep learning capabilities

Usually, a data analysis project is designed to establish insights around a particular scenario or to test a hypothesis.

For example, suppose a university professor collects data from their students, including the number of lectures attended, the hours spent studying, and the final grade achieved on the end of term exam.

The professor could analyze the data to determine if there is a relationship between the amount of studying a student undertakes and the final grade they achieve. The professor might use the data to test a hypothesis that only students who study for a minimum number of hours can expect to achieve a passing grade.

Explore data with NumPy and Pandas

Data scientists can use various tools and techniques to explore, visualize, and manipulate data.

One of the most common ways in which data scientists work with data is to use the Python language and some specific packages for data processing.

What is NumPy

NumPy is a Python library that gives functionality comparable to mathematical tools such as MATLAB and R. While NumPy significantly simplifies the user experience, it also offers comprehensive mathematical functions.

What is Pandas

Pandas is an extremely popular Python library for data analysis and manipulation. Pandas is like excel for Python - providing easy-to-use functionality for data tables.

	name	age	state	num_children	num_pets
0	john	23	iowa	2	0
1	mary	78	dc	2	4
2	peter	22	california	0	0
3	jeff	19	texas	1	5
4	bill	45	washington	2	0
5	lisa	33	dc	1	0

Explore data in a Jupyter notebook

Jupyter notebooks are a popular way of running basic scripts using your web browser. Typically, these notebooks are a single webpage, broken up into text sections and code sections that are executed on the server rather than your local machine. This means you can get started quickly without needing to install Python or other tools.

Testing hypotheses

Data exploration and analysis is typically an *iterative* process, in which the data scientist takes a sample of data and performs the following kinds of task to analyze it and test hypotheses:

Visualize data

Data scientists visualize data to understand it better. This can mean looking at the raw data, summary measures such as averages, or graphing the data. Graphs are a powerful means of viewing data, as we can discern moderately complex patterns quickly without needing to define mathematical summary measures.

Representing data visually

Representing data visually typically means graphing it. This is done to provide a fast qualitative assessment of our data, which can be useful for understanding results, finding outlier values, understanding how numbers are distributed, and so on.

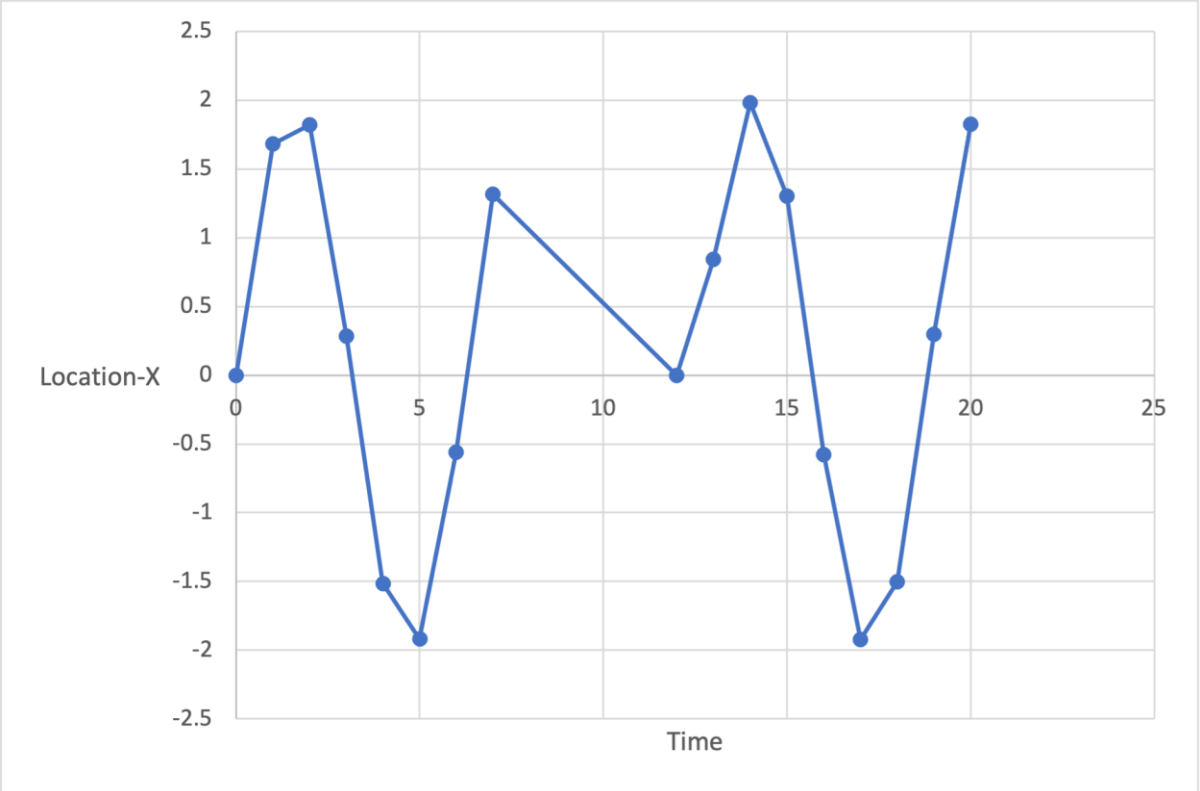
While sometimes we know ahead of time what kind of graph will be most useful, other times we use graphs in an exploratory way. To understand the power of data visualization, consider the data below: the location (x,y) of a self-driving car. In its raw form, it's hard to see any real patterns. The mean or average, tells us that its path was centred around x=0.2 and y=0.3, and the range of numbers appears to be between about -2 and 2.

Time	Location-X	Location-Y
0	0	2
1	1.682942	1.080605
2	1.818595	-0.83229

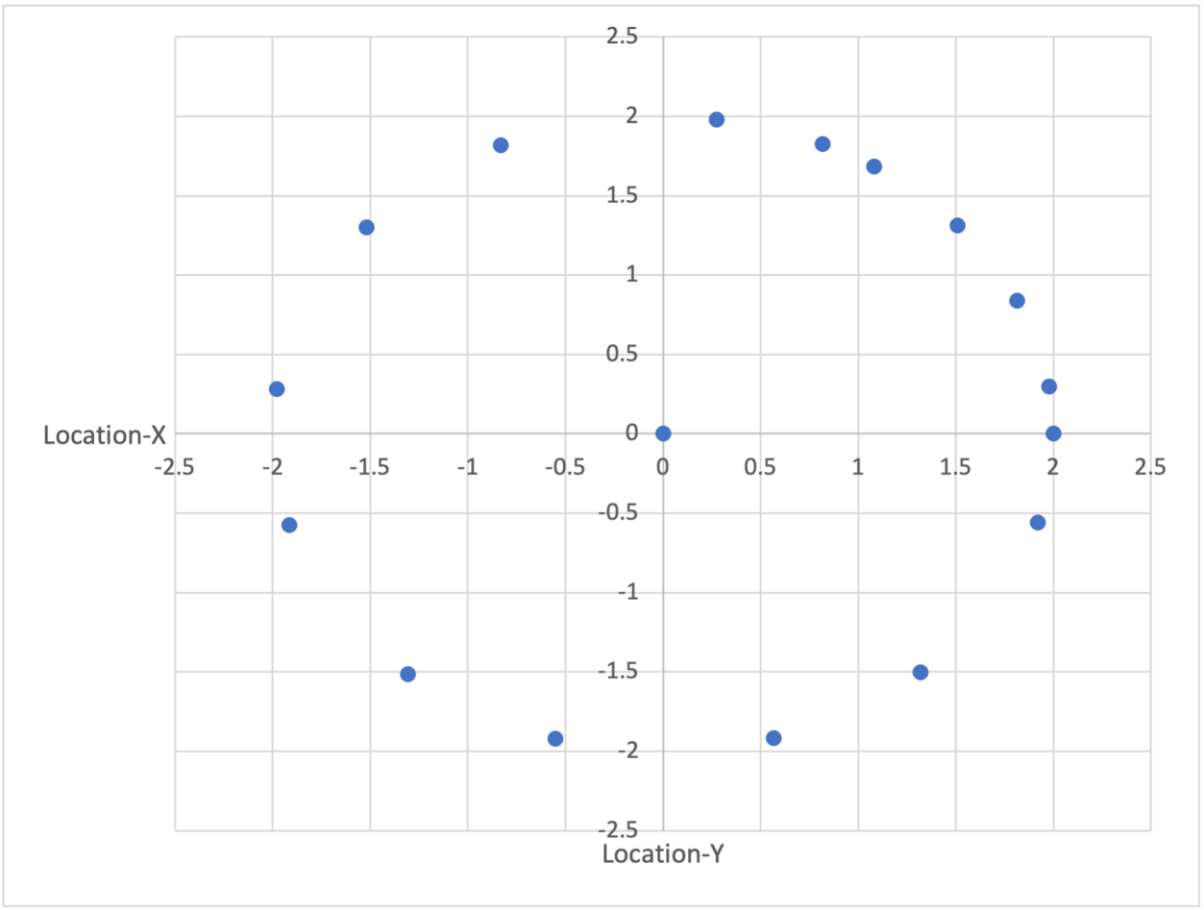
3	0.28224	-1.97998
4	-1.5136	-1.30729
5	-1.91785	0.567324
6	-0.55883	1.920341
7	1.313973	1.507805
12	0.00001	0.00001
13	0.840334	1.814894
14	1.981215	0.273474
15	1.300576	-1.51938
16	-0.57581	-1.91532
17	-1.92279	-0.55033

18	-1.50197	1.320633
19	0.299754	1.977409
20	1.825891	0.816164

If we now plot Location-X over time, we can see that we appear to have some missing values between times 7 and 12.



If we graph X vs Y, we end up with a map of where the car has driven. It's instantly obvious that the car has been driving in a circle, but at some point drove to the center of that circle.



Graphs aren't limited to 2D scatter plots like those above, but can be used to explore other kinds of data, like proportions - shown through pie charts, stacked bar graphs - how data are spread - with histograms, box and whisker plots - and how two data sets differ. Often, when we're trying to understand raw data or results, we may experiment with different types of graphs until we come across one that explains the data in a visually intuitive way.

- Clean data to handle errors, missing values, and other issues.
- Apply statistical techniques to better understand the data, and how the sample might be expected to represent the real-world population of data, allowing for random variation.
- Visualize data to determine relationships between variables, and in the case of a machine learning project, identify *features* that are potentially predictive of the *label*.
- Revise the hypothesis and repeat the process.

Introduction

Regression is where models predict a number.

In machine learning, the goal of regression is to create a model that can predict a numeric, quantifiable value, such as a price, amount, size, or other scalar number.

Regression is a statistical technique of fundamental importance to science because of its ease of interpretation, robustness, and speed in calculation. Regression models provide an excellent foundation to understanding how more complex machine learning techniques work.

In real world situations, particularly when little data are available, regression models are very useful for making predictions. For example, if a company that rents bicycles wants to predict the expected number of rentals on a given day in the future, a regression model can predict this number. A model could be created using existing data such as the number of bicycles that were rented on days where the season, day of the week, and so on, were also recorded.

What is regression?

Regression works by establishing a relationship between variables in the data that represent characteristics—known as the *features*—of the thing being observed, and the variable we're trying to predict—known as the *label*. Recall our company that rents bicycles and wants to predict the expected number of rentals in a given day. In this case, features include things like the day of the week, month, and so on, while the label is the number of bicycle rentals.

To train the model, we start with a data sample containing the features, as well as known values for the label - so in this case we need historical data that includes dates, weather conditions, and the number of bicycle rentals.

We'll then split this data sample into two subsets:

- A *training* dataset to which we'll apply an algorithm that determines a function encapsulating the relationship between the feature values and the known label values.
- A *validation* or *test* dataset that we can use to evaluate the model by using it to generate predictions for the label and comparing them to the actual known label values.

The use of historic data with known label values to train a model makes regression an example of *supervised* machine learning.

A simple example

Let's take a simple example to see how the training and evaluation process works in principle. Suppose we simplify the scenario so that we use a single feature—average daily temperature—to predict the bicycle rentals label.

We start with some data that includes known values for the average daily temperature feature and the bicycle rentals label.

Temperature	Rentals
56	115
61	126
67	137
72	140
76	152
82	156
54	114

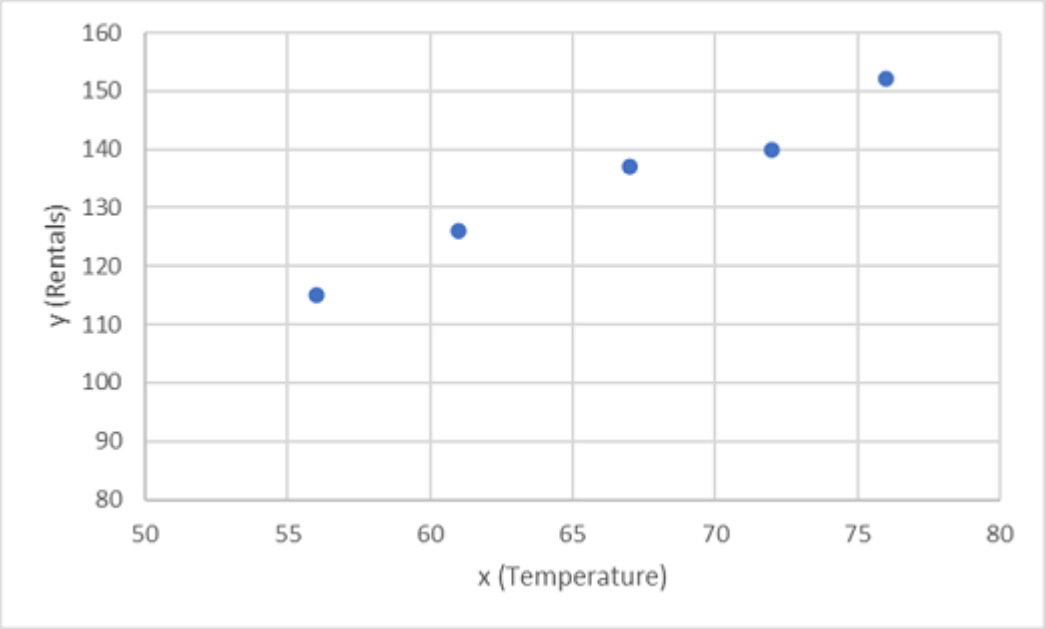
62	129

Now we'll *randomly* select five of these observations and use them to train a regression model. When we're talking about ‘training a model’, what we mean is finding a function (a mathematical equation; let’s call it f) that can use the temperature feature (which we’ll call x) to calculate the number of rentals (which we’ll call y). In other words, we need to define the following function: f(x) = y.

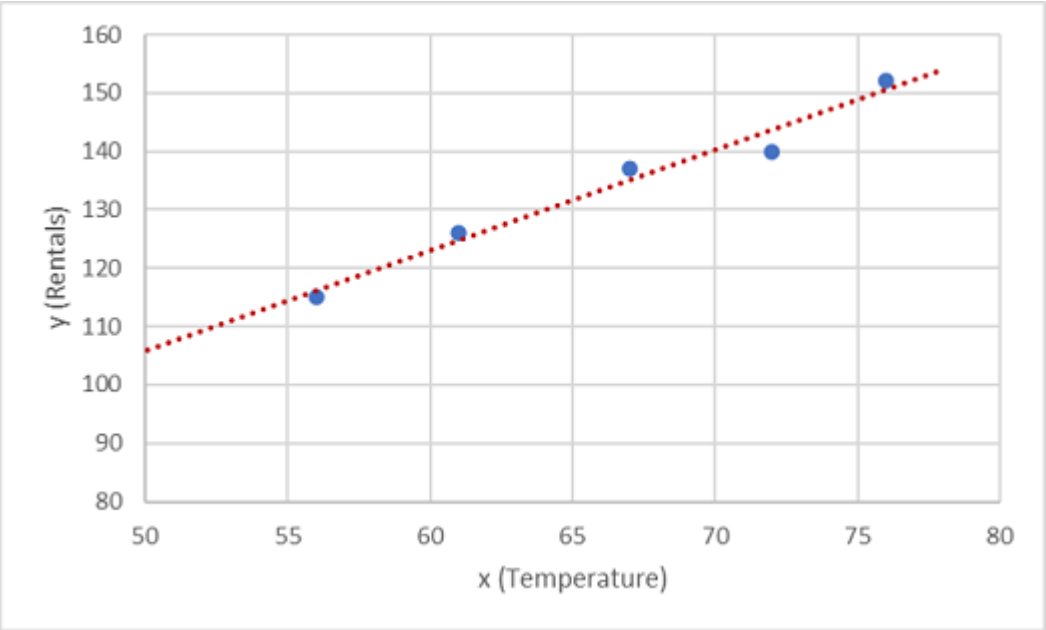
Our training dataset looks like this:

x	**y**
56	115
61	126
67	137
72	140
76	152

Let's start by plotting the training values for *x* and *y* on a chart:



Now we need to fit these values to a function, allowing for some random variation. You can probably see that the plotted points form an almost straight diagonal line - in other words, there's an apparent *linear* relationship between *x* and *y*, so we need to find a linear function that's the best fit for the data sample. There are various algorithms we can use to determine this function, which will ultimately find a straight line with minimal overall variance from the plotted points; like this:



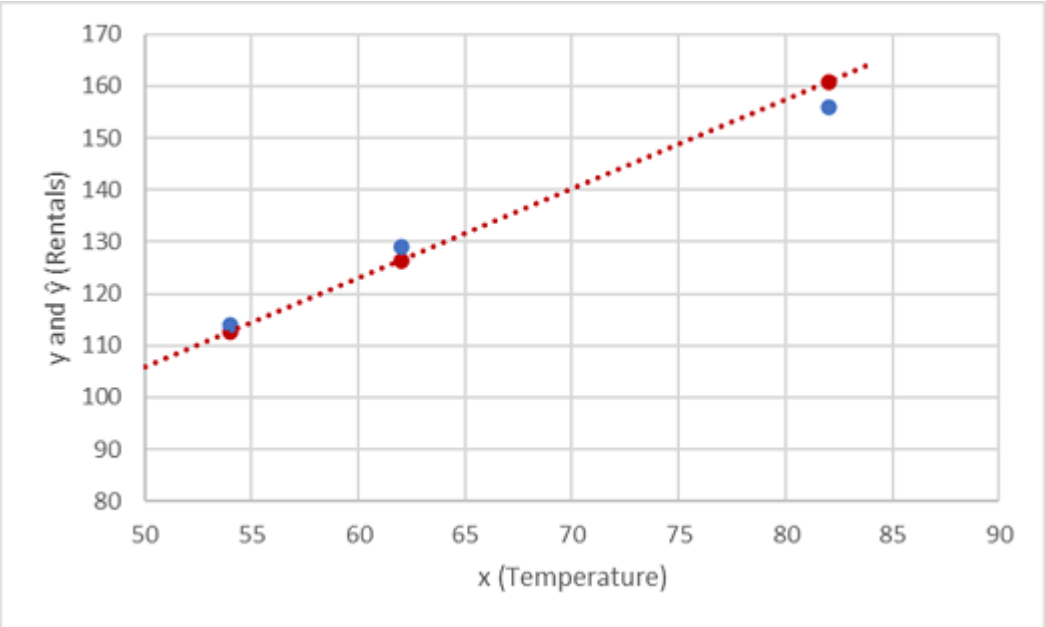
The line represents a linear function that can be used with any value of *x* to apply the *slope* of the line and its *intercept* (where the line crosses the y axis when *x* is 0) to calculate *y*. In this case, if we extended the line to the left we'd find that when *x* is 0, *y* is around 20, and the slope of the line is such that for each unit of *x* you move

along to the right, *y* increases by around 1.7. Our *f* function therefore can be calculated as 20 + 1.7*x*.

Now that we've defined our predictive function, we can use it to predict labels for the validation data we held back and compare the predicted values (which we typically indicate with the symbol *ŷ*, or "y-hat") with the actual known *y* values.

x	**y**	**ŷ**
82	156	159.4
54	114	111.8
62	129	125.4

Let's see how the y and ŷ values compare in a plot:



The plotted points that are on the function line are the predicted *ŷ* values calculated by the function, and the other plotted points are the actual *y* values.

There are various ways we can measure the variance between the predicted and actual values, and we can use these metrics to evaluate how well the model predicts.

Note

Machine learning is based in statistics and math, and it's important to be aware of specific terms that statisticians and mathematicians (and therefore data scientists) use. You can think of the difference between a *predicted* label value and the *actual* label value as a measure of *error*. However, in practice, the "actual" values are based on sample observations (which themselves may be subject to some random variance). To make it clear that we're comparing a *predicted* value ($\hat{y}$) with an *observed* value (y) we refer to the difference between them as the *residuals*. We can summarize the residuals for all of the validation data predictions to calculate the overall *loss* in the model as a measure of its predictive performance.

One of the most common ways to measure the loss is to square the individual residuals, sum the squares, and calculate the mean. Squaring the residuals has the effect of basing the calculation on *absolute* values (ignoring whether the difference is negative or positive) and giving more weight to larger differences. This metric is called the *Mean Squared Error*.

For our validation data, the calculation looks like this:

y	$\hat{y}$	$y - \hat{y}$	$(y - \hat{y})^2$
156	159.4	-3.4	11.56
114	111.8	2.2	4.84
129	125.4	3.6	12.96
	Sum	$\sum$	29.36
	Mean	$\bar{x}$	9.79

So the loss for our model based on the MSE metric is 9.79.

So is that any good? It's difficult to tell because MSE value isn't expressed in a meaningful unit of measurement. We do know that the lower the value is, the less loss there is in the model; and therefore, the better it is predicting. This makes it a useful metric to compare two models and find the one that performs best.

Sometimes, it's more useful to express the loss in the same unit of measurement as the predicted label value itself - in this case, the number of rentals. It's possible to do this by calculating the square root of the MSE, which produces a metric known, unsurprisingly, as the *Root Mean Squared Error* (RMSE).

$\sqrt{9.79} = 3.13$

So our model's RMSE indicates that the loss is just over 3, which you can interpret loosely as meaning that on average, incorrect predictions are wrong by around 3 rentals.

There are many other metrics that can be used to measure loss in a regression. For example, R^2 (R-Squared) (sometimes known as *coefficient of determination*) is the correlation between *x* and *y* squared. This produces a value between 0 and 1 that measures the amount of variance that can be explained by the model. Generally, the closer this value is to 1, the better the model predicts.

Experimenting with models

Regression models are often chosen because they work with small data samples, are robust, easy to interpret, and a variety exist.

Linear regression is the simplest form of regression, with no limit to the number of features used. Linear regression comes in many forms - often named by the number of features used and the shape of the curve that fits.

Decision trees take a step-by-step approach to predicting a variable. If we think of our bicycle example, the decision tree may be first split examples between ones that are during Spring/Summer and Autumn/Winter, make a prediction based on the day of the week. Spring/Summer-Monday may have a bike rental rate of 100 per day, while Autumn/Winter-Monday may have a rental rate of 20 per day.

Ensemble algorithms construct not just one decision tree, but a large number of trees - allowing better predictions on more complex data. Ensemble algorithms, such as Random Forest, are widely used in machine learning and science due to their strong prediction abilities.

Data scientists often experiment with using different models. In the following exercise, we'll experiment with different types of models to compare how they perform on the same data.

Improve models with hyperparameters

Simple models with small datasets can often be fit in a single step, while larger datasets and more complex models must be fit by repeatedly using the model with training data and comparing the output with the expected label. If the prediction is accurate enough, we consider the model trained. If not, we adjust the model slightly and loop again.

Hyperparameters are values that change the way that the model is fit during these loops. Learning rate, for example, is a hyperparameter that sets how much a model is adjusted during each training cycle. A high learning rate means a model can be trained faster, but if it’s too high the adjustments can be so large that the model is never ‘finely tuned’ and not optimal.

Preprocessing data

Preprocessing refers to changes you make to your data before it is passed to the model. We have previously read that preprocessing can involve cleaning your dataset. While this is important, preprocessing can also include changing the format of your data, so it's easier for the model to use. For example, data described as ‘red’, ‘orange’, ‘yellow’, ‘lime’, and ‘green’, may work better if converted into a format more native to computers, such as numbers stating the amount of red and the amount of green.

Scaling features

The most common preprocessing step is to scale features so they fall between zero and one. For example, the weight of a bike and the distance a person travels on a bike may be two very different numbers, but by scaling both numbers to between zero and one allows models to learn more effectively from the data.

Using categories as features

In machine learning, you can also use categorical features such as 'bicycle', 'skateboard’ or 'car'. These features are represented by 0 or 1 values in one-hot vectors - vectors that have a 0 or 1 for each possible value. For example, bicycle, skateboard, and car might respectively be (1,0,0), (0,1,0), and (0,0,1).

Train and evaluate classification model

Classification is a form of machine learning in which you train a model to predict which category an item belongs to. For example, a health clinic might use diagnostic data such as a patient's height, weight, blood pressure, blood-glucose level to predict whether or not the patient is diabetic.

Categorical data has distinct 'classes', rather than numeric values. Some kinds of data can be either numeric or categorical: the time to run a race could be a time in seconds, or we could split times into classes of ‘fast’, ‘medium’ and ‘slow’ - categorical. While other kinds of data can only be categorical, such as a type of shape - ‘circle’, ‘triangle’, or ‘square’.

What is classification?

Completed

100 XP

- 5 minutes

Binary classification is classification with two categories. For example, we could label patients as non-diabetic or diabetic.

The class prediction is made by determining the *probability* for each possible class as a value between 0 -impossible - and 1 - certain. The total probability for all classes is 1, as the patient is definitely either diabetic or non-diabetic. So, if the predicted probability of a patient being diabetic is 0.3, then there is a corresponding probability of 0.7 that the patient is non-diabetic.

A threshold value, usually 0.5, is used to determine the predicted class - so if the *positive* class (in this case, diabetic) has a predicted probability greater than the threshold, then a classification of diabetic is predicted.

Training and evaluating a classification model

Classification is an example of a *supervised* machine learning technique, which means it relies on data that includes known *feature* values (for example, diagnostic measurements for patients) as well as known *label* values (for example, a classification of non-diabetic or diabetic). A classification algorithm is used to fit a subset of the data to a function that can calculate the probability for each class label from the feature values. The remaining data is used to evaluate the model by comparing the predictions it generates from the features to the known class labels.

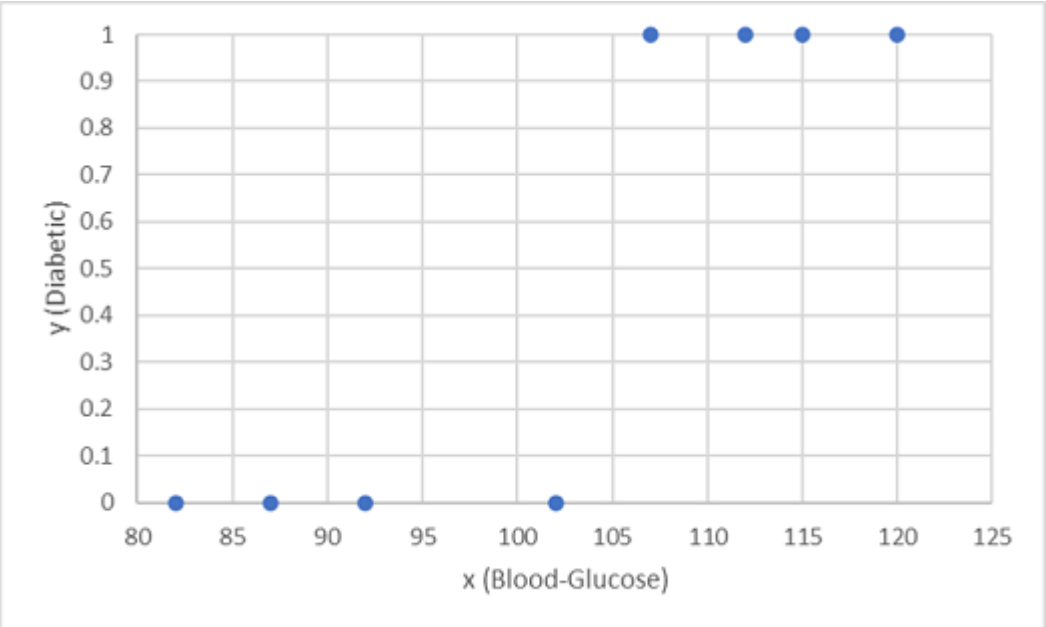
A simple example

Let's explore a simple example to help explain the key principles. Suppose we have the following patient data, which consists of a single feature (blood-glucose level) and a class label 0 for non-diabetic, 1 for diabetic.

Blood-Glucose	Diabetic
82	0
92	0
112	1
102	0
115	1
107	1
87	0
120	1
83	0

119	1
104	1
105	0
86	0
109	1

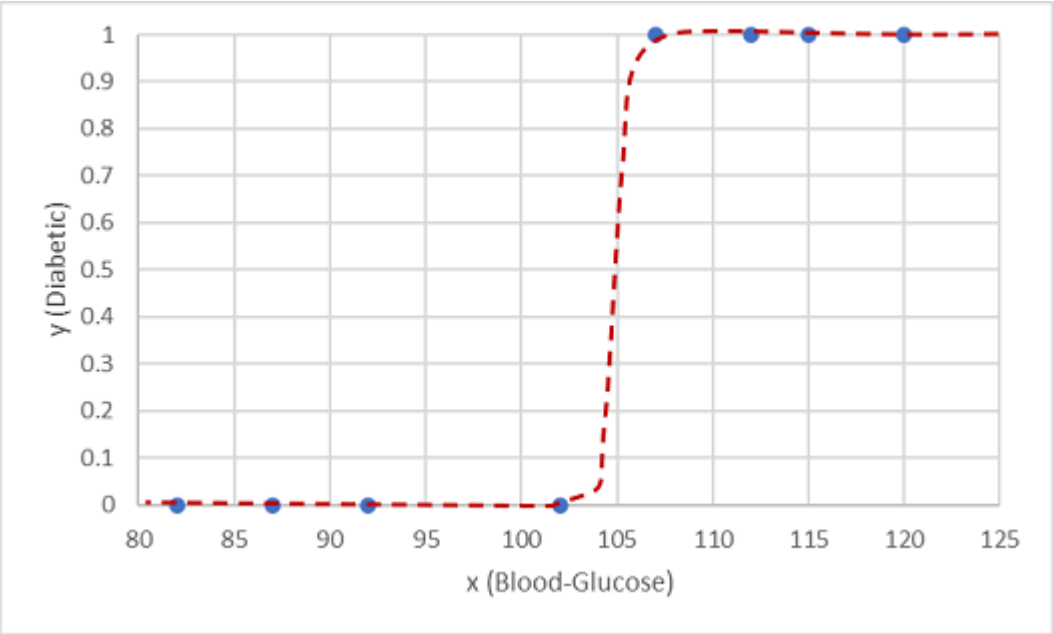
We'll use the first eight observations to train a classification model, and we'll start by plotting the blood-glucose feature (which we'll call *x*) and the predicted diabetic label (which we'll call *y*).



What we need is a function that calculates a probability value for *y* based on *x* (in other words, we need the function *f(x) = y*). You can see from the chart that patients with a low blood-glucose level are all non-diabetic, while patients with a higher blood-glucose level are diabetic. It seems like the higher the blood-glucose level, the more

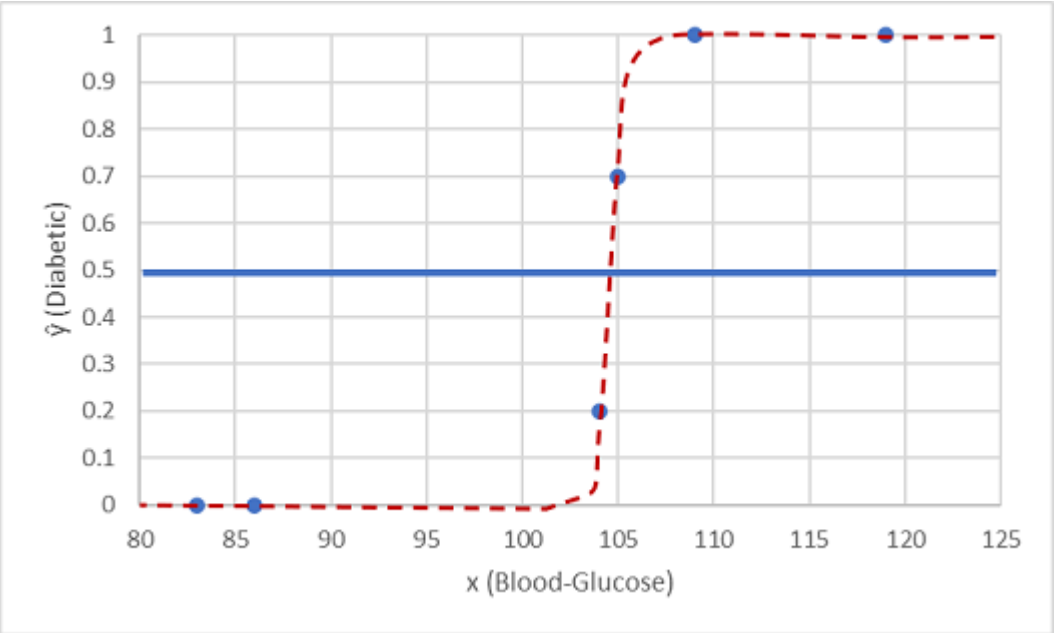
probable it is that a patient is diabetic, with the inflexion point being somewhere between 100 and 110. We need to fit a function that calculates a value between 0 and 1 for *y* to these values.

One such function is a *logistic* function, which forms a sigmoidal (S-shaped) curve, like this:



Now we can use the function to calculate a probability value that *y* is positive, meaning the patient is diabetic, from any value of *x* by finding the point on the function line for *x*. We can set a threshold value of 0.5 as the cut-off point for the class label prediction.

Let's test it with the data values we held-back:



Points plotted below the threshold line will yield a predicted class of 0 - non-diabetic - and points above the line will be predicted as 1 - diabetic.

Now we can compare the label predictions based on the logistic function encapsulated in the model (which we'll call $\hat{y}$, or "y-hat") to the actual class labels (y).

x	y	ŷ
83	0	0
119	1	1
104	1	0
105	0	1
86	0	0
109	1	1

Evaluate classification models

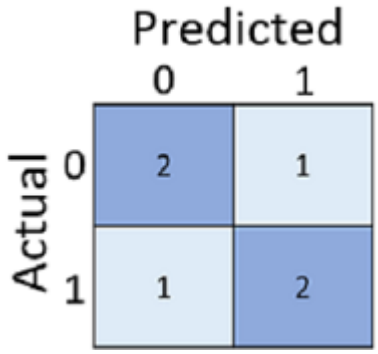
The training accuracy of a classification model is much less important than how well that model will work when given new, unseen data. After all, we train models so that they can be used on new data we find in the real world. So, after we have trained a classification model, we should evaluate how it performs on a set of new, unseen data.

In the previous units, we created a model that would predict whether a patient had diabetes or not based on their blood glucose level. Now, when applied to some data that wasn't part of the training set we get the following predictions:

x	y	ŷ
83	0	0
119	1	1
104	1	0
105	0	1
86	0	0
109	1	1

Recall that x refers to blood glucose level, y refers to whether they're actually diabetic, and ŷ refers to the model’s prediction as to whether they're diabetic or not.

Simply calculating how many predictions were correct is sometimes misleading or too simplistic for us to understand the kinds of errors it will make in the real world. To get more detailed information, we can tabulate the results in a structure called a *confusion matrix*, like this:



The confusion matrix shows the total number of cases where:

- The model predicted 0 and the actual label is 0 (*true negatives*; top left)
- The model predicted 1 and the actual label is 1 (*true positives*; bottom right)
- The model predicted 0 and the actual label is 1 (*false negatives*; bottom left)
- The model predicted 1 and the actual label is 0 (*false positives*; top right)

The cells in the confusion matrix are often shaded so that higher values have a deeper shade. This makes it easier to see a strong diagonal trend from top-left to bottom-right, highlighting the cells where the predicted value and actual value are the same.

From these core values, you can calculate a range of other metrics that can help you evaluate the performance of the model. For example:

- Accuracy: (TP+TN)/(TP+TN+FP+FN) - out all of the predictions, how many were correct?
- Recall: TP/(TP+FN) - of all the cases that *are* positive, how many did the model identify?
- Precision: TP/(TP+FP) - of all the cases that the model predicted to be positive, how many actually *are* positive?

Create multiclass classification models

It's also possible to create *multiclass* classification models, in which there are more than two possible classes. For example, the health clinic might expand the diabetes model to classify patients as:

- Non-diabetic

- Type-1 diabetic
- Type-2 diabetic

The individual class probability values would still add up to a total of 1 as the patient is definitely in only one of the three classes, and the most probable class would be predicted by the model.

Using Multiclass classification models

Multiclass classification can be thought of as a combination of multiple binary classifiers. There are two ways in which you approach the problem:

- One vs Rest (OVR), in which a classifier is created for each possible class value, with a positive outcome for cases where the prediction is this class, and negative predictions for cases where the prediction is any other class. For example, a classification problem with four possible shape classes (square, circle, triangle, hexagon) would require four classifiers that predict:
    - square or not
    - circle or not
    - triangle or not
    - hexagon or not
- One vs One (OVO), in which a classifier for each possible pair of classes is created. The classification problem with four shape classes would require the following binary classifiers:
    - square or circle
    - square or triangle
    - square or hexagon
    - circle or triangle
    - circle or hexagon
    - triangle or hexagon

In both approaches, the overall model must take into account all of these predictions to determine which single category the item belongs to.

Fortunately, in most machine learning frameworks, including scikit-learn, implementing a multiclass classification model is not significantly more complex than binary classification - and in most cases, the estimators used for binary classification implicitly support multiclass classification by abstracting an OVR algorithm, an OVO algorithm, or by allowing a choice of either.

What is clustering?

Clustering is a form of *unsupervised* machine learning in which observations are grouped into clusters based on similarities in their data values, or *features*. This kind of machine learning is considered unsupervised because it does not make use of previously known *label* values to train a model; in a clustering model, the label is the cluster to which the observation is assigned, based purely on its features.

For example, suppose a botanist observes a sample of flowers and records the number of petals and leaves on each flower.

It may be useful to group these flowers into clusters based on similarities between their features.

There are many ways this could be done. For example, if most flowers have the same number of leaves, they could be grouped into those with many vs few petals. Alternatively, if both petal and leaf counts vary considerably there may be a pattern to discover, such as those with many leaves also having many petals. The goal of the clustering algorithm is to find the optimal way to split the dataset into groups. What 'optimal' means depends on both the algorithm used and the dataset that is provided.

Although this flower example may be simple for a human to achieve with only a few samples, as the dataset grows to thousands of samples or to more than two features, clustering algorithms become very useful to quickly dissect a dataset into groups.

Training a clustering model

There are multiple algorithms you can use for clustering. One of the most commonly used algorithms is *K-Means* clustering that, in its simplest form, consists of the following steps:

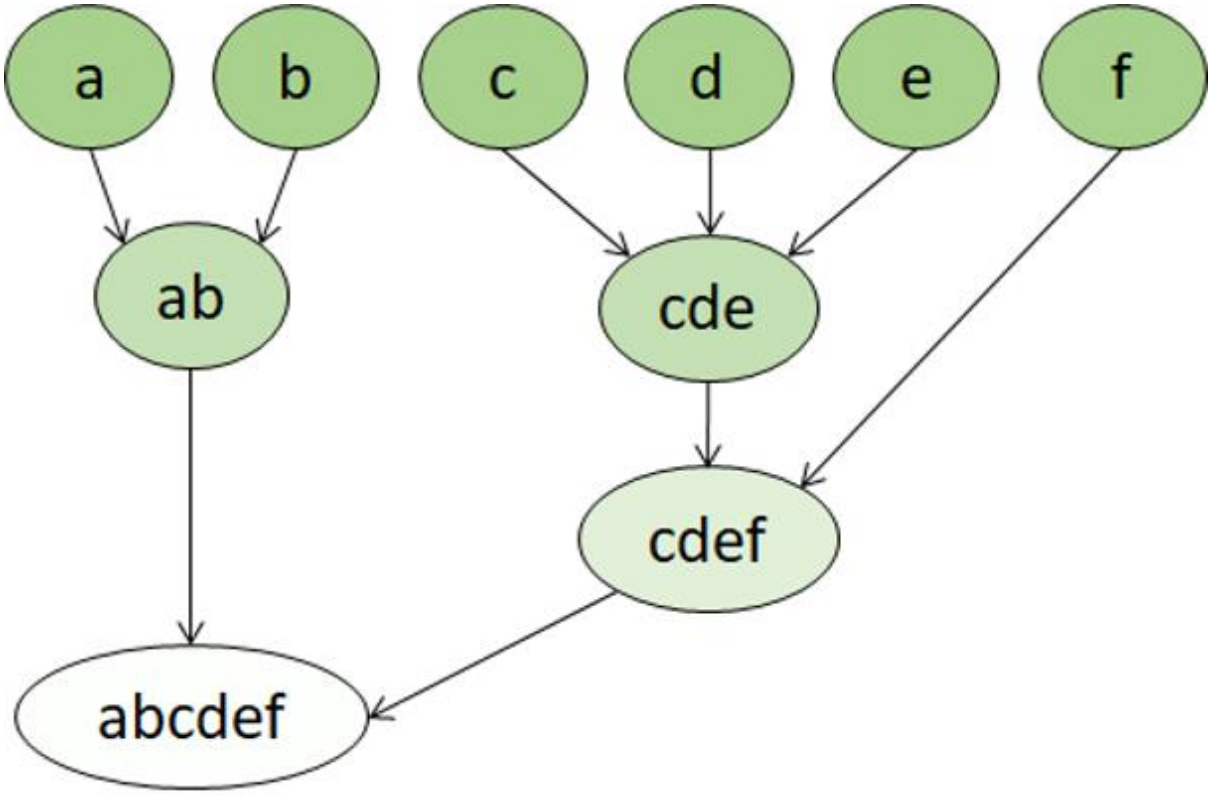
1. The feature values are vectorized to define n-dimensional coordinates (where *n* is the number of features). In the flower example, we have two features (number of petals and number of leaves), so the feature vector has two coordinates that we can use to conceptually plot the data points in two-dimensional space.
2. You decide how many clusters you want to use to group the flowers, and call this value *k*. For example, to create three clusters, you would use a *k* value of 3. Then *k* points are plotted at random coordinates. These points will ultimately be the center points for each cluster, so they're referred to as *centroids*.
3. Each data point (in this case flower) is assigned to its nearest centroid.
4. Each centroid is moved to the center of the data points assigned to it based on the mean distance between the points.
5. After moving the centroid, the data points may now be closer to a different centroid, so the data points are reassigned to clusters based on the new closest centroid.
6. The centroid movement and cluster reallocation steps are repeated until the clusters become stable or a pre-determined maximum number of iterations is reached.

The following animation shows this process:

Hierarchical Clustering

Hierarchical clustering is another type of clustering algorithm in which clusters themselves belong to a larger group, which belong to even larger groups, and so on. The result is that data points can be clusters in differing degrees of precision: with a large number of very small and precise groups, or a small number of larger groups.

For example, if we apply clustering to the meanings of words, we may get a group containing adjectives specific to emotions (‘angry’, ‘happy’, and so on), which itself belongs to a group containing all human-related adjectives (‘happy’, ‘handsome’, ‘young’), and this belongs to an even higher group containing all adjectives (‘happy’, ‘green’, ‘handsome’, ‘hard’ etc.).



Hierarchical clustering is useful for not only breaking data into groups, but understanding the relationships between these groups. A major advantage of hierarchical clustering is that it does not require the number of clusters to be defined in advance, and can sometimes provide more interpretable results than non-hierarchical approaches. The major drawback is that these approaches can take much longer to compute than simpler approaches and sometimes are not suitable for large datasets.

Introduction- Deep Learning

Deep learning is an advanced form of machine learning that tries to emulate the way the human brain learns.

In your brain, you have nerve cells called neurons, which are connected to one another by nerve extensions that pass electrochemical signals through the network.

When the first neuron in the network is stimulated, the input signal is processed, and if it exceeds a particular threshold, the neuron is *activated* and passes the signal on to the neurons to which it is connected. These neurons in turn may be activated and pass the signal on through the rest of the network. Over time, the connections between the neurons are strengthened by frequent use as you learn how to respond effectively. For example, if someone throws a ball towards you, your neuron connections enable you to process the visual information and coordinate your movements to catch the ball. If you perform this action repeatedly, the network of neurons involved in catching a ball will grow stronger as you learn how to be better at catching a ball.

Deep learning emulates this biological process using artificial neural networks that process numeric inputs rather than electrochemical stimuli.

The incoming nerve connections are replaced by numeric inputs that are typically identified as x. When there's more than one input value, x is considered a vector with elements named x_1, x_2, and so on.

Associated with each x value is a *weight* (w), which is used to strengthen or weaken the effect of the x value to simulate learning. Additionally, a *bias* (b) input is added to enable fine-grained control over the network. During the training process, the w and b values will be adjusted to tune the network so that it "learns" to produce correct outputs.

The neuron itself encapsulates a function that calculates a weighted sum of x, w, and b. This function is in turn enclosed in an *activation function* that constrains the result (often to a value between 0 and 1) to determine whether or not the neuron passes an output onto the next layer of neurons in the network.

Deep neural network concepts

Before exploring how to train a deep neural network (DNN) machine learning model, let's consider what we're trying to achieve. Machine learning is concerned with predicting a *label* based on some *features* of a particular observation. In simple terms, a machine learning model is a function that calculates y (the label) from x (the features): $f(x)=y$.

A simple classification example

For example, suppose your observation consists of some measurements of a penguin.

Specifically, the measurements are:

- The length of the penguin's bill.
- The depth of the penguin's bill.
- The length of the penguin's flipper.
- The penguin's weight.

In this case, the features (x) are a vector of four values, or mathematically, $x=[x_1,x_2,x_3,x_4]$.

Let's suppose that the label we're trying to predict (y) is the species of the penguin, and that there are three possible species it could be:

1. *Adelie*
2. *Gentoo*
3. *Chinstrap*

This is an example of a *classification* problem, in which the machine learning model must predict the most probable class to which the observation belongs. A classification model accomplishes this by predicting a label that consists of the probability for each class. In other words, y is a vector of three probability values; one for each of the possible classes: $y=[P(0),P(1),P(2)]$.

You train the machine learning model by using observations for which you already know the true label. For example, you may have the following feature measurements for an *Adelie* specimen:

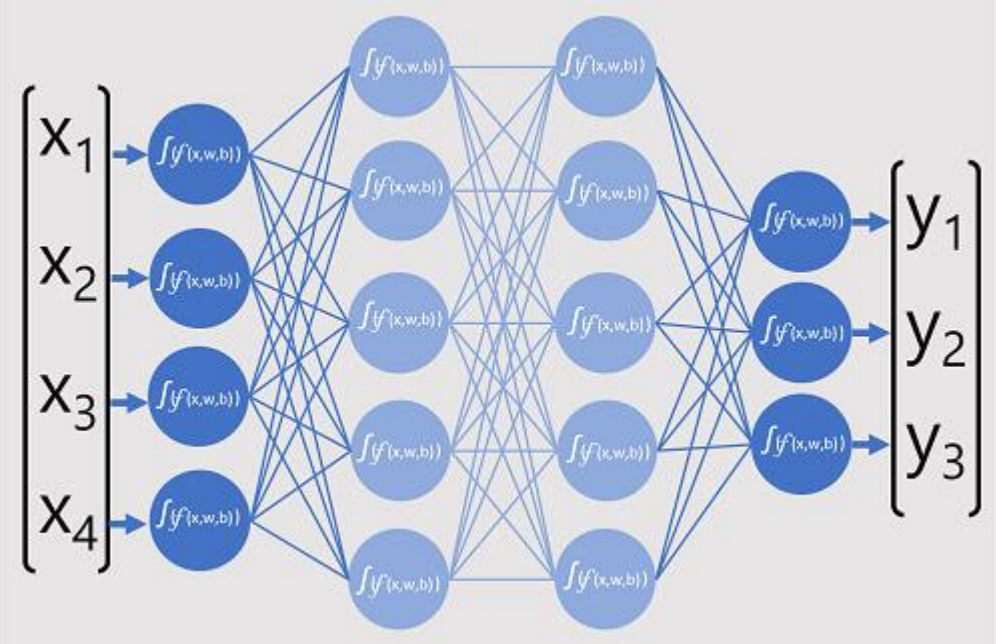
$x=[37.3, 16.8, 19.2, 30.0]$

You already know that this is an example of an *Adelie* (class 0), so a perfect classification function should result in a label that indicates a 100% probability for class 0, and a 0% probability for classes 1 and 2:

y=[1, 0, 0]

A deep neural network model

So how would we use deep learning to build a classification model for the penguin classification model? Let's look at an example:



The deep neural network model for the classifier consists of multiple layers of artificial neurons. In this case, there are four layers:

- An *input* layer with a neuron for each expected input (*x*) value.
- Two so-called *hidden* layers, each containing five neurons.
- An *output* layer containing three neurons - one for each class probability (*y*) value to be predicted by the model.

Because of the layered architecture of the network, this kind of model is sometimes referred to as a *multilayer perceptron*. Additionally, notice that all neurons in the input and hidden layers are connected to all neurons in the subsequent layers - this is an example of a *fully connected network*.

When you create a model like this, you must define an input layer that supports the number of features your model will process, and an output layer that reflects the number of outputs you expect it to produce. You can decide how many hidden layers you want to include and how many neurons are in each of them; but you have no control over the input and output values for these layers - these are determined by the model training process.

Training a deep neural network

The training process for a deep neural network consists of multiple iterations, called *epochs*. For the first epoch, you start by assigning random initialization values for the weight (*w*) and bias *b* values. Then the process is as follows:

1. Features for data observations with known label values are submitted to the input layer. Generally, these observations are grouped into *batches* (often referred to as *mini-batches*).
2. The neurons then apply their function, and if activated, pass the result onto the next layer until the output layer produces a prediction.
3. The prediction is compared to the actual known value, and the amount of variance between the predicted and true values (which we call the *loss*) is calculated.
4. Based on the results, revised values for the weights and bias values are calculated to reduce the loss, and these adjustments are *backpropagated* to the neurons in the network layers.
5. The next epoch repeats the batch training forward pass with the revised weight and bias values, hopefully improving the accuracy of the model (by reducing the loss).

Note

Processing the training features as a batch improves the efficiency of the training process by processing multiple observations simultaneously as a matrix of features with vectors of weights and biases. Linear algebraic functions that operate with matrices and vectors also feature in 3D graphics processing, which is why computers with graphic processing units (GPUs) provide significantly better performance for deep learning model training than central processing unit (CPU) only computers.

A closer look at loss functions and backpropagation

The previous description of the deep learning training process mentioned that the loss from the model is calculated and used to adjust the weight and bias values. How exactly does this work?

Calculating loss

Suppose one of the samples passed through the training process contains features of an *Adelie* specimen (class 0). The correct output from the network would be [1, 0,

0]. Now suppose that the output produced by the network is [0.4, 0.3, 0.3]. Comparing these, we can calculate an absolute variance for each element (in other words, how far is each predicted value away from what it should be) as [0.6, 0.3, 0.3].

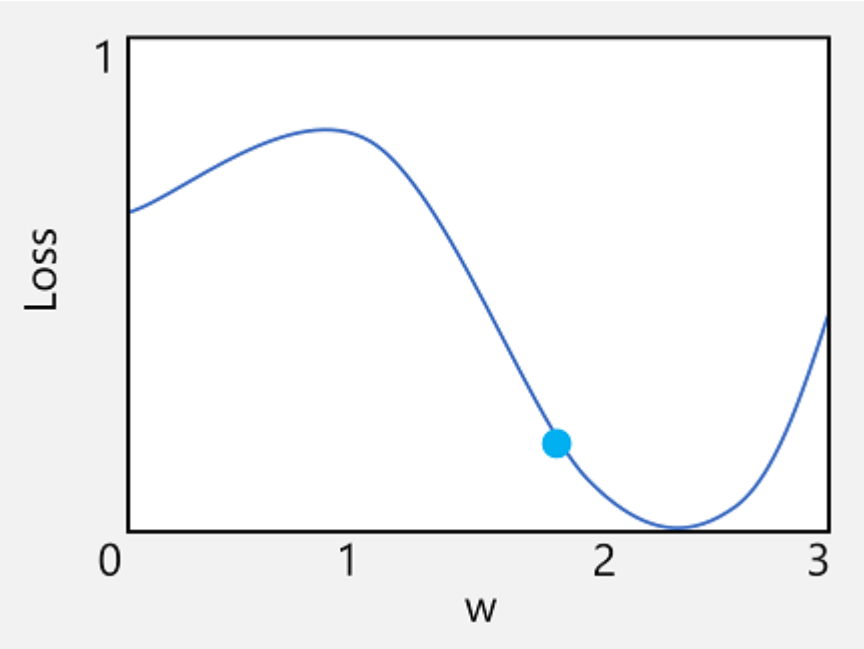
In reality, since we're actually dealing with multiple observations, we typically aggregate the variance - for example by squaring the individual variance values and calculating the mean, so we end up with a single, average loss value, like 0.18.

Optimizers

Now, here's the clever bit. The loss is calculated using a function, which operates on the results from the final layer of the network, which is also a function. The final layer of network operates on the outputs from the previous layers, which are also functions. So in effect, the entire model from the input layer right through to the loss calculation is just one big nested function. Functions have a few really useful characteristics, including:

- You can conceptualize a function as a plotted line comparing its output with each of its variables.
- You can use differential calculus to calculate the *derivative* of the function at any point with respect to its variables.

Let's take the first of these capabilities. We can plot the line of the function to show how an individual weight value compares to loss, and mark on that line the point where the current weight value matches the current loss value.



Now let's apply the second characteristic of a function. The derivative of a function for a given point indicates whether the slope (or *gradient*) of the function output (in this case, loss) is increasing or decreasing with respect to a function variable (in this case, the weight value). A positive derivative indicates that the function is increasing, and a negative derivative indicates that it is decreasing. In this case, at the plotted point for the current weight value, the function has a downward gradient. In other words, increasing the weight will have the effect of decreasing the loss.

We use an *optimizer* to apply this same trick for all of the weight and bias variables in the model and determine in which direction we need to adjust them (up or down) to reduce the overall amount of loss in the model. There are multiple commonly used optimization algorithms, including *stochastic gradient descent (SGD)*, *Adaptive Learning Rate (ADADELTA)*, *Adaptive Momentum Estimation (Adam)*, and others; all of which are designed to figure out how to adjust the weights and biases to minimize loss.

Learning rate

Now, the obvious next question is, by how much should the optimizer adjust the weights and bias values? If you look at the plot for our weight value, you can see that increasing the weight by a small amount will follow the function line down (reducing the loss), but if we increase it by too much, the function line starts to go up again, so we might actually increase the loss; and after the next epoch, we might find we need to reduce the weight.

The size of the adjustment is controlled by a parameter that you set for training called the *learning rate*. A low learning rate results in small adjustments (so it can take more epochs to minimize the loss), while a high learning rate results in large adjustments (so you might miss the minimum altogether).

Convolutional neural networks

While you can use deep learning models for any kind of machine learning, they're particularly useful for dealing with data that consists of large arrays of numeric values - such as images. Machine learning models that work with images are the foundation for an area artificial intelligence called *computer vision*, and deep learning techniques have been responsible for driving amazing advances in this area over recent years.

At the heart of deep learning's success in this area is a kind of model called a *convolutional neural network*, or *CNN*. A CNN typically works by extracting features from images, and then feeding those features into a fully connected neural network to generate a prediction. The feature extraction layers in the network have the effect

of reducing the number of features from the potentially huge array of individual pixel values to a smaller feature set that supports label prediction.

Layers in a CNN

CNNs consist of multiple layers, each performing a specific task in extracting features or predicting labels.

Convolution layers

One of the principal layer types is a *convolutional* layer that extracts important features in images. A convolutional layer works by applying a filter to images. The filter is defined by a *kernel* that consists of a matrix of weight values.

For example, a 3x3 filter might be defined like this:

Copy

```
1  -1   1

-1   0  -1

 1  -1   1
```

An image is also just a matrix of pixel values. To apply the filter, you "overlay" it on an image and calculate a *weighted sum* of the corresponding image pixel values under the filter kernel. The result is then assigned to the center cell of an equivalent 3x3 patch in a new matrix of values that is the same size as the image. For example, suppose a 6 x 6 image has the following pixel values:

Copy

```
255 255 255 255 255 255

255 255 100 255 255 255

255 100 100 100 255 255

100 100 100 100 100 255

255 255 255 255 255 255

255 255 255 255 255 255
```

Applying the filter to the top-left 3x3 patch of the image would work like this:

Copy

```
255 255 255      1  -1   1    (255 x 1)+(255 x -1)+(255 x 1) +

255 255 100  x  -1   0  -1  = (255 x -1)+(255 x 0)+(100 x -1) +   =
155

255 100 100      1  -1   1    (255 x1 )+(100 x -1)+(100 x 1)
```

The result is assigned to the corresponding pixel value in the new matrix like this:

```
?   ?   ?   ?   ?   ?

 ?  155  ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?
```

Now the filter is moved along (*convolved*), typically using a *step* size of 1 (so moving along one pixel to the right), and the value for the next pixel is calculated

Copy

```
255 255 255      1  -1   1    (255 x 1)+(255 x -1)+(255 x 1) +

255 100 255  x  -1   0  -1  = (255 x -1)+(100 x 0)+(255 x -1) +   =
-155

100 100 100      1  -1   1    (100 x1 )+(100 x -1)+(100 x 1)
```

So now we can fill in the next value of the new matrix.

Copy

```
?   ?   ?   ?   ?   ?

 ?  155 -155 ?   ?   ?

 ?   ?   ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?

 ?   ?   ?   ?   ?   ?
```

```
 ?   ?   ?   ?   ?   ?
```

The process repeats until we've applied the filter across all of the 3x3 patches of the image to produce a new matrix of values like this:

Copy

```
?   ?   ?    ?    ?   ?

?  155 -155 155 -155  ?

? -155 310 -155  155  ?

?  310 155  310   0   ?

? -155 -155 -155  0   ?

?   ?   ?    ?    ?   ?
```

Because of the size of the filter kernel, we can't calculate values for the pixels at the edge; so we typically just apply a *padding* value (often 0):

Copy

```
0   0   0    0    0   0

0  155 -155 155 -155  0

0 -155 310 -155  155  0

0  310 155  310   0   0

0 -155 -155 -155  0   0

0   0   0    0    0   0
```

The output of the convolution is typically passed to an activation function, which is often a *Rectified Linear Unit* (ReLU) function that ensures negative values are set to 0:

```
0   0   0    0    0   0

0  155  0   155   0   0

0   0  310   0   155  0

0  310 155  310   0   0
```

0 0 0 0 0 0

0 0 0 0 0 0

The resulting matrix is a *feature map* of feature values that can be used to train a machine learning model.

Note: The values in the feature map can be greater than the maximum value for a pixel (255), so if you wanted to visualize the feature map as an image you would need to *normalize* the feature values between 0 and 255.

The convolution process is shown in the animation below.

1. An image is passed to the convolutional layer. In this case, the image is a simple geometric shape.
2. The image is composed of an array of pixels with values between 0 and 255 (for color images, this is usually a 3-dimensional array with values for red, green, and blue channels).
3. A filter kernel is generally initialized with random weights (in this example, we've chosen values to highlight the effect that a filter might have on pixel values; but in a real CNN, the initial weights would typically be generated from a random Gaussian distribution). This filter will be used to extract a feature map from the image data.
4. The filter is convolved across the image, calculating feature values by applying a sum of the weights multiplied by their corresponding pixel

values in each position. A Rectified Linear Unit (ReLU) activation function is applied to ensure negative values are set to 0.
5. After convolution, the feature map contains the extracted feature values, which often emphasize key visual attributes of the image. In this case, the feature map highlights the edges and corners of the triangle in the image.

Typically, a convolutional layer applies multiple filter kernels. Each filter produces a different feature map, and all of the feature maps are passed onto the next layer of the network.

Pooling layers

After extracting feature values from images, *pooling* (or *downsampling*) layers are used to reduce the number of feature values while retaining the key differentiating features that have been extracted.

One of the most common kinds of pooling is *max pooling* in which a filter is applied to the image, and only the maximum pixel value within the filter area is retained. So for example, applying a 2x2 pooling kernel to the following patch of an image would produce the result 155.

Copy

```
0   0

0  155
```

Note that the effect of the 2x2 pooling filter is to reduce the number of values from 4 to 1.

As with convolutional layers, pooling layers work by applying the filter across the whole feature map. The animation below shows an example of max pooling for an image map.

1. The feature map extracted by a filter in a convolutional layer contains an array of feature values.
2. A pooling kernel is used to reduce the number of feature values. In this case, the kernel size is 2x2, so it will produce an array with quarter the number of feature values.
3. The pooling kernel is convolved across the feature map, retaining only the highest pixel value in each position.

Dropping layers

One of the most difficult challenges in a CNN is the avoidance of *overfitting*, where the resulting model performs well with the training data but doesn't generalize well to new data on which it wasn't trained. One technique you can use to mitigate overfitting is to include layers in which the training process randomly eliminates (or "drops") feature maps. This may seem counterintuitive, but it's an effective way to ensure that the model doesn't learn to be over-dependent on the training images.

Other techniques you can use to mitigate overfitting include randomly flipping, mirroring, or skewing the training images to generate data that varies between training epochs.

Flattening layers

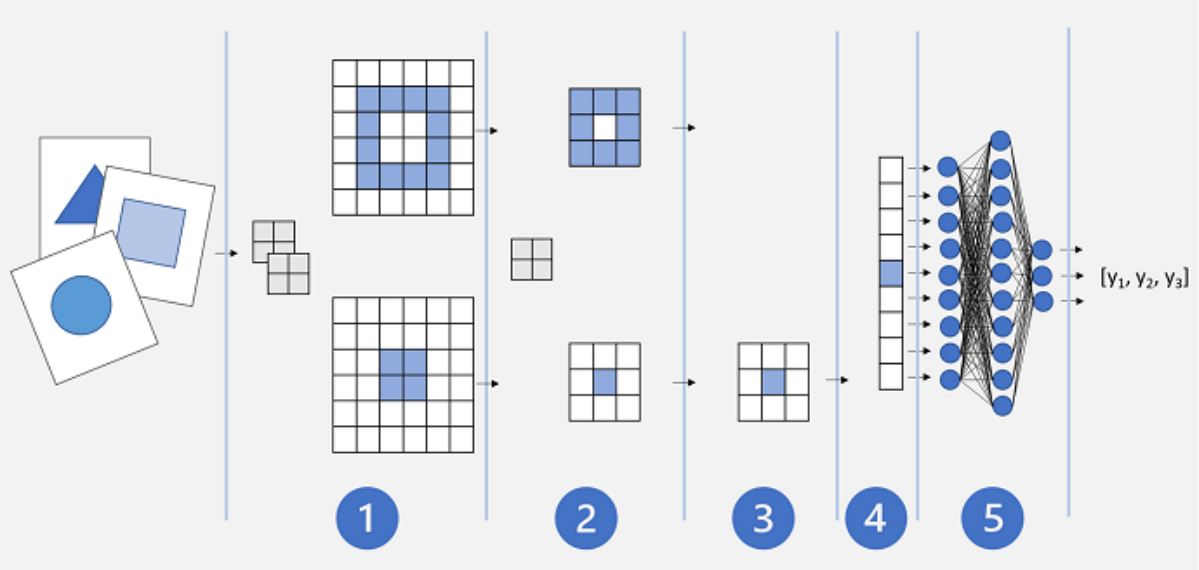
After using convolutional and pooling layers to extract the salient features in the images, the resulting feature maps are multidimensional arrays of pixel values. A

flattening layer is used to flatten the feature maps into a vector of values that can be used as input to a fully connected layer.

Fully connected layers

Usually, a CNN ends with a fully connected network in which the feature values are passed into an input layer, through one or more hidden layers, and generate predicted values in an output layer.

A basic CNN architecture might look similar to this:



1. Images are fed into a convolutional layer. In this case, there are two filters, so each image produces two feature maps.
2. The feature maps are passed to a pooling layer, where a 2x2 pooling kernel reduces the size of the feature maps.
3. A dropping layer randomly drops some of the feature maps to help prevent overfitting.
4. A flattening layer takes the remaining feature map arrays and flattens them into a vector.
5. The vector elements are fed into a fully connected network, which generates the predictions. In this case, the network is a classification model that predicts probabilities for three possible image classes (triangle, square, and circle).

Training a CNN model

As with any deep neural network, a CNN is trained by passing batches of training data through it over multiple epochs, adjusting the weights and bias values based on

the loss calculated for each epoch. In the case of a CNN, backpropagation of adjusted weights includes filter kernel weights used in convolutional layers as well as the weights used in fully connected layers.

Transfer learning

In life, it’s often easier to learn a new skill if you already have expertise in a similar, transferrable skill. For example, it’s probably easier to teach someone how to drive a bus if they have already learned how to drive a car. The driver can build on the driving skills they've already learned in a car, and apply them to driving a bus.

The same principle can be applied to training deep learning models through a technique called transfer learning.

How transfer learning works

A Convolutional Neural Network (CNN) for image classification is typically composed of multiple layers that extract features, and then use a final fully connected layer to classify images based on these features.

A CNN consisting of a set of feature extraction layers and a fully-connected prediction layer

Conceptually, this neural network consists of two distinct sets of layers:

A set of layers from the base model that perform feature extraction.

A fully connected layer that takes the extracted features and uses them for class prediction.

The feature extraction layers apply convolutional filters and pooling to emphasize edges, corners, and other patterns in the images that can be used to differentiate them, and in theory should work for any set of images with the same dimensions as the input layer of the network. The prediction layer maps the features to a set of outputs that represent probabilities for each class label you want to use to classify the images.

By separating the network into these types of layers, we can take the feature extraction layers from a model that has already been trained and append one or more layers to use the extracted features for prediction of the appropriate class labels for your images. This approach enables you to keep the pre-trained weights for the feature extraction layers, which means you only need to train the prediction layers you have added.

There are many established convolutional neural network architectures for image classification that you can use as the base model for transfer learning, so you can build on the work someone else has already done to easily create an effective image classification model.

Run a training script

You can use a ScriptRunConfig to run a script-based experiment that trains a machine learning model.

Writing a script to train a model

When using an experiment to train a model, your script should save the trained model in the outputs folder. For example, the following script trains a model using Scikit-Learn, and saves it in the outputs folder using the joblib package:

Note

This sample code is an incomplete extract that shows the concept of training a model using Scikit-Learn.

Python

Copy

```
from azureml.core import Run

import pandas as pd

import numpy as np

import joblib

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression
```

```
# Get the experiment run context

run = Run.get_context()


# Prepare the dataset

diabetes = pd.read_csv('data.csv')

X, y = diabetes[['Feature1','Feature2','Feature3']].values,
diabetes['Label'].values

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30)


# Train a logistic regression model

reg = 0.1

model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train,
y_train)


# calculate accuracy

y_hat = model.predict(X_test)

acc = np.average(y_hat == y_test)

run.log('Accuracy', np.float(acc))


# Save the trained model

os.makedirs('outputs', exist_ok=True)

joblib.dump(value=model, filename='outputs/model.pkl')


run.complete()
```

To prepare for an experiment that trains a model, a script like this is created and saved in a folder. For example, you could save this script as training_script.py in a

folder named training_folder. Since the script includes code to load training data from data.csv, this file should also be saved in the folder.

Running the script as an experiment

To run the script, create a ScriptRunConfig that references the folder and script file. You generally also need to define a Python (Conda) environment that includes any packages required by the script. In this example, the script uses Scikit-Learn so you must create an environment that includes that. The script also uses Azure Machine Learning to log metrics, so you need to remember to include the azureml-defaults package in the environment.

Python

```
from azureml.core import Experiment, ScriptRunConfig, Environment

from azureml.core.conda_dependencies import CondaDependencies


# Create a Python environment for the experiment

sklearn_env = Environment("sklearn-env")


# Ensure the required packages are installed

packages = CondaDependencies.create(conda_packages=['scikit-
learn','pip'],

                                    pip_packages=['azureml-
defaults'])

sklearn_env.python.conda_dependencies = packages


# Create a script config

script_config = ScriptRunConfig(source_directory='training_folder',

    script='training.py',
environment=sklearn_env)


# Submit the experiment
```

```
experiment = Experiment(workspace=ws, name='training-experiment')

run = experiment.submit(config=script_config)
run.wait_for_completion()
```

Using script parameters

You can increase the flexibility of script-based experiments by using arguments to set variables in the script.

Working with script arguments

To use parameters in a script, you must use a library such as argparse to read the arguments passed to the script and assign them to variables. For example, the following script reads an argument named --reg-rate, which is used to set the regularization rate hyperparameter for the logistic regression algorithm used to train a model.

Python

```
from azureml.core import Run
import argparse
import pandas as pd
import numpy as np
import joblib
import os
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Get the experiment run context
run = Run.get_context()

# Set regularization hyperparameter
parser = argparse.ArgumentParser()
parser.add_argument('--reg-rate', type=float, dest='reg_rate',
default=0.01)
args = parser.parse_args()
reg = args.reg_rate

# Prepare the dataset
diabetes = pd.read_csv('data.csv')
X, y = data[['Feature1','Feature2','Feature3']].values,
data['Label'].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30)

# Train a logistic regression model
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train,
y_train)

# calculate accuracy
y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
run.log('Accuracy', np.float(acc))

# Save the trained model
os.makedirs('outputs', exist_ok=True)
joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()
```

Passing arguments to an experiment script

To pass parameter values to a script being run in an experiment, you need to provide an arguments value containing a list of comma-separated arguments and their values to the ScriptRunConfig, like this:

Python

```
# Create a script config
script_config = ScriptRunConfig(source_directory='training_folder',
                                script='training.py',
                                arguments = ['--reg-rate', 0.1],
                                environment=sklearn_env)
```

Registering models

After running an experiment that trains a model you can use a reference to the Run object to retrieve its outputs, including the trained model.

Retrieving model files

After an experiment run has completed, you can use the run objects get_file_names method to list the files generated. Standard practice is for scripts that train models to save them in the run's outputs folder.

You can also use the run object's download_file and download_files methods to download output files to the local file system.

Python

Copy

```
# "run" is a reference to a completed experiment run


# List the files generated by the experiment

for file in run.get_file_names():

    print(file)


# Download a named file

run.download_file(name='outputs/model.pkl',
output_file_path='model.pkl')
```

Registering a model

Model registration enables you to track multiple versions of a model, and retrieve models for *inferencing* (predicting label values from new data). When you register a model, you can specify a name, description, tags, framework (such as Scikit-Learn or PyTorch), framework version, custom properties, and other useful metadata. Registering a model with the same name as an existing model automatically creates a new version of the model, starting with 1 and increasing in units of 1.

To register a model from a local file, you can use the register method of the Model object as shown here:

Python

Copy

```
from azureml.core import Model


model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model',
                       tags={'data-format': 'CSV'},
                       model_framework=Model.Framework.SCIKITLEARN,
                       model_framework_version='0.20.3')
```

Alternatively, if you have a reference to the Run used to train the model, you can use its register_model method as shown here:

Python

Copy

```
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs
path
                    description='A classification model',
                    tags={'data-format': 'CSV'},
                    model_framework=Model.Framework.SCIKITLEARN,
                    model_framework_version='0.20.3')
```

Viewing registered models

You can view registered models in Azure Machine Learning studio. You can also use the Model object to retrieve details of registered models like this:

Python

Copy

```
from azureml.core import Model


for model in Model.list(ws):
    # Get model name and auto-generated version
    print(model.name, 'version:', model.version)
```

Introduction to datastores

In Azure Machine Learning, *datastores* are abstractions for cloud data sources. They encapsulate the information required to connect to data sources. You can access datastores directly in code by using the Azure Machine Learning SDK, and use it to upload or download data.

Types of Datastore

Azure Machine Learning supports the creation of datastores for multiple kinds of Azure data source, including:

- Azure Storage (blob and file containers)
- Azure Data Lake stores
- Azure SQL Database
- Azure Databricks file system (DBFS)

Note: For a full list of supported datastores, see the Azure Machine Learning documentation.

Built-in Datastores

Every workspace has two built-in datastores (an Azure Storage blob container, and an Azure Storage file container) that are used as system storage by Azure Machine Learning. There's also a third datastore that gets added to your workspace if you make use of the open datasets provided as samples (for example, by creating a designer pipeline based on a sample dataset)

In most machine learning projects, you will likely need to work with data sources of your own - either because you need to store larger volumes of data than the built-in datastores support, or because you need to integrate your machine learning solution with data from existing applications.

Use datastores

To add a datastore to your workspace, you can register it using the graphical interface in Azure Machine Learning studio, or you can use the Azure Machine Learning SDK. For example, the following code registers an Azure Storage blob container as a datastore named blob_data.

Python

```
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()

# Register a new datastore
blob_ds = Datastore.register_azure_blob_container(workspace=ws,
                                                  datastore_name='blob_data',
                                                  container_name='data_container',
                                                  account_name='az_store_acct',
                                                  account_key='123456abcde789…')
```

Managing datastores

You can view and manage datastores in Azure Machine Learning Studio, or you can use the Azure Machine Learning SDK. For example, the following code lists the names of each datastore in the workspace.

Python Copy

```
for ds_name in ws.datastores:
    print(ds_name)
```

You can get a reference to any datastore by using the **Datastore.get()** method as shown here:

Python Copy

```
blob_store = Datastore.get(ws, datastore_name='blob_data')
```

The workspace always includes a *default* datastore (initially, this is the built-in **workspaceblobstore** datastore), which you can retrieve by using the **get_default_datastore()** method of a **Workspace** object, like this:

Python Copy

```
default_store = ws.get_default_datastore()
```

Considerations for datastores

When planning for datastores, consider the following guidelines:

- When using Azure blob storage, *premium* level storage may provide improved I/O performance for large datasets. However, this option will increase cost and may limit replication options for data redundancy.
- When working with data files, although CSV format is very common, Parquet format generally results in better performance.
- You can access any datastore by name, but you may want to consider changing the default datastore (which is initially the built-in workspaceblobstore datastore).

To change the default datastore, use the set_default_datastore() method:

```
Python

ws.set_default_datastore('blob_data')
```

Introduction to datasets

Datasets are versioned packaged data objects that can be easily consumed in experiments and pipelines. Datasets are the recommended way to work with data, and are the primary mechanism for advanced Azure Machine Learning capabilities like data labeling and data drift monitoring.

Types of dataset

Datasets are typically based on files in a datastore, though they can also be based on URLs and other sources. You can create the following types of dataset:

- Tabular: The data is read from the dataset as a table. You should use this type of dataset when your data is consistently structured and you want to work with it in common tabular data structures, such as Pandas dataframes.
- File: The dataset presents a list of file paths that can be read as though from the file system. Use this type of dataset when your data is

unstructured, or when you need to process the data at the file level (for example, to train a convolutional neural network from a set of image files).

Creating and registering datasets

You can use the visual interface in Azure Machine Learning studio or the Azure Machine Learning SDK to create datasets from individual files or multiple file paths. The paths can include wildcards (for example, */files/*.csv*) making it possible to encapsulate data from a large number of files in a single dataset.

After you've created a dataset, you can *register* it in the workspace to make it available for use in experiments and data processing pipelines later.

Creating and registering tabular datasets

To create a tabular dataset using the SDK, use the from_delimited_files method of the Dataset.Tabular class, like this:

Python

```
from azureml.core import Dataset

blob_ds = ws.get_default_datastore()
csv_paths = [(blob_ds, 'data/files/current_data.csv'),
             (blob_ds, 'data/files/archive/*.csv')]
tab_ds = Dataset.Tabular.from_delimited_files(path=csv_paths)
tab_ds = tab_ds.register(workspace=ws, name='csv_table')
```

The dataset in this example includes data from two file paths within the default datastore:

- The current_data.csv file in the data/files folder.
- All .csv files in the data/files/archive/ folder.

After creating the dataset, the code registers it in the workspace with the name csv_table.

Creating and registering file datasets

To create a file dataset using the SDK, use the from_files method of the Dataset.File class, like this:

```python
from azureml.core import Dataset

blob_ds = ws.get_default_datastore()
file_ds = Dataset.File.from_files(path=(blob_ds, 'data/files/images/*.jpg'))
file_ds = file_ds.register(workspace=ws, name='img_files')
```

The dataset in this example includes all .jpg files in the data/files/images path within the default datastore:

After creating the dataset, the code registers it in the workspace with the name img_files.

Retrieving a registered dataset

After registering a dataset, you can retrieve it by using any of the following techniques:

- The datasets dictionary attribute of a Workspace object.
- The get_by_name or get_by_id method of the Dataset class.

Both of these techniques are shown in the following code:

```python
import azureml.core
from azureml.core import Workspace, Dataset

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Get a dataset from the workspace datasets collection
ds1 = ws.datasets['csv_table']

# Get a dataset by name from the datasets class
ds2 = Dataset.get_by_name(ws, 'img_files')
```

Dataset versioning

Datasets can be *versioned*, enabling you to track historical versions of datasets that were used in experiments, and reproduce those experiments with data in the same state.

You can create a new version of a dataset by registering it with the same name as a previously registered dataset and specifying the create_new_version property:

```
Python

img_paths = [(blob_ds, 'data/files/images/*.jpg'),
             (blob_ds, 'data/files/images/*.png')]
file_ds = Dataset.File.from_files(path=img_paths)
file_ds = file_ds.register(workspace=ws, name='img_files', create_new_version=True)
```

Retrieving a specific dataset version

You can retrieve a specific version of a dataset by specifying the version parameter in the get_by_name method of the Dataset class.

```
Python

img_ds = Dataset.get_by_name(workspace=ws, name='img_files', version=2)
```

Use datasets

Datasets are the primary way to pass data to experiments that train models.

Work with tabular datasets

You can read data directly from a tabular dataset by converting it into a Pandas or Spark dataframe:

```
Python

df = tab_ds.to_pandas_dataframe()
# code to work with dataframe goes here, for example:
print(df.head())
```

Pass a tabular dataset to an experiment script

When you need to access a dataset in an experiment script, you must pass the dataset to the script. There are two ways you can do this.

Use a script argument for a tabular dataset

You can pass a tabular dataset as a script argument. When you take this approach, the argument received by the script is the unique ID for the dataset in your workspace. In the script, you can then get the workspace from the run context and use it to retrieve the dataset by it's ID.

ScriptRunConfig:

```
Python

env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pandas]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', tab_ds],
                                environment=env)
```

Script:

```
Python

from azureml.core import Run, Dataset

parser.add_argument('--ds', type=str, dest='dataset_id')
args = parser.parse_args()

run = Run.get_context()
ws = run.experiment.workspace
dataset = Dataset.get_by_id(ws, id=args.dataset_id)
data = dataset.to_pandas_dataframe()
```

Use a named input for a tabular dataset

Alternatively, you can pass a tabular dataset as a *named input*. In this approach, you use the as_named_input method of the dataset to specify a name for the dataset.

Then in the script, you can retrieve the dataset by name from the run context's input_datasets collection without needing to retrieve it from the workspace. Note that if you use this approach, you still need to include a script argument for the dataset, even though you don't actually use it to retrieve the dataset.

ScriptRunConfig:

Python Copy

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pandas]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', tab_ds.as_named_input('my_dataset')],
                                environment=env)
```

Script:

Python Copy

```
from azureml.core import Run

parser.add_argument('--ds', type=str, dest='ds_id')
args = parser.parse_args()

run = Run.get_context()
dataset = run.input_datasets['my_dataset']
data = dataset.to_pandas_dataframe()
```

Work with file datasets

When working with a file dataset, you can use the to_path() method to return a list of the file paths encapsulated by the dataset:

Python

```
for file_path in file_ds.to_path():
    print(file_path)
```

Pass a file dataset to an experiment script

Just as with a Tabular dataset, there are two ways you can pass a file dataset to a script. However, there are some key differences in the way that the dataset is passed.

Use a script argument for a file dataset

You can pass a file dataset as a script argument. Unlike with a tabular dataset, you must specify a mode for the file dataset argument, which can be as_download or as_mount. This provides an access point that the script can use to read the files in the dataset. In most cases, you should use as_download, which copies the files to a temporary location on the compute where the script is being run. However, if you are working with a large amount of data for which there may not be enough storage space on the experiment compute, use as_mount to stream the files directly from their source.

ScriptRunConfig:

Python

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pandas]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', file_ds.as_download()],
                                environment=env)
```

Script:

Python

```
from azureml.core import Run
import glob

parser.add_argument('--ds', type=str, dest='ds_ref')
args = parser.parse_args()
run = Run.get_context()

imgs = glob.glob(args.ds_ref + "/*.jpg")
```

Use a named input for a file dataset

You can also pass a file dataset as a *named input*. In this approach, you use the as_named_input method of the dataset to specify a name before specifying the access mode. Then in the script, you can retrieve the dataset by name from the run context's input_datasets collection and read the files from there. As with tabular datasets, if you use a named input, you still need to include a script argument for the dataset, even though you don't actually use it to retrieve the dataset.

ScriptRunConfig:

```
Python

env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pandas]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', file_ds.as_named_input('my_ds').as_download()],
                                environment=env)
```

Script:

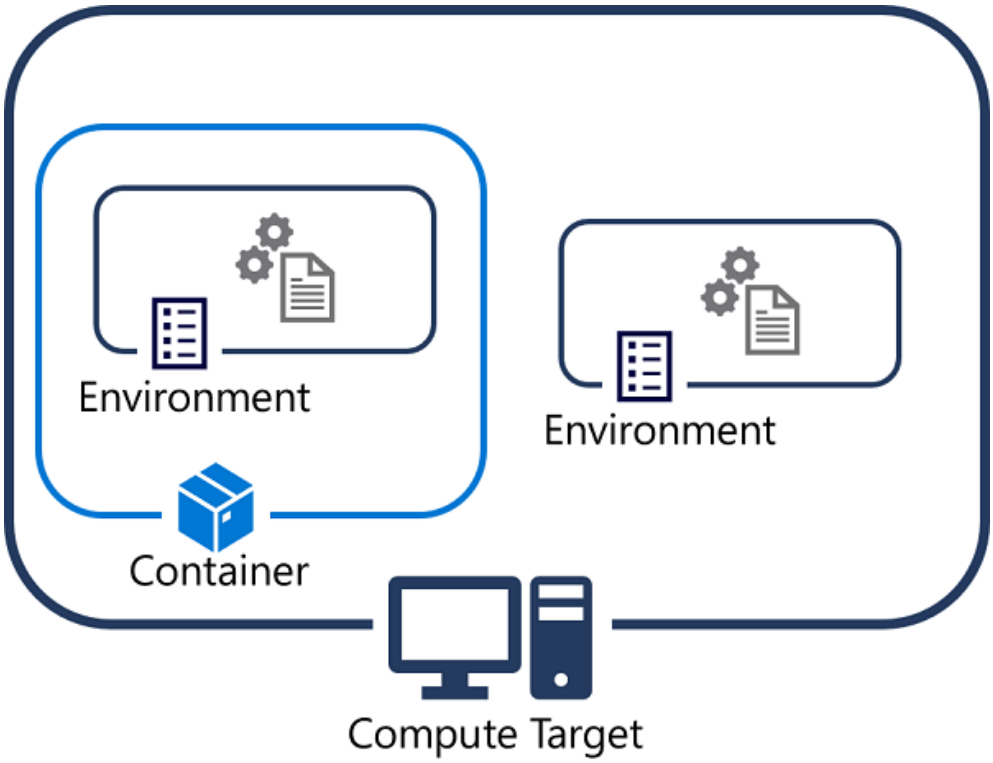
```
Python

from azureml.core import Run
import glob

parser.add_argument('--ds', type=str, dest='ds_ref')
args = parser.parse_args()
run = Run.get_context()

dataset = run.input_datasets['my_ds']
imgs= glob.glob(dataset + "/*.jpg")
```

Introduction-Computing in Azure

In Azure Machine Learning, data scientists can run *experiments* based on scripts that process data, train machine learning models, and perform other data science tasks. The runtime context for each experiment run consists of two elements:

- The *environment* for the script, which includes all packages on which the script depends.
- The *compute target* on which the environment will be deployed and the script run.

Introduction to environments



Python code runs in the context of a *virtual environment* that defines the version of the Python runtime to be used as well as the installed packages available to the code. In most Python installations, packages are installed and managed in environments using Conda or pip.

To improve portability, we usually create environments in docker containers that are in turn be hosted in compute targets, such as your development computer, virtual machines, or clusters in the cloud.

Environments in Azure Machine Learning

In general, Azure Machine Learning handles environment creation and package installation for you - usually through the creation of Docker containers. You can specify the Conda or pip packages you need, and have Azure Machine Learning create an environment for the experiment.

In an enterprise machine learning solution, where experiments may be run in a variety of compute contexts, it can be important to be aware of the environments in which your experiment code is running. Environments are encapsulated by the Environment class; which you can use to create environments and specify runtime configuration for an experiment.

You can have Azure Machine Learning manage environment creation and package installation to define an environment, and then register it for reuse. Alternatively, you can manage your own environments and register them. This makes it possible to define consistent, reusable runtime contexts for your experiments - regardless of where the experiment script is run.

Creating environments

There are multiple ways to create environments in Azure Machine Learning.

Creating an environment from a specification file

You can use a Conda or pip specification file to define the packages required in a Python environment, and use it to create an Environment object.

For example, you could save the following Conda configuration settings in a file named conda.yml:

Bash Copy

```
name: py_env
dependencies:
  - numpy
  - pandas
  - scikit-learn
  - pip:
    - azureml-defaults
```

You could then use the following code to create an Azure Machine Learning environment from the saved specification file:

Python Copy

```
from azureml.core import Environment

env = Environment.from_conda_specification(name='training_environment',
                                           file_path='./conda.yml')
```

Creating an environment from an existing Conda environment

If you have an existing Conda environment defined on your workstation, you can use it to define an Azure Machine Learning environment:

Python Copy

```
from azureml.core import Environment

env = Environment.from_existing_conda_environment(name='training_environment',
                                                  conda_environment_name='py_env')
```

Creating an environment by specifying packages

You can define an environment by specifying the Conda and pip packages you need in a **CondaDependencies** object, like this:

Python Copy

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

env = Environment('training_environment')
deps = CondaDependencies.create(conda_packages=['scikit-learn','pandas','numpy'],
                                pip_packages=['azureml-defaults'])
env.python.conda_dependencies = deps
```


Configuring environment containers

Usually, environments for experiment script are created in containers. The following code configures a script-based experiment to host the env environment created previously in a container (this is the default unless you use a DockerConfiguration with a use_docker attribute of False, in which case the environment is created directly in the compute target)

Python

```
from azureml.core import Experiment, ScriptRunConfig
from azureml.core.runconfig import DockerConfiguration

docker_config = DockerConfiguration(use_docker=True)

script_config = ScriptRunConfig(source_directory='my_folder',
                                script='my_script.py',
                                environment=env,
                                docker_runtime_config=docker_config)
```

Azure Machine Learning uses a library of base images for containers, choosing the appropriate base for the compute target you specify (for example, including Cuda

support for GPU-based compute). If you have created custom container images and registered them in a container registry, you can override the default base images and use your own by modifying the attributes of the environment's docker property..

```python
env.docker.base_image='my-base-image'
env.docker.base_image_registry='myregistry.azurecr.io/myimage'
```

Alternatively, you can have an image created on-demand based on the base image and additional settings in a dockerfile.

```python
env.docker.base_image = None
env.docker.base_dockerfile = './Dockerfile'
```

By default, Azure machine Learning handles Python paths and package dependencies. If your image already includes an installation of Python with the dependencies you need, you can override this behavior by setting **python.user_managed_dependencies** to **True** and setting an explicit Python path for your installation.

```python
env.python.user_managed_dependencies=True
env.python.interpreter_path = '/opt/miniconda/bin/python'
```

Registering and reusing environments

After you've created an environment, you can register it in your workspace and reuse it for future experiments that have the same Python dependencies.

Registering an environment

Use the register method of an Environment object to register an environment:

```python
env.register(workspace=ws)
```

You can view the registered environments in your workspace like this:

```python
from azureml.core import Environment

env_names = Environment.list(workspace=ws)
for env_name in env_names:
    print('Name:',env_name)
```

Retrieving and using an environment

You can retrieve a registered environment by using the get method of the Environment class, and then assign it to a ScriptRunConfig.

For example, the following code sample retrieves the *training_environment* registered environment, and assigns it to a script run configuration:

```python
from azureml.core import Environment, ScriptRunConfig

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_folder',
                                script='my_script.py',
                                environment=training_env)
```

When an experiment based on the estimator is run, Azure Machine Learning will look for an existing environment that matches the definition, and if none is found a new environment will be created based on the registered environment specification.

Introduction to compute targets

In Azure Machine Learning, *Compute Targets* are physical or virtual computers on which experiments are run.

Types of compute

Azure Machine Learning supports multiple types of compute for experimentation and training. This enables you to select the most appropriate type of compute target for your particular needs.

- Local compute - You can specify a local compute target for most processing tasks in Azure Machine Learning. This runs the experiment on the same compute target as the code used to initiate the experiment, which may be your physical workstation or a virtual machine such as an Azure Machine Learning *compute instance* on which you are running a notebook. Local compute is generally a great choice during development and testing with low to moderate volumes of data.
- Compute clusters - For experiment workloads with high scalability requirements, you can use Azure Machine Learning compute clusters; which are multi-node clusters of Virtual Machines that automatically scale up or down to meet demand. This is a cost-effective way to run experiments that need to handle large volumes of data or use parallel processing to distribute the workload and reduce the time it takes to run.
- Attached compute - If you already use an Azure-based compute environment for data science, such as a virtual machine or an Azure Databricks cluster, you can attach it to your Azure Machine Learning workspace and use it as a compute target for certain types of workload.

Note

In Azure Machine Learning studio, you can create another type of compute named *inference clusters*. This kind of compute represents an Azure Kubernetes Service cluster and can only be used to deploy trained models as inferencing services. We'll explore deployment later, but for now we'll focus on compute for experiments and model training.

The ability to assign experiment runs to specific compute targets helps you implement a flexible data science ecosystem in the following ways:

- Code can be developed and tested on local or low-cost compute, and then moved to more scalable compute for production workloads.

- You can run individual processes on the compute target that best fits its needs. For example, by using GPU-based compute to train deep learning models, and switching to lower-cost CPU-only compute to test and register the trained model.

One of the core benefits of cloud computing is the ability to manage costs by paying only for what you use. In Azure Machine Learning, you can take advantage of this principle by defining compute targets that:

- Start on-demand and stop automatically when no longer required.
- Scale automatically based on workload processing needs.

Create compute targets

The most common ways to create or attach a compute target are to use the Compute page in Azure Machine Learning studio, or to use the Azure Machine Learning SDK to provision compute targets in code.

Creating a managed compute target with the SDK

A *managed* compute target is one that is managed by Azure Machine Learning, such as an Azure Machine Learning compute cluster.

To create an Azure Machine Learning compute cluster, use the azureml.core.compute.ComputeTarget class and the AmlCompute class, like this:

Python

```python
from azureml.core import Workspace
from azureml.core.compute import ComputeTarget, AmlCompute

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'aml-cluster'

# Define compute configuration
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_DS11_V2',
                                                       min_nodes=0, max_nodes=4,
                                                       vm_priority='dedicated')

# Create the compute
aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)
aml_cluster.wait_for_completion(show_output=True)
```

In this example, a cluster with up to four nodes that is based on the STANDARD_DS12_v2 virtual machine image will be created. The priority for the virtual machines (VMs) is set to dedicated, meaning they are reserved for use in this cluster (the alternative is to specify *lowpriority*, which has a lower cost but means that the VMs can be preempted if a higher-priority workload requires the compute).
Note

For a full list of AmlCompute configuration options, see the AmlCompute class SDK documentation.

Attaching an unmanaged compute target with the SDK

An *unmanaged* compute target is one that is defined and managed outside of the Azure Machine Learning workspace; for example, an Azure virtual machine or an Azure Databricks cluster.

The code to attach an existing unmanaged compute target is similar to the code used to create a managed compute target, except that you must use the ComputeTarget.attach() method to attach the existing compute based on its target-specific configuration settings.

For example, the following code can be used to attach an existing Azure Databricks cluster:

Python

```
from azureml.core import Workspace
from azureml.core.compute import ComputeTarget, DatabricksCompute

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'db_cluster'

# Define configuration for existing Azure Databricks cluster
db_workspace_name = 'db_workspace'
db_resource_group = 'db_resource_group'
db_access_token = '1234-abc-5678-defg-90...'
db_config = DatabricksCompute.attach_configuration(resource_group=db_resource_group,
                                                   workspace_name=db_workspace_name,
                                                   access_token=db_access_token)

# Create the compute
databricks_compute = ComputeTarget.attach(ws, compute_name, db_config)
databricks_compute.wait_for_completion(True)
```

Checking for an existing compute target

In many cases, you will want to check for the existence of a compute target, and only create a new one if there isn't already one with the specified name. To accomplish this, you can catch the ComputeTargetException exception, like this:

Python

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

compute_name = "aml-cluster"

# Check if the compute target exists
try:
    aml_cluster = ComputeTarget(workspace=ws, name=compute_name)
    print('Found existing cluster.')
except ComputeTargetException:
    # If not, create it
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_DS11_V2',
                                                           max_nodes=4)
    aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)

aml_cluster.wait_for_completion(show_output=True)
```

Use compute targets

After you've created or attached compute targets in your workspace, you can use them to run specific workloads; such as experiments.

To use a particular compute target, you can specify it in the appropriate parameter for an experiment run configuration or estimator. For example, the following code configures an estimator to use the compute target named *aml-cluster*:

Python

```
from azureml.core import Environment, ScriptRunConfig

compute_name = 'aml-cluster'

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=training_env,
                                compute_target=compute_name)
```

When an experiment is submitted, the run will be queued while the *aml-cluster* compute target is started and the specified environment created on it, and then the run will be processed on the compute environment.

Instead of specifying the name of the compute target, you can specify a ComputeTarget object, like this:

Python

```
from azureml.core import Environment, ScriptRunConfig
from azureml.core.compute import ComputeTarget

compute_name = "aml-cluster"

training_cluster = ComputeTarget(workspace=ws, name=compute_name)

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=training_env,
                                compute_target=training_cluster)
```

Orchestrate machine learning with pipelines

Introduction

In Azure Machine learning, you run workloads as experiments that leverage data assets and compute resources. In an enterprise data science process, you'll generally want to separate the overall process into individual tasks, and orchestrate these tasks as *pipelines* of connected steps. Pipelines are key to implementing an effective Machine Learning Operationalization (ML Ops) solution in Azure, so you'll explore how to define and run them in this module.

Note

The term *pipeline* is used extensively in machine learning, often with different meanings. For example, in Scikit-Learn, you can define pipelines that combine data preprocessing transformations with a training algorithm; and in Azure DevOps, you can define a build or release pipeline to perform the build and configuration tasks required to deliver software. The focus of this module is on Azure Machine Learning pipelines, which encapsulate steps that can be run as an experiment. However, bear in mind that it's perfectly feasible to have an Azure DevOps pipeline with a task that initiates an Azure Machine Learning pipeline, which in turn includes a step that trains a model based on a Scikit-Learn pipeline!

Introduction to pipelines

In Azure Machine Learning, a *pipeline* is a workflow of machine learning tasks in which each task is implemented as a *step*.

Steps can be arranged sequentially or in parallel, enabling you to build sophisticated flow logic to orchestrate machine learning operations. Each step can be run on a specific compute target, making it possible to combine different types of processing as required to achieve an overall goal.

A pipeline can be executed as a process by running the pipeline as an experiment. Each step in the pipeline runs on its allocated compute target as part of the overall experiment run.

You can publish a pipeline as a REST endpoint, enabling client applications to initiate a pipeline run. You can also define a schedule for a pipeline, and have it run automatically at periodic intervals.

Pipeline steps

An Azure Machine Learning pipeline consists of one or more *steps* that perform tasks. There are many kinds of steps supported by Azure Machine Learning pipelines, each with its own specialized purpose and configuration options.

Common kinds of step in an Azure Machine Learning pipeline include:

- PythonScriptStep: Runs a specified Python script.
- DataTransferStep: Uses Azure Data Factory to copy data between data stores.
- DatabricksStep: Runs a notebook, script, or compiled JAR on a databricks cluster.
- AdlaStep: Runs a U-SQL job in Azure Data Lake Analytics.
- ParallelRunStep - Runs a Python script as a distributed task on multiple compute nodes.

Note: For a full list of supported step types, see azure.pipeline.steps package documentation.

To create a pipeline, you must first define each step, and then create a pipeline that includes the steps. The specific configuration of each step depends on the step type. For example, the following code defines two PythonScriptStep steps to prepare data, and then train a model.

Python

```python
from azureml.pipeline.steps import PythonScriptStep

# Step to run a Python script
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster')

# Step to train a model
step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'train_model.py',
                         compute_target = 'aml-cluster')
```

After defining the steps, you can assign them to a pipeline, and run it as an experiment:

Python

```python
from azureml.pipeline.core import Pipeline
from azureml.core import Experiment

# Construct the pipeline
train_pipeline = Pipeline(workspace = ws, steps = [step1,step2])

# Create an experiment and run the pipeline
experiment = Experiment(workspace = ws, name = 'training-pipeline')
pipeline_run = experiment.submit(train_pipeline)
```

Pass data between pipeline steps

Completed

100 XP

- 5 minutes

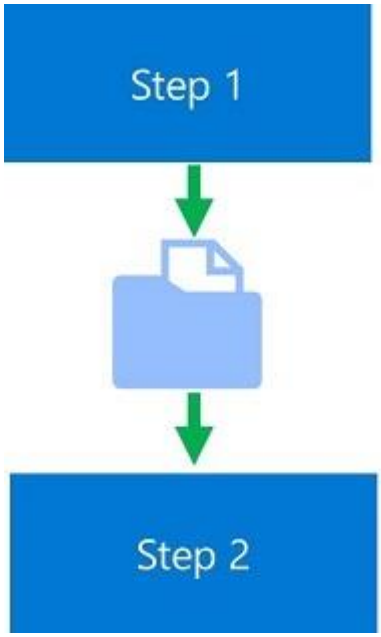
Often, a pipeline line includes at least one step that depends on the output of a preceding step. For example, you might use a step that runs a python script to preprocess some data, which must then be used in a subsequent step to train a model.

The OutputFileDatasetConfig Object

The OutputFileDatasetConfig object is a special kind of dataset that:

- References a location in a datastore for interim storage of data.
- Creates a data dependency between pipeline steps.

You can view a OutputFileDatasetConfig object as an intermediary store for data that must be passed from a step to a subsequent step.



OutputFileDatasetConfig Step Inputs and Outputs

To use a OutputFileDatasetConfig object to pass data between steps, you must:

1. Define a named OutputFileDatasetConfig object that references a location in a datastore. If no explicit datastore is specified, the default datastore is used.
2. Pass the OutputFileDatasetConfig object as a script argument in steps that run scripts.
3. Include code in those scripts to write to the OutputFileDatasetConfig argument as an output or read it as an input.

For example, the following code defines a OutputFileDatasetConfig object that for the preprocessed data that must be passed between the steps.

Python

```python
from azureml.data import OutputFileDatasetConfig
from azureml.pipeline.steps import PythonScriptStep, EstimatorStep

# Get a dataset for the initial data
raw_ds = Dataset.get_by_name(ws, 'raw_dataset')

# Define a PipelineData object to pass data between steps
data_store = ws.get_default_datastore()
prepped_data = OutputFileDatasetConfig('prepped')

# Step to run a Python script
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster',
                         # Script arguments include PipelineData
                         arguments = ['--raw-ds', raw_ds.as_named_input('raw_data'),
                                      '--out_folder', prepped_data])

# Step to run an estimator
step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'train_model.py',
                         compute_target = 'aml-cluster',
                         # Pass as script argument
                         arguments=['--training-data', prepped_data.as_input()])
```

In the scripts themselves, you can obtain a reference to the OutputFileDatasetConfig object from the script argument, and use it like a local folder.

Python

```python
# code in data_prep.py
from azureml.core import Run
import argparse
import os

# Get the experiment run context
run = Run.get_context()

# Get arguments
parser = argparse.ArgumentParser()
parser.add_argument('--raw-ds', type=str, dest='raw_dataset_id')
parser.add_argument('--out_folder', type=str, dest='folder')
args = parser.parse_args()
output_folder = args.folder

# Get input dataset as dataframe
raw_df = run.input_datasets['raw_data'].to_pandas_dataframe()

# code to prep data (in this case, just select specific columns)
prepped_df = raw_df[['col1', 'col2', 'col3']]

# Save prepped data to the PipelineData location
os.makedirs(output_folder, exist_ok=True)
output_path = os.path.join(output_folder, 'prepped_data.csv')
prepped_df.to_csv(output_path)
```

Reuse pipeline steps

Pipelines with multiple long-running steps can take a significant time to complete. Azure Machine Learning includes some caching and reuse features to reduce this time.

Managing step output reuse

By default, the step output from a previous pipeline run is reused without rerunning the step provided the script, source directory, and other parameters for the step haven't changed. Step reuse can reduce the time it takes to run a pipeline, but it can lead to stale results when changes to downstream data sources haven't been accounted for.

To control reuse for an individual step, you can set the allow_reuse parameter in the step configuration, like this:

```python
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster',
                         runconfig = run_config,
                         inputs=[raw_ds.as_named_input('raw_data')],
                         outputs=[prepped_data],
                         arguments = ['--folder', prepped_data]),
                         # Disable step reuse
                         allow_reuse = False)
```

Forcing all steps to run

When you have multiple steps, you can force all of them to run regardless of individual reuse configuration by setting the **regenerate_outputs** parameter when submitting the pipeline experiment:

```python
pipeline_run = experiment.submit(train_pipeline, regenerate_outputs=True)
```

Publish pipelines

After you've created a pipeline, you can publish it to create a REST endpoint through which the pipeline can be run on demand.

Publishing a pipeline

To publish a pipeline, you can call its publish method, as shown here:

Python Copy

```
published_pipeline = pipeline.publish(name='training_pipeline',
                                      description='Model training pipeline',
                                      version='1.0')
```

Alternatively, you can call the **publish** method on a successful run of the pipeline:

Python Copy

```
# Get the most recent run of the pipeline
pipeline_experiment = ws.experiments.get('training-pipeline')
run = list(pipeline_experiment.get_runs())[0]

# Publish the pipeline from the run
published_pipeline = run.publish_pipeline(name='training_pipeline',
                                      description='Model training pipeline',
                                      version='1.0')
```

After the pipeline has been published, you can view it in Azure Machine Learning studio. You can also determine the URI of its endpoint like this:

Python Copy

```
rest_endpoint = published_pipeline.endpoint
print(rest_endpoint)
```

Using a published pipeline

To initiate a published endpoint, you make an HTTP request to its REST endpoint, passing an authorization header with a token for a service principal with permission to run the pipeline, and a JSON payload specifying the experiment name. The pipeline is run asynchronously, so the response from a successful REST call includes the run ID. You can use the run ID to track the run in Azure Machine Learning studio.

For example, the following Python code makes a REST request to run a pipeline and displays the returned run ID.

Python

```
import requests

response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "run_training_pipeline"})
run_id = response.json()["Id"]
print(run_id)
```

Use pipeline parameters

You can increase the flexibility of a pipeline by defining parameters.

Defining parameters for a pipeline

To define parameters for a pipeline, create a PipelineParameter object for each parameter, and specify each parameter in at least one step.

For example, you could use the following code to include a parameter for a regularization rate in the script used by an estimator:

Python

```
from azureml.pipeline.core.graph import PipelineParameter

reg_param = PipelineParameter(name='reg_rate', default_value=0.01)

...

step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'train_model.py',
                         compute_target = 'aml-cluster',
                         # Pass parameter as script argument
                         arguments=['--in_folder', prepped_data,
                                    '--reg', reg_param],
                         inputs=[prepped_data])
```

Note

You must define parameters for a pipeline before publishing it.

Running a pipeline with a parameter

After you publish a parameterized pipeline, you can pass parameter values in the JSON payload for the REST interface:

Python

```
response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "run_training_pipeline",
                               "ParameterAssignments": {"reg_rate": 0.1}})
```

Schedule pipelines

After you've published a pipeline, you can initiate it on demand through its REST endpoint, or you can have the pipeline run automatically based on a periodic schedule or in response to data updates.

Scheduling a pipeline for periodic intervals

To schedule a pipeline to run at periodic intervals, you must define a ScheduleRecurrence that determines the run frequency, and use it to create a Schedule.

For example, the following code schedules a daily run of a published pipeline.

Python

```
from azureml.pipeline.core import ScheduleRecurrence, Schedule

daily = ScheduleRecurrence(frequency='Day', interval=1)
pipeline_schedule = Schedule.create(ws, name='Daily Training',
                                    description='trains model every day',
                                    pipeline_id=published_pipeline.id,
                                    experiment_name='Training_Pipeline',
                                    recurrence=daily)
```

Triggering a pipeline run on data changes

To schedule a pipeline to run whenever data changes, you must create a Schedule that monitors a specified path on a datastore, like this:

```python
from azureml.core import Datastore
from azureml.pipeline.core import Schedule

training_datastore = Datastore(workspace=ws, name='blob_data')
pipeline_schedule = Schedule.create(ws, name='Reactive Training',
                                    description='trains model on data change',
                                    pipeline_id=published_pipeline.id,
                                    experiment_name='Training_Pipeline',
                                    datastore=training_datastore,
                                    path_on_datastore='data/training')
```

Exercise - Create a pipeline

Now it's your chance to create and run an Azure Machine Learning pipeline.

In this exercise, you will:

- Create an Azure Machine Learning pipeline.
- Publish a pipeline as a REST service.
- Schedule a pipeline.

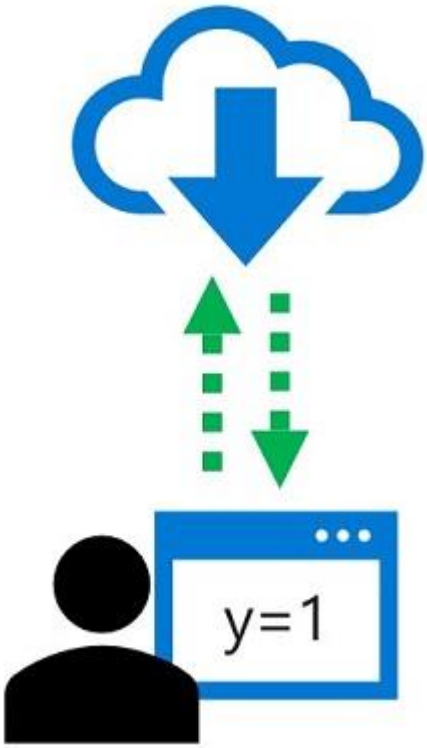
Deploy real-time machine learning services with Azure Machine Learning

In this module, you will learn how to:

- Deploy a model as a real-time inferencing service.
- Consume a real-time inferencing service.
- Troubleshoot service deployment

Introduction

In machine learning, *inferencing* refers to the use of a trained model to predict labels for new data on which the model has not been trained. Often, the model is deployed as part of a service that enables applications to request immediate, or *real-time*, predictions for individual, or small numbers of data observations.



In Azure Machine Learning, you can create real-time inferencing solutions by deploying a model as a service, hosted in a containerized platform, such as Azure Kubernetes Services (AKS).

Deploy a model as a real-time service

You can deploy a model as a real-time web service to several kinds of compute target, including local compute, an Azure Machine Learning compute instance, an Azure Container Instance (ACI), an Azure Kubernetes Service (AKS) cluster, an Azure Function, or an Internet of Things (IoT) module. Azure Machine Learning uses *containers* as a deployment mechanism, packaging the model and the code to use it as an image that can be deployed to a container in your chosen compute target.

Note

Deployment to a local service, a compute instance, or an ACI is a good choice for testing and development. For production, you should deploy to a target that meets the specific performance, scalability, and security needs of your application architecture.

To deploy a model as a real-time inferencing service, you must perform the following tasks:

1. Register a trained model

After successfully training a model, you must register it in your Azure Machine Learning workspace. Your real-time service will then be able to load the model when required.

To register a model from a local file, you can use the register method of the Model object as shown here:

```python
from azureml.core import Model

classification_model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```python
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model')
```

2. Define an inference configuration

The model will be deployed as a service that consist of:

- A script to load the model and return predictions for submitted data.
- An environment in which the script will be run.

You must therefore define the script and environment for the service.

Create an entry script

Create the *entry script* (sometimes referred to as the *scoring script*) for the service as a Python (.py) file. It must include two functions:

- init(): Called when the service is initialized.
- run(raw_data): Called when new data is submitted to the service.

Typically, you use the init function to load the model from the model registry, and use the run function to generate predictions from the input data. The following example script shows this pattern:

Python

```
import json
import joblib
import numpy as np
import os

# Called when the service is loaded
def init():
    global model
    # Get the path to the registered model file and load it
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'model.pkl')
    model = joblib.load(model_path)

# Called when a request is received
def run(raw_data):
    # Get the input data as a numpy array
    data = np.array(json.loads(raw_data)['data'])
    # Get a prediction from the model
    predictions = model.predict(data)
    # Return the predictions as any JSON serializable format
    return predictions.tolist()
```

Save the script in a folder so you can easily identify it later. For example, you might save the script above as *score.py* in a folder named *service_files*.

Create an environment

Your service requires a Python environment in which to run the entry script, which you can define by creating an Environment that contains the required packages:

```python
from azureml.core import Environment

service_env = Environment(name='service-env')
python_packages = ['scikit-learn', 'numpy'] # whatever packages your entry script uses
for package in python_packages:
    service_env.python.conda_dependencies.add_pip_package(package)
```

Combine the script and environment in an InferenceConfig

After creating the entry script and environment, you can combine them in an **InferenceConfig** for the service like this:

```python
from azureml.core.model import InferenceConfig

classifier_inference_config = InferenceConfig(source_directory = 'service_files',
                                              entry_script="score.py",
                                              environment=service_env)
```

3. Define a deployment configuration

Now that you have the entry script and environment, you need to configure the compute to which the service will be deployed. If you are deploying to an AKS cluster, you must create the cluster and a compute target for it before deploying:

```python
from azureml.core.compute import ComputeTarget, AksCompute

cluster_name = 'aks-cluster'
compute_config = AksCompute.provisioning_configuration(location='eastus')
production_cluster = ComputeTarget.create(ws, cluster_name, compute_config)
production_cluster.wait_for_completion(show_output=True)
```

With the compute target created, you can now define the deployment configuration, which sets the target-specific compute specification for the containerized deployment:

```python
from azureml.core.webservice import AksWebservice

classifier_deploy_config = AksWebservice.deploy_configuration(cpu_cores = 1,
                                                              memory_gb = 1)
```

The code to configure an ACI deployment is similar, except that you do not need to explicitly create an ACI compute target, and you must use the deploy_configuration class from the azureml.core.webservice.AciWebservice namespace. Similarly, you can use the azureml.core.webservice.LocalWebservice namespace to configure a local Docker-based service.

Note

To deploy a model to an Azure Function, you do not need to create a deployment configuration. Instead, you need to package the model based on the type of function trigger you want to use. This functionality is in preview at the time of writing. For more details, see Deploy a machine learning model to Azure Functions in the Azure Machine Learning documentation.

4. Deploy the model

After all of the configuration is prepared, you can deploy the model. The easiest way to do this is to call the deploy method of the Model class, like this:

```python
from azureml.core.model import Model

model = ws.models['classification_model']
service = Model.deploy(workspace=ws,
                       name = 'classifier-service',
                       models = [model],
                       inference_config = classifier_inference_config,
                       deployment_config = classifier_deploy_config,
                       deployment_target = production_cluster)
service.wait_for_deployment(show_output = True)
```

For ACI or local services, you can omit the deployment_target parameter (or set it to None).

For more information about deploying models with Azure Machine Learning, see Deploy machine learning models to Azure in the documentation.

Consume a real-time inferencing service

After deploying a real-time service, you can consume it from client applications to predict labels for new data cases.

Use the Azure Machine Learning SDK

For testing, you can use the Azure Machine Learning SDK to call a web service through the run method of a WebService object that references the deployed

service. Typically, you send data to the run method in JSON format with the following structure:

JSON

```
{
  "data":[
      [0.1,2.3,4.1,2.0], // 1st case
      [0.2,1.8,3.9,2.1],  // 2nd case,
      ...
  ]
}
```

The response from the run method is a JSON collection with a prediction for each case that was submitted in the data. The following code sample calls a service and displays the response:

Python

```
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Call the web service, passing the input data
response = service.run(input_data = json_data)

# Get the predictions
predictions = json.loads(response)

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i], predictions[i])
```

Use a REST endpoint

In production, most client applications will not include the Azure Machine Learning SDK, and will consume the service through its REST interface. You can determine the endpoint of a deployed service in Azure Machine Learning studio, or by retrieving the scoring_uri property of the Webservice object in the SDK, like this:

Python

```
endpoint = service.scoring_uri
print(endpoint)
```

With the endpoint known, you can use an HTTP POST request with JSON data to call the service. The following example shows how to do this using Python:

Python

```
import requests
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Set the content type in the request headers
request_headers = { 'Content-Type':'application/json' }

# Call the service
response = requests.post(url = endpoint,
                         data = json_data,
                         headers = request_headers)

# Get the predictions from the JSON response
predictions = json.loads(response.json())

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i]), predictions[i] )
```

Authentication

In production, you will likely want to restrict access to your services by applying authentication. There are two kinds of authentication you can use:

- Key: Requests are authenticated by specifying the key associated with the service.
- Token: Requests are authenticated by providing a JSON Web Token (JWT).

By default, authentication is disabled for ACI services, and set to key-based authentication for AKS services (for which primary and secondary keys are

automatically generated). You can optionally configure an AKS service to use token-based authentication (which is not supported for ACI services).

Assuming you have an authenticated session established with the workspace, you can retrieve the keys for a service by using the get_keys method of the WebService object associated with the service:

```
Python

primary_key, secondary_key = service.get_keys()
```

For token-based authentication, your client application needs to use service-principal authentication to verify its identity through Azure Active Directory (Azure AD) and call the get_token method of the service to retrieve a time-limited token.

To make an authenticated call to the service's REST endpoint, you must include the key or token in the request header like this:

Python

```python
import requests
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Set the content type in the request headers
request_headers = { "Content-Type":"application/json",
                    "Authorization":"Bearer " + key_or_token }

# Call the service
response = requests.post(url = endpoint,
                         data = json_data,
                         headers = request_headers)

# Get the predictions from the JSON response
predictions = json.loads(response.json())

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i]), predictions[i] )
```

Troubleshoot service deployment

Completed

100 XP

- 5 minutes

There are a lot of elements to a real-time service deployment, including the trained model, the runtime environment configuration, the scoring script, the container image, and the container host. Troubleshooting a failed deployment or an error when consuming a deployed service can be complex.

Check the service state

As an initial troubleshooting step, you can check the status of a service by examining its state:

```
Python

from azureml.core.webservice import AksWebservice

# Get the deployed service
service = AksWebservice(name='classifier-service', workspace=ws)

# Check its state
print(service.state)
```

Note

To view the state of a service, you must use the compute-specific service type (for example AksWebservice) and not a generic WebService object.

For an operational service, the state should be *Healthy*.

Review service logs

If a service is not healthy, or you are experiencing errors when using it, you can review its logs:

```
Python

print(service.get_logs())
```

The logs include detailed information about the provisioning of the service, and the requests it has processed. They can often provide an insight into the cause of unexpected errors.

Deploy to a local container

Deployment and runtime errors can be easier to diagnose by deploying the service as a container in a local Docker instance, like this:

```python
from azureml.core.webservice import LocalWebservice

deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, 'test-svc', [model], inference_config, deployment_config)
```

You can then test the locally deployed service using the SDK:

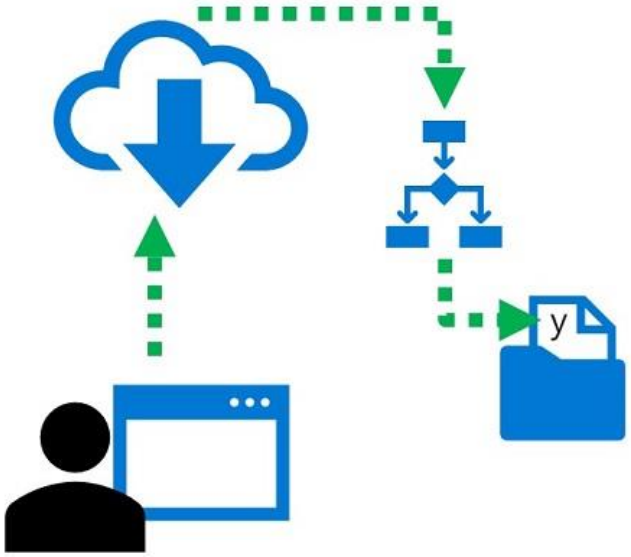
```python
print(service.run(input_data = json_data))
```

You can then troubleshoot runtime issues by making changes to the scoring file that is referenced in the inference configuration, and reloading the service without redeploying it (something you can only do with a local service):

```python
service.reload()
print(service.run(input_data = json_data))
```

Introduction

In many production scenarios, long-running tasks that operate on large volumes of data are performed as *batch* operations. In machine learning, *batch inferencing* is used to apply a predictive model to multiple cases asynchronously - usually writing the results to a file or database.



In Azure Machine Learning, you can implement batch inferencing solutions by creating a pipeline that includes a step to read the input data, load a registered model, predict labels, and write the results as its output.

Learning objectives

In this module, you will learn how to:

- Publish batch inference pipeline for a trained model.
- Use a batch inference pipeline to generate predictions.

Creating a batch inference pipeline

To create a batch inferencing pipeline, perform the following tasks:

1. Register a model

To use a trained model in a batch inferencing pipeline, you must register it in your Azure Machine Learning workspace.

To register a model from a local file, you can use the register method of the Model object as shown in the following example code:

Python Copy

```
from azureml.core import Model

classification_model = Model.register(workspace=your_workspace,
                                      model_name='classification_model',
                                      model_path='model.pkl', # local path
                                      description='A classification model')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown in the following example code:

Python Copy

```
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model')
```

2. Create a scoring script

Batch inferencing service requires a scoring script to load the model and use it to predict new values. It must include two functions:

- init(): Called when the pipeline is initialized.
- run(mini_batch): Called for each batch of data to be processed.

Typically, you use the init function to load the model from the model registry, and use the run function to generate predictions from each batch of data and return the results. The following example script shows this pattern:

Python

```
import os
import numpy as np
from azureml.core import Model
import joblib

def init():
    # Runs when the pipeline step is initialized
    global model

    # load the model
    model_path = Model.get_model_path('classification_model')
    model = joblib.load(model_path)

def run(mini_batch):
    # This runs for each batch
    resultList = []

    # process each file in the batch
    for f in mini_batch:
        # Read comma-delimited data into an array
        data = np.genfromtxt(f, delimiter=',')
        # Reshape into a 2-dimensional array for model input
        prediction = model.predict(data.reshape(1, -1))
        # Append prediction to results
        resultList.append("{}: {}".format(os.path.basename(f), prediction[0]))
    return resultList
```

3. Create a pipeline with a ParallelRunStep

Azure Machine Learning provides a type of pipeline step specifically for performing parallel batch inferencing. Using the ParallelRunStep class, you can read batches of files from a File dataset and write the processing output to a OutputFileDatasetConfig. Additionally, you can set the output_action setting for the step to "append_row", which will ensure that all instances of the step being run in parallel will collate their results to a single output file named *parallel_run_step.txt*:

```
from azureml.pipeline.steps import ParallelRunConfig, ParallelRunStep
from azureml.data import OutputFileDatasetConfig
from azureml.pipeline.core import Pipeline

# Get the batch dataset for input
batch_data_set = ws.datasets['batch-data']

# Set the output location
output_dir = OutputFileDatasetConfig(name='inferences')

# Define the parallel run step step configuration
parallel_run_config = ParallelRunConfig(
    source_directory='batch_scripts',
    entry_script="batch_scoring_script.py",
    mini_batch_size="5",
    error_threshold=10,
    output_action="append_row",
    environment=batch_env,
    compute_target=aml_cluster,
    node_count=4)

# Create the parallel run step
parallelrun_step = ParallelRunStep(
    name='batch-score',
    parallel_run_config=parallel_run_config,
    inputs=[batch_data_set.as_named_input('batch_data')],
    output=output_dir,
    arguments=[],
    allow_reuse=True
)
# Create the pipeline
pipeline = Pipeline(workspace=ws, steps=[parallelrun_step])
```

4. Run the pipeline and retrieve the step output

After your pipeline has been defined, you can run it and wait for it to complete. Then you can retrieve the parallel_run_step.txt file from the output of the step to view the results, as shown in the following code example:

Python

```python
from azureml.core import Experiment

# Run the pipeline as an experiment
pipeline_run = Experiment(ws, 'batch_prediction_pipeline').submit(pipeline)
pipeline_run.wait_for_completion(show_output=True)

# Get the outputs from the first (and only) step
prediction_run = next(pipeline_run.get_children())
prediction_output = prediction_run.get_output_data('inferences')
prediction_output.download(local_path='results')

# Find the parallel_run_step.txt file
for root, dirs, files in os.walk('results'):
    for file in files:
        if file.endswith('parallel_run_step.txt'):
            result_file = os.path.join(root,file)

# Load and display the results
df = pd.read_csv(result_file, delimiter=":", header=None)
df.columns = ["File", "Prediction"]
print(df)
```

Publishing a batch inference pipeline

You can publish a batch inferencing pipeline as a REST service, as shown in the following example code:

```
published_pipeline = pipeline_run.publish_pipeline(name='Batch_Prediction_Pipeline',
                                                   description='Batch pipeline',
                                                   version='1.0')
rest_endpoint = published_pipeline.endpoint
```

Once published, you can use the service endpoint to initiate a batch inferencing job, as shown in the following example code:

Python Copy

```
import requests

response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "Batch_Prediction"})
run_id = response.json()["Id"]
```

You can also schedule the published pipeline to have it run automatically, as shown in the following example code:

Python Copy

```
from azureml.pipeline.core import ScheduleRecurrence, Schedule

weekly = ScheduleRecurrence(frequency='Week', interval=1)
pipeline_schedule = Schedule.create(ws, name='Weekly Predictions',
                                    description='batch inferencing',
                                    pipeline_id=published_pipeline.id,
                                    experiment_name='Batch_Prediction',
                                    recurrence=weekly)
```

Exercise - Create a batch inference pipeline

Now it's your chance to create and run a batch inferencing pipeline.

In this exercise, you will:

- Create a batch inferencing pipeline.
- Publish the pipeline as a REST service.
- Run the pipeline through its REST endpoint.

Tune Hyperparameters-Introduction

In machine learning, models are trained to predict unknown labels for new data based on correlations between known labels and features found in the training data. Depending on the algorithm used, you may need to specify *hyperparameters* to configure how the model is trained. For example, the *logistic regression* algorithm uses a *regularization rate* hyperparameter to counteract overfitting; and deep learning techniques for convolutional neural networks (CNNs) use hyperparameters

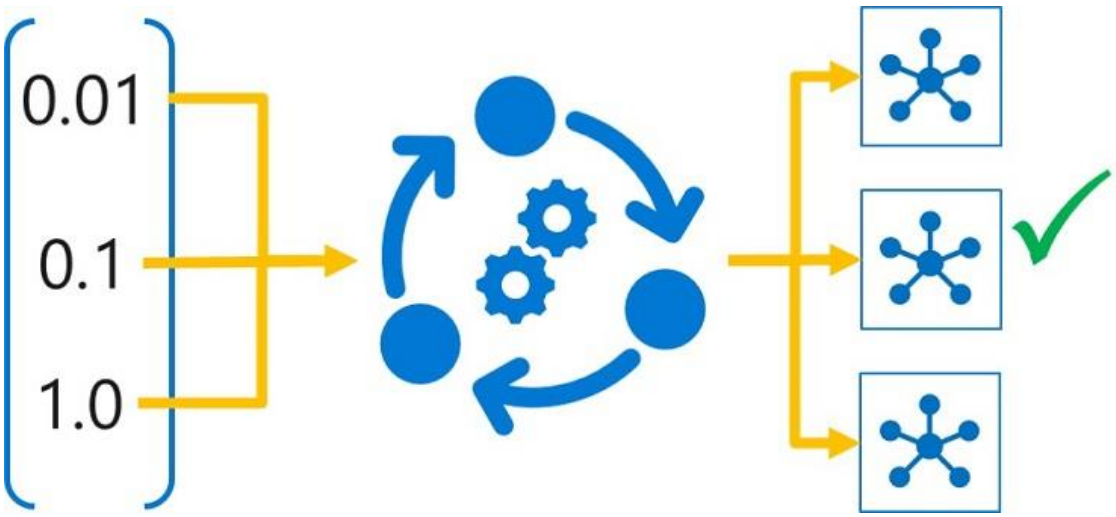
like *learning rate* to control how weights are adjusted during training, and *batch size* to determine how many data items are included in each training batch.

Note

Machine Learning is an academic field with its own particular terminology. Data scientists refer to the values determined from the training features as *parameters*, so a different term is required for values that are used to configure training behavior but which are *not* derived from the training data - hence the term *hyperparameter*.

The choice of hyperparameter values can significantly affect the resulting model, making it important to select the best possible values for your particular data and predictive performance goals.

Tuning hyperparameters



Hyperparameter tuning is accomplished by training the multiple models, using the same algorithm and training data but different hyperparameter values. The resulting model from each training run is then evaluated to determine the performance metric for which you want to optimize (for example, *accuracy*), and the best-performing model is selected.

In Azure Machine Learning, you achieve this through an experiment that consists of a *hyperdrive* run, which initiates a child run for each hyperparameter combination to be tested. Each child run uses a training script with parameterized hyperparameter values to train a model, and logs the target performance metric achieved by the trained model.

Learning objectives

In this module, you will learn how to:

- Define a hyperparameter search space.
- Configure hyperparameter sampling.
- Select an early-termination policy.
- Run a hyperparameter tuning experiment.

Defining a search space

The set of hyperparameter values tried during hyperparameter tuning is known as the *search space*. The definition of the range of possible values that can be chosen depends on the type of hyperparameter.

Discrete hyperparameters

Some hyperparameters require *discrete* values - in other words, you must select the value from a particular set of possibilities. You can define a search space for a discrete parameter using a choice from a list of explicit values, which you can define as a Python list (`choice([10,20,30])`), a range (`choice(range(1,10))`), or an arbitrary set of comma-separated values (`choice(30,50,100)`)

You can also select discrete values from any of the following discrete distributions:

- qnormal
- quniform
- qlognormal
- qloguniform

Continuous hyperparameters

Some hyperparameters are *continuous* - in other words you can use any value along a scale. To define a search space for these kinds of value, you can use any of the following distribution types:

- normal
- uniform
- lognormal
- loguniform

Defining a search space

To define a search space for hyperparameter tuning, create a dictionary with the appropriate parameter expression for each named hyperparameter. For example, the following search space indicates that the batch_size hyperparameter can have the value 16, 32, or 64, and the learning_rate hyperparameter can have any value from a normal distribution with a mean of 10 and a standard deviation of 3.

Python

```
from azureml.train.hyperdrive import choice, normal

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': normal(10, 3)
              }
```

Configuring sampling

The specific values used in a hyperparameter tuning run depend on the type of *sampling* used.

Grid sampling

Grid sampling can only be employed when all hyperparameters are discrete, and is used to try every possible combination of parameters in the search space.

For example, in the following code example, grid sampling is used to try every possible combination of discrete *batch_size* and *learning_rate* value:

Python

```
from azureml.train.hyperdrive import GridParameterSampling, choice

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': choice(0.01, 0.1, 1.0)
              }

param_sampling = GridParameterSampling(param_space)
```

Random sampling

Random sampling is used to randomly select a value for each hyperparameter, which can be a mix of discrete and continuous values as shown in the following code example:

```python
from azureml.train.hyperdrive import RandomParameterSampling, choice, normal

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': normal(10, 3)
              }

param_sampling = RandomParameterSampling(param_space)
```

Bayesian sampling

Bayesian sampling chooses hyperparameter values based on the Bayesian optimization algorithm, which tries to select parameter combinations that will result in improved performance from the previous selection. The following code example shows how to configure Bayesian sampling:

```python
from azureml.train.hyperdrive import BayesianParameterSampling, choice, uniform

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': uniform(0.05, 0.1)
              }

param_sampling = BayesianParameterSampling(param_space)
```

You can only use Bayesian sampling with choice, uniform, and quniform parameter expressions, and you can't combine it with an early-termination policy.

Configuring early termination

With a sufficiently large hyperparameter search space, it could take many iterations (child runs) to try every possible combination. Typically, you set a maximum number

of iterations, but this could still result in a large number of runs that don't result in a better model than a combination that has already been tried.

To help prevent wasting time, you can set an early termination policy that abandons runs that are unlikely to produce a better result than previously completed runs. The policy is evaluated at an *evaluation_interval* you specify, based on each time the target performance metric is logged. You can also set a *delay_evaluation* parameter to avoid evaluating the policy until a minimum number of iterations have been completed.
Note

Early termination is particularly useful for deep learning scenarios where a deep neural network (DNN) is trained iteratively over a number of *epochs*. The training script can report the target metric after each epoch, and if the run is significantly underperforming previous runs after the same number of intervals, it can be abandoned.

Bandit policy

You can use a bandit policy to stop a run if the target performance metric underperforms the best run so far by a specified margin.

```
Python
```

```python
from azureml.train.hyperdrive import BanditPolicy

early_termination_policy = BanditPolicy(slack_amount = 0.2,
                                        evaluation_interval=1,
                                        delay_evaluation=5)
```

This example applies the policy for every iteration after the first five, and abandons runs where the reported target metric is 0.2 or more worse than the best performing run after the same number of intervals.

You can also apply a bandit policy using a slack *factor*, which compares the performance metric as a ratio rather than an absolute value.

Median stopping policy

A median stopping policy abandons runs where the target performance metric is worse than the median of the running averages for all runs.

```python
from azureml.train.hyperdrive import MedianStoppingPolicy

early_termination_policy = MedianStoppingPolicy(evaluation_interval=1,
                                                delay_evaluation=5)
```

Truncation selection policy

A truncation selection policy cancels the lowest performing *X*% of runs at each evaluation interval based on the *truncation_percentage* value you specify for *X*.

```python
from azureml.train.hyperdrive import TruncationSelectionPolicy

early_termination_policy = TruncationSelectionPolicy(truncation_percentage=10,
                                                     evaluation_interval=1,
                                                     delay_evaluation=5)
```

Running a hyperparameter tuning experiment

In Azure Machine Learning, you can tune hyperparameters by running a *hyperdrive* experiment.

Creating a training script for hyperparameter tuning

To run a hyperdrive experiment, you need to create a training script just the way you would do for any other training experiment, except that your script *must*:

- Include an argument for each hyperparameter you want to vary.
- Log the target performance metric. This enables the hyperdrive run to evaluate the performance of the child runs it initiates, and identify the one that produces the best performing model.

For example, the following example script trains a logistic regression model using a --regularization argument to set the *regularization rate* hyperparameter, and logs the *accuracy* metric with the name Accuracy:

```
import argparse
import joblib
from azureml.core import Run
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Get regularization hyperparameter
parser = argparse.ArgumentParser()
parser.add_argument('--regularization', type=float, dest='reg_rate', default=0.01)
args = parser.parse_args()
reg = args.reg_rate

# Get the experiment run context
run = Run.get_context()

# load the training dataset
data = run.input_datasets['training_data'].to_pandas_dataframe()

# Separate features and labels, and split for training/validatiom
X = data[['feature1','feature2','feature3','feature4']].values
y = data['label'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

# Train a logistic regression model with the reg hyperparameter
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_train)

# calculate and log accuracy
y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
run.log('Accuracy', np.float(acc))

# Save the trained model
os.makedirs('outputs', exist_ok=True)
joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()
```

Zoc

Note

Note that in the Scikit-Learn LogisticRegression class, C is the *inverse* of the regularization rate; hence C=1/reg.

Configuring and running a hyperdrive experiment

To prepare the hyperdrive experiment, you must use a HyperDriveConfig object to configure the experiment run, as shown in the following example code:

Python

```
from azureml.core import Experiment
from azureml.train.hyperdrive import HyperDriveConfig, PrimaryMetricGoal

# Assumes ws, script_config and param_sampling are already defined

hyperdrive = HyperDriveConfig(run_config=script_config,
                              hyperparameter_sampling=param_sampling,
                              policy=None,
                              primary_metric_name='Accuracy',
                              primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
                              max_total_runs=6,
                              max_concurrent_runs=4)

experiment = Experiment(workspace = ws, name = 'hyperdrive_training')
hyperdrive_run = experiment.submit(config=hyperdrive)
```

Monitoring and reviewing hyperdrive runs

You can monitor hyperdrive experiments in Azure Machine Learning studio, or by using the Jupyter Notebooks RunDetails widget.

The experiment will initiate a child run for each hyperparameter combination to be tried, and you can retrieve the logged metrics these runs using the following code:

Python

```
for child_run in run.get_children():
    print(child_run.id, child_run.get_metrics())
```

You can also list all runs in descending order of performance like this:

Python

```
for child_run in hyperdrive_run.get_children_sorted_by_primary_metric():
    print(child_run)
```

To retrieve the best performing run, you can use the following code:

Python

```
best_run = hyperdrive_run.get_best_run_by_primary_metric()
```

Automate machine learning model selection with Azure Machine Learning

Introduction

Automated Machine Learning enables you to try multiple algorithms and preprocessing transformations with your data. This, combined with scalable cloud-based compute makes it possible to find the best performing model for your data without the huge amount of time-consuming manual trial and error that would otherwise be required.

Azure Machine Learning includes support for automated machine learning through a visual interface in Azure Machine Learning studio. You can use the Azure Machine Learning SDK to run automated machine learning experiments.

Learning objectives

In this module, you will learn how to:

- Use Azure Machine Learning's automated machine learning capabilities to determine the best performing algorithm for your data.
- Use automated machine learning to preprocess data for training.
- Run an automated machine learning experiment.

Automated machine learning tasks and algorithms

You can use automated machine learning in Azure Machine Learning to train models for the following types of machine learning tasks:

- Classification
- Regression
- Time Series Forecasting

Task-specific algorithms

Azure Machine Learning includes support for numerous commonly used algorithms for these tasks, including:

Classification algorithms

- Logistic Regression
- Light Gradient Boosting Machine (GBM)
- Decision Tree
- Random Forest
- Naive Bayes
- Linear Support Vector Machine (SVM)
- XGBoost
- Deep Neural Network (DNN) Classifier
- Others...

Regression algorithms

- Linear Regression
- Light Gradient Boosting Machine (GBM)
- Decision Tree
- Random Forest
- Elastic Net
- LARS Lasso
- XGBoost
- Others...

Forecasting algorithms

- Linear Regression
- Light Gradient Boosting Machine (GBM)
- Decision Tree
- Random Forest
- Elastic Net
- LARS Lasso
- XGBoost
- Others...

More Information: For a full list of supported algorithms, see How to define a machine learning task.

Restrict algorithm selection

By default, automated machine learning will randomly select from the full range of algorithms for the specified task. You can choose to block individual algorithms from being selected; which can be useful if you know that your data is not suited to a particular type of algorithm, or you have to comply with a policy that restricts the type of machine learning algorithms you can use in your organization.

Preprocessing and featurization

As well as trying a selection of algorithms, automated machine learning can apply preprocessing transformations to your data; improving the performance of the model.

Scaling and normalization

Automated machine learning applies scaling and normalization to numeric data automatically, helping prevent any large-scale features from dominating training. During an automated machine learning experiment, multiple scaling or normalization techniques will be applied.

Optional featurization

You can choose to have automated machine learning apply preprocessing transformations, such as:

- Missing value imputation to eliminate nulls in the training dataset.
- Categorical encoding to convert categorical features to numeric indicators.
- Dropping high-cardinality features, such as record IDs.
- Feature engineering (for example, deriving individual date parts from DateTime features)
- Others...
  More Information: For more information about the preprocessing support in automated machine learning, see What is automated machine learning.

Running automated machine learning experiments

To run an automated machine learning experiment, you can either use the user interface in Azure Machine Learning studio, or submit an experiment using the SDK.

Configure an automated machine learning experiment

The user interface provides an intuitive way to select options for your automated machine learning experiment. When using the SDK, you have greater flexibility, and you can set experiment options using the AutoMLConfig class, as shown in the following example.

Python

```
from azureml.train.automl import AutoMLConfig

automl_run_config = RunConfiguration(framework='python')
automl_config = AutoMLConfig(name='Automated ML Experiment',
                             task='classification',
                             primary_metric = 'AUC_weighted',
                             compute_target=aml_compute,
                             training_data = train_dataset,
                             validation_data = test_dataset,
                             label_column_name='Label',
                             featurization='auto',
                             iterations=12,
                             max_concurrent_iterations=4)
```

Specify data for training

Automated machine learning is designed to enable you to simply bring your data, and have Azure Machine Learning figure out how best to train a model from it.

When using the Automated Machine Learning user interface in Azure Machine Learning studio, you can create or select an Azure Machine Learning *dataset* to be used as the input for your automated machine learning experiment.

When using the SDK to run an automated machine learning experiment, you can submit the data in the following ways:

- Specify a dataset or dataframe of *training data* that includes features and the label to be predicted.
- Optionally, specify a second *validation data* dataset or dataframe that will be used to validate the trained model. if this is not provided, Azure Machine Learning will apply cross-validation using the training data.

Alternatively:

- Specify a dataset, dataframe, or numpy array of *X* values containing the training features, with a corresponding *y* array of label values.
- Optionally, specify *X_valid* and *y_valid* datasets, dataframes, or numpy arrays of *X_valid* values to be used for validation.

Specify the primary metric

One of the most important settings you must specify is the primary_metric. This is the target performance metric for which the optimal model will be determined. Azure Machine Learning supports a set of named metrics for each type of task. To retrieve the list of metrics available for a particular task type, you can use the get_primary_metrics function as shown here:

```
Python

from azureml.train.automl.utilities import get_primary_metrics

get_primary_metrics('classification')
```

Submit an automated machine learning experiment

You can submit an automated machine learning experiment like any other SDK-based experiment.

You can monitor automated machine learning experiment runs in Azure Machine Learning studio, or in the Jupyter Notebooks RunDetails widget.

```python
from azureml.core.experiment import Experiment

automl_experiment = Experiment(ws, 'automl_experiment')
automl_run = automl_experiment.submit(automl_config)
```

Retrieve the best run and its model

You can easily identify the best run in Azure Machine Learning studio, and download or deploy the model it generated. To accomplish this programmatically with the SDK, you can use code like the following example:

```python
best_run, fitted_model = automl_run.get_output()
best_run_metrics = best_run.get_metrics()
for metric_name in best_run_metrics:
    metric = best_run_metrics[metric_name]
    print(metric_name, metric)
```

Explore preprocessing steps

Automated machine learning uses scikit-learn pipelines to encapsulate preprocessing steps with the model. You can view the steps in the fitted model you obtained from the best run using the code above like this:

```python
for step_ in fitted_model.named_steps:
    print(step_)
```

Introduction-Differential privacy

Data science projects, including machine learning projects, involve analysis of data; and often that data includes sensitive personal details that should be kept private. In practice, most reports that are published from the data include aggregations of the data, which you may think would provide some privacy – after all, the aggregated results do not reveal the individual data values.

However, consider a case where multiple analyses of the data result in reported aggregations that when combined, could be used to work out information about individuals in the source dataset. Suppose 10 participants share data about their location and salary, from which two reports are produced:

- An aggregated salary report that tells us the average salaries in New York, San Francisco, and Seattle
- A worker location report that tells us that 10% of the study participants (in other words, a single person) is based in Seattle.

From these two reports, we can easily determine the specific salary of the Seattle-based participant. Anyone reviewing both studies who happens to know a person from Seattle that participated, now knows that person's salary.

In this module, you'll explore differential privacy, a technique that can help protect an individual's data against this kind of exposure.

In this module, you will learn how to:

Articulate the problem of data privacy

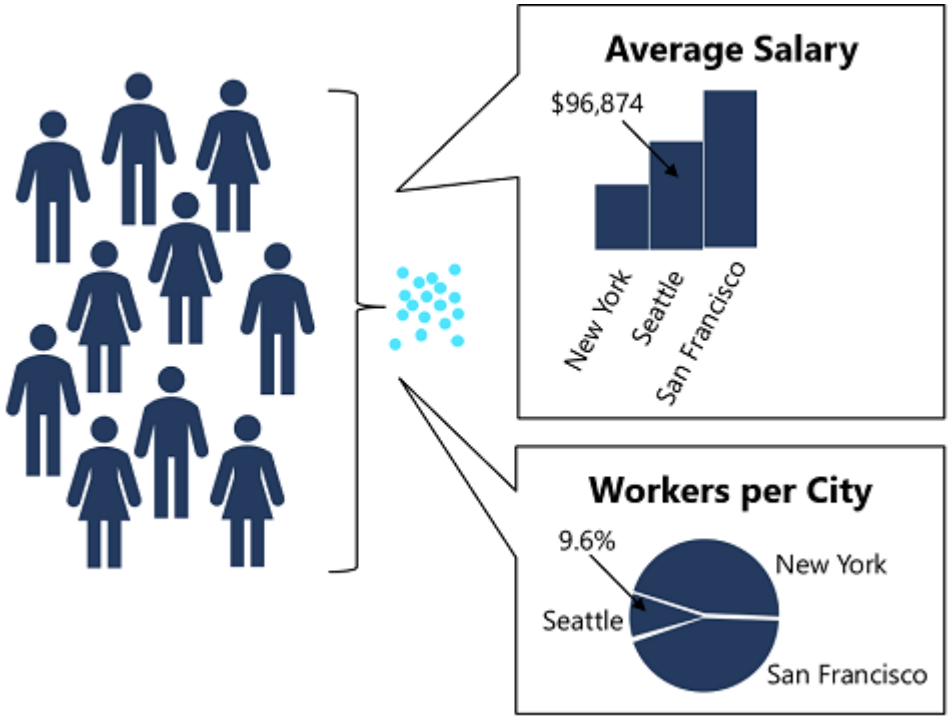
Describe how differential privacy works

Configure parameters for differential privacy

Perform differentially private data analysis

Understand differential privacy

Differential privacy seeks to protect individual data values by adding statistical "noise" to the analysis process. The math involved in adding the noise is complex, but the principle is fairly intuitive – the noise ensures that data aggregations stay statistically consistent with the actual data values allowing for some random variation, but make it impossible to work out the individual values from the aggregated data. In addition, the noise is different for each analysis, so the results are non-deterministic – in other words, two analyses that perform the same aggregation may produce slightly different results.



Configure data privacy parameters

One way that an individual can protect their personal data is simply to not participate in a study – this is known as their "opt-out" option. However, there are a few considerations for this as a solution:

- Even if you decide to opt out a study may still produce results that affect you. For example, you may choose to opt-out of a study that compares the heart disease diagnoses across a group of people on the basis that doing so may reveal a heart disease diagnosis that causes your health insurance premiums to rise. If the study finds a correlation between people who drink coffee and higher risk of heart disease, and your insurance company knows that you are a coffee drinker, your rate may rise even though you didn’t personally participate in the study.

- The benefits of participation in the study may outweigh any negative impact. For example, if you're paid $100 to participate in a study that results in your health insurance rate rising by $10 per year, it will be more than 10 years before you make a net loss. This may be a worthwhile tradeoff to you (particularly if your rate may rise as a result of the study even if you don't participate!)
- The only way for the opt-out option to work for every individual, is for every individual not to take part – which makes the whole study pointless!

The amount of variation caused by adding noise is configurable through a parameter called epsilon. This value governs the amount of additional risk that your personal data can be identified through rejecting the opt-out option and participating in a study. The key thing is that it applies this privacy principle for everyone participating in the study. A low epsilon value provides the most privacy, at the expense of less accuracy when aggregating the data. A higher epsilon value results in aggregations that are more true to the actual data distribution, but in which the individual contribution of a single individual to the aggregated value is less obscured by noise.

Explain machine learning models with Azure Machine Learning

Introduction

As machine learning becomes increasingly integral to decisions that affect health, safety, economic wellbeing, and other aspects of people's lives, it's important to be able to understand how models make predictions; and to be able to explain the rationale for machine learning based decisions.

Explaining models is difficult because of the range of machine learning algorithm types and the nature of how machine learning works, but model interpretability has become a key element of helping to make model predictions explainable.

Learning objectives

In this module, you will learn how to:

- Interpret *global* and *local* feature importance.
- Use an explainer to interpret a model.
- Create model explanations in a training experiment.
- Visualize model explanations.

Feature importance

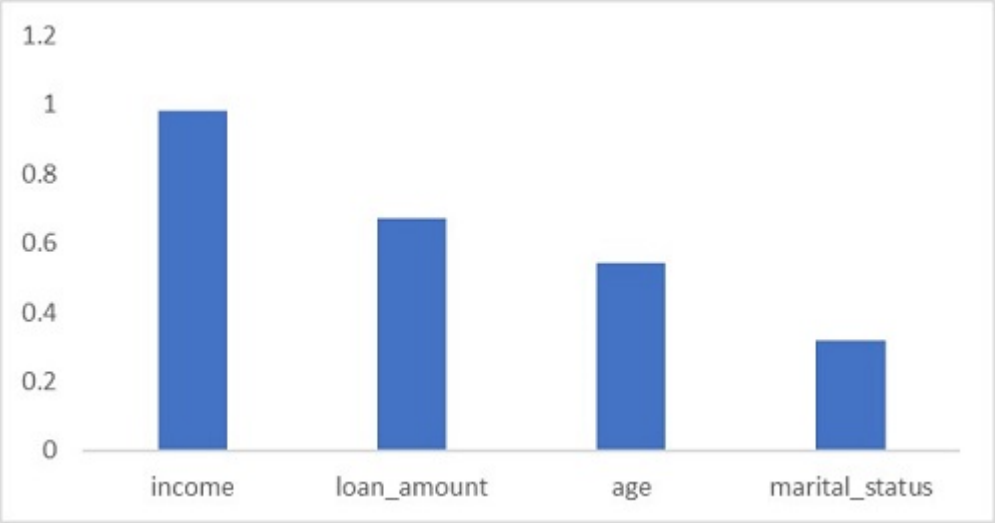
Model explainers use statistical techniques to calculate *feature importance*. This enables you to quantify the relative influence each feature in the training dataset has on label prediction. Explainers work by evaluating a test data set of feature cases and the labels the model predicts for them.

Global feature importance

Global feature importance quantifies the relative importance of each feature in the test dataset as a whole. It provides a general comparison of the extent to which each feature in the dataset influences prediction.

For example, a binary classification model to predict loan default risk might be trained from features such as loan amount, income, marital status, and age to predict a label of 1 for loans that are likely to be repaid, and 0 for loans that have a significant risk of default (and therefore shouldn't be approved). An explainer might then use a sufficiently representative test dataset to produce the following global feature importance values:

- income: 0.98
- loan amount: 0.67
- age: 0.54
- marital status 0.32



It's clear from these values, that in respect to the overall predictions generated by the model for the test dataset, income is the most important feature for predicting whether or not a borrower will default on a loan, followed by the loan amount, then age, and finally marital status.

Local feature importance

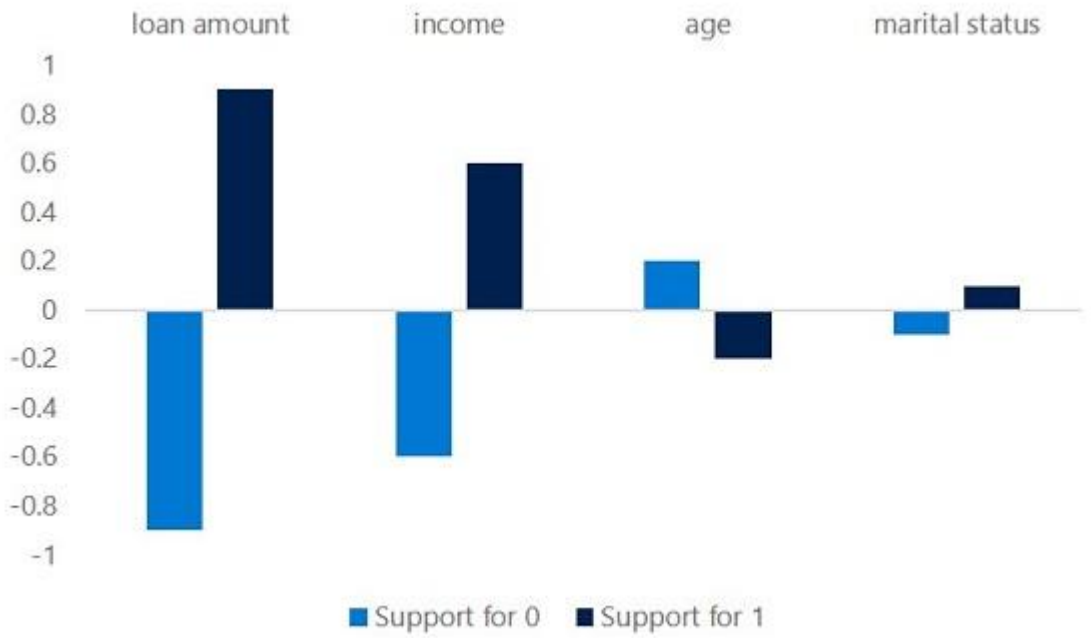
Local feature importance measures the influence of each feature value for a specific individual prediction.

For example, suppose Sam applies for a loan, which the machine learning model approves (by predicting that Sam won't default on the loan repayment). You could use an explainer to calculate the local feature importance for Sam's application to determine which factors influenced the prediction. You might get a result like this:

Feature	Support for 0	Support for 1
loan amount	-0.9	0.9
income	-0.6	0.6

age	0.2	-0.2
marital status	-0.1	0.1

Because this is a *classification* model, each feature gets a local importance value for each possible class, indicating the amount of support for that class based on the feature value. Since this is a *binary* classification model, there are only two possible classes (0 and 1). Each feature's support for one class results in correlatively negative level of support for the other.



In Sam's case, the overall support for class 0 is -1.4, and the support for class 1 is correspondingly 1.4; so support for class 1 is higher than for class 0, and the loan is approved. The most important feature for a prediction of class 1 is loan amount, followed by income - these are the opposite order from their global feature importance values (which indicate that income is the most important factor for the data sample as a whole). There could be multiple reasons why local importance for an individual prediction varies from global importance for the overall dataset; for example, Sam might have a lower income than average, but the loan amount in this case might be unusually small.

For a multi-class classification model, a local importance values for each possible class is calculated for every feature, with the total across all classes always being 0. For example, a model might predict the species of a penguin based on features like its bill length, bill width, flipper length, and weight. Suppose there are three species of penguin, so the model predicts one of three class labels (0, 1, or 2). For an individual prediction, the flipper length feature might have local importance values of 0.5 for class 0, 0.3 for class 1, and -0.8 for class 2 - indicating that the flipper length moderately supports a prediction of class 0, slightly supports a prediction of class 1, and strongly supports a prediction that this particular penguin is *not* class 2.

For a regression model, there are no classes so the local importance values simply indicate the level of influence each feature has on the predicted scalar label.

Using explainers

You can use the Azure Machine Learning SDK to create explainers for models, even if they were not trained using an Azure Machine Learning experiment.

Creating an explainer

To interpret a local model, you must install the azureml-interpret package and use it to create an explainer. There are multiple types of explainer, including:

- MimicExplainer - An explainer that creates a *global surrogate model* that approximates your trained model and can be used to generate explanations. This explainable model must have the same kind of architecture as your trained model (for example, linear or tree-based).
- TabularExplainer - An explainer that acts as a wrapper around various SHAP explainer algorithms, automatically choosing the one that is most appropriate for your model architecture.
- PFIExplainer - a *Permutation Feature Importance* explainer that analyzes feature importance by shuffling feature values and measuring the impact on prediction performance.

The following code example shows how to create an instance of each of these explainer types for a hypothetical model named loan_model:

Python

```python
# MimicExplainer
from interpret.ext.blackbox import MimicExplainer
from interpret.ext.glassbox import DecisionTreeExplainableModel

mim_explainer = MimicExplainer(model=loan_model,
                               initialization_examples=X_test,
                               explainable_model = DecisionTreeExplainableModel,
                               features=['loan_amount','income','age','marital_status'],
                               classes=['reject', 'approve'])


# TabularExplainer
from interpret.ext.blackbox import TabularExplainer

tab_explainer = TabularExplainer(model=loan_model,
                               initialization_examples=X_test,
                               features=['loan_amount','income','age','marital_status'],
                               classes=['reject', 'approve'])


# PFIExplainer
from interpret.ext.blackbox import PFIExplainer

pfi_explainer = PFIExplainer(model = loan_model,
                               features=['loan_amount','income','age','marital_status'],
                               classes=['reject', 'approve'])
```

Explaining global feature importance

To retrieve global importance values for the features in your model, you call the explain_global() method of your explainer to get a global explanation, and then use the get_feature_importance_dict() method to get a dictionary of the feature importance values. The following code example shows how to retrieve global feature importance

Python

```
# MimicExplainer
global_mim_explanation = mim_explainer.explain_global(X_train)
global_mim_feature_importance = global_mim_explanation.get_feature_importance_dict()


# TabularExplainer
global_tab_explanation = tab_explainer.explain_global(X_train)
global_tab_feature_importance = global_tab_explanation.get_feature_importance_dict()


# PFIExplainer
global_pfi_explanation = pfi_explainer.explain_global(X_train, y_train)
global_pfi_feature_importance = global_pfi_explanation.get_feature_importance_dict()
```

Note

The code is the same for MimicExplainer and TabularExplainer. The PFIExplainer requires the actual labels that correspond to the test features.

Explaining local feature importance

To retrieve local feature importance from a MimicExplainer or a TabularExplainer, you must call the explain_local() method of your explainer, specifying the subset of cases you want to explain. Then you can use the get_ranked_local_names() and get_ranked_local_values() methods to retrieve dictionaries of the feature names and importance values, ranked by importance. The following code example shows how to retrieve local feature importance:

```
# MimicExplainer
local_mim_explanation = mim_explainer.explain_local(X_test[0:5])
local_mim_features = local_mim_explanation.get_ranked_local_names()
local_mim_importance = local_mim_explanation.get_ranked_local_values()


# TabularExplainer
local_tab_explanation = tab_explainer.explain_local(X_test[0:5])
local_tab_features = local_tab_explanation.get_ranked_local_names()
local_tab_importance = local_tab_explanation.get_ranked_local_values()
```

Creating explanations

When you use an estimator or a script to train a model in an Azure Machine Learning experiment, you can create an explainer and upload the explanation it generates to the run for later analysis.

Creating an explanation in the experiment script

To create an explanation in the experiment script, you'll need to ensure that the azureml-interpret and azureml-contrib-interpret packages are installed in the run environment. Then you can use these to create an explanation from your trained model and upload it to the run outputs. The following code example shows how code to generate and upload a model explanation can be incorporated into an experiment script.

```
# Import Azure ML run library
from azureml.core.run import Run
from azureml.contrib.interpret.explanation.explanation_client import ExplanationClient
from interpret.ext.blackbox import TabularExplainer
# other imports as required

# Get the experiment run context
run = Run.get_context()

# code to train model goes here

# Get explanation
explainer = TabularExplainer(model, X_train, features=features, classes=labels)
explanation = explainer.explain_global(X_test)

# Get an Explanation Client and upload the explanation
explain_client = ExplanationClient.from_run(run)
explain_client.upload_model_explanation(explanation, comment='Tabular Explanation')

# Complete the run
run.complete()
```

Viewing the explanation

You can view the explanation you created for your model in the Explanations tab for the run in Azure Machine learning studio.

You can also use the ExplanationClient object to download the explanation in Python.

```python
from azureml.contrib.interpret.explanation.explanation_client import ExplanationClient

client = ExplanationClient.from_run_id(workspace=ws,
                                       experiment_name=experiment.experiment_name,
                                       run_id=run.id)
explanation = client.download_model_explanation()
feature_importances = explanation.get_feature_importance_dict()
```

Visualizing explanations

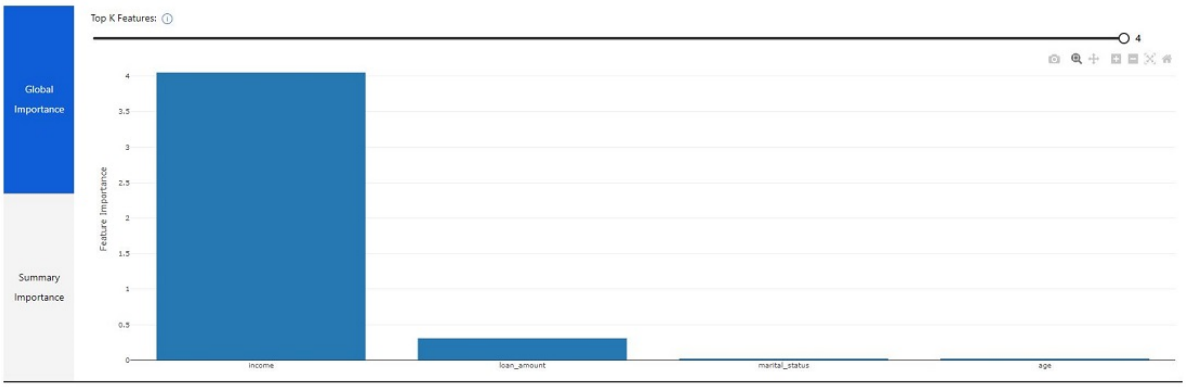
Model explanations in Azure Machine Learning studio include multiple visualizations that you can use to explore feature importance.

Note

Visualizations are only available for experiment runs that were configured to generate and upload explanations. When using automated machine learning, only the run producing the best model has explanations generated by default.

Visualizing global feature importance

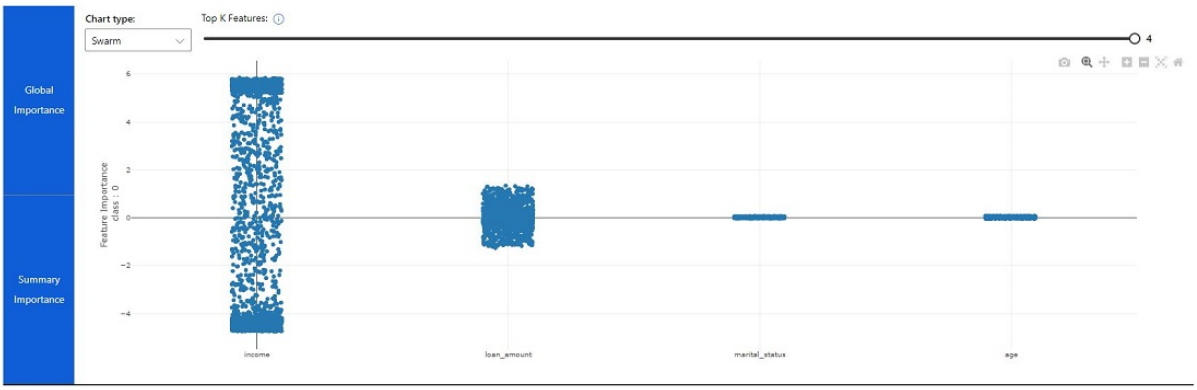
The first visualization on the Explanations tab for a run shows global feature importance.



You can use the slider to show only the top *N* features.

Visualizing summary importance

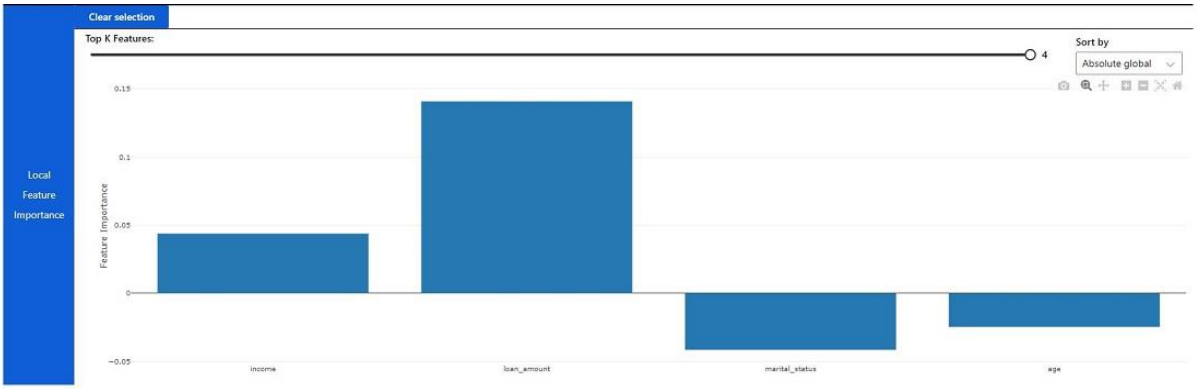
Switching to the Summary Importance visualization shows the distribution of individual importance values for each feature across the test dataset.



You can view the features as a *swarm* plot (shown above), a *box* plot, or a *violin* plot.

Visualizing local feature importance

Selecting an individual data point shows the local feature importance for the case to which the data point belongs.



Introduction- Mitigate unfairness in Models

Machine learning models are increasingly used to inform decisions that affect people's lives. For example, a prediction made by a machine learning model might influence:

- Approval for a loan, insurance, or other financial services.
- Acceptance into a school or college course.
- Eligibility for a medical trial or experimental treatment.
- Inclusion in a marketing promotion.
- Selection for employment or promotion.

With such critical decisions in the balance, confidence that the machine learning models we rely on predict, and don't discriminate for or against subsets of the population based on ethnicity, gender, age, or other factors.

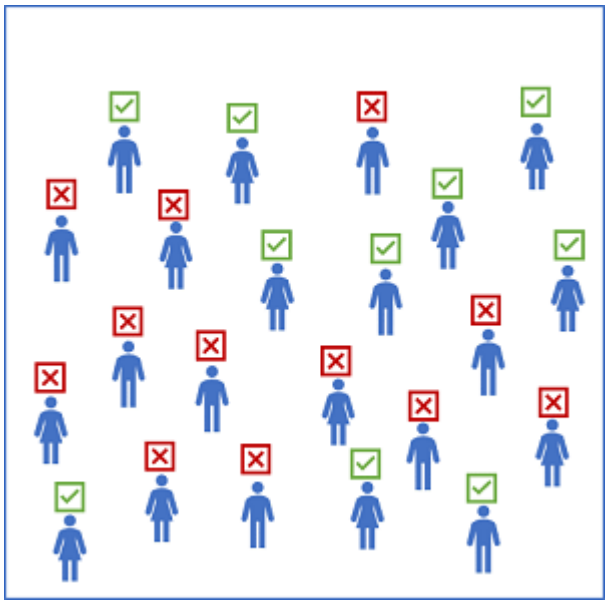
Consider model fairness

When we consider the concept of *fairness* concerning predictions made by machine learning models, it helps to be clear about what we mean by "fair".

For example, suppose a classification model is used to predict the probability of successful loan repayment and therefore influences whether or not the loan is approved. The model will likely be trained using features that reflect the characteristics of the applicant, such as:

- Age
- Employment status
- Income
- Savings
- Current debt

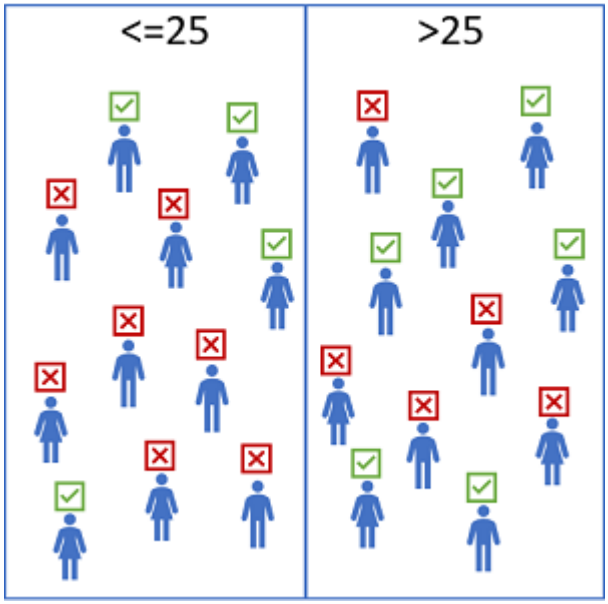
These features are used to train a binary classification model that predicts whether an applicant will repay a loan.



Suppose the model predicts that around 45% of applicants will successfully repay their loans. However, on reviewing loan approval records, you begin to suspect that fewer loans are approved for applicants aged 25 or younger than for applicants who are over 25. How can you be sure the model is *fair* to applicants in both age groups?

Measuring disparity in predictions

One way to start evaluating the fairness of a model is to compare *predictions* for each group within a *sensitive feature*. For the loan approval model, *Age* is a sensitive feature that we care about, so we could split the data into subsets for each age group and compare the *selection rate* (the proportion of positive predictions) for each group.



Let's say we find that the model predicts that 36% of applicants aged 25 or younger will repay a loan, but it predicts successful repayments for 54% of applicants aged over 25. There's a disparity in predictions of 18%.

At first glance, this comparison seems to confirm that there's bias in the model that discriminates against younger applicants. However, when you consider the population as a whole, it may be that younger people generally earn less than people more established in their careers, have lower levels of savings and assets, and have a higher rate of defaulting on loans.

The important point to consider here is that just because we want to ensure fairness regarding age, it doesn't necessarily follow that age is not a factor in loan repayment probability. It's possible that in general, younger people are less likely to repay a loan than older people. To get the full picture, we need to look a little deeper into the predictive performance of the model for each subset of the population.

Measuring disparity in prediction performance

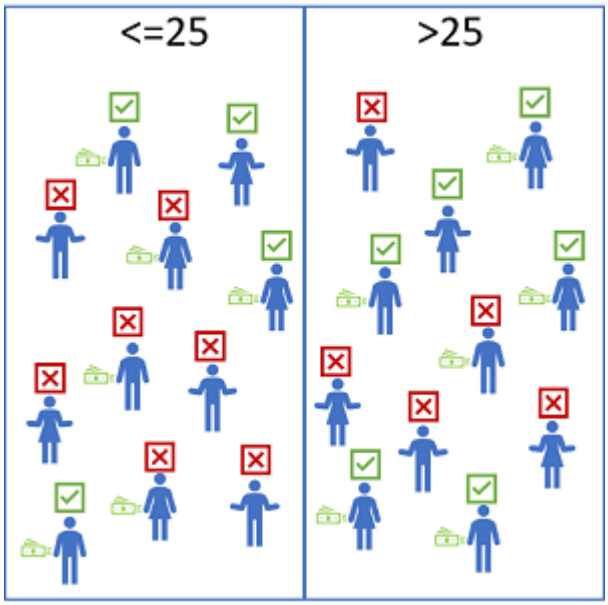
When you train a machine learning model using a supervised technique, like regression or classification, you use metrics achieved against hold-out validation

data to evaluate the overall predictive performance of the model. For example, you might evaluate a classification model based on *accuracy*, *precision*, or *recall*.

To evaluate the fairness of a model, you can apply the same predictive performance metric to subsets of the data, based on the sensitive features on which your population is grouped, and measure the disparity in those metrics across the subgroups.

For example, suppose the loan approval model exhibits an overall *recall* metric of 0.67 - in other words, it correctly identifies 67% of cases where the applicant repaid the loan. The question is whether or not the model provides a similar rate of correct predictions for different age groups.

To find out, we group the data based on the sensitive feature (*Age*) and measure the predictive performance metric (*recall*) for those groups. Then we can compare the metric scores to determine the disparity between them.



Let's say that we find that the recall for validation cases where the applicant is 25 or younger is 0.50, and recall for cases where the applicant is over 25 is 0.83. In other words, the model correctly identified 50% of the people in the 25 or younger age group who successfully repaid a loan (and therefore misclassified 50% of them as loan defaulters), but found 83% of loan repayers in the older age group (misclassifying only 17% of them). The disparity in prediction performance between the groups is 33%, with the model predicting significantly more false negatives for the younger age group.

Potential causes of disparity

When you find a disparity between prediction rates or prediction performance metrics across sensitive feature groups, it's worth considering potential causes. These might include:

- Data imbalance. Some groups may be overrepresented in the training data, or the data may be skewed so that cases within a specific group aren't representative of the overall population.
- Indirect correlation. The sensitive feature itself may not be predictive of the label, but there may be a hidden correlation between the sensitive feature and some other feature that influences the prediction. For example, there's likely a correlation between age and credit history, and there's likely a correlation between credit history and loan defaults. If the credit history feature is not included in the training data, the training algorithm may assign a predictive weight to age without accounting for credit history, which might make a difference to loan repayment probability.
- Societal biases. Subconscious biases in the data collection, preparation, or modeling process may have influenced feature selection or other aspects of model design.

Mitigating bias

Optimizing for fairness in a machine learning model is a *sociotechnical* challenge. In other words, it's not always something you can achieve purely by applying technical corrections to a training algorithm. However, there are some strategies you can adopt to mitigate bias, including:

- Balance training and validation data. You can apply over-sampling or under-sampling techniques to balance data and use stratified splitting algorithms to maintain representative proportions for training and validation.
- Perform extensive feature selection and engineering analysis. Make sure you fully explore the interconnected correlations in your data to try to differentiate features that are directly predictive from features that encapsulate more complex, nuanced relationships. You can use the model interpretability support in Azure Machine Learning to understand how individual features influence predictions.
- Evaluate models for disparity based on significant features. You can't easily address the bias in a model if you can't quantify it.
- Trade-off overall predictive performance for the lower disparity in predictive performance between sensitive feature groups. A model that is 99.5% accurate with comparable performance across all groups is

often more desirable than a model that is 99.9% accurate but discriminates against a particular subset of cases.

The rest of this module explores the Fairlearn package - a Python package that you can use to evaluate and mitigate unfairness in machine learning models.

Analyze model fairness with Fairlearn

Fairlearn is a Python package that you can use to analyze models and evaluate disparity between predictions and prediction performance for one or more sensitive features.

It works by calculating group metrics for the sensitive features you specify. The metrics themselves are based on standard scikit-learn model evaluation metrics, such as *accuracy*, *precision*, or *recall* for classification models.

The Fairlearn API is extensive, offering multiple ways to explore disparity in metrics across sensitive feature groupings. For a binary classification model, you might start by comparing the selection rate (the number of positive predictions for each group) by using the selection_rate function. This function returns the overall selection rate for the test dataset. You can also use standard sklearn.metrics functions (such as accuracy_score, precision_score, or recall_score) to get an overall view of how the model performs.

Then, you can define one or more *sensitive features* in your dataset with which you want to group subsets of the population and compare selection rate and predictive performance. Fairlearn includes a MetricFrame function that enables you to create a dataframe of multiple metrics by the group.

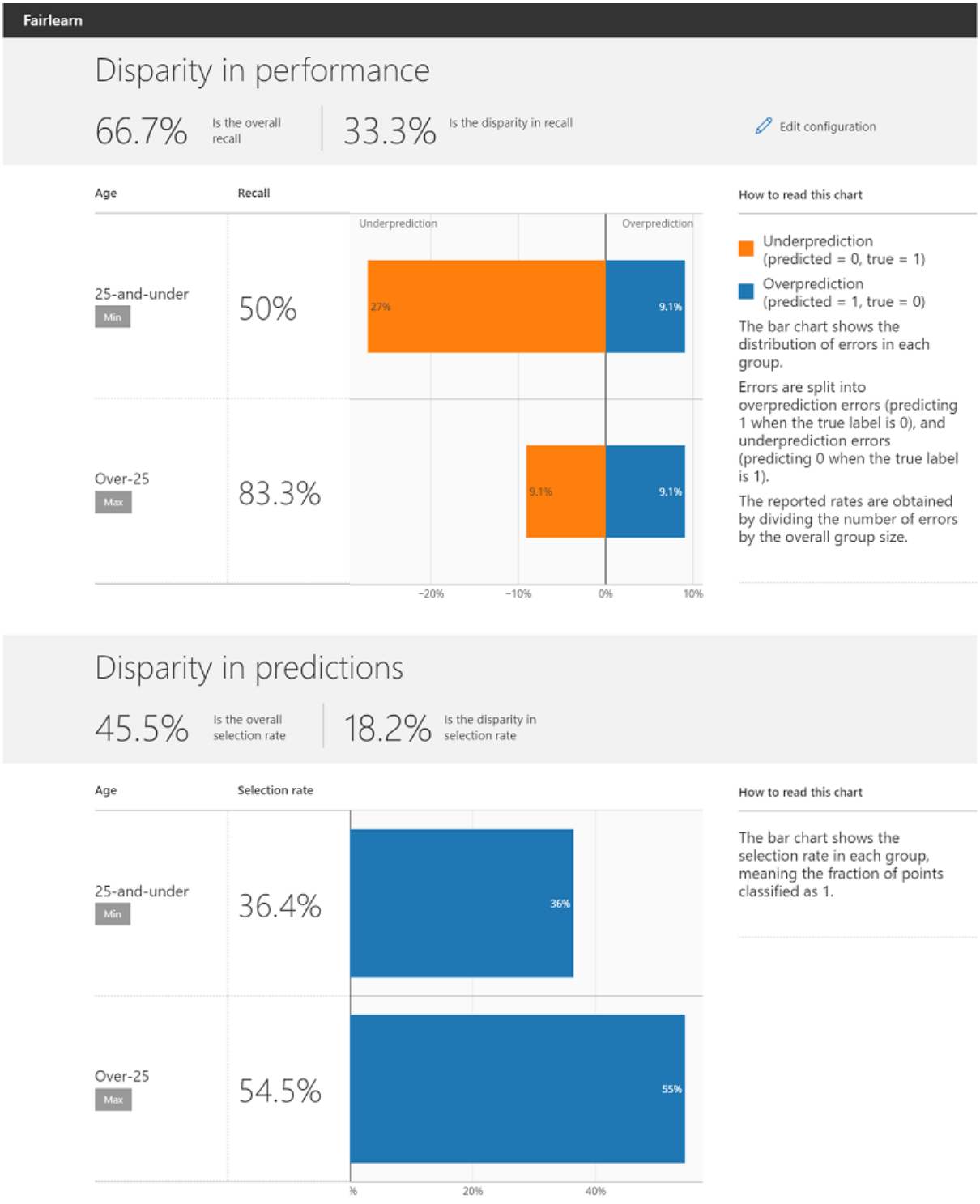
For example, in a binary classification model for loan repayment prediction, where the sensitive feature Age consists of two possible categorical values (25-and-under and over-25), a MetricFrame for these groups might be similar to the following table:

Age	selection_rate	accuracy	recall	precision
25 or younger	0.298178	0.89619	0.825926	0.825926

Over 25	0.708995	0.888889	0.937984	0.902985

Visualizing metrics in a dashboard

It's often easier to compare metrics visually, so Fairlearn provides an interactive dashboard widget that you can use in a notebook to display group metrics for a model. The widget enables you to choose a sensitive feature and performance metric to compare, and then calculates and visualizes the metrics and disparity, like this:



Integration with Azure Machine Learning

Fairlearn integrates with Azure Machine Learning by enabling you to run an experiment in which the dashboard metrics are uploaded to your Azure Machine Learning workspace. This enables you to share the dashboard in Azure Machine Learning studio so that your data science team can track and compare disparity metrics for models registered in the workspace.

Mitigate unfairness with Fairlearn

In addition to enabling you to analyze disparity in selection rates and predictive performance across sensitive features, Fairlearn provides support for mitigating unfairness in models.

Mitigation algorithms and parity constraints

The mitigation support in Fairlearn is based on the use of algorithms to create alternative models that apply *parity constraints* to produce comparable metrics across sensitive feature groups. Fairlearn supports the following mitigation techniques.

Technique	Description	Model type support
Exponentiated Gradient	A *reduction* technique that applies a cost-minimization approach to learning the optimal trade-off of overall predictive performance and fairness disparity	Binary classification and regression
Grid Search	A simplified version of the Exponentiated Gradient algorithm that works efficiently with small numbers of constraints	Binary classification and regression
Threshold Optimizer	A *post-processing* technique that applies a constraint to an existing classifier, transforming the prediction as appropriate	Binary classification

The choice of parity constraint depends on the technique being used and the specific fairness criteria you want to apply. Constraints in Fairlearn include:

- Demographic parity: Use this constraint with any of the mitigation algorithms to minimize disparity in the selection rate across sensitive feature groups. For example, in a binary classification scenario, this

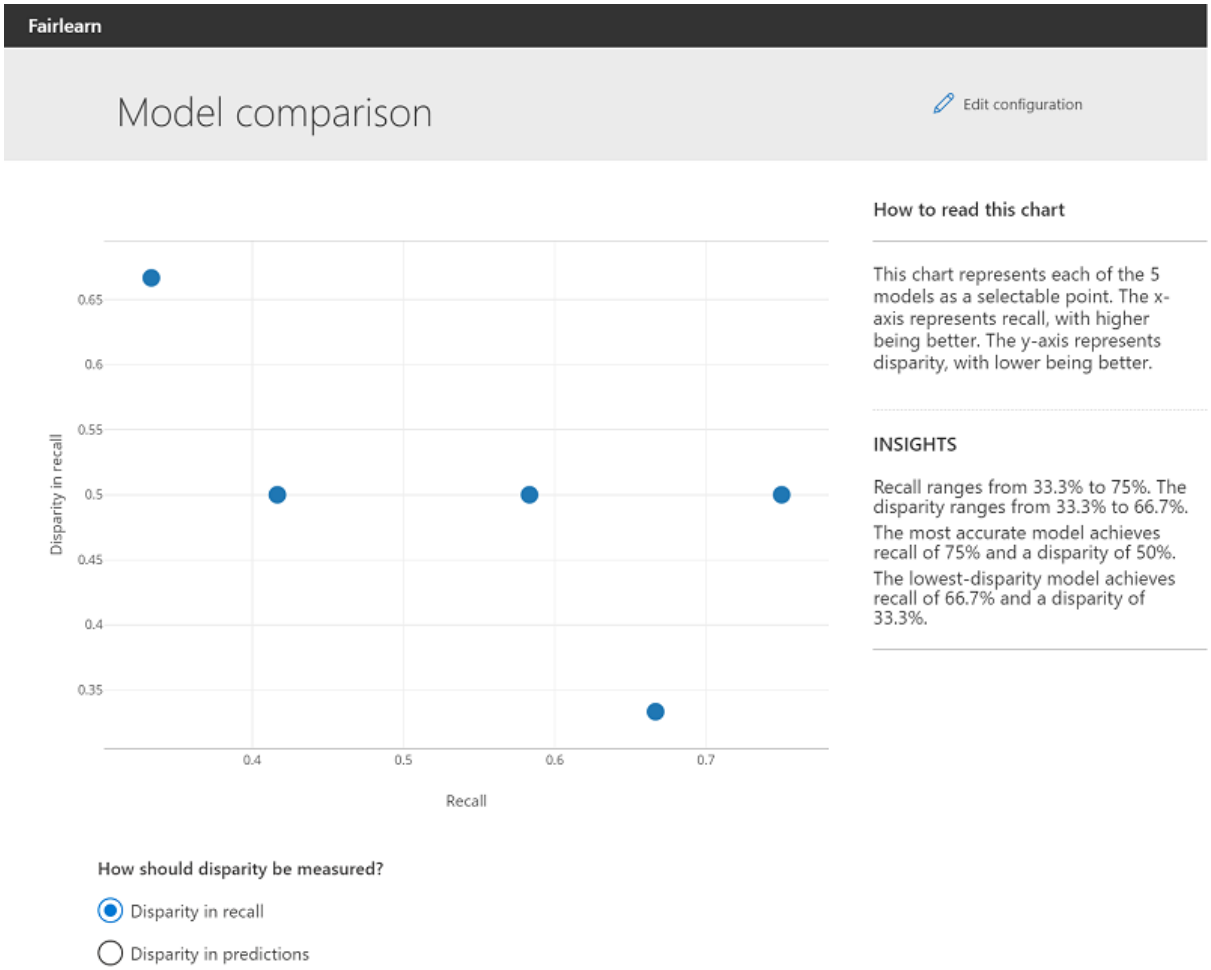
constraint tries to ensure that an equal number of positive predictions are made in each group.

- True positive rate parity: Use this constraint with any of the mitigation algorithms to minimize disparity in *true positive rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive predictions.
- False-positive rate parity: Use this constraint with any of the mitigation algorithms to minimize disparity in *false_positive_rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of false-positive predictions.
- Equalized odds: Use this constraint with any of the mitigation algorithms to minimize disparity in combined *true positive rate* and *false_positive_rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive and false-positive predictions.
- Error rate parity: Use this constraint with any of the reduction-based mitigation algorithms (Exponentiated Gradient and Grid Search) to ensure that the error for each sensitive feature group does not deviate from the overall error rate by more than a specified amount.
- Bounded group loss: Use this constraint with any of the reduction-based mitigation algorithms to restrict the loss for each sensitive feature group in a *regression* model.

Training and evaluating mitigated models

A common approach to mitigation is to use one of the algorithms and constraints to train multiple models, and then compare their performance, selection rate, and disparity metrics to find the optimal model for your needs. Often, the choice of the model involves a trade-off between raw predictive performance and fairness - based on your definition of fairness for a given scenario. Generally, fairness is measured by a reduction in the disparity of feature selection (for example, ensuring that an equal proportion of members from each gender group is approved for a bank loan) or by a reduction in the disparity of performance metric (for example, ensuring that a model is equally accurate at identifying repayers and defaulters in each age group).

Fairlearn enables you to train mitigated models and visualize them using the dashboard, like this.



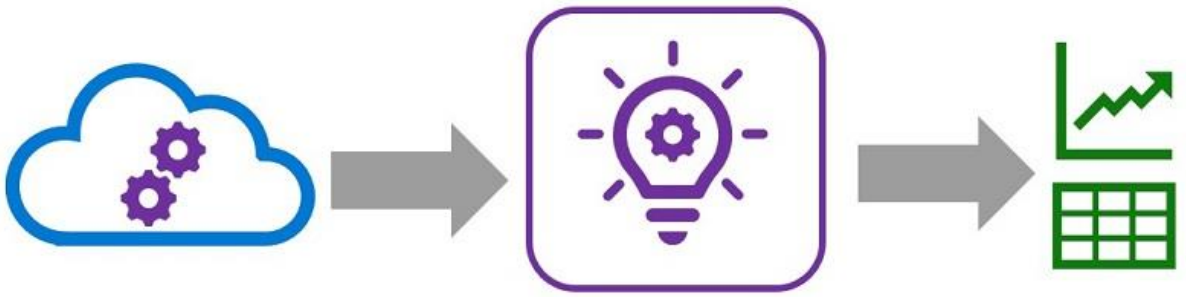
You can select an individual model in the scatterplot to see its details, enabling you to explore the options and select the best model for your fairness requirements.

Integration with Azure Machine Learning

Just as when analyzing an individual model, you can register all of the models found during your mitigation testing and upload the dashboard metrics to Azure Machine Learning.

Introduction-Monitor Models

Application Insights is an application performance management service in Microsoft Azure that enables the capture, storage, and analysis of telemetry data from applications.



You can use Application Insights to monitor telemetry from many kinds of application, including applications that are not running in Azure. All that's required is a low-overhead instrumentation package to capture and send the telemetry data to Application Insights. The necessary package is already included in Azure Machine Learning Web services, so you can use it to capture and review telemetry from models published with Azure Machine Learning.

Enable Application Insights

To log telemetry in application insights from an Azure machine learning service, you must have an Application Insights resource associated with your Azure Machine Learning workspace, and you must configure your service to use it for telemetry logging.

Associate Application Insights with a workspace

When you create an Azure Machine Learning workspace, you can select an Azure Application Insights resource to associate with it. If you do not select an existing Application Insights resource, a new one is created in the same resource group as your workspace.

You can determine the Application Insights resource associated with your workspace by viewing the Overview page of the workspace blade in the Azure portal, or by using the get_details() method of a Workspace object as shown in the following code example:

Python Copy

```
from azureml.core import Workspace

ws = Workspace.from_config()
ws.get_details()['applicationInsights']
```

Enable Application Insights for a service

When deploying a new real-time service, you can enable Application Insights in the deployment configuration for the service, as shown in this example:

Python Copy

```
dep_config = AciWebservice.deploy_configuration(cpu_cores = 1,
                                                memory_gb = 1,
                                                enable_app_insights=True)
```

If you want to enable Application Insights for a service that is already deployed, you can modify the deployment configuration for Azure Kubernetes Service (AKS) based services in the Azure portal. Alternatively, you can update any web service by using the Azure Machine Learning SDK, like this:

Python Copy

```
service = ws.webservices['my-svc']
service.update(enable_app_insights=True)
```

Capture and view telemetry

Application Insights automatically captures any information written to the standard output and error logs, and provides a query capability to view data in these logs.

Write log data

To capture telemetry data for Application insights, you can write any values to the standard output log in the scoring script for your service by using a `print` statement, as shown in the following example:

Python

```
def init():
    global model
    model = joblib.load(Model.get_model_path('my_model'))
def run(raw_data):
    data = json.loads(raw_data)['data']
    predictions = model.predict(data)
    log_txt = 'Data:' + str(data) + ' - Predictions:' + str(predictions)
    print(log_txt)
    return predictions.tolist()
```

Azure Machine Learning creates a *custom dimension* in the Application Insights data model for the output you write.

Query logs in Application Insights

To analyze captured log data, you can use the Log Analytics query interface for Application Insights in the Azure portal. This interface supports a SQL-like query syntax that you can use to extract fields from logged data, including custom dimensions created by your Azure Machine Learning service.

For example, the following query returns the timestamp and customDimensions.Content fields from log traces that have a message field value of *STDOUT* (indicating the data is in the standard output log) and a customDimensions.["Service Name"] field value of *my-svc*:

```
SQL

traces
|where message == "STDOUT"
  and customDimensions.["Service Name"] = "my-svc"
| project  timestamp, customDimensions.Content
```

This query returns the logged data as a table:

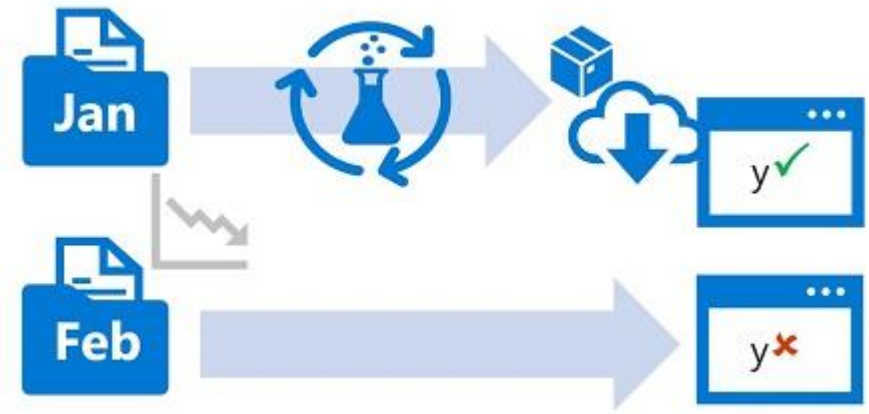
timestamp	customDimensions_Content
01/02/2020...	Data:[[1, 2, 2.5, 3.1], [0, 1, 1,7, 2.1]] - Predictions:[0 1]
01/02/2020...	Data:[[3, 2, 1.7, 2.0]] - Predictions:[0]

Monitor data drift with Azure Machine Learning

You typically train a machine learning model using a historical dataset that is representative of the new data that your model will receive for inferencing. However, over time there may be trends that change the profile of the data, making your model less accurate.

For example, suppose a model is trained to predict the expected gas mileage of an automobile based on the number of cylinders, engine size, weight, and other

features. Over time, as car manufacturing and engine technologies advance, the typical fuel-efficiency of vehicles might improve dramatically; making the predictions made by the model trained on older data less accurate.



This change in data profiles between training and inferencing is known as *data drift*, and it can be a significant issue for predictive models used in production. It is therefore important to be able to monitor data drift over time, and retrain models as required to maintain predictive accuracy.

Learning objectives

In this module, you will learn how to:

- Create a data drift monitor.
- Schedule data drift monitoring.
- View data drift monitoring results.
-

Creating a data drift monitor

Azure Machine Learning supports data drift monitoring through the use of *datasets*. You can capture new feature data in a dataset and compare it to the dataset with which the model was trained.

Monitor data drift by comparing datasets

It's common for organizations to continue to collect new data after a model has been trained. For example, a health clinic might use diagnostic measurements from

previous patients to train a model that predicts diabetes likelihood, but continue to collect the same diagnostic measurements from all new patients. The clinic's data scientists could then periodically compare the growing collection of new data to the original training data, and identify any changing data trends that might affect model accuracy.

To monitor data drift using registered datasets, you need to register two datasets:

- A *baseline* dataset - usually the original training data.
- A *target* dataset that will be compared to the baseline based on time intervals. This dataset requires a column for each feature you want to compare, and a timestamp column so the rate of data drift can be measured.

Note

You can configure a deployed service to collect new data submitted to the model for inferencing, which is saved in Azure blob storage and can be used as a target dataset for data drift monitoring. See Collect data from models in production in the Azure Machine Learning documentation for more information.

After creating these datasets, you can define a *dataset monitor* to detect data drift and trigger alerts if the rate of drift exceeds a specified threshold. You can create dataset monitors using the visual interface in Azure Machine Learning studio, or by using the DataDriftDetector class in the Azure Machine Learning SDK as shown in the following example code:

Python

```
from azureml.datadrift import DataDriftDetector

monitor = DataDriftDetector.create_from_datasets(workspace=ws,
                                                 name='dataset-drift-detector',
                                                 baseline_data_set=train_ds,
                                                 target_data_set=new_data_ds,
                                                 compute_target='aml-cluster',
                                                 frequency='Week',
                                                 feature_list=['age','height', 'bmi'],
                                                 latency=24)
```

After creating the dataset monitor, you can *backfill* to immediately compare the baseline dataset to existing data in the target dataset, as shown in the following example, which backfills the monitor based on weekly changes in data for the previous six weeks:

```python
import datetime as dt

backfill = monitor.backfill( dt.datetime.now() - dt.timedelta(weeks=6), dt.datetime.now())
```

Scheduling alerts

When you define a data monitor, you specify a schedule on which it should run. Additionally, you can specify a threshold for the rate of data drift and an operator email address for notifications if this threshold is exceeded.

Configure data drift monitor schedules

Data drift monitoring works by running a comparison at scheduled frequency, and calculating data drift metrics for the features in the dataset that you want to monitor. You can define a schedule to run every Day, Week, or Month.

For dataset monitors, you can specify a latency, indicating the number of hours to allow for new data to be collected and added to the target dataset. For deployed model data drift monitors, you can specify a schedule_start time value to indicate when the data drift run should start (if omitted, the run will start at the current time).

Configure alerts

Data drift is measured using a calculated *magnitude* of change in the statistical distribution of feature values over time. You can expect some natural random variation between the baseline and target datasets, but you should monitor for large changes that might indicate significant data drift.

You can define a threshold for data drift magnitude above which you want to be notified, and configure alert notifications by email. The following code shows an example of scheduling a data drift monitor to run every week, and send an alert if the drift magnitude is greater than 0.3:

```python
alert_email = AlertConfiguration('data_scientists@contoso.com')
monitor = DataDriftDetector.create_from_datasets(ws, 'dataset-drift-detector',
                                                 baseline_data_set, target_data_set,
                                                 compute_target=cpu_cluster,
                                                 frequency='Week', latency=2,
                                                 drift_threshold=.3,
                                                 alert_configuration=alert_email)
```

Introduction-Exploring security

Machine learning requires large quantities of data to train effective models. Some of this data can contain individuals' or companies' sensitive information that must be secured to remain private. Security challenges surrounding machine learning are not just limited to data, and the setup of creating a secure work environment can be daunting. Azure Machine Learning (Azure ML) provides several mechanisms that offers granular control of the network environment, resources used within it, and the data being accessed. Proper measures can protect both from outside attackers and from internal issues like negligence or immature processes.

Many attacks on machine learning systems involve accessing models through insecure networks to steal or manipulate data to affect model performance and access sensitive data. By building better, more secure training data stores, locking down machine learning platforms, and controlling access to inputs and outputs, we can ensure data remains protected. These features are useful for data scientists, administrators, and operations engineers who want to create secure configurations compliant with their companies' policies.

What is role-based access control?

Azure Role-Based Access Control (Azure RBAC) is an authorization system that allows fine-grained access management of Azure Machine Learning resources. It enables you to manage team members' access to Azure cloud resources by assigning roles. These roles determine what assets team members can use in the workspace, and what they can do with those resources. This can be important if your team works with sensitive data such as hospital records that contain private medical information, or if you wish to restrict access to critical assets for junior team members within Azure Machine Learning. Using these roles, you can realistically

reflect your organization's structure, ensuring responsibilities and critical assets are portioned to the correct individuals.

User roles

Azure RBAC roles can be assigned to individuals and groups. The rights assigned to each group are allocated as a permission-based system, with set access and restrictions being clearly defined. This control is applied at the workspace level and can only be changed by administrators or owners of the specific workspace within Azure Machine Learning. An Azure Machine Learning workspace, like other Azure resources, comes with three default roles when it is created. You can add users to the workspace and assign one of these roles:

- Owners have full access to the workspace, including the ability to view, create, edit, or delete assets in a workspace. Owners can also change role assignments.
- Contributors can view, create, edit, or delete assets in a workspace. For example, contributors can create an experiment, create or attach a compute cluster, submit a run, and deploy a web service.
- Readers can only perform read-only actions in the workspace. Readers can list and view assets, including datastore credentials in a workspace. Readers can't create or update these assets.

Custom roles

If the default roles do not meet your organization's need for more selective access control, you can create your own Custom roles. Custom roles give you the flexibility to develop permission-based rules for individuals' or groups that provide access while defining your own security stipulations to secure data or resources. You can make a role available at a specific workspace level, a specific resource group level, or a subscription level by defining the scope of your custom role, which we can see in the example JSON below.

Custom roles can be created by defining possible actions permitted and NotActions to restrict specific activities or access. You can create custom roles using Azure portal, Azure PowerShell, Azure CLI, or the REST API. Below we can see a custom role JSON request for a Data Scientist:

```
{
    "Name": "Data Scientist Custom",
    "IsCustom": true,
    "Description": "Can run experiment but can't create or delete compute or deploy production endpoints.",
    "Actions": [
        "Microsoft.MachineLearningServices/workspaces/*/read",
        "Microsoft.MachineLearningServices/workspaces/*/action",
        "Microsoft.MachineLearningServices/workspaces/*/delete",
        "Microsoft.MachineLearningServices/workspaces/*/write"
    ],
    "NotActions": [
        "Microsoft.MachineLearningServices/workspaces/delete",
        "Microsoft.MachineLearningServices/workspaces/write",
        "Microsoft.MachineLearningServices/workspaces/computes/*/write",
        "Microsoft.MachineLearningServices/workspaces/computes/*/delete",
        "Microsoft.Authorization/*",
        "Microsoft.MachineLearningServices/workspaces/computes/listKeys/action",
        "Microsoft.MachineLearningServices/workspaces/listKeys/action",
        "Microsoft.MachineLearningServices/workspaces/services/aks/write",
        "Microsoft.MachineLearningServices/workspaces/services/aks/delete",
        "Microsoft.MachineLearningServices/workspaces/endpoints/pipelines/write"
    ],
    "AssignableScopes": [
        "/subscriptions/<subscription_id>/resourceGroups/<resource_group_name>/providers/Microsoft.MachineLear
    ]
}
```

Note the *Actions* and *NotActions* above which define the permissions for the custom role and the assigned scope that is at the specific workspace level. In this example, the data scientist's actions are defined through a wildcard (represented by the * sign) which extends a permission to everything that matches the action string you provide. So, in the above example the wildcard string adds all permissions related to any read, write, action, or deletion within the workspace.

However, if we look at the NotActions above, we can see restrictions on the deletion or creation of workspaces or new compute resources, amongst others. These restrictions take precedence over explicitly allowed actions.

System

A common challenge for developers is the management of secrets and credentials used to secure communication between different components. Azure Machine Learning relies on Azure Active Directory (Azure AD) for authentication and/or communication between other Azure cloud resources. Azure AD is a cloud-based identity and access management service that helps your employees' sign-in and access cloud resources on Azure.

Authentication with Azure AD

In general, there are three authentication workflows that you can use when connecting to the workspace:

Interactive: You use your account in Azure Active Directory to either manually authenticate or obtain an authentication token. Interactive authentication is used during experimentation and iterative development. It enables you to control access to resources (such as a web service) on a per-user basis.

Service principal: You create a service principal account in Azure Active Directory, and use it to authenticate or obtain an authentication token. A service principal is used when you need an automated process to authenticate to the service. For example, a continuous integration and deployment script that trains and tests a model every time the training code changes needs ongoing access and so would benefit from a service principal account.

Azure CLI session: You use an active Azure CLI session to authenticate. Azure CLI authentication is used during experimentation and iterative development, or when you need an automated process to authenticate to the service using a pre-authenticated session. You can log in to Azure via the Azure CLI on your local workstation, without storing credentials in code or prompting the user to authenticate.

Managed identity: When using the Azure Machine Learning SDK on an Azure Virtual Machine, you can use a managed identity for Azure. This workflow allows the VM to connect to the workspace using the managed identity, without storing credentials in code or prompting the user to authenticate. Azure Machine Learning compute clusters can also be configured to use a managed identity to access the workspace when training models.

Managed Identities

When configuring the Azure Machine Learning workspace in a trustworthy manner, it is vital to ensure that different cloud services associated with the workspace have the correct level of access. Managed identities allow you to authenticate services by providing an automatically managed identity for applications or services to use when connecting to Azure cloud services. Managed identities work with any service that supports Azure AD authentication, and provides activity logs so admins can see user activity such as log-in times, when operations were started, and by whom.

There are two types of managed identities:

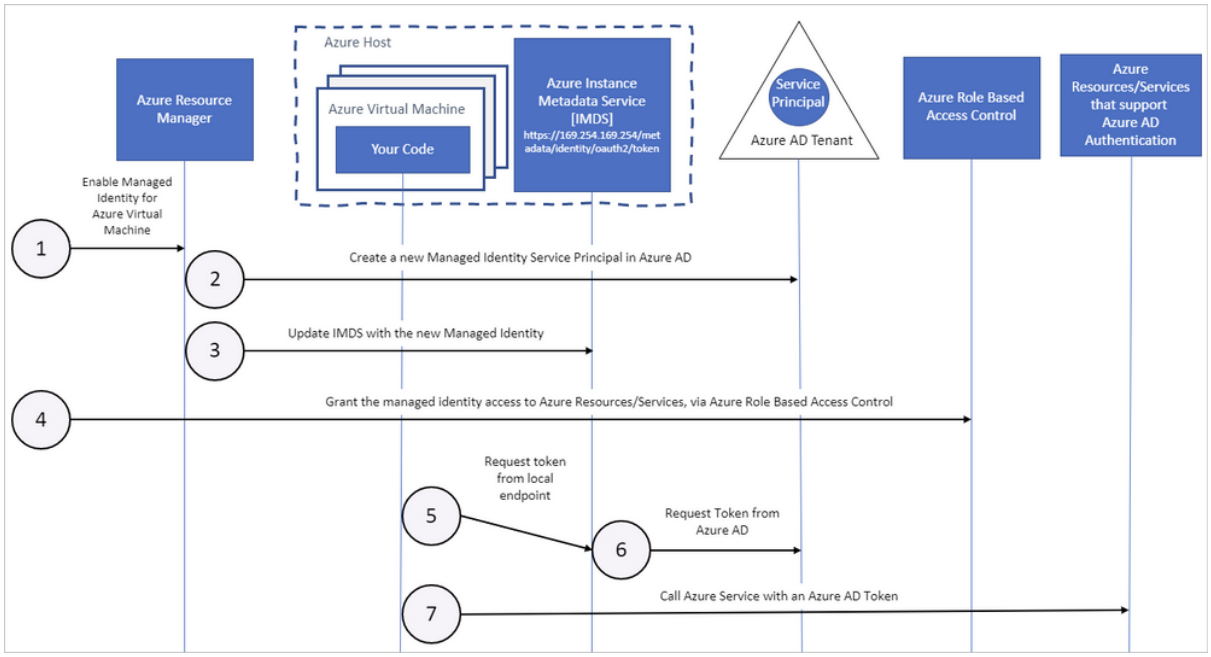
- System-assigned: Some Azure services allow you to enable a managed identity directly on a service instance. When you enable a system-assigned managed identity, an identity is created in Azure AD tied to that service instance's lifecycle. By design, only that Azure resource can use

this identity to request tokens from Azure AD, and when the resource is deleted, Azure automatically deletes the identity for you.

- User-assigned: You may also create a managed identity as a standalone Azure resource. You can create a user-assigned managed identity and assign it to one or more instances of an Azure service. The identity is managed separately from the resources that use it and will persist if a resource using it is removed. For simplicity, we recommend using system-assigned roles unless you require a custom access solution.

Once you have a managed identity, you can request tokens via a token endpoint on a resource such as a virtual machine. These tokens then work with users existing authorization from Azure RBAC to permit them to do actions, such as pull keys from Azure Key Vault or any other secret. Similarly, suppose that user has an Azure RBAC role that permits using an Azure storage solution. In that case, that token can be used to authenticate and either read or push data to storage without any credentials in their code. Operations on managed identities may be performed by using an Azure Resource Manager (ARM) template, the Azure portal, the Azure CLI, PowerShell, and REST APIs.

Below we can see the typical workflow for a managed identity within a virtual machine:



Identities with compute clusters

Azure Machine Learning compute clusters can use managed identities to authenticate access to Azure resources within Azure Machine Learning without including credentials in your code. This is useful to quickly provide the minimum

required permissions to access resources while securing other critical resources. For example, during machine learning workflow, the workspace needs access to Azure Container Registry for Docker images, and storage accounts for training data. By default, the System-assigned identity is enabled directly on the Azure Machine Learning compute clusters and these resources will all be available to use. Once the compute cluster is deleted, Azure will automatically clean up the credentials and the identity in Azure Active Directory. Compute clusters can also support custom user-assigned identities assigned to multiple resources and will persist after resources are deleted.

During cluster creation or when editing compute cluster details, in the Advanced settings, toggle Assign a managed identity and specify a system-assigned identity or user-assigned identity. Note that Azure Machine Learning compute clusters support only one system-assigned identity or multiple user-assigned identities, not both concurrently.
Note

Managed identities can only be used with Azure Machine Learning when using the Azure Machine Learning SDK on an Azure Virtual Machine or Azure Machine Learning compute cluster. This workflow allows the virtual machine to connect to the workspace using the managed identity instead of the individual user's Azure AD account, and without storing credentials in code.

Keys and secrets with Azure Key Vault

Most networked applications need to work with secrets, such as database connection strings or passwords. When perform machine learning on Azure, we normally use secrets to access training data or look at results. Particularly when working with private data, it’s important to make sure the secrets are properly managed.

Don’t store secrets in source code

Storing secrets in source code is impractical and a security anti-pattern. This is for multiple reasons:

- Changing passwords means updating source code, which can mean rebuilding and re-publishing applications.
- Hard-coded secrets make it awkward to work with different environments, such as staging and production environments. This also increases the risk of inadvertent modification or destruction of production environment data during development.

- All people with access to the source code gain access to all secrets. This makes it near impossible to ensure that only senior team members have access to sensitive resources. It also means that any sharing, or leak, of your source code also provides outside parties with your security keys.
- Source control, such as git, will typically retain old passwords in history. This means future team members gain access to all historical passwords.

One of the best alternatives to storing secrets in source code is to make them available in the application environment. In this pattern, your application requests secrets from the environment and then uses these to connect to the requisite resources. The aforementioned drawbacks of storing secrets in source code are eliminated, so long as each environment has different secret values, such as different passwords to access certain resources.

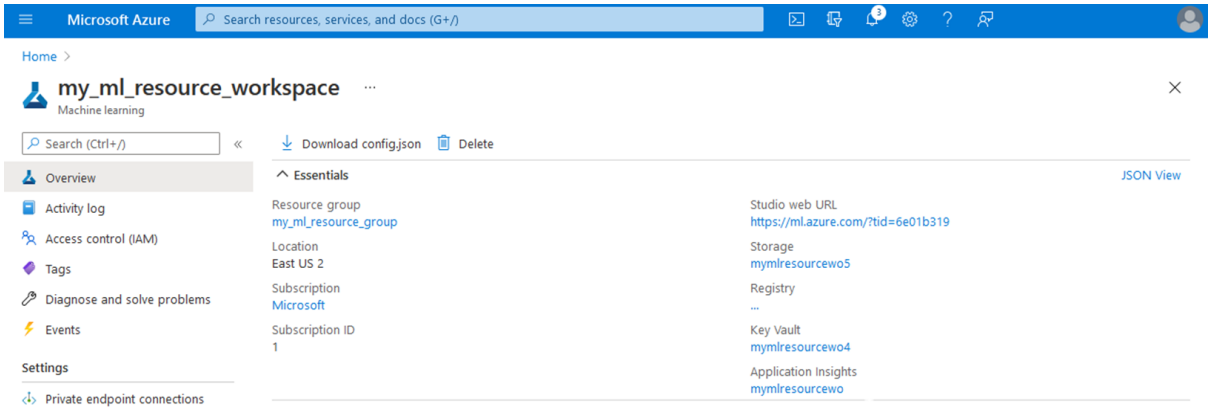
Azure Key Vault

Azure Key Vault provides secure storage of generic secrets for applications in Azure-hosted environments. Any type of secret can be stored, so long as its value is no larger than 25kb and it can be read and returned as a string. Secrets are named, and their content type (such as password or certificate) can optionally be stored alongside the value to provide a hint that assists in its interpretation when retrieved.

Secrets stored in Azure Key Vault are encrypted, optionally at the hardware level. This is handled transparently, and requires no action from the user or the application requesting the secrets. They can also be temporarily disabled, and automatically activate or expire on a certain date.

How Key Vault works with Azure Machine Learning

When you create an Azure Machine Learning workspace, this automatically creates a Key Vault. To view the Azure Key Vault associated with your workspace, open the workspace’s Overview tab. Your key vault appears on the right hand side.



When you first create your workspace, your Key Vault will be automatically created. The Key Vault can be accessed through your application. For example, you can use the Azure Shell to set an environmental variable holding Key Store's name, and save a password to that key store like so:

Bash

```bash
# export the name of the vault to an environmental variable
export KEY_VAULT_NAME=<your-unique-keyvault-name>

# Save a new secret, called ExamplePassword
az keyvault secret set --vault-name $KEY_VAULT_NAME --name "ExamplePassword" --value "hVFkk965BuUv"
```

This password is stored securely and is encrypted. As an example, a Python application using Azure Machine Learning's SDK can access this key as follows:

Python

```python
'''
Simple example of obtaining a secret from the keyvault.
Assumes azure-identity and azure-keyvault-secrets have been
pip installed
'''

import os
from azure.keyvault.secrets import SecretClient
from azure.identity import DefaultAzureCredential

# Get the key vault name
keyVaultName = os.environ["KEY_VAULT_NAME"]


# Create a client to access the secret
credential = DefaultAzureCredential()
client = SecretClient(vault_url= f"https://{keyVaultName}.vault.azure.net", credential=credential)

# Get a secret and print it to the console
# Note that printing out passwords is bad practice and only
# performed here for learning purposes
retrieved_secret = client.get_secret("ExamplePassword")
print(f"Your secret is '{retrieved_secret.value}'")
```

```
Bash

Your secret is 'hVFkk965BuUv'
```

Working with remote runs

The above provides a generic solution to using Key Vault. Typically, with Azure Machine Learning, you will be executing code through a remote run.

The standard flow for using secrets in this context is:

- Log in to Azure and connect to your workspace,
- Set a secret in Workspace Key Vault,
- Submit a remote run, then
- Within the remote run, get the secret from Key Vault and use it.

When using the Python SDK and a run, secrets can be easily accessed directly. This is because a submitted run is aware of its workspace. For example:

```
Python

# Code in submitted run
from azureml.core import Experiment, Run

run = Run.get_context()
secret_value = run.get_secret(name=" ExamplePassword")
```

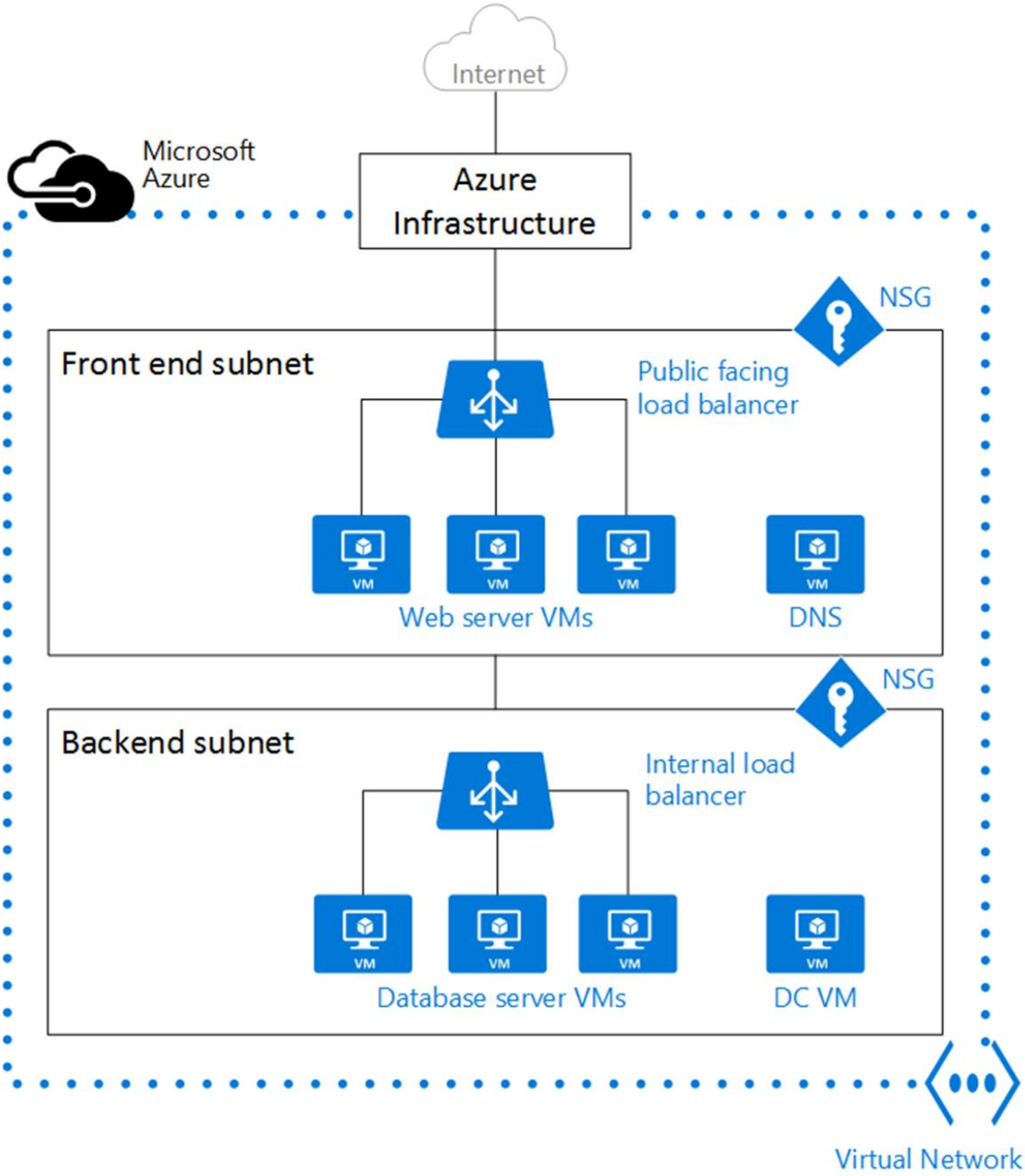
Secure your Azure Machine Learning network

Before you begin training models, it's essential to secure your Azure Machine Learning network from outside intrusion. Without first securing your network, you can leave your data and models exposed to potentially malicious actors and lead to data

theft or attacks that could negatively change model behavior. These alterations can often be difficult to spot due to the often large nature of datasets or parameters influencing model behavior. We will begin by separating your model training from the wider net to its own virtual network to avoid these problems.

Virtual networks and security groups

To secure the Azure Machine Learning workspace and compute resources, we will use a virtual network (VNet). An Azure VNet is the fundamental building block for your private network in Azure. VNet enables Azure resources, such as Azure Blob Storage and Azure Container Registry, to securely communicate with each other, the internet, and on-premises networks. VNet is similar to a traditional network that you'd operate in your own data center, but brings with it additional benefits of Azure's infrastructure such as scale, availability, and isolation. With a VNet, you can enhance security between Azure resources and filter network traffic to ensure only trusted users have access to the network.



In the above image, we can see a typical structure for a Virtual Network (VNet) comprised of:

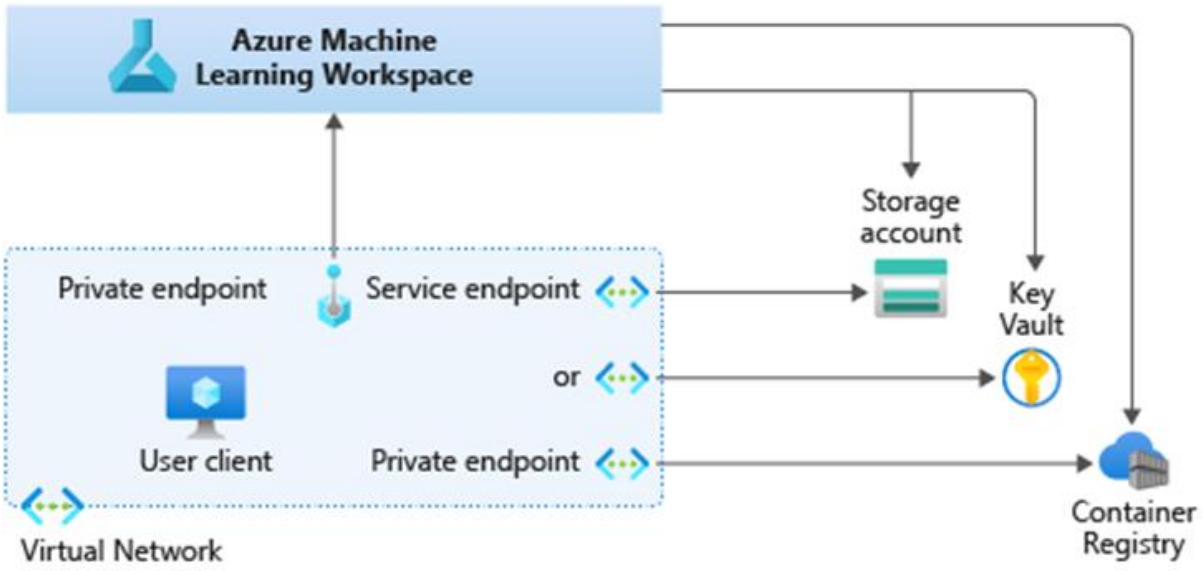
- IP address space: When creating a VNet, you must specify a custom private IP address space using public and private (RFC 1918) addresses.
- Subnets: Shown above as separate virtual machines (VMs), subnets enable you to segment the virtual network into one or more sub-networks and allocate a portion of the virtual network's address space to each subnet, enhancing security and performance.

- Network interfaces (NIC) are the interconnection between a VM and a virtual network (VNet). When you create a VM in the Azure portal, a network interface is automatically created for you.
- Network security groups (NSG) can contain multiple inbound and outbound security rules that enable you to filter traffic to and from resources by source and destination IP address, port, and protocol.
- Load balancers can be configured to efficiently handle inbound and outbound traffic to VMs and VNets, while also offering metrics to monitor the health of VMs.

Paths into a VNet

Integrating Azure services to an Azure virtual network enables private access to the service from virtual machines or compute resources in the virtual network. You can integrate Azure services in your virtual network with the following options:

- Service endpoints provide the identity of your virtual network to the Azure service. Once you enable service endpoints in your virtual network, you can add a virtual network rule to secure the Azure service resources to your virtual network. Service endpoints use public IP addresses.
- Private endpoints are network interfaces that securely connect you to a service powered by Azure Private Link. Private endpoint uses a private IP address from your VNet, effectively bringing the Azure services into your VNet.



You can connect your on-premises computers and networks to a VNet through a virtual private network (VPN) in several ways. A Point-to-site VPN is a connection

between a virtual network and a single computer in your network. The communication is sent through an encrypted tunnel over the internet. Each computer that wants to establish connectivity with a VNet must configure its connection, so it's best used if you only have a few users who need to connect to the VNet. This connection type is great if you're just getting started as it requires little or no changes to your existing network.

While a Site-to-site VPN can be established between your on-premises VPN device and an Azure VPN Gateway that's deployed in a virtual network. This connection type enables any on-premises resource that you authorize to access a virtual network. The communication between your on-premises VPN device and an Azure VPN gateway is sent through an encrypted tunnel over the internet.

ExpressRoute can be used instead of a VPN If you wish to speed up creating private connections to Azure services. The service allows you to create private connections between Microsoft data centers and infrastructure on your premises or in another facility. ExpressRoute connections are separate from the public internet and offer high security, reliability, and speeds with lower latency than typical connections over the internet. ExpressRoute has a range of pricing options depending on your estimated bandwidth requirements.

Securing the workspace and resources

To begin securing your network, you will need to connect to your workspace via a private endpoint (private IP), which we will cover in the next exercise. The private endpoint can be added to a workspace through the *Azure Machine Learning Python SDK, Azure CLI,* or within the *Networking tab of the Azure portal.* From there, you can then limit access to your workspace to only occur over the private IP addresses.

However, this alone will not ensure end-to-end security by itself, so make sure other Azure services you are communicating with are also behind the VNet. Since communication to the workspace is then set to be only allowed from the VNet, any development environments that use the workspace must be members of the VNet unless you have configured the network to allow public IP connections.

The following methods can be used to connect to the secure workspace:

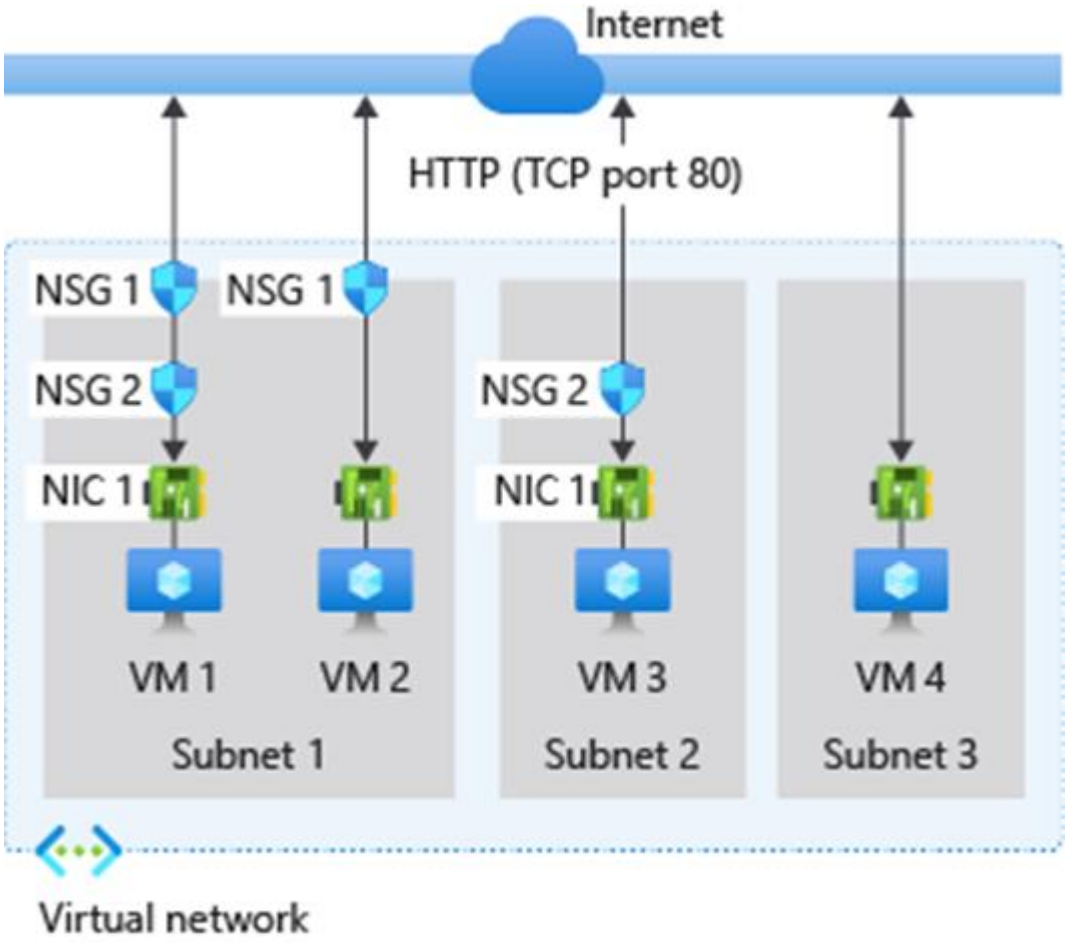
- Azure VPN gateway - Connects on-premises networks to the VNet over a private connection. Connection is made over the public internet. There are two types of VPN gateways that you might use:
    - Point-to-site: Each client computer uses a VPN client to connect to the VNet.
    - Site-to-site: A VPN device connects the VNet to your on-premises network.

- ExpressRoute - Connects on-premises networks into the cloud over a private connection. Connection is made using a connectivity provider.
- Azure Bastion - In this scenario, you create an Azure Virtual Machine (sometimes called a jump box) inside the VNet. You then connect to the VM using Azure Bastion. Bastion allows you to connect to the VM using either an RDP or SSH session from your local web browser. You then use the jump box as your development environment. Since it is inside the VNet, it can directly access the workspace.

Security groups and traffic

You can use an Azure network security group (NSG) to filter network traffic to and from Azure resources in an Azure virtual network. A network security group contains security rules that allow or deny inbound network traffic to, or outbound network traffic from, several types of Azure resources. NSGs are useful for controlling the traffic flow between VM subnets or limiting outbound communication by resources within an Azure VNet to the internet, which is enabled by default.

The following picture illustrates different scenarios for how network security groups might be deployed to allow network traffic to and from the internet over TCP port 80:

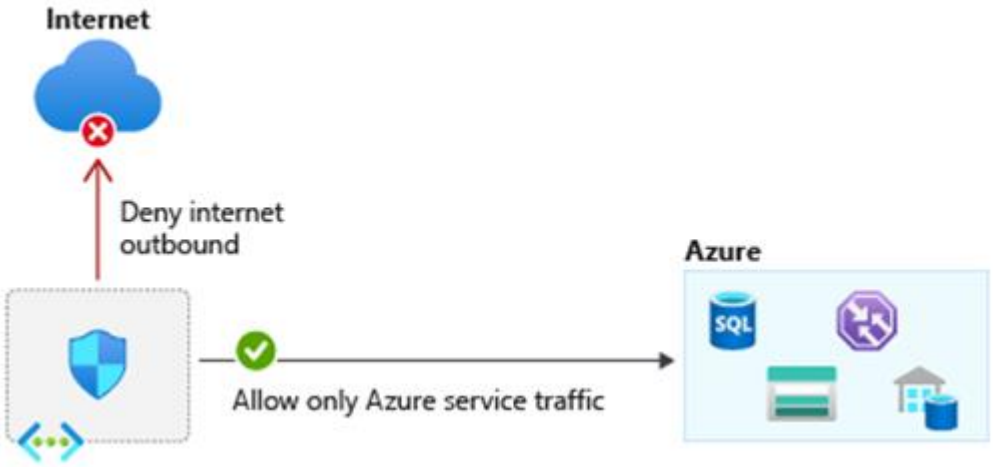


Application security groups (ASG) can also be used to configure network security as a natural extension of an application's structure, allowing you to group virtual machines and define network security policies based on those groups. You can reuse your security policy at scale without manual maintenance of explicit IP addresses. The platform handles the complexity of explicit IP addresses and multiple rule sets, simplifying the NSG rule definition process immensely.

Service tags

A service tag represents a group of IP address prefixes from a given Azure service. Microsoft manages the address prefixes encompassed by the service tag and automatically updates the service tag as addresses change, minimizing the complexity of frequent updates to network security rules.

You can use service tags in place of specific IP addresses when you create security rules to define network access controls on network security groups or Azure Firewall. By specifying the service tag name, such as ApiManagement, in the appropriate source or destination field of a rule, you can allow or deny the traffic for the corresponding service.



Network Security Group (NSG)				
Action	Name	Source	Destination	Port
Allow	AllowStorage	VirtualNetwork	Storage	Any
Allow	AllowSQL	VirtualNetwork	Sql.EastUS	Any
Deny	DenyAllOutBound	Any	Any	Any

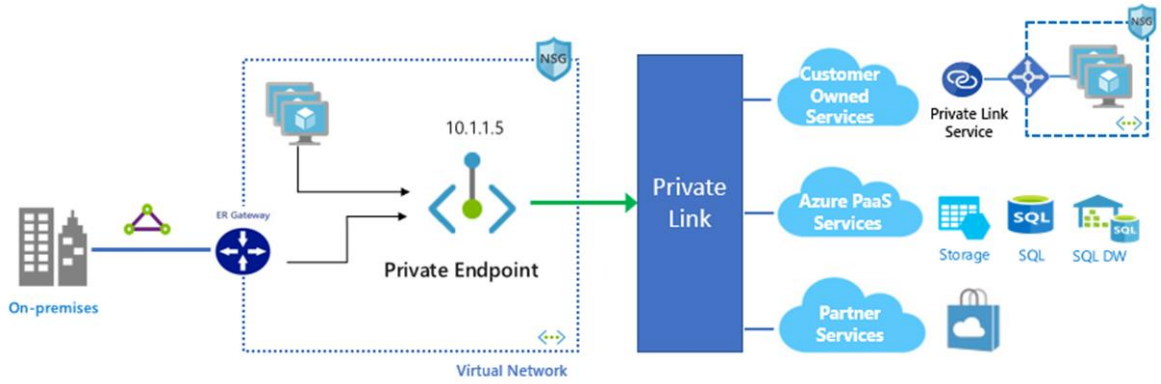
Private endpoints & Private Link

The Azure Machine Learning workspace can use Azure Private Link to create a private endpoint behind the VNet. Azure Private Link is a technology designed to provide connectivity to selected PaaS services. This provides a set of private IP addresses that can be used to access the workspace from within the VNet. Some of the services that Azure Machine Learning relies on can also use Azure Private Link, but some rely on network security groups or user-defined routing.

There are two key components of Azure Private Link:

- Private endpoint – a network interface connected to your virtual network, assigned with a private IP address. It is used to connect privately and securely to a service powered by Azure Private Link or a Private Link Service that you or a partner might own.
- Private Link Service – your own service, powered by Azure Private Link that runs behind an Azure Standard Load Balancer, enabled for Private Link access. This service can be privately connected with and consumed using Private Endpoints deployed in the user's virtual network.

When you create a private endpoint for your Azure resource, it provides secure connectivity between clients on your virtual network and your Azure resource. You can use private endpoints to communicate and ingress events directly from your virtual network to Azure resources securely over a private link without going through the public internet, boosting security. The private endpoint is assigned an IP address from the IP address range of your virtual network. The service is flexible, allowing connections between VNets with overlapping address spaces and connecting resources running in other regions, offering global reach.



Azure Private Link – Connecting Azure Services privately to your Network

Introduction- Azure Databricks

Azure Databricks is a Microsoft analytics service, part of the Microsoft Azure cloud platform. It offers an integration between Microsoft Azure and the Apache Spark's Databricks implementation. Azure Databricks natively integrates with Azure security and data services. In this module, you will learn how to work with the key features of Azure Databricks.

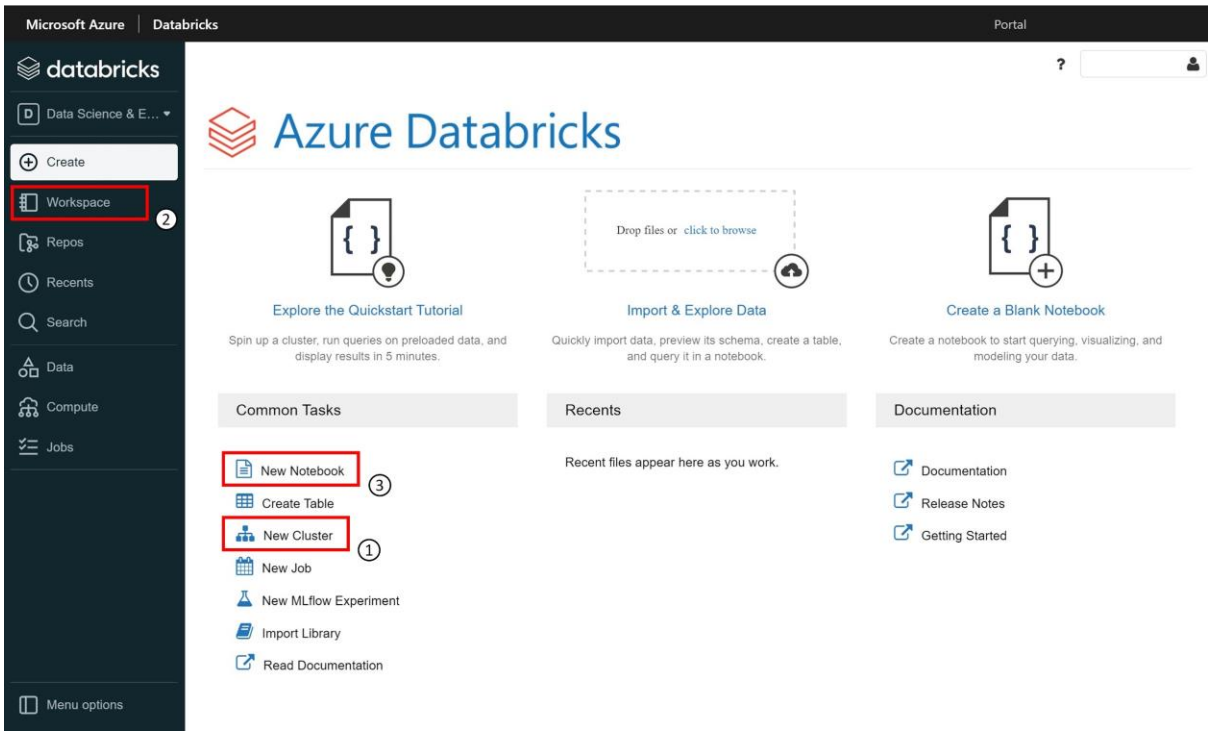
Understand Azure Databricks

Azure Databricks runs on top of a proprietary data processing engine called Databricks Runtime, an optimized version of Apache Spark. It allows up to 50x performance for Apache Spark workloads.

Apache Spark is the core technology. Spark is an open-source analytics engine for large-scale data processing. It provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

In a nutshell: Azure Databricks offers a fast, easy, and collaborative Spark based analytics service. It is used to accelerate big data analytics, artificial intelligence, performant data lakes, interactive data science, machine learning, and collaboration.

The main concepts in Azure Databricks



The landing page shows the fundamental concepts to be used in Databricks:

1. The cluster: a set of computational resources on which we run the code.

2. The workspace: groups all the Databricks elements, clusters, notebooks, data.
3. The notebook: a document that contains runnable code, descriptive text, and visualizations.

Provision Azure Databricks workspaces and clusters

Two of the key concepts you need to be familiar with when working with Azure Databricks are workspaces and clusters.

Workspaces

A workspace is an environment for accessing all of your Databricks elements:

- It groups objects (like notebooks, libraries, experiments) into folders,
- Provides access to your data,
- Provides access to the computations resources used (clusters, jobs).

Clusters

A cluster is a set of computational resources on which you run your code (as notebooks or jobs). We can run ETL pipelines, or machine learning, data science, analytics workloads on the cluster.

We can create:

- An all-purpose cluster. Multiple users can share such clusters to do collaborative interactive analysis.
- A job cluster to run a specific job. The cluster will be terminated when the job completes (A job is a way of running a notebook or JAR either immediately or on a scheduled basis).

Before we can use a cluster, we have to choose one of the available runtimes.

Databricks runtimes are the set of core components that run on Azure Databricks clusters. Azure Databricks offers several types of runtimes:

- Databricks Runtime: includes Apache Spark, components and updates that optimize the usability, performance, and security for big data analytics.
- Databricks Runtime for Machine Learning: a variant that adds multiple machine learning libraries such as TensorFlow, Keras, and PyTorch.
- Databricks Light: for jobs that don't need the advanced performance, reliability, or autoscaling of the Databricks Runtime.

Working with data in a workspace

An Azure Databricks *database* is a collection of tables. An Azure Databricks *table* is a collection of structured data.

We can cache, filter, and perform any operations supported by Apache Spark DataFrames on Azure Databricks tables. We can query tables with Spark APIs and Spark SQL.

To access our data:

- We can import our files to DBFS using the UI.
- We can mount and use supported data sources via DBFS.

We can then use Spark or local APIs to access the data.

We will be able to use a DBFS file path in our notebook to access our data, independent of its data source.

It is possible to import existing data or code in the workspace.

If we use small data files on the local machine that we want to analyze with Azure Databricks, we can import them to DBFS using the UI. There are two ways to upload data to DBFS with the UI:

- Upload files to the FileStore in the Upload Data UI.
- Upload data to a table with the Create table UI, which is also accessible via the Import & Explore Data box on the landing page.

We may also read data on cluster nodes using Spark APIs. We can read data imported to DBFS into Apache Spark DataFrames. For example, if you import a CSV file, you can read the data using this code

```
df = spark.read.csv('/FileStore/tables/nyc_taxi.csv', header="true", inferSchema="true")
```

We can also read data imported to DBFS in programs running on the Spark driver node using local file APIs. For example:

```
df = spark.read.csv('/dbfs/FileStore/tables/nyc_taxi.csv', header="true", inferSchema="true")
```

Importing data

To add data, we can go to the landing page and select Import & Explore Data.

To get the data in a table, there are multiple options available:

- Upload a local file and import the data.
- Use data already existing under DBFS.
- Mount external data sources, like Azure Storage, Azure Data Lake and more.

To create a table based on a local file, we can select Upload File to upload data from your local machine.

Using DBFS mounted data

Databricks File System (DBFS) is a distributed file system mounted into a Databricks workspace and available on Databricks clusters. DBFS is an abstraction on top of scalable object storage and offers the following benefits:

- Allows you to mount storage objects so that you can seamlessly access data without requiring credentials.
- Allows you to interact with object storage using directory and file semantics instead of storage URLs.
- Persists files to object storage, so you won't lose data after you terminate a cluster.

The default storage location in DBFS is known as the DBFS root.

We can use the DBFS to access:

- Local files (previously imported). For example, the tables you imported above are available under `/FileStore`
- Remote files, objects kept in separate storages as if they were on the local file system

For example, to mount a remote Azure storage account as a DBFS folder, we can use the `dbutils` module:

Work with notebooks in Azure Databricks

Completed

100 XP

- 3 minutes

A notebook is a web-based interface to a document that contains:

- Runnable code
- Descriptive text
- Visualizations

A notebook is a collection of runnable cells (commands). When you use a notebook, you are primarily developing and running cells.

Runnable cells operate on files and tables. Cells can be run in sequence, referring to the output of previously run cells.

To create a notebook, we can select Workspace, browse into the desired folder, right-click, and choose Create, then select Notebook.

The supported magic commands are:

- `%python`
- `%r`
- `%scala`
- `%sql`

Notebooks also support a few auxiliary magic commands:

- `%sh`: Allows you to run shell code in your notebook
- `%fs`: Allows you to use dbutils filesystem commands
- `%md`: Allows you to include various types of documentation, including text, images, and mathematical formulas and equations.

Introduction-Working with Data Bricks

In Azure Databricks, data scientists use DataFrames to structure their data. A DataFrame is equivalent to a relational table in Spark SQL. In this module, you will learn what DataFrames are and how to use them.

Understand dataframes

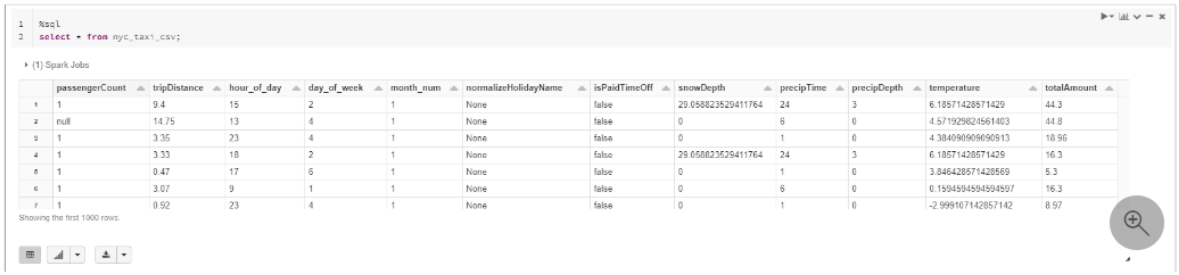
Spark uses 3 different APIs: RDDs, DataFrames, and DataSets. The architectural foundation is the resilient distributed dataset (RDD). The DataFrame API was released as an abstraction on top of the RDD, followed later by the Dataset API. We'll only use DataFrames in our notebook examples.

DataFrames are the distributed collections of data, organized into rows and columns. Each column in a DataFrame has a name and an associated type.

Spark DataFrames can be created from various sources, such as CSV files, JSON, Parquet files, Hive tables, log tables, and external databases.

Using Spark to load table data

Assuming we have this data available in a table:



We can use Spark to load the table data by using the `sql` method:

```python
df = spark.sql("SELECT * FROM nyc_taxi_csv")
```

Using Spark to load file/DBFS data

We can also read the data from the original files we've uploaded; or indeed from any other file available in the DBFS. The code is the same regardless of whether a file is local or in remote storage that was mounted, thanks to DBFS mountpoints.

Python Copy

```
df = spark.read.csv('dbfs:/FileStore/tables/nyc_taxi.csv', header=True, inferSchema=True)
```

Spark supports many different data formats, such as CSV, JSON, XML, Parquet, Avro, ORC and more.

DataFrame size

To get the number of rows available in a DataFrame, we can use the `count()` method.

Python

```
df.count
```

DataFrame structure

To get the schema metadata for a given DataFrame, we can use the `printSchema()` method.

Each column in a given DataFrame has a name, a type, and a nullable flag.

Python Copy

```
df.printSchema
```

DataFrame contents

Spark has a built-in function that allows to print the rows inside a DataFrame: `show()`

Python Copy

```
df.show
df.show(100, truncate=False) #show more lines, do not truncate
```

By default it will only show the first 20 lines in your DataFrame and it will truncate long columns. Additional parameters are available to override these settings.

Query dataframes

DataFrames allow the processing of huge amounts of data. Spark uses an optimization engine to generate logical queries. Data is distributed over your cluster and you get huge performance for massive amounts of data.

Spark SQL is a component that introduced the DataFrames, which provides support for structured and semi-structured data.

Spark has multiple interfaces (APIs) for dealing with DataFrames:

- We have seen the `.sql()` method, which allows to run arbitrary SQL queries on table data.
- Another option is to use the Spark domain-specific language for structured data manipulation, available in Scala, Java, Python, and R.

DataFrame API

The Apache Spark DataFrame API provides a rich set of functions (select columns, filter, join, aggregate, and so on) that allow you to solve common data analysis problems efficiently.

A complex operation where tables are joined, filtered, and restructured is easy to write, easy to understand, type safe, feels natural for people with prior sql experience, and comes with the added speed of parallel processing given by the Spark engine.

To load or save data use `read` and `write`:

Python Copy

```
df = spark.read.format('json').load('sample/trips.json')
df.write.format('parquet').bucketBy(100, 'year', 'month').mode("overwrite").saveAsTable('table1'))
```

To get the available data in a DataFrame use `select`:

Python Copy

```
df.select('*')
df.select('tripDistance', 'totalAmount')
```

To extract the first rows, use `take`:

Python Copy

```
df.take(15)
```

To order the data, use the `sort` method:

Python Copy

```
df.sort(df.tripDistance.desc())
```

To combine the rows in multiple DataFrames use `union`:

Python Copy

```
df1.union(df2)
```

This operation is equivalent to `UNION ALL` in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by `distinct()`.

The dataframes must have the same structure/schema.

To add or update columns use `withColumn` or `withColumnRenamed`:

Python Copy

```
df.withColumn('isHoliday', False)
df.withColumnRenamed('isDayOff', 'isHoliday')
```

To use aliases for the whole DataFrame or specific columns:

Python Copy

```
df.alias("myTrips")
df.select(df.passengerCount.alias("numberOfPassengers"))
```

To create a temporary view:

```python
df.createOrReplaceTempView("tripsView")
```

To aggregate on the entire DataFrame without groups use `agg`:

```python
df.agg({"age": "max"})
```

To do more complex queries, use `filter`, `groupBy` and `join`:

```python
people \
  .filter(people.age > 30) \
  .join(department, people.deptId == department.id) \
  .groupBy(department.name, "gender")
  .agg({"salary": "avg", "age": "max"})
```

These join types are supported: inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, and left_anti.

Note that `filter` is an alias for `where`.

To use columns aggregations using windows:

```python
w = Window.partitionBy("name").orderBy("age").rowsBetween(-1, 1)
df.select(rank().over(w), min('age').over(window))
```

To use a list of conditions for a column and return an expression use `when`:

```python
df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3, -1).otherwise(0)).show()
```

To check the presence of data use `isNull` or `isNotNull`:

```python
df.filter(df.passengerCount.isNotNull())
df.filter(df.totalAmount.isNull())
```

To clean the data use `dropna`, `fillna` or `dropDuplicates`:

Python Copy

```
df1.fillna(1) #replace nulls with specified value
df2.dropna #drop rows containing null values
df3.dropDuplicates #drop duplicate rows
```

To get statistics about the DataFrame use `summary` or `describe`:

Python Copy

```
df.summary().show()
df.summary("passengerCount", "min", "25%", "75%", "max").show()
df.describe(['age']).show()
```

Available statistics are:

- Count
- Mean
- Stddev
- Min
- Max
- Arbitrary approximate percentiles specified as a percentage (for example, 75%).

To find correlations between specific columns use `corr`. This operation currently only supports the Pearson Correlation Coefficient:

Python

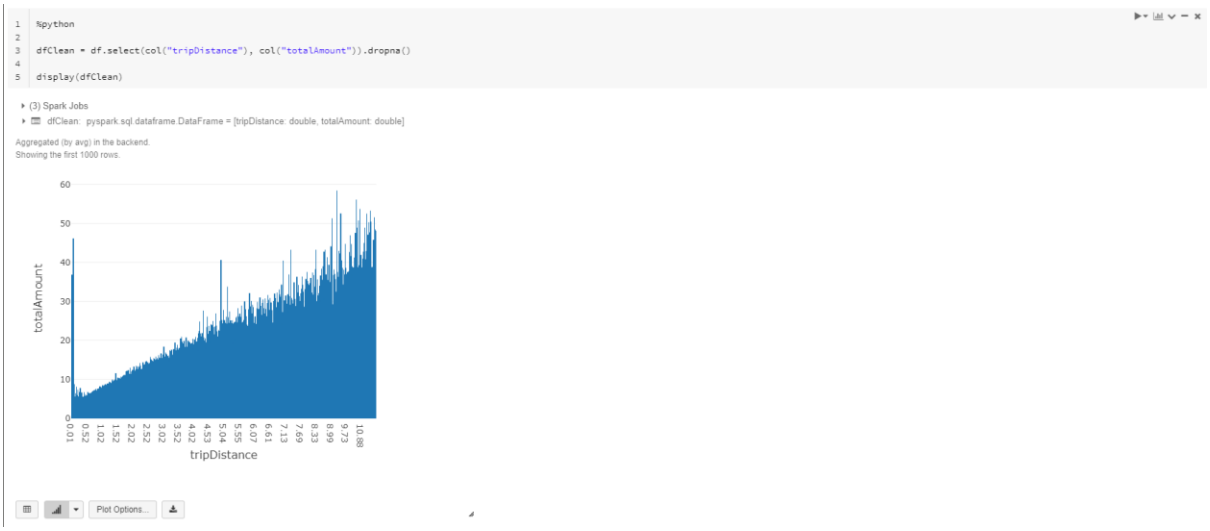
```
df.corr('tripDistance', 'totalAmount')
```

Visualize data

Spark has a built-in `show` function, which allows to print the rows in a DataFrame.

Azure Databricks adds its own display capabilities and adds various other types of visualizations out-of-the-box using the `display` and `displayHTML` functions.

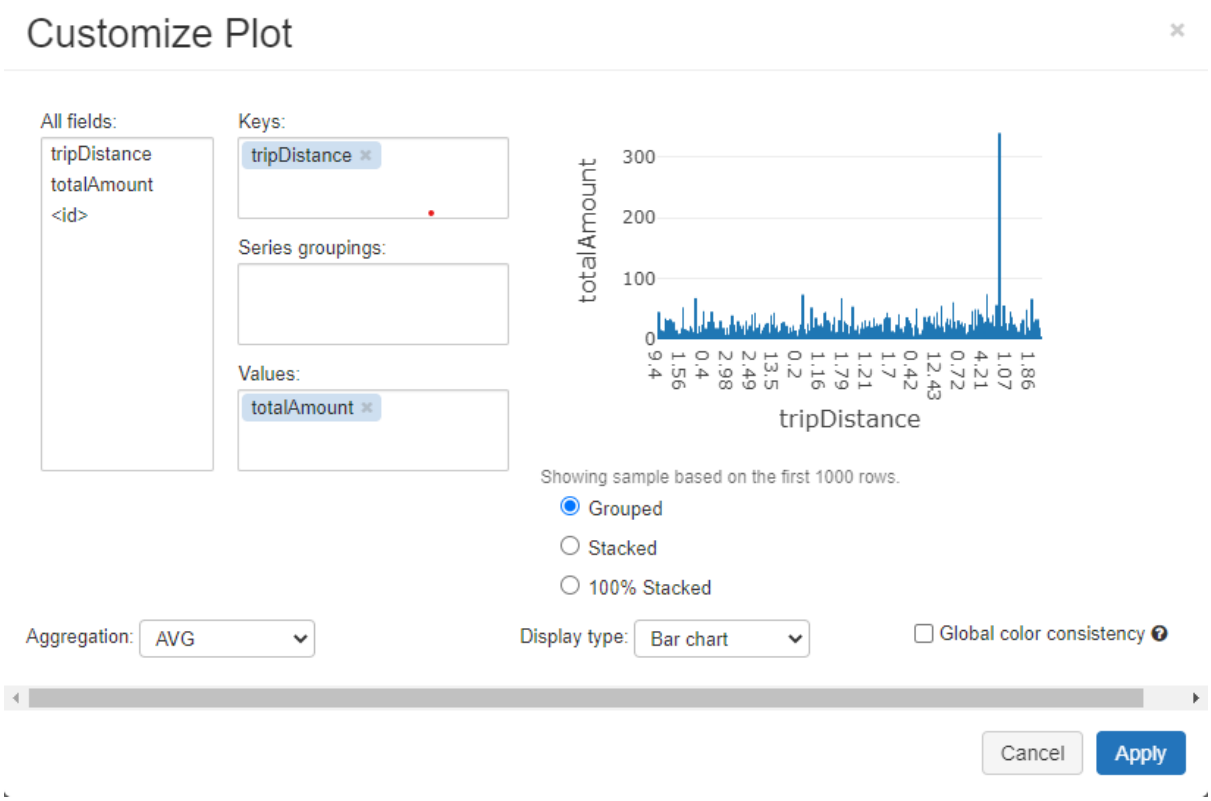
The same data we've seen previously as a table can be displayed as a bar chart, pie, histogram, or other graphs. Even maps or images can be displayed:



Plot options

The following display options are available:

- We can choose the DataFrame columns to be used as axes (keys, values).
- We can choose to group our series of data.
- We can choose the aggregations to be used with our grouped data (avg, sum, count, min, max).



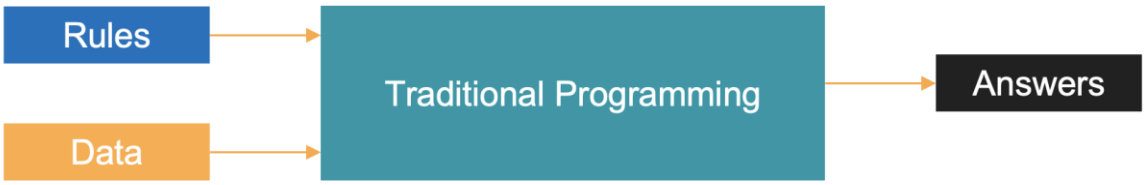
Prepare data for machine learning with Azure Databricks

Machine learning is a data science technique used to extract patterns from data allowing computers to identify related data, forecast future outcomes, behaviors, and trends.

Machine learning as the new programming paradigm

Traditional programming

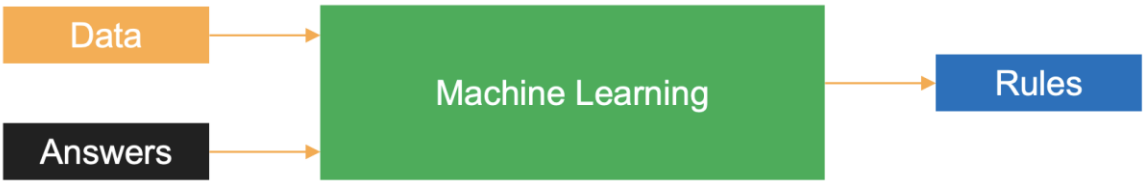
In traditional programming, the inputs of hard-coded rules and data are used to arrive at the output of answers.



You provide the traditional program with rules and data, and in return, it gives your results or answers.

Machine learning

The result of training a machine learning algorithm is that the algorithm has learned the rules to map the input data to answers.



In machine learning, you train the algorithm with data and answers, also known as labels, and the algorithm learns the rules to map the data to their respective labels.

Perform data cleaning

Big Data has become part of the lexicon of organizations worldwide, as more and more organizations look to leverage data to drive more informed business decisions. With this evolution in business decision-making, the amount of raw data collected, along with the number and diversity of data sources, is growing at an astounding rate.

Raw data, however, is often noisy and unreliable and may contain missing values and outliers. Using such data for Machine Learning can produce misleading results. Thus, data cleaning of the raw data is one of the most important steps in preparing data for Machine Learning. As Machine Learning algorithm learns the rules from data, having clean and consistent data is an important factor in influencing the predictive abilities of the underlying algorithms.

The most common type of data available for machine learning is in tabular format. The tabular data is typically available in the form of rows and columns. In tabular data, the row describes a single observation, and each column describes different properties of the observation. Column values can be continuous, discrete, datetime, or text. Columns that are chosen as inputs to the Machine Learning models are also known as model features.

Data cleaning deals with issues in the data quality such as errors, missing values and outliers. There are several techniques in dealing with data quality issues and we will discuss some of the common approaches below.

Imputation of null values

Null values refer to unknown or missing data. Strategies for dealing with this scenario include:

- Dropping these records: Works when you do not need to use the information for downstream workloads.
- Adding a placeholder (for example, -1): Allows you to see missing data later on without violating a schema.
- Basic imputing: Allows you to have a "best guess" of what the data could have been, often by using the *mean* or *median* of non-missing data for numerical data type, or *most_frequent* value of non-missing data for categorical data type.
- Advanced imputing: Determines the "best guess" of what data should be using more advanced strategies such as clustering machine learning algorithms or oversampling techniques such as SMOTE (Synthetic Minority Over-sampling Technique).

Converting data types

In some situations, the columns have inconsistent data types. For example, a column can have a combination of numbers presented as strings, like `"44.5"` and `"25.1"`. As part of data cleaning you often have to convert the data in the column to its correct data type.

Duplicate records

In some situations, you find duplicate records in the table. The easiest solution is to drop the duplicate records.

Outliers

An outlier is defined as an observation that is significantly different to all other observations in a given column. There are several ways to identify outliers, and one common approach is to compute the *Z-score* for an observation *x*.

You can use similar strategies as imputing null values to deal with outliers. However, it is important to note that outliers are not necessarily invalid data and, in some situations, it is perfectly valid to retain the outliers in your training data.

Perform feature engineering

Completed

100 XP

- 3 minutes

Machine learning models are as strong as the data they are trained on. Often it is important to derive features from existing raw data that better represent the nature of the data and thus help improve the predictive power of the machine learning algorithms. This process of generating new predictive features from existing raw data is commonly referred to as feature engineering.

Feature engineering

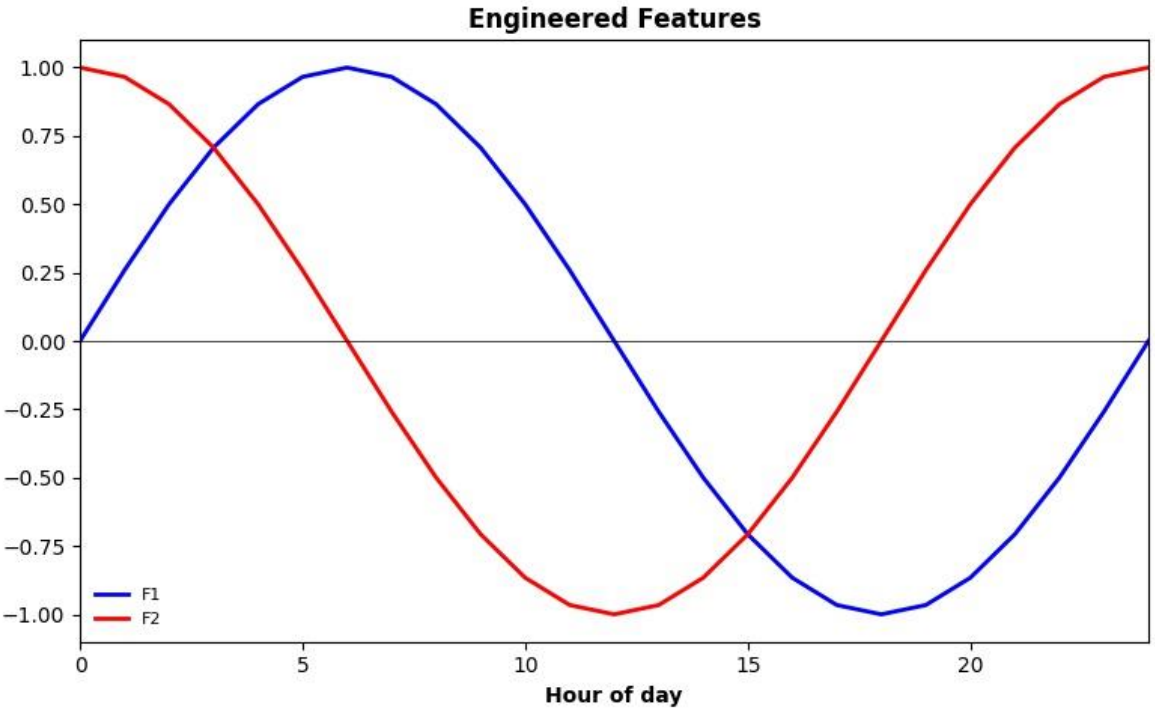
There are many valid approaches to feature engineering and some of the most popular ones, categorized by data type, are as follows:

- Aggregation (count, sum, average, mean, median, and the like)
- Part-of (year of date, month of date, week of date, and the like)
- Binning (grouping entities into bins and then applying aggregations)
- Flagging (boolean conditions resulting in True of False)
- Frequency-based (calculating the frequencies of the levels of one or more categorical variables)
- Embedding (transforming one or more categorical or text features into a new set of features, possibly with a different cardinality)
- Deriving by example

Feature engineering is not limited to the above list and can involve domain knowledge-based approaches for deriving features.

Let’s work with an example to understand the process of feature engineering. In our example, we are working with a system that gives us weather data on an hourly basis, and we have a column in the data that is `hour of day`. The `hour of day` column is of type integer and it can assume any integer value in the range `[0, 23]`.

The question is, how best to represent this data to a machine learning algorithm that can learn its cyclical nature? One approach is to engineer a set of new features that transforms the `hour of day` column using sine and cosine functions. These derived features are plotted in the figure below for the range `[0, 24]`:



The cosine function provides symmetrically equal weights to corresponding AM and PM hours, and the sine function provides symmetrically opposite weights to corresponding AM and PM hours. Both functions capture the cyclical nature of `hour of day`.

Perform data scaling

Scaling numerical features is an important part of preprocessing data for machine learning. Typically the range of values each input feature takes vary greatly between features. There are many machine learning algorithms that are sensitive to the magnitude of the input features and thus without feature scaling, higher weights might get assigned to features with higher magnitudes irrespective of the importance of the feature on the predicted output.

There are two common approaches to scaling numerical features:

- Normalization
- Standardization

We will discuss each of these approaches below.

Normalization

Normalization mathematically rescales the data into the range [0, 1].

For example, for each individual value, you can subtract the minimum value for that input in the training dataset, and then divide by the range of the values in the training dataset. The range of the values is the difference between the maximum value and the minimum value.

Standardization

Standardization rescales the data to have mean = 0 and standard deviation = 1.

For the numeric input, you first compute the mean and standard deviation using all the data available in the training dataset. Then, for each individual input value, you scale that value by subtracting the mean and then dividing by the standard deviation.

Perform data encoding

A common type of data that is prevalent in machine learning is called *categorical* data. Categorical data implies discrete or a limited set of values. For example, a person's gender or ethnicity is considered as categorical. Let's consider the following data table:

SKU	Make	Color	Quantity	Price
908721	G	Blue	789	45.33
456552	T	Red	244	22.91
789921	A	Green	387	25.92
872266	G	Blue	154	17.56

In the table above, the row describes a single observation, and each column describes different properties of the observation. In the table, we have two types of data, numeric data such as `Quantity` and `Price`, and categorical data such as `Make` and `Color`. In the previous lesson, we looked at examples of how to scale numeric data types. Furthermore, it is important to note that in machine learning, we ultimately always work with numbers or specifically, `vectors`. In this context, a vector is either an array of numbers, or nested arrays of numbers. So how does one encode categorical data for the purposes of machine learning? We will look at two common approaches for encoding categorical data:

- Ordinal encoding
- One-hot encoding

Ordinal encoding

Ordinal encoding, converts categorical data into integer codes ranging from 0 to (number of categories – 1). For example, the categories `Make` and `Color` from the above table are encoded as:

Make	Encoding
A	0
G	1
T	2

Color	Encoding
Red	0

Green	1
Blue	2

Using the above encoding, the transformed table is shown below:

SKU	Make	Color	Quantity	Price
908721	1	2	789	45.33
456552	2	0	244	22.91
789921	0	1	387	25.92
872266	1	2	154	17.56

One-hot encoding

One-hot encoding is often the recommended approach, and it involves transforming each categorical value into n (= number of categories) binary values, with one of them 1, and all others 0. For example, the above table can be transformed as:

SKU	A	G	T	Red	Green	Blue	Quantity	Price
908721	0	1	0	0	0	1	789	45.33

456552	0	0	1	1	0	0	244	22.91
789921	1	0	0	0	1	0	387	25.92
872266	0	1	0	0	0	1	154	17.56

One-hot encoding is often preferred over ordinal encoding because it encodes each category item with equal weight. In our above example, the ordinal encoder assigned color `Green = 1` and color `Blue = 2`, and that can imply that color Blue is twice as important as color Green. Whereas, with one-hot encoding each color is weighted equally.

Exercise - Prepare data for machine learning

Now, it's your chance to use Azure Databricks to prepare data for Machine Learning.

In this exercise, you will:

- Handling missing data.
- Feature Engineering.
- Scaling Numeric features.
- Encoding Categorical Features.

Understand Spark ML

Azure Databricks supports several libraries for machine learning. There's one key library, which has two approaches that are native to Apache Spark: MLLib and Spark ML.

MLLib

MLLib is a legacy approach for machine learning on Apache Spark. It builds off of Spark's Resilient Distributed Dataset (RDD) data structure. This data structure forms the foundation of Apache Spark, but additional data structures on top of the RDD, such as DataFrames, have reduced the need to work directly with RDDs.

As of Apache Spark 2.0, the library entered a maintenance mode. This means that MLLib is still available and has not been deprecated, but there will be no new functionality added to the library. Instead, customers are advised to move to the `org.apache.spark.ml` library, commonly referred to as Spark ML.

Spark ML

Spark ML is the primary library for machine learning development in Apache Spark. It supports DataFrames in its API, versus the classic RDD approach. This makes Spark ML an easier library to work with for data scientists, as Spark DataFrames share many common ideas with the DataFrames used in Pandas and R.

The most confusing part about MLLib versus Spark ML is that they are both the same library. The difference is that the "classic" MLLib namespace is `org.apache.spark.mllib` whereas the Spark ML namespace is `org.apache.spark.ml`. Whenever possible, use the Spark ML namespace when performing new data science activities.

Train and validate a model

The process of training and validating a machine learning model using Spark ML is fairly straightforward. The steps are as follows:

1. Splitting data.
2. Training a model.
3. Validating a model.

Splitting data

The first step involves splitting data between training and validation datasets. Doing so allows a data scientist to train a model with a representative portion of the data, while still retaining some percentage as a hold-out dataset. This hold-out dataset can be useful for determining whether the training model is overfitting - that is, latching onto the peculiarities of the training dataset rather than finding generally applicable relationships between variables.

DataFrames support a `randomSplit()` method, which makes this process of splitting data simple.

Training a model

Training a model relies on three key abstractions: a transformer, an estimator, and a pipeline.

A transformer takes a DataFrame as an input and returns a new DataFrame as an output. Transformers are helpful for performing feature engineering and feature selection, as the result of a transformer is another DataFrame. An example of this might be to read in a text column, map that text column into a set of feature vectors, and output a DataFrame with the newly mapped column. Transformers will implement a `.transform()` method.

An estimator takes a DataFrame as an input and returns a model. It takes a DataFrame as an input and returns a model, which is itself a transformer. An example of an estimator is the `LinearRegression` machine learning algorithm. It accepts a DataFrame and produces a `Model`. Estimators implement a `.fit()` method.

Pipelines combine together estimators and transformers and implement a `.fit()` method. By breaking out the training process into a series of stages, it's easier to combine multiple algorithms.

Validating a model

Once a model has been trained, it becomes possible to validate its results. Spark ML includes built-in summary statistics for models based on the algorithm of choice. Using linear regression for example, the model contains a `summary` object, which includes scores such as Root Mean Square Error (RMSE), Mean Absolute Error (MAE), and coefficient of determination (R^2, pronounced R-squared). These will be the summary measures based on the training data.

From there, with a validation dataset, it is possible to calculate summary statistics on a never-before-seen set of data, running the model's `transform()` function against the validation dataset. From there, use evaluators such as the `RegressionEvaluator` to calculate measures such as RMSE, MAE, and R^2.

Use other machine learning frameworks

Azure Databricks supports machine learning frameworks other than Spark ML and MLLib. For example, Azure Databricks offers support for popular libraries like TensorFlow and PyTorch.

It is possible to install these libraries directly, but the best recommendation is to use the Databricks Runtime for Machine Learning. This runtime comes with various machine learning libraries pre-installed, including TensorFlow, PyTorch, Keras, and XGBoost. It also includes libraries essential for distributed training, allowing data scientists to take advantage of the distributed nature of Apache Spark.

For libraries, which do not support distributed training, it is also possible to use a single node cluster. For example, PyTorch and TensorFlow both support single node use.

Understand capabilities of MLflow

MLflow is an open-source product designed to manage the Machine Learning development lifecycle. That is, MLflow allows data scientists to train models, register those models, deploy the models to a web server, and manage model updates.

The importance of MLflow

MLflow is an important part of machine learning with Azure Databricks, as it integrates key operational processes with the Azure Databricks interface. MLflow makes it easy for data scientists to train models and make them available without writing a great deal of code.

As a side note, MLflow will also operate on workloads outside of Azure Databricks. The examples in this module will all use Azure Databricks but this is not a requirement.

MLflow product components

There are four components to MLflow:

- MLflow Tracking
- MLflow Projects
- MLflow Models

- MLflow Model Registry

MLflow Tracking

MLflow Tracking allows data scientists to work with experiments. For each run in an experiment, a data scientist may log parameters, versions of libraries used, evaluation metrics, and generated output files when training machine learning models.

MLflow Tracking provides the ability to audit the results of prior model training executions.

▾ Parameters

Name	Value
alpha	0.01
l1_ratio	1.0

▾ Metrics

Name	Value
mae	51.05
r2	0.395
rmse	63.25

MLflow Projects

An MLflow Project is a way of packaging up code in a manner, which allows for consistent deployment and the ability to reproduce results. MLflow supports several environments for projects, including via Conda, Docker, and directly on a system.

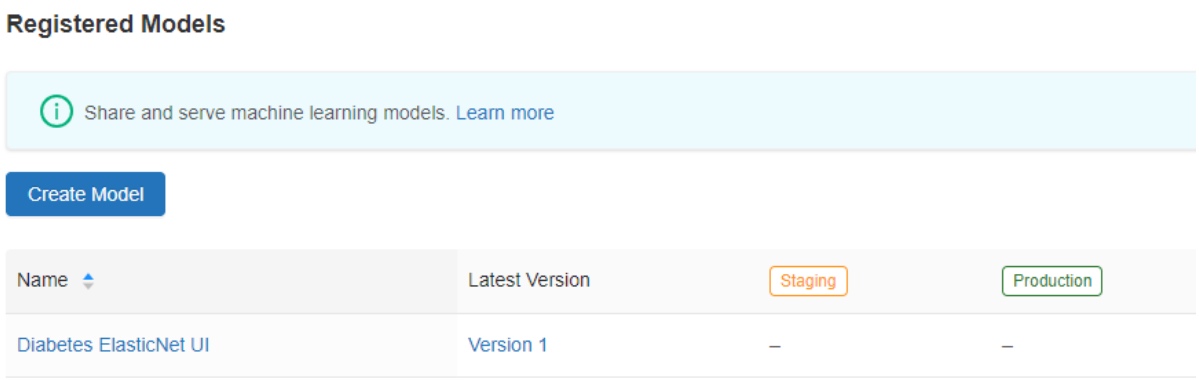
MLflow Models

MLflow offers a standardized format for packaging models for distribution. This standardized model format allows MLflow to work with models generated from

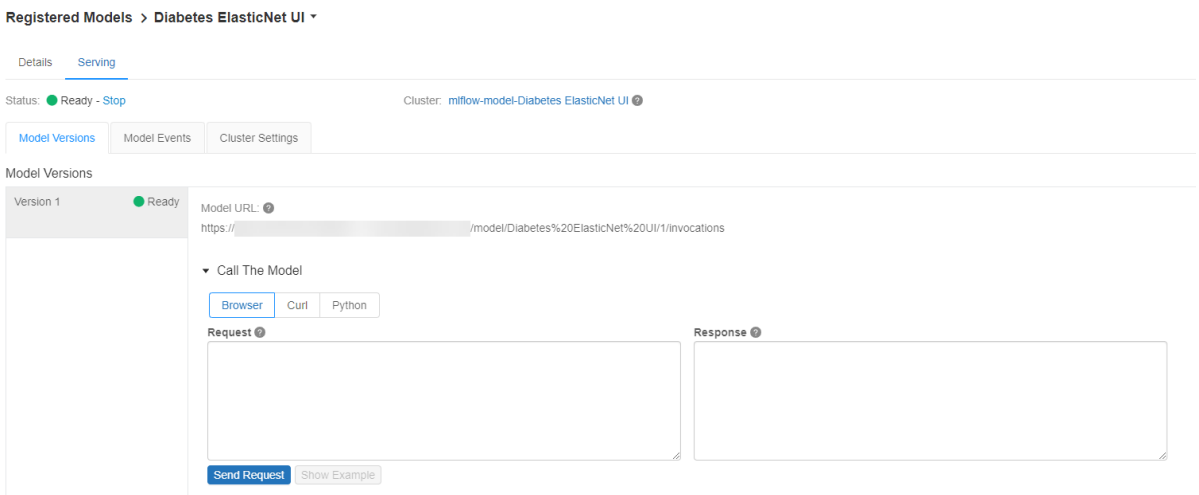
several popular libraries, including `scikit-learn`, `Keras`, `MLlib`, `ONNX`, and more. Review the MLflow Models documentation for information on the full set of supported model flavors.

MLflow Model Registry

The MLflow Model Registry allows data scientists to register models in a registry.



From there, MLflow Models and MLflow Projects combine with the MLflow Model Registry to allow operations team members to deploy models in the registry, serving them either through a REST API or as part of a batch inference solution using Azure Databricks.



Use MLflow terminology

There are several terms, which will be important to understand when working with MLflow. Most of these terms are fairly common in the data science space. Other products, such as Azure Machine Learning, use very similar terminology to allow for

simplified cross-product development of skills. The following sections include key terms and concepts for each MLflow product.

MLflow Tracking

MLflow Tracking is built around runs, that is, executions of code for a data science task. Each run contains several key attributes, including:

- Parameters: Key-value pairs, which represent inputs. Use parameters to track hyperparameters, that is, inputs to functions, which affect the machine learning process.
- Metrics: Key-value pairs, which represent how the model is performing. This can include evaluation measures such as Root Mean Square Error, and metrics can be updated throughout the course of a run. This allows a data scientist, for example, to track Root Mean Square Error for each epoch of a neural network.
- Artifacts: Output files. Artifacts may be stored in any format, and can include models, images, log files, data files, or anything else, which might be important for model analysis and understanding.

These runs can be combined together into experiments, which are intended to collect and organize runs. For example, a data scientist may create an experiment to train a classifier against a particular data set. Each run might try a different algorithm or different set of inputs. The data scientist can then review the individual runs in order to determine which run generated the best model.

MLflow Projects

A project in MLflow is a method of packaging data science code. This allows other data scientists or automated processes to use the code in a consistent manner.

Each project includes at least one entry point, which is a file (either .py or .sh) that is intended to act as the starting point for project use. Projects also specify details about the environment. This includes the specific packages (and versions of packages) used in developing the project, as new versions of packages may include breaking changes.

MLflow Models

A model in MLflow is a directory containing an arbitrary set of files along with an MLmodel file in the root of the directory.

MLflow allows models to be of a particular flavor, which is a descriptor of which tool or library generated a model. This allows MLflow to work with a wide variety of modeling libraries, such as `scikit-learn`, `Keras`, `MLlib`, `ONNX`, and many more. Each model has a signature, which describes the expected inputs and outputs for the model.

MLflow Model Registry

The MLflow Model Registry allows a data scientist to keep track of a model from MLflow Models. In other words, the data scientist registers a model with the Model Registry, storing details such as the name of the model. Each registered model may have multiple versions, which allow a data scientist to keep track of model changes over time.

It is also possible to stage models. Each model version may be in one stage, such as Staging, Production, or Archived. Data scientists and administrators may transition a model version from one stage to the next.

Describe considerations for model management

The two key steps for model management in MLflow are registration and versioning of models.

With registration, a data scientist stores the details of a model in the MLflow Model Registry, along with a name for ease of access. Users can retrieve the model from the registry and use that model to perform inference on new data sets. Next to that, it´s possible to serve models on Azure Databricks or in Azure Machine Learning, automatically generating a REST API to interact with the model.

Once a model is out in production, there is still more work to do. As models change over time, model management becomes a process of training new candidate models, comparing to the current version and prior candidate models, and determining whether a candidate is worthy of becoming the next production model. MLflow's versioning system makes model management easy by labeling new versions of models and retaining information on prior model versions automatically. MLflow allows a data scientist to perform testing on various model versions and ensure that new models are performing better than older models.

The importance of model registration

Model registration allows MLflow and Azure Databricks to keep track of models. This is important for two reasons. First, registering a model allows you to serve the model for real-time or batch scoring. This makes the process of using a trained model easy, as now data scientists will not need to develop application code; the serving process builds that wrapper and exposes a REST API or method for batch scoring automatically.

Second, registering a model allows you to create new versions of that model over time. This gives you the opportunity to track model changes and even perform comparisons between different historical versions of models. This helps answer a question of whether your model changes are significant - that is, newer models are definitely better than older models - or if the newer models are *chasing noise* and are not actually better than their predecessors.

Introduction-Azure Data Bricks experiment

If you choose to train models using Azure Databricks and track your work using MLflow, you can add an integration with Azure Machine Learning to store model training metrics and artifacts and keep a clear overview of your work. Using Azure Machine Learning as the backend for your MLflow experiments that run on Azure Databricks compute gives you the benefit of having a centralized and scalable workspace where you can access all your assets to run experiments or review them. In this module, you will learn about the integration between all these products and how you can manage your work from the Azure Machine Learning workspace.

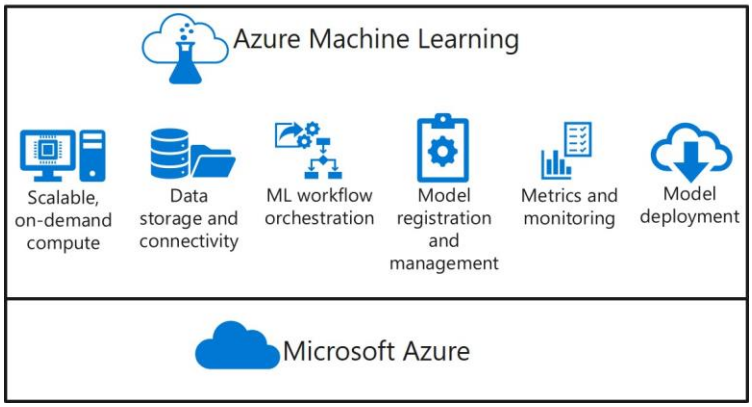
Learning objectives

After completing this module, you'll be able to:

- Describe Azure Machine Learning.
- Run an experiment.
- Log metrics with MLflow.
- Run Pipeline Step on Databricks Compute.

Describe Azure Machine Learning

Azure Machine Learning is a platform for operating machine learning workloads in the cloud.



Built on the Microsoft Azure cloud platform, Azure Machine Learning enables you to manage:

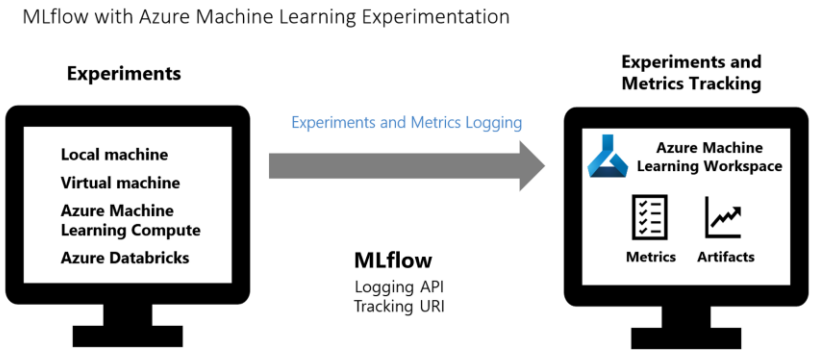
- Scalable on-demand compute for machine learning workloads.
- Data storage and connectivity to ingest data from a wide range of sources.
- Machine learning workflow orchestration to automate model training, deployment, and management processes.
- Model registration and management, so you can track multiple versions of models and the data on which they were trained.
- Metrics and monitoring for training experiments, datasets, and published services.
- Model deployment for real-time and batch inferencing.

Run Azure Databricks experiments in Azure Machine Learning

MLflow is an open-source library for managing the life cycle of your machine learning experiments. MLFlow Tracking is a component of MLflow that logs and tracks your training run metrics and model artifacts, no matter your experiment's environment.

A recommended approach for running Azure Machine Learning (AML) Experiments on Azure Databricks cluster is to use MLflow Tracking and connect Azure Machine Learning as the backend for MLflow experiments.

The following diagram illustrates that with MLflow Tracking, you track an experiment's run metrics and store model artifacts in your Azure Machine Learning workspace.



Track AML Experiments in Azure Databricks

When running AML experiments in Azure Databricks, there are three key steps:

1. Configure MLflow tracking URI to use AML.
2. Configure a MLflow experiment.
3. Run your experiment.

1. Configure MLflow tracking URI to use AML

In order to configure MLflow Tracking and connect Azure Machine Learning as the backend for MLFlow experiments, you need to follow these steps as shown in the code snippet:

- Get your AML workspace object.
- From your AML workspace object, get the unique tracking URI address.
- Setup MLflow tracking URI to point to AML workspace.

```python
import mlflow
from azureml.core import Workspace

# Get your AML workspace
ws = Workspace.from_config()

# Get the unique tracking URI address to the AML workspace
tracking_uri = ws.get_mlflow_tracking_uri()

# Set up MLflow tracking URI to point to AML workspace
mlflow.set_tracking_uri(tracking_uri)
```

2. Configure a MLflow experiment

Provide the name for the MLflow experiment as shown below. Note that the same experiment name will appear in Azure Machine Learning.

```python
experiment_name = 'MLflow-AML-Exercise'
mlflow.set_experiment(experiment_name)
```

3. Run your experiment

Once the experiment is set up, you can start your training run with `start_run()` as shown below:

```python
with mlflow.start_run() as run:
    ...
    ...
```

Log metrics in Azure Machine Learning with MLflow

In the previous unit, we discussed how to set up Azure Machine Learning as the backend for MLflow experiments. We also looked at how to start your model training on Azure Databricks as a MLflow experiment. In this section, we will look at how to log model metrics and artifacts to the MLflow logging API. These logged metrics and artifacts are then captured in an Azure Machine Learning workspace that provides a centralized, secure, and scalable location to store training metrics and artifacts.

In your MLflow experiment, once you train and evaluate your model, you can use the MLflow logging API, `mlflow.log_metric()`, to start logging your model metrics as shown below:

Python

```python
with mlflow.start_run() as run:
    ...
    ...
    # Make predictions on hold-out data
    y_predict = clf.predict(X_test)
    y_actual = y_test.values.flatten().tolist()

    # Evaluate and log model metrics on hold-out data
    rmse = math.sqrt(mean_squared_error(y_actual, y_predict))
    mlflow.log_metric('rmse', rmse)
    mae = mean_absolute_error(y_actual, y_predict)
    mlflow.log_metric('mae', mae)
    r2 = r2_score(y_actual, y_predict)
    mlflow.log_metric('R2 score', r2)
```

Next, you can use MLflow's `log_artifact()` API to save model artifacts such as your `Predicted vs True` curve as shown:

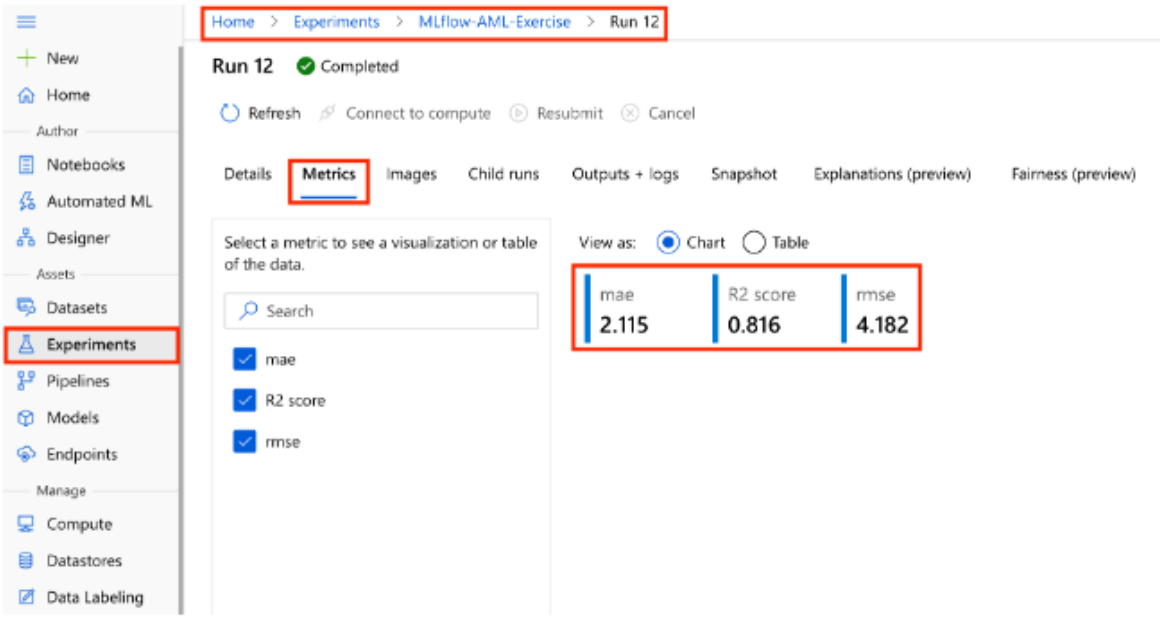
Python

```python
import matplotlib.pyplot as plt

with mlflow.start_run() as run:
    ...
    ...
    plt.scatter(y_actual, y_predict)
    plt.savefig("./outputs/results.png")
    mlflow.log_artifact("./outputs/results.png")
```

Reviewing experiment metrics and artifacts in Azure ML Studio

Since Azure Machine Learning is set up as the backend for MLflow experiments, you can review all the training metrics and artifacts from within the Azure Machine

Learning Studio. From within the studio, navigate to the `Experiments` tab, and open the experiment run that corresponds to the MLflow experiment. In the `Metrics` tab of the run, you will observe the model metrics that were logged via MLflow tracking APIs.



Next, when you open the `Outputs + logs` tab you will observe the model artifacts that were logged via MLflow tracking APIs.

In summary, using MLflow integration with Azure Machine Learning, you can run experiments in Azure Databricks and leverage Azure Machine Learning workspace capabilities of centralized, secure, and scalable solution to store model training metrics and artifacts.

Run Azure Machine Learning pipelines on Azure Databricks compute

Azure Machine Learning supports multiple types of compute for experimentation and training. Specifically, you can run an Azure Machine Learning pipeline on Databricks compute.

What is an Azure Machine Learning pipeline?

In Azure Machine Learning, a pipeline is a workflow of machine learning tasks in which each task is implemented as a step. Steps can be arranged sequentially or in parallel, enabling you to build sophisticated flow logic to orchestrate machine learning operations. Each step can be run on a specific compute target, making it possible to combine different types of processing as required to achieve an overall goal.

Running pipeline step on Databricks Compute

Azure Machine Learning supports a specialized pipeline step called DatabricksStep with which you can run a notebook, script, or compiled JAR on an Azure Databricks cluster. In order to run a pipeline step on a Databricks cluster, you need to do the following steps:

a. Attach Azure Databricks Compute to Azure Machine Learning workspace.
b. Define DatabricksStep in a pipeline.
c. Submit the pipeline.

Attaching Azure Databricks Compute

The following code example can be used to attach an existing Azure Databricks cluster:

```python
from azureml.core import Workspace
from azureml.core.compute import ComputeTarget, DatabricksCompute

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'db_cluster'

# Define configuration for existing Azure Databricks cluster
db_workspace_name = 'db_workspace'
db_resource_group = 'db_resource_group'
# Get the access token from the Databricks workspace
db_access_token = '1234-abc-5678-defg-90...'
db_config = DatabricksCompute.attach_configuration(resource_group=db_resource_group,
                                                   workspace_name=db_workspace_name,
                                                   access_token=db_access_token)

# Create the compute
databricks_compute = ComputeTarget.attach(ws, compute_name, db_config)
databricks_compute.wait_for_completion(True)
```

Defining DatabricksStep in a pipeline

To create a pipeline, you must first define each step and then create a pipeline that includes the steps. The specific configuration of each step depends on the step type. For example, the following code defines a DatabricksStep step to run a python script, `process_data.py`, on the attached Databricks compute.

```python
from azureml.pipeline.core import Pipeline
from azureml.pipeline.steps import DatabricksStep

script_directory = "./scripts"
script_name = "process_data.py"

dataset_name = "nyc-taxi-dataset"

spark_conf = {"spark.databricks.delta.preview.enabled": "true"}

databricksStep = DatabricksStep(name = "process_data",
                                run_name = "process_data",
                                python_script_params=["--dataset_name", dataset_name],
                                spark_version = "7.3.x-scala2.12",
                                node_type = "Standard_DS3_v2",
                                spark_conf = spark_conf,
                                num_workers = 1,
                                python_script_name = script_name,
                                source_directory = script_directory,
                                pypi_libraries = [PyPiLibrary(package = 'scikit-learn'),
                                                  PyPiLibrary(package = 'scipy'),
                                                  PyPiLibrary(package = 'azureml-sdk'),
                                                  PyPiLibrary(package = 'azureml-dataprep[pandas]')],
                                compute_target = databricks_compute,
                                allow_reuse = False
                               )
```

The above step defines the configuration to create a new Databricks job cluster to run the Python script. The cluster is created on the fly to run the script and the cluster is subsequently deleted after the step execution is completed.

Submit the pipeline

After defining the step, you can assign it to a pipeline, and run it as an experiment

```python
from azureml.pipeline.core import Pipeline
from azureml.core import Experiment

# Construct the pipeline
pipeline = Pipeline(workspace = ws, steps = [databricksStep])

# Create an experiment and run the pipeline
experiment = Experiment(workspace = ws, name = "process-data-pipeline")
pipeline_run = experiment.submit(pipeline)
```

Introduction

After training your model on Azure Databricks Compute, you may want to deploy your model so that it can be consumed by your business or end user. You can easily deploy your model by using Azure Machine Learning. In this module, you will learn how to deploy models using Azure Databricks and Azure Machine Learning.

Learning objectives

After completing this module, you'll be able to:

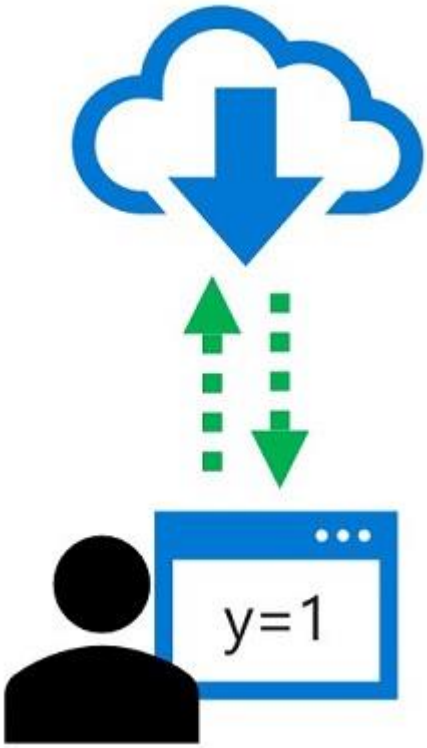
- Describe considerations for model deployment.
- Plan for deployment endpoints.
- Deploy a model as an inferencing webservice.
- Troubleshoot model deployment.

Describe considerations for model deployment

In machine learning, Model Deployment can be considered as a process by which you integrate your trained machine learning models into a production environment such that your business or end-user applications can use the model predictions to make decisions or gain insights into your data. The most common way you deploy a model using Azure Machine Learning from Azure Databricks, is to deploy the model as a real-time inferencing service. Here the term inferencing refers to the use of a trained model to make predictions on new input data on which the model has not been trained.

What is Real-Time Inferencing?

The model is deployed as part of a service that enables applications to request immediate, or *real-time*, predictions for individual, or small numbers of data observations.



In Azure Machine learning, you can create real-time inferencing solutions by deploying a model as a real-time service, hosted in a containerized platform such as Azure Kubernetes Services (AKS).

Plan for Azure Machine Learning deployment endpoints

After you have trained your machine learning model and evaluated it to the point where you are ready to use it outside your own development or test environment, you need to deploy it somewhere. Azure Machine Learning service simplifies this process. You can use the service components and tools to register your model and deploy it to one of the available compute targets so it can be made available as a web service in the Azure cloud, or on an IoT Edge device.

Available compute targets

You can use the following compute targets to host your web service deployment:

Compute target	Usage	Description
Local web service	Testing/debug	Good for limited testing and troubleshooting.
Azure Kubernetes Service (AKS)	Real-time inference	Good for high-scale production deployments. Provides autoscaling, and fast response times.
Azure Container Instances (ACI)	Testing	Good for low scale, CPU-based workloads.
Azure Machine Learning Compute Clusters	Batch inference	Run batch scoring on serverless compute. Supports normal and low-priority VMs.
Azure IoT Edge	(Preview) IoT module	Deploy & serve ML models on IoT devices.

Deploy a model to Azure Machine Learning

Completed

100 XP

- 3 minutes

As we discussed in the previous unit, you can deploy a model to several kinds of compute target: including local compute, an Azure Container Instance (ACI), an Azure Kubernetes Service (AKS) cluster, or an Internet of Things (IoT) module. Azure Machine Learning uses containers as a deployment mechanism, packaging the model and the code to use it as an image that can be deployed to a container in your chosen compute target.

To deploy a model as an inferencing webservice, you must perform the following tasks:

1. Register a trained model.
2. Define an Inference Configuration.
3. Define a Deployment Configuration.
4. Deploy the Model.

1. Register a trained model

After successfully training a model, you must register it in your Azure Machine Learning workspace. Your real-time service will then be able to load the model when required.

To register a model from a local file, you can use the register method of the Model object as shown here:

Python

```
from azureml.core import Model

model = Model.register(workspace=ws,
                       model_name='nyc-taxi-fare',
                       model_path='model.pkl', # local path
                       description='Model to predict taxi fares in NYC.')
```

2. Define an Inference Configuration

The model will be deployed as a service that consists of:

- A script to load the model and return predictions for submitted data.
- An environment in which the script will be run.

You must therefore define the script and environment for the service.

Creating an Entry Script

Create the *entry script* (sometimes referred to as the *scoring script*) for the service as a Python (.py) file. It must include two functions:

- init(): Called when the service is initialized.
- run(raw_data): Called when new data is submitted to the service.

Typically, you use the init function to load the model from the model registry, and use the run function to generate predictions from the input data. The following example script shows this pattern:

Python

```
import json
import joblib
import numpy as np
from azureml.core.model import Model

# Called when the service is loaded
def init():
    global model
    # Get the path to the registered model file and load it
    model_path = Model.get_model_path('nyc-taxi-fare')
    model = joblib.load(model_path)

# Called when a request is received
def run(raw_data):
    # Get the input data as a numpy array
    data = np.array(json.loads(raw_data)['data'])
    # Get a prediction from the model
    predictions = model.predict(data)
    # Return the predictions as any JSON serializable format
    return predictions.tolist()
```

Creating an Environment

Azure Machine Learning environments are an encapsulation of the environment where your machine learning training happens. They define Python packages, environment variables, Docker settings and other attributes in declarative fashion. The below code snippet shows an example of how you can create an environment for your deployment:

```python
from azureml.core import Environment
from azureml.core.environment import CondaDependencies

my_env_name="nyc-taxi-env"
myenv = Environment.get(workspace=ws, name='AzureML-Minimal').clone(my_env_name)
conda_dep = CondaDependencies()
conda_dep.add_pip_package("numpy==1.18.1")
conda_dep.add_pip_package("pandas==1.1.5")
conda_dep.add_pip_package("joblib==0.14.1")
conda_dep.add_pip_package("scikit-learn==0.24.1")
conda_dep.add_pip_package("sklearn-pandas==2.1.0")
myenv.python.conda_dependencies=conda_dep
```

Combining the Script and Environment in an InferenceConfig

After creating the entry script and environment, you can combine them in an InferenceConfig for the service like this:

```python
from azureml.core.model import InferenceConfig

from azureml.core.model import InferenceConfig
inference_config = InferenceConfig(entry_script='score.py',
                                   source_directory='.',
                                   environment=myenv)
```

3. Define a Deployment Configuration

Now that you have the entry script and environment, you need to configure the compute to which the service will be deployed. If you are deploying to an AKS cluster, you must create the cluster and a compute target for it before deploying:

```
Python
from azureml.core.compute import ComputeTarget, AksCompute

cluster_name = 'aks-cluster'
compute_config = AksCompute.provisioning_configuration(location='eastus')
production_cluster = ComputeTarget.create(ws, cluster_name, compute_config)
production_cluster.wait_for_completion(show_output=True)
```

With the compute target created, you can now define the deployment configuration, which sets the target-specific compute specification for the containerized deployment:

```
Python
from azureml.core.webservice import AksWebservice

deploy_config = AksWebservice.deploy_configuration(cpu_cores = 1,
                                                   memory_gb = 1)
```

The code to configure an ACI deployment is similar, except that you do not need to explicitly create an ACI compute target, and you must use the deploy_configuration class from the azureml.core.webservice.AciWebservice namespace. Similarly, you can use the azureml.core.webservice.LocalWebservice namespace to configure a local Docker-based service.

4. Deploy the Model

After all of the configuration is prepared, you can deploy the model. The easiest way to do this is to call the deploy method of the Model class, like this:

```python
from azureml.core.model import Model

service = Model.deploy(workspace=ws,
                       name = 'nyc-taxi-service',
                       models = [model],
                       inference_config = inference_config,
                       deployment_config = deploy_config,
                       deployment_target = production_cluster)
service.wait_for_deployment(show_output = True)
```

For ACI or local services, you can omit the deployment_target parameter (or set it to None).

Note

More Information: For more information about deploying models with Azure Machine Learning, see Deploy models with Azure Machine Learning in the documentation.

Troubleshoot model deployment

Completed

100 XP

- 3 minutes

There are a lot of elements to a service deployment, including the trained model, the runtime environment configuration, the scoring script, the container image, and the container host. Troubleshooting a failed deployment, or an error when consuming a deployed service can be complex.

Check the service state

As an initial troubleshooting step, you can check the status of a service by examining its state:

```python
from azureml.core.webservice import AksWebservice

# Get the deployed service
service = AksWebservice(name='classifier-service', workspace=ws)

# Check its state
print(service.state)
```

Review service logs

If a service is not healthy, or you are experiencing errors when using it, you can review its logs:

```python
print(service.get_logs())
```

The logs include detailed information about the provisioning of the service, and the requests it has processed; and can often provide an insight into the cause of unexpected errors.

Deploy to a local container

Deployment and runtime errors can be easier to diagnose by deploying the service as a container in a local Docker instance, like this:

```python
from azureml.core.webservice import LocalWebservice

deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, 'test-svc', [model], inference_config, deployment_config)
```

You can then test the locally deployed service using the SDK:

```python
print(service.run(input_data = json_data))
```

You can then troubleshoot runtime issues by making changes to the scoring file that is referenced in the inference configuration, and reloading the service without redeploying it (something you can only do with a local service):

```python
service.reload()
print(service.run(input_data = json_data))
```

Tune hyperparameters with Azure Databricks

AdvancedData ScientistDatabricks

In Azure Databricks, you can automate the process of hyperparameter tuning to more easily identify the best model.

Learning objectives

After completing this module, you will be able to:

- Understand hyperparameter tuning and its role in machine learning.

- Learn how to use the two open-source tools - automated MLflow and Hyperopt - to automate the process of model selection and hyperparameter tuning.

Introduction

Hyperparameters are parameters defined before model training that can influence the model's performance. There are different hyperparameters available to fine-tune depending on the algorithm used to train a model, which can be done through a process called hyperparameter tuning.

In this module, you'll learn how to use Azure Databricks with MLflow to do hyperparameter tuning and model selection.

Learning objectives

After completing this module, you'll be able to:

- Understand hyperparameter tuning and its role in machine learning.
- Learn how to use the two open-source tools - automated MLflow and Hyperopt - to automate the process of model selection and hyperparameter tuning.

Understand hyperparameter tuning

Building machine learning solutions involves testing many different models. Let's explore two concepts that can help with finding the optimal model:

- Hyperparameter tuning
- Cross-validation

Hyperparameter tuning

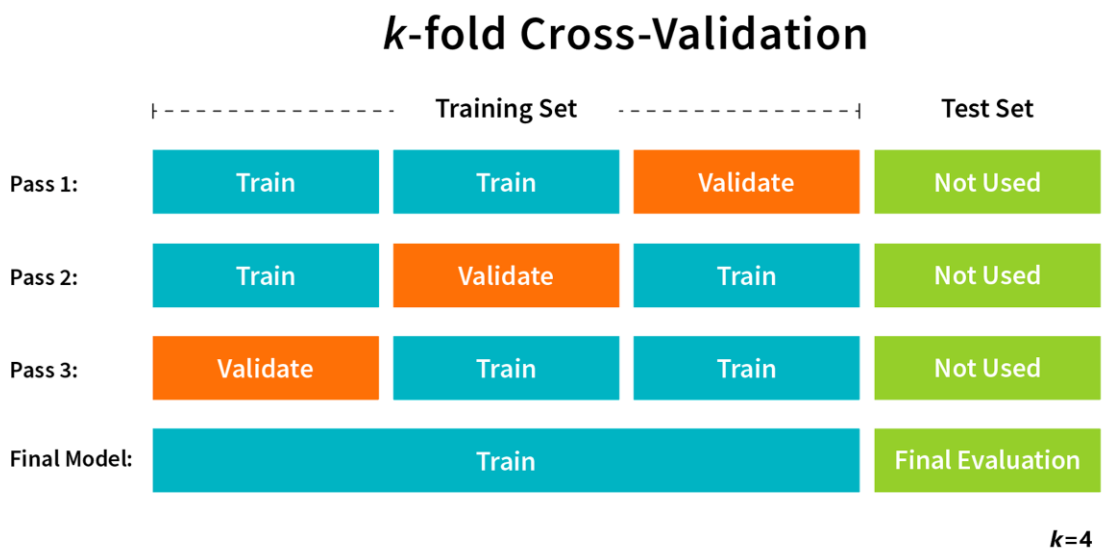
A hyperparameter is a parameter used in a machine learning algorithm that is set before the learning process begins. In other words, a machine learning algorithm can't learn hyperparameters from the data itself. Hyperparameters are tested and validated by training multiple models. Common hyperparameters include the number of iterations and the complexity of the model. Hyperparameter tuning is the process

of choosing the hyperparameter that has the best result on our loss function, or the way we penalize an algorithm for being wrong.

Cross-validation

When you train and evaluate a model on the same data, it can lead to overfitting. Overfitting is where the model performs well on data it has already seen but fails to predict anything useful on data it has not already seen. To avoid overfitting, you can use the train/test split where the dataset is divided between a training set used to train the model and a test set to evaluate the model's performance on unseen data.

If you train many different models with different hyperparameters and then evaluate their performance on the test set, you would still risk overfitting because you may choose the hyperparameter that just so happens to perform the best on the data you have in your dataset. To solve overfitting when using hyperparameters, you can use *k* subsets of your training set to train the model, a process called *k*-fold cross-validation. A model is then trained on k-1 folds of the training data and the last fold is used to evaluate its performance.



Within Azure Databricks, there are two approaches to tune hyperparameters, which will be discussed in the next units:

- Automated MLflow tracking.
- Hyperparameter tuning with Hyperopt.

Automated MLflow for model tuning

To choose the best model trained during hyperparameter tuning, you want to compare all models by evaluating their metrics. One common and simple approach to track model training in Azure Databricks is by using the open-source platform MLflow.

Use automated MLflow

As you train multiple models with hyperparameter tuning, you want to avoid the need to make explicit API calls to log all necessary information about the different models to MLflow. To make tracking hyperparameter tuning easier, the Databricks Runtime for Machine Learning also supports *automated* MLflow Tracking. When you use automated MLflow for model tuning, the hyperparameter values and evaluation metrics are automatically logged in MLflow and a hierarchy will be created for the different runs that represent the distinct models you train.

To use automated MLflow tracking, you have to do the following:

- Use a Python notebook to host your code.
- Attach the notebook to a cluster with Databricks Runtime or Databricks Runtime for Machine Learning.
- Set up the hyperparameter tuning with `CrossValidator` or `TrainValidationSplit`.

MLflow will automatically create a main or parent run that contains the information for the method you chose: `CrossValidator` or `TrainValidationSplit`. MLflow will also create child runs that are nested under the main or parent run. Each child run will represent a trained model and you can see which hyperparameter values were used and the resulting evaluation metrics.

Run tuning code

When you want to run code that will train multiple models with different hyperparameter settings, you can go through the following steps:

- List the available hyperparameters for a specific algorithm.
- Set up the search space and sampling method.

- Run the code with automated MLflow, using `CrossValidator` or `TrainValidationSplit`.

List the available hyperparameters

You can explore the hyperparameters of a specific machine learning algorithm by using the `.explainParams()` method on a model. For example, if we want to train a linear regression model `lr`, we can use the following command to view the available hyperparameters:

Python

```
print(lr.explainParams())
```

The `.explainParams()` method will return a list of hyperparameters you can choose from, including the name of the hyperparameter, a description, and the default value. Three of the hyperparamaters available for the linear regression model are:

- maxIter: max number of iterations (>= 0). (default: 100)
- fitIntercept: whether to fit an intercept term. (default: True)
- standardization: whether to standardize the training features before fitting the model. (default: True)

Set up the search space and sampling method

After you select the hyperparameters, you can use `ParamGridBuilder()` to specify the search space. The search space is the range of values of the hyperparameters you want to try out. You can then specify how you want to choose values from that search space to train individual models with which is known as the sampling method. The most straight-forward sampling method is known as grid sampling. The grid sampling method tries all possible combinations of values for the hyperparameters listed.

By default, the individual models will be trained in serial. It is possible to train models with different hyperparamater values in parallel. You can find more information on setting up the parameter grid in the documentation here.

Note

Since grid search works through exhaustively building a model for each combination of hyperparameters, it quickly becomes a lot of different unique combinations. As each model training can consume a lot of compute power, be careful with the configuration you set up.

If we continue the example with the linear regression model `lr`, the following code shows how to set up a grid search to try out all possible combinations of parameters:

```python
from pyspark.ml.tuning import ParamGridBuilder

paramGrid = (ParamGridBuilder()
  .addGrid(lr.maxIter, [1, 10, 100])
  .addGrid(lr.fitIntercept, [True, False])
  .addGrid(lr.standardization, [True, False])
  .build()
)
```

Run code and invoke automated MLflow

To test how the model performs and to generate evaluation metrics, you can use a test dataset. If you want to train multiple models on the same training dataset and the same test dataset, you can use the `TrainValidationSplit` method to run your code, build the models, and log them automatically with MLflow.

In case you want to take extra measures to prevent overfitting, you can use the `CrossValidator` method to train the models with different training datasets for each model and different test datasets to calculate the evaluation metrics.

To build the models for the linear regression model `lr` used in the examples above, you can create a `RegressionEvaluator()` to evaluate the grid search experiments, which will help decide which model is best. The settings for the hyperparameter tuning experiment can be set by using the `CrossValidator()` method as is done in the example below.

Python

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator

evaluator = RegressionEvaluator(
  labelCol = "medv",
  predictionCol = "prediction"
)

cv = CrossValidator(
  estimator = pipeline,             # Estimator (individual model or pipeline)
  estimatorParamMaps = paramGrid,   # Grid of parameters to try (grid search)
  evaluator=evaluator,              # Evaluator
  numFolds = 3,                     # Set k to 3
  seed = 42                         # Seed to sure our results are the same if ran again
)

cvModel = cv.fit(trainDF)
```

Once all models have been trained, you can get the best model with the following code:

Python Copy

```
bestModel = cvModel.bestModel
```

Alternatively, you can look at all models you trained through the UI of MLflow. Just remember that there will be a parent run for the complete experiment and child runs for each individual model that has been trained.

Hyperparameter tuning with Hyperopt

Another open-source tool that allows you to automate the process of hyperparameter tuning and model selection is Hyperopt. Hyperopt is simple to use, but using it efficiently requires care. The main advantage to using Hyperopt is that it is flexible and it can optimize any Python model with hyperparameters.

Use Hyperopt

Hyperopt is already installed if you create a compute with the Databricks Runtime ML. To use it when training a Python model, you should follow these basic steps:

1. Define an objective function to minimize.
2. Define the hyperparameter search space.
3. Specify the search algorithm.

4. Run the Hyperopt function fmin().

Define an objective function to minimize

The objective function represents what the main purpose is of training multiple models through hyperparameter tuning. Often, the objective is to minimize training or validation loss.

When defining a function, you can make use of any evaluation metric that can be calculated with the algorithm you selected. For example, if we use a support vector machine classifier from the scikit-learn library, you can vary the value for the regularization parameter `C`. The objective is to have the model with the highest accuracy. Since Hyperopt wants a function that it needs to minimize, you can define the objective function as the negative accuracy so that a lower score actually means a higher accuracy.

In the following example, the regularization parameter `C` is defined as the input, a support vector machine classifier model is trained, the accuracy is calculated, and the objective function is defined as the negative accuracy, which is the value Hyperopt will try to minimize.

Python

```
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

def objective(C):
    clf = SVC(C)

    accuracy = cross_val_score(clf, X, y).mean()

    return {'loss': -accuracy, 'status': STATUS_OK}
```

Define the hyperparameter search space

When tuning hyperparameters, you need to define a search space. If you want to make use of Hyperopt's Bayesian approach to sampling, there is a set of expressions you can use to define the search space that is compatible with Hyperopt's approach to sampling.

Some examples of the expressions used to define the search space are:

- `hp.choice(label, options)`: Returns one of the `options` you listed.
- `hp.randint(label, upper)`: Returns a random integer in the range [0, upper].
- `hp.uniform(label, low, high)`: Returns a value uniformly between `low` and `high`.
- `hp.normal(label, mu, sigma)`: Returns a real value that's normally distributed with mean `mu` and standard deviation `sigma`.

For the complete list of expressions, see the Hyperopt documentation.

Select the search algorithm

There are two main choices in how Hyperopt will sample over the search space:

- `hyperopt.tpe.suggest`: Tree of Parzen Estimators (TPE), a Bayesian approach, which iteratively and adaptively selects new hyperparameter settings to explore based on past results.
- `hyperopt.rand.suggest`: Random search, a non-adaptive approach that samples over the search space.

Run the Hyperopt function fmin()

Finally, to execute a Hyperopt run, you can use the function `fmin()`. The `fmin()` function takes the following arguments:

- `fn`: The objective function.
- `space`: The search space.
- `algo`: The search algorithm you want Hyperopt to use.
- `max_evals`: The maximum number of models to train.
- `max_queue_len`: The number of hyperparameter settings generated ahead of time. This can save time when using the TPE algorithm.
- `trials`: A `SparkTrials` or `Trials` object. `SparkTrials` is used for single-machine algorithms such as scikit-learn. `Trials` is used for distributed training algorithms such as MLlib methods or Horovod. When using `SparkTrials` or Horovod, automated MLflow tracking is enabled and hyperparameters and evaluation metrics are automatically logged in MLflow.

For more information on how to configure `fmin()` and `SparkTrials`, read about the Hyperopt concepts here.

Understand Horovod

Horovod can help data scientists when training *deep learning* models. Before we can explore Horovod, let's review what deep learning is and what the problem with training deep learning models can be.

A quick review of deep learning

Deep learning is a subfield of machine learning and refers to models that consist of multiple layers, also known as deep neural networks. The training process starts with data being submitted to the input layer in batches.

The data is analyzed by the input layer and passed on to the next layer until it reaches the output layer and produces a prediction. Predictions are compared to the actual known value and based on these results, weights, and bias values are revised to improve the model.

Batches are processed by the network over multiple iterations or epochs. Each epoch, the model tries to further improve predictions by updating the weight and bias values.

Deep learning with Azure Databricks

Within Azure Databricks, we can train deep learning models using the popular open-source frameworks for Python: TensorFlow, PyTorch, and Keras. When we use any of these single-node frameworks to train a deep learning model, we should use a *single-node* cluster in Azure Databricks.

Deep learning models benefit from having more data: the more data, the more likely it is we will get a better or more accurate model. Although it is advised to train deep learning models on single-node clusters, your model or your data may be too large to fit in the memory of a single machine. A single-node cluster being insufficient is the problem that data scientists may face when training deep learning models. Thankfully, Horovod can help in these scenarios.

Understand Horovod

Horovod is an open-source distributed training framework and is the alternative to training a model on a single-node cluster. Horovod allows data scientists to distribute the training process and make use of Spark's parallel processing.

Since deep learning models contain layers that need to be processed sequentially, and use intermediary results to improve the model at the end of an epoch, the parallel processing of deep learning models can quickly become complicated. Horovod is designed to take care of the infrastructure management so that data scientists can focus on training models.

Horovod is named after a traditional dance in which partners hold hands while dancing in a circle. Horovod owes this name to the way it allows worker nodes to communicate with other worker nodes, to avoid a bottle-neck at the driver node.

When Horovod is used on top of one of the deep learning frameworks (TensorFlow, PyTorch or Keras), it trains multiple models on different batches of the input dataset on separate workers. In other words, multiple models are trained in parallel on separate workers using different subsets of the data.

At the end of an epoch, the weights are communicated between workers and the average weight of all workers is calculated. Next, a new epoch can start using the new average weight and during which again, multiple models are trained in parallel.

HorovodRunner for distributed deep learning

You want to train deep learning models using the open-source frameworks TensorFlow, Keras, or PyTorch in Azure Databricks. You tried using a single-node cluster but your model or data is too large for the cluster. In that case, you may choose to use Horovod on top of your work so far to distribute the training. Let's explore how we can do that using HorovodRunner.

Trigger Horovod in Azure Databricks with HorovodRunner

HorovodRunner is a general API, which triggers Horovod jobs. The benefit of using HorovodRunner instead of the Horovod framework directly, is that HorovodRunner has been designed to distribute deep learning jobs across *Spark* workers. As a result, HorovodRunner is more stable for long-running deep learning training jobs on Azure Databricks.

Horovod process

To distribute the training of a deep learning model using HorovodRunner, you should do the following:

- Prepare and test single-node code with TensorFlow, Keras, or PyTorch.
- Migrate the code to Horovod.
- Use HorovodRunner to run the code and distribute your work.

Run single-node training

Before working with Horovod and HorovodRunner, the code used to train the deep learning model should be tested on a single-node cluster. Once it works, make sure to wrap the main training procedure into a single Python function. This function will be used later on to initiate the distributed execution of your code.

Migrate to Horovod

Once you have tested your single-node code to train a deep learning model, you have to migrate it to Horovod before you can trigger the job with HorovodRunner.

1. Import the Horovod framework as `hvd`.
2. Initialize the Horovod library with `hvd.init()`.
3. Pin one GPU per process. Pinning is necessary to disable random mapping of workers and avoid clashes. Pinning is skipped when using CPUs.
4. Specify how you want to partition or sample the data so that each worker uses a unique subset of the data to train a model. As a best practice, make sure the subsets are all the same size. Depending on the input dataset, there are several techniques to do the sampling. For example, you could use Petastorm to work with datasets in Apache Parquet format. Learn more about the open-source library Petastorm here.
5. Scale the learning rate by the number of workers to make sure the weights are adjusted correctly after each epoch.
6. Use the Horovod distributed optimizer to handle the communication between workers.
7. Broadcast the initial parameters so all workers start with the same parameters.
8. Save checkpoints only on worker 0 to prevent conflicts between workers.

Use HorovodRunner

To run HovorodRunner, you have to create a `HorovodRunner` instance in which you specify how many nodes (defined by the argument `np`) you want to distribute your work to. You can specify to use one node if you want to test on a single-node cluster with `np=-1`. Finally, you can trigger the Horovod training job with HorovodRunner by invoking the Python function you created for your training code.

DONE