**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

**Install and import packages**

Use the following to install necessary packages if you don't already have them.

shellCopy

pip install "azureml-dataprep[pandas]>=1.1.0,<1.2.0"

Import the SDK.

PythonCopy

import azureml.dataprep as dprep

**Important**

Ensure you install the latest version. This tutorial will not work with version number lower than 1.1.0

**Load data**

Download two different NYC taxi data sets into dataflow objects. The datasets have slightly different fields. The auto\_read\_file() method automatically recognizes the input file type.

PythonCopy

from IPython.display import display

dataset\_root = "https://dprepdata.blob.core.windows.net/demo"

green\_path = "/".join([dataset\_root, "green-small/\*"])

yellow\_path = "/".join([dataset\_root, "yellow-small/\*"])

green\_df\_raw = dprep.read\_csv(path=green\_path, header=dprep.PromoteHeadersMode.GROUPED)

# auto\_read\_file automatically identifies and parses the file type, which is useful when you don't know the file type.

yellow\_df\_raw = dprep.auto\_read\_file(path=yellow\_path)

display(green\_df\_raw.head(5))

display(yellow\_df\_raw.head(5))

A Dataflow object is similar to a dataframe, and represents a series of lazily-evaluated, immutable operations on data. Operations can be added by invoking the different transformation and filtering methods available. The result of adding an operation to a Dataflow is always a new Dataflow object.

**Cleanse data**

Now you populate some variables with shortcut transforms to apply to all dataflows. The drop\_if\_all\_null variable is used to delete records where all fields are null. The useful\_columns variable holds an array of column descriptions that are kept in each dataflow.

PythonCopy

all\_columns = dprep.ColumnSelector(term=".\*", use\_regex=True)

drop\_if\_all\_null = [all\_columns, dprep.ColumnRelationship(dprep.ColumnRelationship.ALL)]

useful\_columns = [

"cost", "distance", "dropoff\_datetime", "dropoff\_latitude", "dropoff\_longitude",

"passengers", "pickup\_datetime", "pickup\_latitude", "pickup\_longitude", "store\_forward", "vendor"

]

You first work with the green taxi data to get it into a valid shape that can be combined with the yellow taxi data. Call the replace\_na(), drop\_nulls(), and keep\_columns()functions by using the shortcut transform variables you created. Additionally, rename all the columns in the dataframe to match the names in the useful\_columns variable.

PythonCopy

green\_df = (green\_df\_raw

.replace\_na(columns=all\_columns)

.drop\_nulls(\*drop\_if\_all\_null)

.rename\_columns(column\_pairs={

"VendorID": "vendor",

"lpep\_pickup\_datetime": "pickup\_datetime",

"Lpep\_dropoff\_datetime": "dropoff\_datetime",

"lpep\_dropoff\_datetime": "dropoff\_datetime",

"Store\_and\_fwd\_flag": "store\_forward",

"store\_and\_fwd\_flag": "store\_forward",

"Pickup\_longitude": "pickup\_longitude",

"Pickup\_latitude": "pickup\_latitude",

"Dropoff\_longitude": "dropoff\_longitude",

"Dropoff\_latitude": "dropoff\_latitude",

"Passenger\_count": "passengers",

"Fare\_amount": "cost",

"Trip\_distance": "distance"

})

.keep\_columns(columns=useful\_columns))

green\_df.head(5)

|  | **vendor** | **pickup\_datetime** | **dropoff\_datetime** | **store\_forward** | **pickup\_longitude** | **pickup\_latitude** | **dropoff\_longitude** | **dropoff\_latitude** | **passengers** | **distance** | **cost** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2 | 2013-08-01 08:14:37 | 2013-08-01 09:09:06 | N | 0 | 0 | 0 | 0 | 1 | .00 | 21.25 |
| **1** | 2 | 2013-08-01 09:13:00 | 2013-08-01 11:38:00 | N | 0 | 0 | 0 | 0 | 2 | .00 | 74.5 |
| **2** | 2 | 2013-08-01 09:48:00 | 2013-08-01 09:49:00 | N | 0 | 0 | 0 | 0 | 1 | .00 | 1 |
| **3** | 2 | 2013-08-01 10:38:35 | 2013-08-01 10:38:51 | N | 0 | 0 | 0 | 0 | 1 | .00 | 3.25 |
| **4** | 2 | 2013-08-01 11:51:45 | 2013-08-01 12:03:52 | N | 0 | 0 | 0 | 0 | 1 | .00 | 8.5 |

Run the same transformation steps on the yellow taxi data. These functions ensure that null data is removed from the data set, which will help increase machine learning model accuracy.

PythonCopy

yellow\_df = (yellow\_df\_raw

.replace\_na(columns=all\_columns)

.drop\_nulls(\*drop\_if\_all\_null)

.rename\_columns(column\_pairs={

"vendor\_name": "vendor",

"VendorID": "vendor",

"vendor\_id": "vendor",

"Trip\_Pickup\_DateTime": "pickup\_datetime",

"tpep\_pickup\_datetime": "pickup\_datetime",

"Trip\_Dropoff\_DateTime": "dropoff\_datetime",

"tpep\_dropoff\_datetime": "dropoff\_datetime",

"store\_and\_forward": "store\_forward",

"store\_and\_fwd\_flag": "store\_forward",

"Start\_Lon": "pickup\_longitude",

"Start\_Lat": "pickup\_latitude",

"End\_Lon": "dropoff\_longitude",

"End\_Lat": "dropoff\_latitude",

"Passenger\_Count": "passengers",

"passenger\_count": "passengers",

"Fare\_Amt": "cost",

"fare\_amount": "cost",

"Trip\_Distance": "distance",

"trip\_distance": "distance"

})

.keep\_columns(columns=useful\_columns))

yellow\_df.head(5)

Call the append\_rows() function on the green taxi data to append the yellow taxi data. A new combined dataframe is created.

PythonCopy

combined\_df = green\_df.append\_rows([yellow\_df])

**Convert types and filter**

Examine the pickup and drop-off coordinates summary statistics to see how the data is distributed. First, define a TypeConverter object to change the latitude and longitude fields to decimal type. Next, call the keep\_columns() function to restrict output to only the latitude and longitude fields, and then call the get\_profile() function. These function calls create a condensed view of the dataflow to just show the lat/long fields, which makes it easier to evaluate missing or out-of-scope coordinates.

PythonCopy

decimal\_type = dprep.TypeConverter(data\_type=dprep.FieldType.DECIMAL)

combined\_df = combined\_df.set\_column\_types(type\_conversions={

"pickup\_longitude": decimal\_type,

"pickup\_latitude": decimal\_type,

"dropoff\_longitude": decimal\_type,

"dropoff\_latitude": decimal\_type

})

combined\_df.keep\_columns(columns=[

"pickup\_longitude", "pickup\_latitude",

"dropoff\_longitude", "dropoff\_latitude"

]).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** | **Standard Deviation** | **Mean** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pickup\_longitude** | FieldType.DECIMAL | -115.179337 | 0.000000 | 7722.0 | 0.0 | 7722.0 | 0.0 | 0.0 | 0.0 | -88.114046 | -73.961840 | -73.961964 | -73.947693 | -73.922097 | -73.846670 | 0.000000 | 0.000000 | 0.000000 | 18.792672 | -68.833579 |
| **pickup\_latitude** | FieldType.DECIMAL | 0.000000 | 40.919121 | 7722.0 | 0.0 | 7722.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 40.682889 | 40.675541 | 40.721075 | 40.756159 | 40.803909 | 40.849406 | 40.870681 | 40.891244 | 10.345967 | 37.936742 |
| **dropoff\_longitude** | FieldType.DECIMAL | -115.179337 | 0.000000 | 7722.0 | 0.0 | 7722.0 | 0.0 | 0.0 | 0.0 | -87.699611 | -73.984734 | -73.985777 | -73.956250 | -73.928948 | -73.866208 | 0.000000 | 0.000000 | 0.000000 | 18.696526 | -68.896978 |
| **dropoff\_latitude** | FieldType.DECIMAL | 0.000000 | 41.008934 | 7722.0 | 0.0 | 7722.0 | 0.0 | 0.0 | 0.0 | 0.000000 | 40.662763 | 40.654851 | 40.717821 | 40.756534 | 40.784688 | 40.852437 | 40.879289 | 40.937291 | 10.290780 | 37.963774 |

From the summary statistics output, you see there are missing coordinates and coordinates that aren't in New York City (this is determined from subjective analysis). Filter out coordinates for locations that are outside the city border. Chain the column filter commands within the filter() function and define the minimum and maximum bounds for each field. Then call the get\_profile() function again to verify the transformation.

PythonCopy

latlong\_filtered\_df = (combined\_df

.drop\_nulls(

columns=["pickup\_longitude", "pickup\_latitude", "dropoff\_longitude", "dropoff\_latitude"],

column\_relationship=dprep.ColumnRelationship(dprep.ColumnRelationship.ANY)

)

.filter(dprep.f\_and(

dprep.col("pickup\_longitude") <= -73.72,

dprep.col("pickup\_longitude") >= -74.09,

dprep.col("pickup\_latitude") <= 40.88,

dprep.col("pickup\_latitude") >= 40.53,

dprep.col("dropoff\_longitude") <= -73.72,

dprep.col("dropoff\_longitude") >= -74.09,

dprep.col("dropoff\_latitude") <= 40.88,

dprep.col("dropoff\_latitude") >= 40.53

)))

latlong\_filtered\_df.keep\_columns(columns=[

"pickup\_longitude", "pickup\_latitude",

"dropoff\_longitude", "dropoff\_latitude"

]).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** | **Standard Deviation** | **Mean** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pickup\_longitude** | FieldType.DECIMAL | -74.078156 | -73.736481 | 7059.0 | 0.0 | 7059.0 | 0.0 | 0.0 | 0.0 | -74.076314 | -73.962542 | -73.962893 | -73.948975 | -73.927856 | -73.866662 | -73.830438 | -73.823160 | -73.769750 | 0.048711 | -73.913865 |
| **pickup\_latitude** | FieldType.DECIMAL | 40.575485 | 40.879852 | 7059.0 | 0.0 | 7059.0 | 0.0 | 0.0 | 0.0 | 40.632884 | 40.713105 | 40.711600 | 40.721403 | 40.758142 | 40.805145 | 40.848855 | 40.867567 | 40.877690 | 0.048348 | 40.765226 |
| **dropoff\_longitude** | FieldType.DECIMAL | -74.085747 | -73.720871 | 7059.0 | 0.0 | 7059.0 | 0.0 | 0.0 | 0.0 | -74.078828 | -73.985650 | -73.985813 | -73.959041 | -73.936681 | -73.884846 | -73.815507 | -73.776697 | -73.733471 | 0.055961 | -73.920718 |
| **dropoff\_latitude** | FieldType.DECIMAL | 40.583530 | 40.879734 | 7059.0 | 0.0 | 7059.0 | 0.0 | 0.0 | 0.0 | 40.597741 | 40.695376 | 40.695115 | 40.727549 | 40.758160 | 40.788378 | 40.850372 | 40.867968 | 40.878586 | 0.050462 | 40.759487 |

**Split and rename columns**

Look at the data profile for the store\_forward column. This field is a boolean flag that is Ywhen the taxi did not have a connection to the server after the trip, and thus had to store the trip data in memory, and later forward it to the server when connected.

PythonCopy

latlong\_filtered\_df.keep\_columns(columns='store\_forward').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** | **Standard Deviation** | **Mean** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **store\_forward** | FieldType.STRING | N | Y | 7059.0 | 99.0 | 6960.0 | 0.014025 | 0.0 | 0.0 |  |  |  |  |  |  |  |  |  |  |  |

Notice that the data profile output in the store\_forward column shows that the data is inconsistent and there are missing or null values. Use the replace() and fill\_nulls()functions to replace these values with the string "N":

PythonCopy

replaced\_stfor\_vals\_df = latlong\_filtered\_df.replace(columns="store\_forward", find="0", replace\_with="N").fill\_nulls("store\_forward", "N")

Execute the replace function on the distance field. The function reformats distance values that are incorrectly labeled as .00, and fills any nulls with zeros. Convert the distance field to numerical format. These incorrect data points are likely anomalies in the data collection system on the taxi cabs.

PythonCopy

replaced\_distance\_vals\_df = replaced\_stfor\_vals\_df.replace(columns="distance", find=".00", replace\_with=0).fill\_nulls("distance", 0)

replaced\_distance\_vals\_df = replaced\_distance\_vals\_df.to\_number(["distance"])

Split the pickup and dropoff datetime values into the respective date and time columns. Use the split\_column\_by\_example() function to make the split. In this case, the optional example parameter of the split\_column\_by\_example() function is omitted. Therefore, the function automatically determines where to split based on the data.

PythonCopy

time\_split\_df = (replaced\_distance\_vals\_df

.split\_column\_by\_example(source\_column="pickup\_datetime")

.split\_column\_by\_example(source\_column="dropoff\_datetime"))

time\_split\_df.head(5)

|  | **vendor** | **pickup\_datetime** | **pickup\_datetime\_1** | **pickup\_datetime\_2** | **dropoff\_datetime** | **dropoff\_datetime\_1** | **dropoff\_datetime\_2** | **store\_forward** | **pickup\_longitude** | **pickup\_latitude** | **dropoff\_longitude** | **dropoff\_latitude** | **passengers** | **distance** | **cost** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2 | 2013-08-01 17:22:00 | 2013-08-01 | 17:22:00 | 2013-08-01 17:22:00 | 2013-08-01 | 17:22:00 | N | -73.937767 | 40.758480 | -73.937767 | 40.758480 | 1 | 0.0 | 2.5 |
| **1** | 2 | 2013-08-01 17:24:00 | 2013-08-01 | 17:24:00 | 2013-08-01 17:25:00 | 2013-08-01 | 17:25:00 | N | -73.937927 | 40.757843 | -73.937927 | 40.757843 | 1 | 0.0 | 2.5 |
| **2** | 2 | 2013-08-06 06:51:19 | 2013-08-06 | 06:51:19 | 2013-08-06 06:51:36 | 2013-08-06 | 06:51:36 | N | -73.937721 | 40.758404 | -73.937721 | 40.758369 | 1 | 0.0 | 3.3 |
| **3** | 2 | 2013-08-06 13:26:34 | 2013-08-06 | 13:26:34 | 2013-08-06 13:26:57 | 2013-08-06 | 13:26:57 | N | -73.937691 | 40.758419 | -73.937790 | 40.758358 | 1 | 0.0 | 3.3 |
| **4** | 2 | 2013-08-06 13:27:53 | 2013-08-06 | 13:27:53 | 2013-08-06 13:28:08 | 2013-08-06 | 13:28:08 | N | -73.937805 | 40.758396 | -73.937775 | 40.758450 | 1 | 0.0 | 3.3 |

Rename the columns generated by the split\_column\_by\_example() function to use meaningful names.

PythonCopy

renamed\_col\_df = (time\_split\_df

.rename\_columns(column\_pairs={

"pickup\_datetime\_1": "pickup\_date",

"pickup\_datetime\_2": "pickup\_time",

"dropoff\_datetime\_1": "dropoff\_date",

"dropoff\_datetime\_2": "dropoff\_time"

}))

renamed\_col\_df.head(5)

Call the get\_profile() function to see the full summary statistics after all cleansing steps.

PythonCopy

renamed\_col\_df.get\_profile()

**Transform data**

Split the pickup and dropoff date further into the day of the week, day of the month, and month values. To get the day of the week value, use the derive\_column\_by\_example()function. The function takes an array parameter of example objects that define the input data, and the preferred output. The function automatically determines your preferred transformation. For the pickup and dropoff time columns, split the time into the hour, minute, and second by using the split\_column\_by\_example() function with no example parameter.

After you generate the new features, use the drop\_columns() function to delete the original fields as the newly generated features are preferred. Rename the rest of the fields to use meaningful descriptions.

Transforming the data in this way to create new time-based features will improve machine learning model accuracy. For example, generating a new feature for the weekday will help establish a relationship between the day of the week and the taxi fare price, which is often more expensive on certain days of the week due to high demand.

PythonCopy

transformed\_features\_df = (renamed\_col\_df

.derive\_column\_by\_example(

source\_columns="pickup\_date",

new\_column\_name="pickup\_weekday",

example\_data=[("2009-01-04", "Sunday"), ("2013-08-22", "Thursday")]

)

.derive\_column\_by\_example(

source\_columns="dropoff\_date",

new\_column\_name="dropoff\_weekday",

example\_data=[("2013-08-22", "Thursday"), ("2013-11-03", "Sunday")]

)

.split\_column\_by\_example(source\_column="pickup\_time")

.split\_column\_by\_example(source\_column="dropoff\_time")

# The following two calls to split\_column\_by\_example reference the column names generated from the previous two calls.

.split\_column\_by\_example(source\_column="pickup\_time\_1")

.split\_column\_by\_example(source\_column="dropoff\_time\_1")

.drop\_columns(columns=[

"pickup\_date", "pickup\_time", "dropoff\_date", "dropoff\_time",

"pickup\_date\_1", "dropoff\_date\_1", "pickup\_time\_1", "dropoff\_time\_1"

])

.rename\_columns(column\_pairs={

"pickup\_date\_2": "pickup\_month",

"pickup\_date\_3": "pickup\_monthday",

"pickup\_time\_1\_1": "pickup\_hour",

"pickup\_time\_1\_2": "pickup\_minute",

"pickup\_time\_2": "pickup\_second",

"dropoff\_date\_2": "dropoff\_month",

"dropoff\_date\_3": "dropoff\_monthday",

"dropoff\_time\_1\_1": "dropoff\_hour",

"dropoff\_time\_1\_2": "dropoff\_minute",

"dropoff\_time\_2": "dropoff\_second"

}))

transformed\_features\_df.head(5)

|  | **vendor** | **pickup\_datetime** | **pickup\_weekday** | **pickup\_hour** | **pickup\_minute** | **pickup\_second** | **dropoff\_datetime** | **dropoff\_weekday** | **dropoff\_hour** | **dropoff\_minute** | **dropoff\_second** | **store\_forward** | **pickup\_longitude** | **pickup\_latitude** | **dropoff\_longitude** | **dropoff\_latitude** | **passengers** | **distance** | **cost** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2 | 2013-08-01 17:22:00 | Thursday | 17 | 22 | 00 | 2013-08-01 17:22:00 | Thursday | 17 | 22 | 00 | N | -73.937767 | 40.758480 | -73.937767 | 40.758480 | 1 | 0.0 | 2.5 |
| **1** | 2 | 2013-08-01 17:24:00 | Thursday | 17 | 24 | 00 | 2013-08-01 17:25:00 | Thursday | 17 | 25 | 00 | N | -73.937927 | 40.757843 | -73.937927 | 40.757843 | 1 | 0.0 | 2.5 |
| **2** | 2 | 2013-08-06 06:51:19 | Tuesday | 06 | 51 | 19 | 2013-08-06 06:51:36 | Tuesday | 06 | 51 | 36 | N | -73.937721 | 40.758404 | -73.937721 | 40.758369 | 1 | 0.0 | 3.3 |
| **3** | 2 | 2013-08-06 13:26:34 | Tuesday | 13 | 26 | 34 | 2013-08-06 13:26:57 | Tuesday | 13 | 26 | 57 | N | -73.937691 | 40.758419 | -73.937790 | 40.758358 | 1 | 0.0 | 3.3 |
| **4** | 2 | 2013-08-06 13:27:53 | Tuesday | 13 | 27 | 53 | 2013-08-06 13:28:08 | Tuesday | 13 | 28 | 08 | N | -73.937805 | 40.758396 | -73.937775 | 40.758450 | 1 | 0.0 | 3.3 |

Notice that the data shows that the pickup and dropoff date and time components produced from the derived transformations are correct. Drop the pickup\_datetime and dropoff\_datetime columns because they're no longer needed (granular time features like hour, minute and second are more useful for model training).

PythonCopy

processed\_df = transformed\_features\_df.drop\_columns(columns=["pickup\_datetime", "dropoff\_datetime"])

Use the type inference functionality to automatically check the data type of each field, and display the inference results.

PythonCopy

type\_infer = processed\_df.builders.set\_column\_types()

type\_infer.learn()

type\_infer

The resulting output of type\_infer is as follows.

Copy

Column types conversion candidates:

'pickup\_weekday': [FieldType.STRING],

'pickup\_hour': [FieldType.DECIMAL],

'pickup\_minute': [FieldType.DECIMAL],

'pickup\_second': [FieldType.DECIMAL],

'dropoff\_hour': [FieldType.DECIMAL],

'dropoff\_minute': [FieldType.DECIMAL],

'dropoff\_second': [FieldType.DECIMAL],

'store\_forward': [FieldType.STRING],

'pickup\_longitude': [FieldType.DECIMAL],

'dropoff\_longitude': [FieldType.DECIMAL],

'passengers': [FieldType.DECIMAL],

'distance': [FieldType.DECIMAL],

'vendor': [FieldType.STRING],

'dropoff\_weekday': [FieldType.STRING],

'pickup\_latitude': [FieldType.DECIMAL],

'dropoff\_latitude': [FieldType.DECIMAL],

'cost': [FieldType.DECIMAL]

The inference results look correct based on the data. Now apply the type conversions to the dataflow.

PythonCopy

type\_converted\_df = type\_infer.to\_dataflow()

type\_converted\_df.get\_profile()

Before you package the dataflow, run two final filters on the data set. To eliminate incorrectly captured data points, filter the dataflow on records where both the cost and distancevariable values are greater than zero. This step will significantly improve machine learning model accuracy, because data points with a zero cost or distance represent major outliers that throw off prediction accuracy.

PythonCopy

final\_df = type\_converted\_df.filter(dprep.col("distance") > 0)

final\_df = final\_df.filter(dprep.col("cost") > 0)

You now have a fully transformed and prepared dataflow object to use in a machine learning model. The SDK includes object serialization functionality, which is used as shown in the following code.

PythonCopy

import os

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

final\_df.save(file\_path)