Now you're ready to start building your model with Azure Machine Learning service. In this part of the tutorial, you use the prepared data and automatically generate a regression model to predict taxi fare prices. By using the automated machine learning capabilities of the service, you define your machine learning goals and constraints. You launch the automated machine learning process. Then allow the algorithm selection and hyperparameter tuning to happen for you. The automated machine learning technique iterates over many combinations of algorithms and hyperparameters until it finds the best model based on your criterion.

In this tutorial, you learn the following tasks:

* Set up a Python environment and import the SDK packages.
* Configure an Azure Machine Learning service workspace.
* Autotrain a regression model.
* Run the model locally with custom parameters.
* Explore the results.

If you don’t have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning service](https://aka.ms/AMLFree) today.

**Note**

Code in this article was tested with Azure Machine Learning SDK version 1.0.0.

**Set up your development environment**

All the setup for your development work can be accomplished in a Python notebook. Setup includes the following actions:

* Install the SDK
* Import Python packages
* Configure your workspace

**Install and import packages**

If you are following the tutorial in your own Python environment, use the following to install necessary packages.

shellCopy

pip install azureml-sdk[automl,notebooks] matplotlib

Import the Python packages you need in this tutorial:

PythonCopy

import azureml.core

import pandas as pd

from azureml.core.workspace import Workspace

import logging

import os

**Configure workspace**

Create a workspace object from the existing workspace. A [Workspace](https://docs.microsoft.com/python/api/azureml-core/azureml.core.workspace.workspace?view=azure-ml-py) is a class that accepts your Azure subscription and resource information. It also creates a cloud resource to monitor and track your model runs.

Workspace.from\_config() reads the file **config.json** and loads the details into an object named ws. ws is used throughout the rest of the code in this tutorial.

After you have a workspace object, specify a name for the experiment. Create and register a local directory with the workspace. The history of all runs is recorded under the specified experiment and in the [Azure portal](https://portal.azure.com/).

PythonCopy

ws = Workspace.from\_config()

# choose a name for the run history container in the workspace

experiment\_name = 'automated-ml-regression'

# project folder

project\_folder = './automated-ml-regression'

output = {}

output['SDK version'] = azureml.core.VERSION

output['Subscription ID'] = ws.subscription\_id

output['Workspace'] = ws.name

output['Resource Group'] = ws.resource\_group

output['Location'] = ws.location

output['Project Directory'] = project\_folder

pd.set\_option('display.max\_colwidth', -1)

pd.DataFrame(data=output, index=['']).T

**Explore data**

Use the data flow object created in the previous tutorial. To summarize, part 1 of this tutorial cleaned the NYC Taxi data so it could be used in a machine learning model. Now, you use various features from the data set and allow an automated model to build relationships between the features and the price of a taxi trip. Open and run the data flow and review the results:

PythonCopy

import azureml.dataprep as dprep

file\_path = os.path.join(os.getcwd(), "dflows.dprep")

dflow\_prepared = dprep.Dataflow.open(file\_path)

dflow\_prepared.get\_profile()

|  | **Type** | **Min** | **Max** | **Count** | **Missing count** | **Not missing count** | **Percent missing** | **Error count** | **Empty count** | **0.1% quantile** | **1% quantile** | **5% quantile** | **25% quantile** | **50% quantile** | **75% quantile** | **95% quantile** | **99% quantile** | **99.9% quantile** | **Mean** | **Standard deviation** | **Variance** | **Skewness** | **Kurtosis** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **vendor** | FieldType.STRING | 1 | VTS | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **pickup\_weekday** | FieldType.STRING | Friday | Wednesday | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **pickup\_hour** | FieldType.DECIMAL | 0 | 23 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 2.90047 | 2.69355 | 9.72889 | 16 | 19.3713 | 22.6974 | 23 | 23 | 14.2731 | 6.59242 | 43.46 | -0.693723 | -0.570403 |
| **pickup\_minute** | FieldType.DECIMAL | 0 | 59 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 4.99701 | 4.95833 | 14.1528 | 29.3832 | 44.6825 | 56.4444 | 58.9909 | 59 | 29.427 | 17.4333 | 303.921 | 0.0120999 | -1.20981 |
| **pickup\_second** | FieldType.DECIMAL | 0 | 59 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 5.28131 | 5 | 14.7832 | 29.9293 | 44.725 | 56.7573 | 59 | 59 | 29.7443 | 17.3595 | 301.351 | -0.0252399 | -1.19616 |
| **dropoff\_weekday** | FieldType.STRING | Friday | Wednesday | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **dropoff\_hour** | FieldType.DECIMAL | 0 | 23 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 2.57153 | 2 | 9.58795 | 15.9994 | 19.6184 | 22.8317 | 23 | 23 | 14.2105 | 6.71093 | 45.0365 | -0.687292 | -0.61951 |
| **dropoff\_minute** | FieldType.DECIMAL | 0 | 59 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 5.44383 | 4.84694 | 14.1036 | 28.8365 | 44.3102 | 56.6892 | 59 | 59 | 29.2907 | 17.4108 | 303.136 | 0.0222514 | -1.2181 |
| **dropoff\_second** | FieldType.DECIMAL | 0 | 59 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0 | 5.07801 | 5 | 14.5751 | 29.5972 | 45.4649 | 56.2729 | 59 | 59 | 29.772 | 17.5337 | 307.429 | -0.0212575 | -1.226 |
| **store\_forward** | FieldType.STRING | N | Y | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **pickup\_longitude** | FieldType.DECIMAL | -74.0781 | -73.7459 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | -74.0578 | -73.9639 | -73.9656 | -73.9508 | -73.9255 | -73.8529 | -73.8302 | -73.8238 | -73.7697 | -73.9123 | 0.0503757 | 0.00253771 | 0.352172 | -0.923743 |
| **pickup\_latitude** | FieldType.DECIMAL | 40.5755 | 40.8799 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 40.632 | 40.7117 | 40.7115 | 40.7213 | 40.7565 | 40.8058 | 40.8478 | 40.8676 | 40.8778 | 40.7649 | 0.0494674 | 0.00244702 | 0.205972 | -0.777945 |
| **dropoff\_longitude** | FieldType.DECIMAL | -74.0857 | -73.7209 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | -74.0775 | -73.9875 | -73.9882 | -73.9638 | -73.935 | -73.8755 | -73.8125 | -73.7759 | -73.7327 | -73.9202 | 0.0584627 | 0.00341789 | 0.623622 | -0.262603 |
| **dropoff\_latitude** | FieldType.DECIMAL | 40.5835 | 40.8797 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 40.5973 | 40.6928 | 40.6911 | 40.7226 | 40.7567 | 40.7918 | 40.8495 | 40.868 | 40.8787 | 40.7583 | 0.0517399 | 0.00267701 | 0.0390404 | -0.203525 |
| **passengers** | FieldType.DECIMAL | 1 | 6 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 6 | 6 | 2.39249 | 1.83197 | 3.3561 | 0.763144 | -1.23467 |
| **distance** | FieldType.DECIMAL | 0.01 | 32.34 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 0.0108744 | 0.743898 | 0.738194 | 1.243 | 2.40168 | 4.74478 | 10.5136 | 14.9011 | 21.8035 | 3.5447 | 3.2943 | 10.8524 | 1.91556 | 4.99898 |
| **cost** | FieldType.DECIMAL | 0.1 | 88 | 6148.0 | 0.0 | 6148.0 | 0.0 | 0.0 | 0.0 | 2.33837 | 5.00491 | 5 | 6.93129 | 10.524 | 17.4811 | 33.2343 | 50.0093 | 63.1753 | 13.6843 | 9.66571 | 93.426 | 1.78518 | 4.13972 |

You prepare the data for the experiment by adding columns to dflow\_x to be features for our model creation. You define dflow\_y to be our prediction value, **cost**:

PythonCopy

dflow\_X = dflow\_prepared.keep\_columns(['pickup\_weekday','pickup\_hour', 'distance','passengers', 'vendor'])

dflow\_y = dflow\_prepared.keep\_columns('cost')

**Split the data into train and test sets**

Now you split the data into training and test sets by using the train\_test\_split function in the sklearn library. This function segregates the data into the x, **features**, dataset for model training and the y, **values to predict**, dataset for testing. The test\_size parameter determines the percentage of data to allocate to testing. The random\_state parameter sets a seed to the random generator, so that your train-test splits are always deterministic:

PythonCopy

from sklearn.model\_selection import train\_test\_split

x\_df = dflow\_X.to\_pandas\_dataframe()

y\_df = dflow\_y.to\_pandas\_dataframe()

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_df, y\_df, test\_size=0.2, random\_state=223)

# flatten y\_train to 1d array

y\_train.values.flatten()

The purpose of this step is to have data points to test the finished model that haven't been used to train the model, in order to measure true accuracy. In other words, a well-trained model should be able to accurately make predictions from data it hasn't already seen. You now have the necessary packages and data ready for autotraining your model.

**Automatically train a model**

To automatically train a model, take the following steps:

1. Define settings for the experiment run. Attach your training data to the configuration, and modify settings that control the training process.
2. Submit the experiment for model tuning. After submitting the experiment, the process iterates through different machine learning algorithms and hyperparameter settings, adhering to your defined constraints. It chooses the best-fit model by optimizing an accuracy metric.

**Define settings for autogeneration and tuning**

Define the experiment parameter and model settings for autogeneration and tuning. View the full list of [settings](https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-configure-auto-train). Submitting the experiment with these default settings will take approximately 10-15 min, but if you want a shorter run time, reduce either iterations or iteration\_timeout\_minutes.

| **Property** | **Value in this tutorial** | **Description** |
| --- | --- | --- |
| **iteration\_timeout\_minutes** | 10 | Time limit in minutes for each iteration. Reduce this value to decrease total runtime. |
| **iterations** | 30 | Number of iterations. In each iteration, a new machine learning model is trained with your data. This is the primary value that affects total run time. |
| **primary\_metric** | spearman\_correlation | Metric that you want to optimize. The best-fit model will be chosen based on this metric. |
| **preprocess** | True | By using **True**, the experiment can preprocess the input data (handling missing data, converting text to numeric, etc.) |
| **verbosity** | logging.INFO | Controls the level of logging. |
| **n\_cross\_validations** | 5 | Number of cross-validation splits to perform when validation data is not specified. |

PythonCopy

automl\_settings = {

"iteration\_timeout\_minutes" : 10,

"iterations" : 30,

"primary\_metric" : 'spearman\_correlation',

"preprocess" : True,

"verbosity" : logging.INFO,

"n\_cross\_validations": 5

}

Use your defined training settings as a parameter to an AutoMLConfig object. Additionally, specify your training data and the type of model, which is regression in this case.

PythonCopy

from azureml.train.automl import AutoMLConfig

# local compute

automated\_ml\_config = AutoMLConfig(task = 'regression',

debug\_log = 'automated\_ml\_errors.log',

path = project\_folder,

X = x\_train.values,

y = y\_train.values.flatten(),

\*\*automl\_settings)

**Train the automatic regression model**

Start the experiment to run locally. Pass the defined automated\_ml\_config object to the experiment. Set the output to True to view progress during the experiment:

PythonCopy

from azureml.core.experiment import Experiment

experiment=Experiment(ws, experiment\_name)

local\_run = experiment.submit(automated\_ml\_config, show\_output=True)

The output shown updates live as the experiment runs. For each iteration, you see the model type, the run duration, and the training accuracy. The field BEST tracks the best running training score based on your metric type.

Copy

Parent Run ID: AutoML\_02778de3-3696-46e9-a71b-521c8fca0651

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ITERATION: The iteration being evaluated.

PIPELINE: A summary description of the pipeline being evaluated.

DURATION: Time taken for the current iteration.

METRIC: The result of computing score on the fitted pipeline.

BEST: The best observed score thus far.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ITERATION PIPELINE DURATION METRIC BEST

0 MaxAbsScaler ExtremeRandomTrees 0:00:08 0.9447 0.9447

1 StandardScalerWrapper GradientBoosting 0:00:09 0.9536 0.9536

2 StandardScalerWrapper ExtremeRandomTrees 0:00:09 0.8580 0.9536

3 StandardScalerWrapper RandomForest 0:00:08 0.9147 0.9536

4 StandardScalerWrapper ExtremeRandomTrees 0:00:45 0.9398 0.9536

5 MaxAbsScaler LightGBM 0:00:08 0.9562 0.9562

6 StandardScalerWrapper ExtremeRandomTrees 0:00:27 0.8282 0.9562

7 StandardScalerWrapper LightGBM 0:00:07 0.9421 0.9562

8 MaxAbsScaler DecisionTree 0:00:08 0.9526 0.9562

9 MaxAbsScaler RandomForest 0:00:09 0.9355 0.9562

10 MaxAbsScaler SGD 0:00:09 0.9602 0.9602

11 MaxAbsScaler LightGBM 0:00:09 0.9553 0.9602

12 MaxAbsScaler DecisionTree 0:00:07 0.9484 0.9602

13 MaxAbsScaler LightGBM 0:00:08 0.9540 0.9602

14 MaxAbsScaler RandomForest 0:00:10 0.9365 0.9602

15 MaxAbsScaler SGD 0:00:09 0.9602 0.9602

16 StandardScalerWrapper ExtremeRandomTrees 0:00:49 0.9171 0.9602

17 SparseNormalizer LightGBM 0:00:08 0.9191 0.9602

18 MaxAbsScaler DecisionTree 0:00:08 0.9402 0.9602

19 StandardScalerWrapper ElasticNet 0:00:08 0.9603 0.9603

20 MaxAbsScaler DecisionTree 0:00:08 0.9513 0.9603

21 MaxAbsScaler SGD 0:00:08 0.9603 0.9603

22 MaxAbsScaler SGD 0:00:10 0.9602 0.9603

23 StandardScalerWrapper ElasticNet 0:00:09 0.9603 0.9603

24 StandardScalerWrapper ElasticNet 0:00:09 0.9603 0.9603

25 MaxAbsScaler SGD 0:00:09 0.9603 0.9603

26 TruncatedSVDWrapper ElasticNet 0:00:09 0.9602 0.9603

27 MaxAbsScaler SGD 0:00:12 0.9413 0.9603

28 StandardScalerWrapper ElasticNet 0:00:07 0.9603 0.9603

29 Ensemble 0:00:38 0.9622 0.9622

**Explore the results**

Explore the results of automatic training with a Jupyter widget or by examining the experiment history.

**Option 1: Add a Jupyter widget to see results**

If you use a Jupyter notebook, use this Jupyter notebook widget to see a graph and a table of all results:

PythonCopy

from azureml.widgets import RunDetails

RunDetails(local\_run).show()

**Option 2: Get and examine all run iterations in Python**

You can also retrieve the history of each experiment and explore the individual metrics for each iteration run. By examining RMSE (root\_mean\_squared\_error) for each individual model run, you see that most iterations are predicting the taxi fair cost within a reasonable margin ($3-4).

PythonCopy

children = list(local\_run.get\_children())

metricslist = {}

for run in children:

properties = run.get\_properties()

metrics = {k: v for k, v in run.get\_metrics().items() if isinstance(v, float)}

metricslist[int(properties['iteration'])] = metrics

rundata = pd.DataFrame(metricslist).sort\_index(1)

rundata

|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **...** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **explained\_variance** | 0.811037 | 0.880553 | 0.398582 | 0.776040 | 0.663869 | 0.875911 | 0.115632 | 0.586905 | 0.851911 | 0.793964 | ... | 0.850023 | 0.883603 | 0.883704 | 0.880797 | 0.881564 | 0.883708 | 0.881826 | 0.585377 | 0.883123 | 0.886817 |
| **mean\_absolute\_error** | 2.189444 | 1.500412 | 5.480531 | 2.626316 | 2.973026 | 1.550199 | 6.383868 | 4.414241 | 1.743328 | 2.294601 | ... | 1.797402 | 1.415815 | 1.418167 | 1.578617 | 1.559427 | 1.413042 | 1.551698 | 4.069196 | 1.505795 | 1.430957 |
| **median\_absolute\_error** | 1.438417 | 0.850899 | 4.579662 | 1.765210 | 1.594600 | 0.869883 | 4.266450 | 3.627355 | 0.954992 | 1.361014 | ... | 0.973634 | 0.774814 | 0.797269 | 1.147234 | 1.116424 | 0.783958 | 1.098464 | 2.709027 | 1.003728 | 0.851724 |
| **normalized\_mean\_absolute\_error** | 0.024908 | 0.017070 | 0.062350 | 0.029878 | 0.033823 | 0.017636 | 0.072626 | 0.050219 | 0.019833 | 0.026105 | ... | 0.020448 | 0.016107 | 0.016134 | 0.017959 | 0.017741 | 0.016076 | 0.017653 | 0.046293 | 0.017131 | 0.016279 |
| **normalized\_median\_absolute\_error** | 0.016364 | 0.009680 | 0.052101 | 0.020082 | 0.018141 | 0.009896 | 0.048538 | 0.041267 | 0.010865 | 0.015484 | ... | 0.011077 | 0.008815 | 0.009070 | 0.013052 | 0.012701 | 0.008919 | 0.012497 | 0.030819 | 0.011419 | 0.009690 |
| **normalized\_root\_mean\_squared\_error** | 0.047968 | 0.037882 | 0.085572 | 0.052282 | 0.065809 | 0.038664 | 0.109401 | 0.071104 | 0.042294 | 0.049967 | ... | 0.042565 | 0.037685 | 0.037557 | 0.037643 | 0.037513 | 0.037560 | 0.037465 | 0.072077 | 0.037249 | 0.036716 |
| **normalized\_root\_mean\_squared\_log\_error** | 0.055353 | 0.045000 | 0.110219 | 0.065633 | 0.063589 | 0.044412 | 0.123433 | 0.092312 | 0.046130 | 0.055243 | ... | 0.046540 | 0.041804 | 0.041771 | 0.045175 | 0.044628 | 0.041617 | 0.044405 | 0.079651 | 0.042799 | 0.041530 |
| **r2\_score** | 0.810900 | 0.880328 | 0.398076 | 0.775957 | 0.642812 | 0.875719 | 0.021603 | 0.586514 | 0.851767 | 0.793671 | ... | 0.849809 | 0.880142 | 0.880952 | 0.880586 | 0.881347 | 0.880887 | 0.881613 | 0.548121 | 0.882883 | 0.886321 |
| **root\_mean\_squared\_error** | 4.216362 | 3.329810 | 7.521765 | 4.595604 | 5.784601 | 3.398540 | 9.616354 | 6.250011 | 3.717661 | 4.392072 | ... | 3.741447 | 3.312533 | 3.301242 | 3.308795 | 3.297389 | 3.301485 | 3.293182 | 6.335581 | 3.274209 | 3.227365 |
| **root\_mean\_squared\_log\_error** | 0.243184 | 0.197702 | 0.484227 | 0.288349 | 0.279367 | 0.195116 | 0.542281 | 0.405559 | 0.202666 | 0.242702 | ... | 0.204464 | 0.183658 | 0.183514 | 0.198468 | 0.196067 | 0.182836 | 0.195087 | 0.349935 | 0.188031 | 0.182455 |
| **spearman\_correlation** | 0.944743 | 0.953618 | 0.857965 | 0.914703 | 0.939846 | 0.956159 | 0.828187 | 0.942069 | 0.952581 | 0.935477 | ... | 0.951287 | 0.960335 | 0.960195 | 0.960279 | 0.960288 | 0.960323 | 0.960161 | 0.941254 | 0.960293 | 0.962158 |
| **spearman\_correlation\_max** | 0.944743 | 0.953618 | 0.953618 | 0.953618 | 0.953618 | 0.956159 | 0.956159 | 0.956159 | 0.956159 | 0.956159 | ... | 0.960303 | 0.960335 | 0.960335 | 0.960335 | 0.960335 | 0.960335 | 0.960335 | 0.960335 | 0.960335 | 0.962158 |

12 rows × 30 columns

**Retrieve the best model**

Select the best pipeline from our iterations. The get\_output method on automl\_classifier returns the best run and the fitted model for the last fit invocation. By using the overloads on get\_output, you can retrieve the best run and fitted model for any logged metric or a particular iteration:

PythonCopy

best\_run, fitted\_model = local\_run.get\_output()

print(best\_run)

print(fitted\_model)

**Test the best model accuracy**

Use the best model to run predictions on the test dataset to predict taxi fares. The function predict uses the best model and predicts the values of y, **trip cost**, from the x\_testdataset. Print the first 10 predicted cost values from y\_predict:

PythonCopy

y\_predict = fitted\_model.predict(x\_test.values)

print(y\_predict[:10])

Create a scatter plot to visualize the predicted cost values compared to the actual cost values. The following code uses the distance feature as the x-axis and trip cost as the y-axis. To compare the variance of predicted cost at each trip distance value, the first 100 predicted and actual cost values are created as separate series. Examining the plot shows that the distance/cost relationship is nearly linear, and the predicted cost values are in most cases very close to the actual cost values for the same trip distance.

PythonCopy

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(14, 10))

ax1 = fig.add\_subplot(111)

distance\_vals = [x[4] for x in x\_test.values]

y\_actual = y\_test.values.flatten().tolist()

ax1.scatter(distance\_vals[:100], y\_predict[:100], s=18, c='b', marker="s", label='Predicted')

ax1.scatter(distance\_vals[:100], y\_actual[:100], s=18, c='r', marker="o", label='Actual')

ax1.set\_xlabel('distance (mi)')

ax1.set\_title('Predicted and Actual Cost/Distance')

ax1.set\_ylabel('Cost ($)')

plt.legend(loc='upper left', prop={'size': 12})

plt.rcParams.update({'font.size': 14})

plt.show()

Calculate the root mean squared error of the results. Use the y\_test dataframe. Convert it to a list to compare to the predicted values. The function mean\_squared\_errortakes two arrays of values and calculates the average squared error between them. Taking the square root of the result gives an error in the same units as the y variable, **cost**. It indicates roughly how far the taxi fare predictions are from the actual fares:

PythonCopy

from sklearn.metrics import mean\_squared\_error

from math import sqrt

rmse = sqrt(mean\_squared\_error(y\_actual, y\_predict))

rmse

Copy

3.2204936862688798

Run the following code to calculate mean absolute percent error (MAPE) by using the full y\_actual and y\_predict datasets. This metric calculates an absolute difference between each predicted and actual value and sums all the differences. Then it expresses that sum as a percent of the total of the actual values:

PythonCopy

sum\_actuals = sum\_errors = 0

for actual\_val, predict\_val in zip(y\_actual, y\_predict):

abs\_error = actual\_val - predict\_val

if abs\_error < 0:

abs\_error = abs\_error \* -1

sum\_errors = sum\_errors + abs\_error

sum\_actuals = sum\_actuals + actual\_val

mean\_abs\_percent\_error = sum\_errors / sum\_actuals

print("Model MAPE:")

print(mean\_abs\_percent\_error)

print()

print("Model Accuracy:")

print(1 - mean\_abs\_percent\_error)

Copy

Model MAPE:

0.10545153869569586

Model Accuracy:

0.8945484613043041

From the final prediction accuracy metrics, you see that the model is fairly good at predicting taxi fares from the data set's features, typically within +- $3.00. The traditional machine learning model development process is highly resource-intensive, and requires significant domain knowledge and time investment to run and compare the results of dozens of models. Using automated machine learning is a great way to rapidly test many different models for your scenario.

**Clean up resources**

**Important**

The resources you created can be used as prerequisites to other Azure Machine Learning service tutorials and how-to articles.

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.
2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.