

Model connectivity & development / Tutorial - Supervised Machine Learning / Train a model in Code Repositories

2b. Tutorial - Train a model in Code Repositories

Before starting this step of the tutorial, you should have completed the [modeling project set up](#). In this tutorial, you can choose to either train a model [in a Jupyter® notebook](#) or in Code Repositories. Jupyter® notebooks are recommended for fast and iterative model development whereas code repositories are recommended for production-grade data and model pipelines.

In this step of the tutorial, we will train a model in [Code Repositories](#). This step will cover:

1. [Creating a code repository for model training](#)
2. [Splitting feature data for testing and training](#)
3. [Authoring model training logic in Code Repositories](#)
4. [Testing inference logic in Code Repositories](#)
5. [Viewing a model and submit it to a modeling objective](#)

2b.1 How to create a code repository for model training

The Code Repositories application in Foundry is a web-based development environment for authoring production-grade data and machine learning pipelines. Foundry provides a templated repository for machine learning called the `Model Training` template.

Action: In the `code` folder you created during the previous step of this tutorial, select **+ New > Code repository**. Your code repository should be named in relation to the model that you are training. In this case, name the repository "median_house_price_model_repo". Select **Model Training** template, then **Initialize**.

The screenshot shows the Palantir Foundry interface. At the top, there's a navigation bar with icons for file operations, a share button, and a three-line menu icon. Below the header, the title "Initialize Repository" is displayed, followed by a placeholder "Name your repository".

The main content area is titled "Data transforms" and contains several sections:

- Data transforms**: Description: Transform and integrate datasets or train machine learning models using your preferred language.
- Model Integration**: Description: Templates for machine learning (ML) and artificial intelligence (AI) in Foundry. This section is highlighted with a red box and contains a dropdown menu labeled "Model Training".
- Language template**: Description: Model Training
- Python Library**: Description: Create a Python library and share it with other pipeline developers.
- Functions**: Description: Parent template for a Functions repository.
- Foundry UDF Definitions**: Description: Templates for creating user-defined functions (UDFs) in Foundry.
- Empty repository**: Description: Create an empty repository and initialize it with a README file.

A blue "Initialize repository" button is located on the right side of the "Language template" section. In the top right corner of the interface, there are two buttons: "EN" and "AB XY".

The model training template contains an example structure that we will adapt for this tutorial. You can expand the files on the left side to see an example project.

Foundry Modeling

This template provides the structure for training a model in Foundry. To train a model in Foundry, you will need to implement two components:

1. Logic to train a **model** in Foundry
2. A **ModelAdapter** - this is the logic that describes how Foundry can interact with the **model** for serialization and inference

A model trained in Foundry can be saved natively as a resource that can be used in:

- the Modeling Objectives application for evaluation and deployment to a REST API or batch deployment environment
- downstream inference builds
- further training or transfer learning to a different application

Saving a trained model in Foundry

Models can be saved to **ModelErrorOutput** with the **publish** method.

```
from transforms.api import transform, Input
from palantir_models.transforms import ModelOutput
from palantir_models.models import ModelVersionChangeType
from model_adapter.example_adapter import ExampleModelAdapter
@transform(
    training_data_input=Input("/path/to/training_data"),
    model_output=ModelErrorOutput("/path/to/model")
)
def compute(training_data_input, model_output):
    trained_model = train_model(training_data_input)

    wrapped_model = ExampleModelAdapter(trained_model)
    model_output.publish(
        model_adapter=wrapped_model,
        change_type=ModelVersionChangeType.MINOR
    )
```

Running Inference in Foundry

To use a model in production, it is recommended to submit the model to the Modeling Objectives application for evaluation, review and hosting. Alternatively, you can use the model in a python transform as

2b.2 How to split feature data for testing and training

The first step in a supervised machine learning project is to split our labeled feature data into separate datasets for training and testing. Eventually, we will want to create performance metrics (estimates of how well our model performs on new data) so we can decide whether this model is good enough to use in a production setting and so we can communicate how much to trust the results of this model with other stakeholders. We must use separate data for this validation to help ensure that the performance metrics are representative of what we will see in the real world.

As such, we are going to write a [Python transform](#) that takes our labeled feature data and splits this into our two training and testing datasets.

```
1 from transforms.api import transform, Input, Output
2
3
4 @transform(
5     features_and_labels_input=Input("<YOUR_PROJECT_PATH>/data/housing_features_and_labels"),
6     training_output=Output("<YOUR_PROJECT_PATH>/data/housing_training_data"),
7     testing_output=Output("<YOUR_PROJECT_PATH>/data/housing_test_data"),
8 )
```

```
9 def compute(features_and_labels_input, training_output, testing_output):
10     # Converts this TransformInput to a PySpark DataFrame
11     features_and_labels = features_and_labels_input.dataframe()
12
13     # Randomly split the PySpark dataframe with 80% training data and 20% testing data
14     training_data, testing_data = features_and_labels.randomSplit([0.8, 0.2], seed=0)
15
16     # Write training and testing data back to Foundry
17     training_output.write_dataframe(training_data)
18     testing_output.write_dataframe(testing_data)
```

Action: Open the `feature_engineering.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the datasets you uploaded in the previous step of this tutorial. Select **Build** at the top left to run the code. You can, optionally, select **Preview** to test this logic on a subset of the data for faster iteration.

The screenshot shows a code editor interface with a sidebar and a main content area. The sidebar on the left lists project files and folders:

- transforms-model-training
- conda_recipe
- src
- main
- model_adapters
- model_training
- feature_engineering.py

The main content area displays the code for `feature_engineering.py`:

```
1  from transforms.api import transform, Input, Output
2
3
4  @transform(
5      features_and_labels_input=Input("/Public/Model Training Tutorial/data/housing_features_and_labels"),
6      training_output=Output("/Public/Model Training Tutorial/data/housing_training_data"),
7      testing_output=Output("/Public/Model Training Tutorial/data/housing_test_data"),
8  )
9  def compute(features_and_labels_input, training_output, testing_output):
10     # Converts this TransformInput to a PySpark DataFrame
11     features_and_labels = features_and_labels_input.dataframe()
12
13     # Randomly split the PySpark dataframe with 80% training data and 20% testing data
14     training_data, testing_data = features_and_labels.randomSplit([0.8, 0.2], seed=0)
15
16     # Write training and testing data back to Foundry
17     training_output.write_dataframe(training_data)
18     testing_output.write_dataframe(testing_data)
19
```

Below the code editor is a build status panel titled "Build - Aug 4, 2023 11:51 AM". It shows the following items:

- Checks • just now
- Build initialization • just now
- feature_engineering.py

The status for "feature_engineering.py" is "Running Checks".

You can continue with 2b.3 while this build executes.

2b.3 How to author model training logic in Code Repositories

Models in Foundry are comprised of two components, model artifacts (the model files produced in a model training job), and a model adapter (a Python class that describes how Foundry should interact with the model artifacts to perform inference).

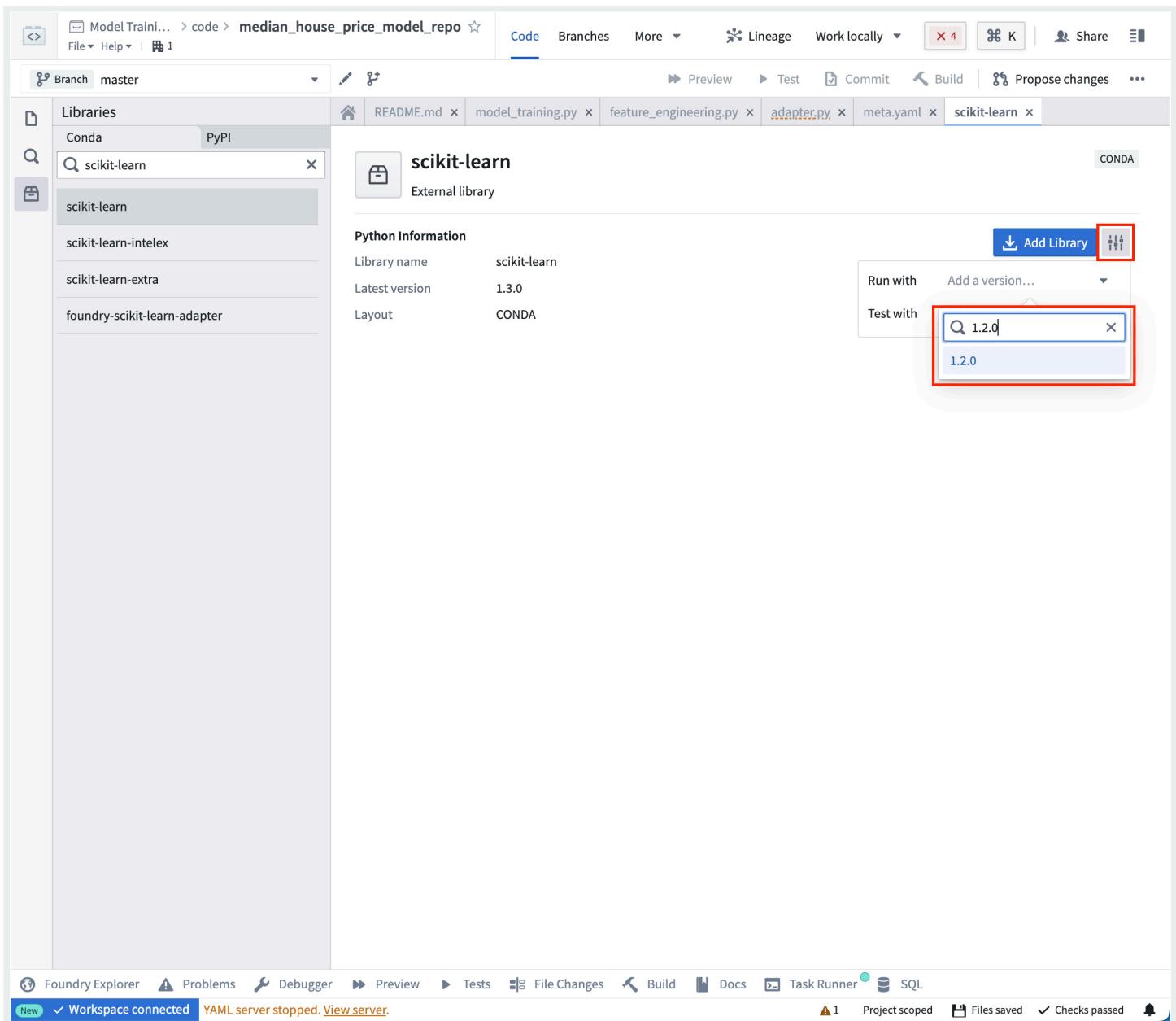
The model training template consists of two modules, `model_training` for the training job and `model_adapters` for the model adapter.

Model dependencies

Model training will almost always require adding Python dependencies that contain model training, serialization, inference, or evaluation logic. Foundry supports adding dependency specifications through conda. These dependency specifications are used to create a resolved Python environment for executing model training jobs.

In Foundry, these resolved dependencies are automatically packaged with your models to ensure that your model automatically has all of the logic required to perform inference (generate predictions). In this example, we will use `pandas` and `scikit-learn` to produce our model and `dill` to save our model.

Action: On the left side bar, select **Libraries** and add dependencies for `scikit-learn = 1.2.0`, `pandas = 1.5.2` and `dill = 0.3.7`. Then select **Commit** to create a resolved Python environment.



Model adapter logic

Model adapters provide a standard interface for all models in Foundry. The standard interface ensures that all models can be used immediately in production applications as Foundry will handle the infrastructure to load the model, its Python dependencies, expose its API, and interface with your model.

To enable this, you must create an instance of a `ModelAdapter` class to act as this communication layer.

There are 4 functions to implement:

1. **Model save and load:** In order to reuse your model, you need to define how your model should be saved and loaded. Palantir provides many default methods of serialization (saving), and in more complex cases you can implement custom serialization logic.
2. **api:** Defines the API of your model and tells Foundry what type of input data your model requires.
3. **predict:** Called by Foundry to provide data to your model. This is where you can pass input data to the model and generate inferences (predictions).

```
1 import palantir_models as pm
2 from palantir_models_serializers import DillSerializer
3
4
5 class SklearnRegressionAdapter(pm.ModelAdapter):
6
7     @pm.auto_serialize(
8         model=DillSerializer()
9     )
10    def __init__(self, model):
11        self.model = model
12
13    @classmethod
14    def api(cls):
15        columns = [
16            ('median_income', float),
17            ('housing_median_age', float),
18            ('total_rooms', float),
19        ]
20        return {"df_in": pm.Pandas(columns)}, \
21               {"df_out": pm.Pandas(columns + [('prediction', float)])}
22
23    def predict(self, df_in):
24        df_in['prediction'] = self.model.predict(
25            df_in[['median_income', 'housing_median_age', 'total_rooms']]
26        )
27        return df_in
```

Action Open the `model_adapters/adapter.py` file and paste the above logic into the file.

The screenshot shows the Foundry IDE interface. The top navigation bar includes 'File', 'Help', 'Branch master', 'Code', 'Branches', 'More', 'Preview', 'Test', 'Commit', 'Build', 'Propose changes', and a three-dot menu. The left sidebar has sections for 'Files' (with a search bar), 'transform-model-training', 'conda_recipe', 'src', 'main', 'model_adapters', 'model_training', and 'README.md'. The main area displays the 'adapter.py' file content:

```
1 import palantir_models as pm
2 from palantir_models_serializers import DillSerializer
3
4
5 class SklearnRegressionAdapter(pm.ModelAdapter):
6     @pm.auto_serialize()
7     def __init__(self, model):
8         self.model = model
9
10    @classmethod
11    def api(cls):
12        columns = [
13            ('median_income', float),
14            ('housing_median_age', float),
15            ('total_rooms', float),
16        ]
17        inputs = {"df_in": pm.Pandas(columns=columns)}
18        outputs = {"df_out": pm.Pandas(columns=columns + [('prediction', float)])}
19        return inputs, outputs
20
21    def predict(self, df_in):
22        predictions = self.model.predict(
23            df_in[['median_income', 'housing_median_age', 'total_rooms']])
24        df_in['prediction'] = predictions
25        return df_in
26
27
28
29
30
```

The bottom navigation bar includes 'Foundry Explorer', 'Problems', 'Debugger', 'Preview', 'Tests', 'File Changes', 'Build', 'Docs', 'Task Runner', 'SQL', and a status bar with 'Workspace connected', 'AIP autocomplete', 'Project scoped', 'Files saved', and 'Checks started running'.

Model training Logic

Now that our dependencies are set and we have written a model adapter, we can train a model in Foundry.

```
1 from transforms.api import transform, Input
2 from palantir_models.transforms import ModelOutput
3 from main.model_adapters.adapter import SklearnRegressionAdapter
4
5
6 @transform(
7     training_data_input=Input("<YOUR_PROJECT_PATH>/data/housing_training_data"),
```

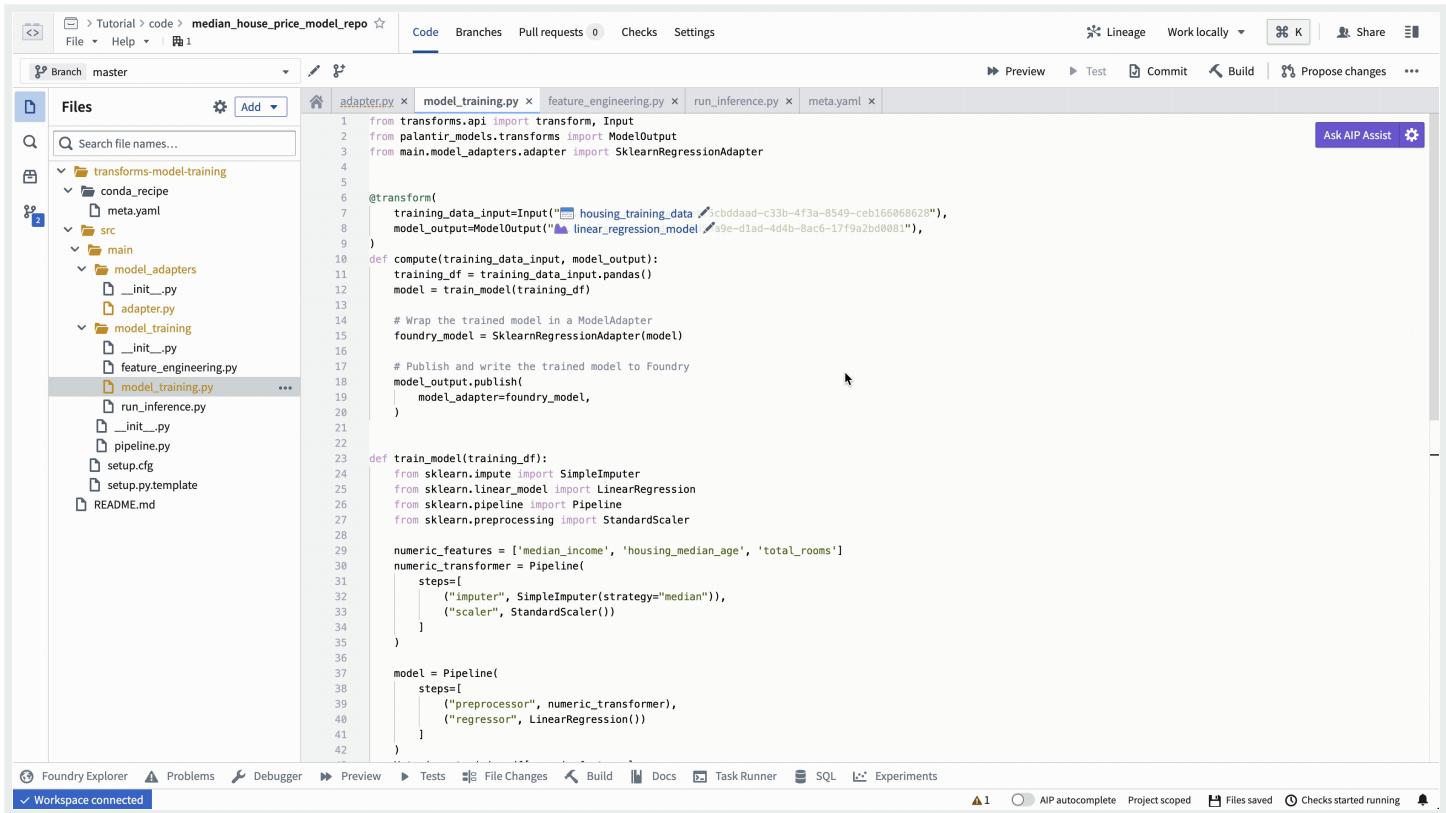
```

8     model_output=ModelOutput("<YOUR_PROJECT_PATH>/models/linear_regression_model"),
9 )
10 def compute(training_data_input, model_output):
11     training_df = training_data_input.pandas()
12     model = train_model(training_df)
13
14     # Wrap the trained model in a ModelAdapter
15     foundry_model = SklearnRegressionAdapter(model)
16
17     # Publish and write the trained model to Foundry
18     model_output.publish(
19         model_adapter=foundry_model
20     )
21
22
23 def train_model(training_df):
24     from sklearn.impute import SimpleImputer
25     from sklearn.linear_model import LinearRegression
26     from sklearn.pipeline import Pipeline
27     from sklearn.preprocessing import StandardScaler
28
29     numeric_features = ['median_income', 'housing_median_age', 'total_rooms']
30     numeric_transformer = Pipeline(
31         steps=[
32             ("imputer", SimpleImputer(strategy="median")),
33             ("scaler", StandardScaler())
34         ]
35     )
36
37     model = Pipeline(
38         steps=[
39             ("preprocessor", numeric_transformer),
40             ("classifier", LinearRegression())
41         ]
42     )
43     X_train = training_df[numeric_features]
44     y_train = training_df['median_house_value']
45     model.fit(X_train, y_train)
46
47     return model

```

Optional: When you are iterating on model training and model adapter logic, it can be useful to test your changes on a subset of your training data before running a build. Select **Preview** at the

top left to test your code.



```
1 from transforms.api import transform, Input
2 from palantir_models.transforms import ModelOutput
3 from main.model_adapters.adapter import SklearnRegressionAdapter
4
5
6 @transform(
7     training_data_input=Input("housing_training_data/cbdaad-c33b-4f3a-8549-ceb1660682"),
8     model_output=ModelOutput("linear_regression_model/a9e-d1ad-4d4b-8ac6-17f9a2bd0081"),
9 )
10 def compute(training_data_input, model_output):
11     training_df = training_data_input.pandas()
12     model = train_model(training_df)
13
14     # Wrap the trained model in a ModelAdapter
15     foundry_model = SklearnRegressionAdapter(model)
16
17     # Publish and write the trained model to Foundry
18     model_output.publish(
19         model_adapter=foundry_model,
20     )
21
22
23 def train_model(training_df):
24     from sklearn.impute import SimpleImputer
25     from sklearn.linear_model import LinearRegression
26     from sklearn.pipeline import Pipeline
27     from sklearn.preprocessing import StandardScaler
28
29     numeric_features = ['median_income', 'housing_median_age', 'total_rooms']
30     numeric_transformer = Pipeline(
31         steps=[
32             ("imputer", SimpleImputer(strategy="median")),
33             ("scaler", StandardScaler())
34         ]
35     )
36
37     model = Pipeline(
38         steps=[
39             ("preprocessor", numeric_transformer),
40             ("regressor", LinearRegression())
41         ]
42     )
43
```

Foundry Explorer Problems Debugger Preview Tests File Changes Build Docs TaskRunner SQL Experiments

Workspace connected AIP autocomplete Project scoped Files saved Checks started running

Action: Open the `model_training/model_training.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the training dataset and model folder you created in the [step 1.1](#). Select **Build** at the top left to run the code.

The screenshot shows a code editor interface for a GitHub repository named 'median_house_price_model_repo'. The 'Code' tab is selected. In the center, there is a code editor window displaying 'model_training.py' with the following content:

```

1  from transforms.api import transform, Input
2  from palantir_models.transforms import ModelOutput
3  from palantir_models.models import ModelVersionChangeType
4  from main.model_adapters.adapter import SklearnRegressionAdapter
5
6  Replace paths with RIDs
7  @transform(
8      training_data_input=Input("housing_training_data", "tutorial/data/housing_training_data"),
9      model_output=ModelOutput("linear_regression_model", "tutorial/models/linear_regression_model"),
10 )
11 def compute(training_data_input, model_output):
12     training_df = training_data_input.pandas()
13     model = train_model(training_df)
14
15     # Wrap the trained model in a ModelAdapter
16     foundry_model = SklearnRegressionAdapter(model)
17
18     # Publish and write the trained model to Foundry
19     model_output.publish(
20         model_adapter=foundry_model,
21         change_type=ModelVersionChangeType.MAJOR
22     )
23
24
25 def train_model(training_df):
26     from sklearn.impute import SimpleImputer
27     from sklearn.linear_model import LinearRegression
28     from sklearn.pipeline import Pipeline
29     from sklearn.preprocessing import StandardScaler
30
31     numeric_features = ['median_income', 'housing_median_age', 'total_rooms']
32     numeric_transformer = Pipeline(
33         steps=[
34             ("imputer", SimpleImputer(strategy="median")),
35             ("scaler", StandardScaler())
36         ]
37     )
38
39     model = Pipeline(
40         steps=[
41             ("numeric_transformer", numeric_transformer),

```

Below the code editor, a 'Build - Aug 30, 2023 5:20 PM' section shows the build status:

- Checks • just now (green)
- Build initialization • just now (green)
- model_training.py (green)

The status bar at the bottom includes links for Foundry Explorer, Problems, Debugger, Preview, Tests, File Changes, Build, Docs, Task Runner, SQL, and Modeling & AI.

(Optional) Log metrics and hyperparameters to a model experiment

[Model experiments](#) is a lightweight framework for logging metrics and hyperparameters produced during a model training run, which can then be published alongside a model and persisted in the model page.

[Learn more about creating and writing to experiments.](#)

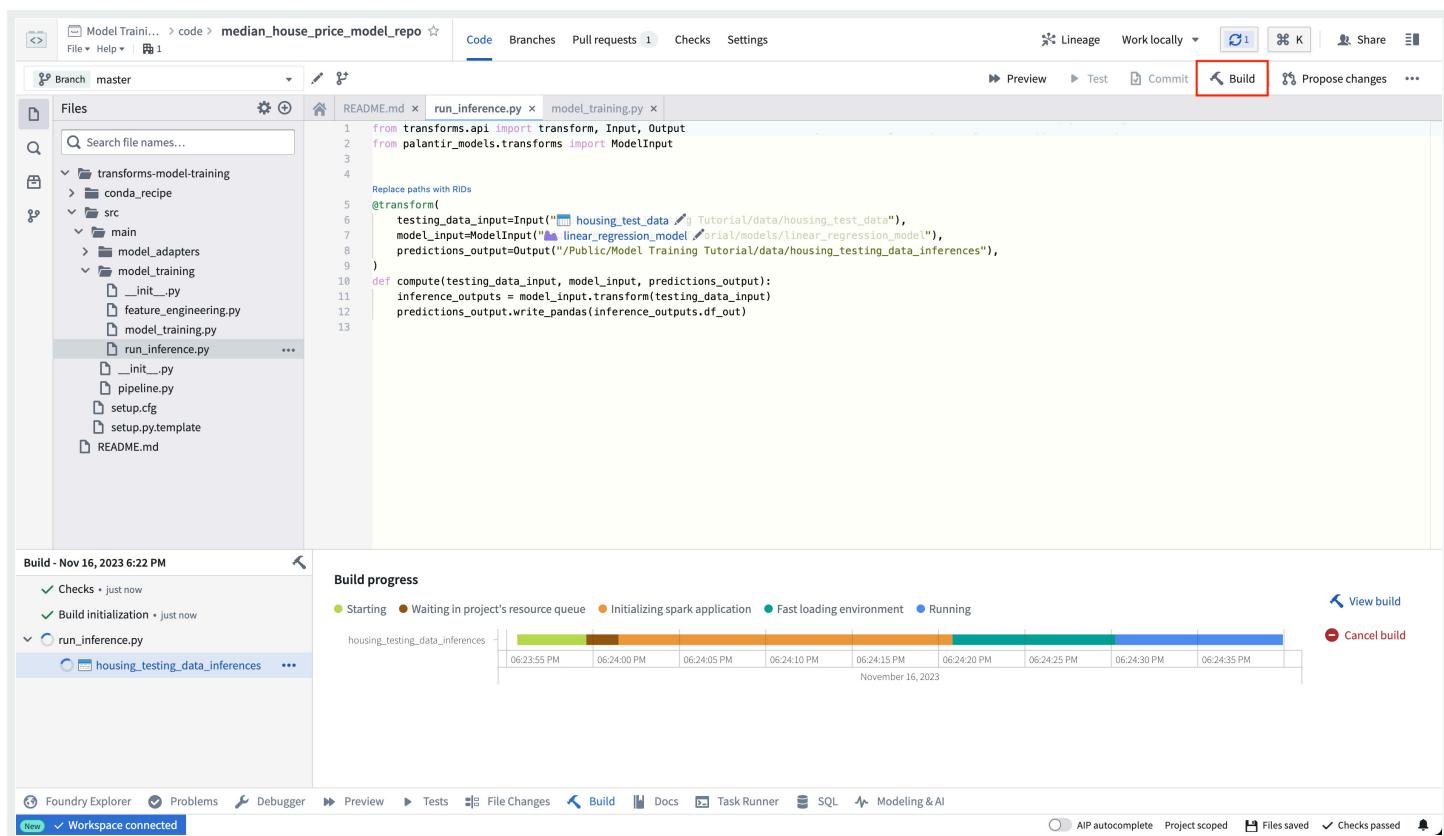
2b.4 How to test inference logic in Code Repositories

Once your model training logic has finished you can generate predictions (also known as inferences) directly in your code repository.

```
1 from transforms.api import transform, Input, Output
2 from palantir_models.transforms import ModelInput
3
4
5 @transform(
6     testing_data_input=Input("<YOUR_PROJECT_PATH>/data/housing_test_data"),
7     model_input=ModelInput("<YOUR_PROJECT_PATH>/models/linear_regression_model"),
8     predictions_output=Output("<YOUR_PROJECT_PATH>/data/housing_testing_data_inferences")
9 )
10 def compute(testing_data_input, model_input, predictions_output):
11     inference_outputs = model_input.transform(testing_data_input)
12     predictions_output.write_pandas(inference_outputs.df_out)
```



Action: Open the `model_training/run_inference.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the model asset and test dataset you created earlier. Select **Build** at the top left to run the code.



The screenshot shows a GitHub Codespace interface with the following details:

- Repository:** median_house_price_model_repo
- Code Tab:** The 'Code' tab is active, showing the contents of the `run_inference.py` file.
- Build Button:** The 'Build' button in the top right corner is highlighted with a red box.
- Build Progress:** The 'Build - Nov 16, 2023 6:22 PM' section shows the build progress for the `housing_testing_data_inferences` task. The progress bar indicates the task is currently running.
- Status Bar:** The status bar at the bottom shows 'AIP autocomplete' and 'Project scoped'.

Once your build is complete, you can review the generated predictions in the build output panel.

The screenshot shows the Foundry IDE interface. At the top, there's a navigation bar with 'Model Traini...' and 'median_house_price_model_repo'. Below it is a toolbar with 'Code', 'Branches', 'Pull requests 1', 'Checks', 'Settings', and various icons for lineage, work locally, share, and propose changes. The main area has tabs for 'README.md', 'run_inference.py', and 'model_training.py'. On the left is a file tree for 'transforms-model-training' and 'src' (including 'main', 'model_adapters', 'model_training', and 'run_inference.py'). The right side shows the code for 'run_inference.py' with syntax highlighting. Below the code editor is a 'Build - Nov 16, 2023 6:22 PM' section with a 'Build finished' status and a preview of the 'housing_testing_data_inferences' CSV file. The CSV data is as follows:

	total_rooms	total_bedrooms	population	households	median_income	median_house_val...	id	prediction
	Double	Double	Double	Double	Double	Double	Long	Double
1	1741	401	753	377	2.0064	77900	12882	168051.55285154055
2	821	170	477	129	3.1500	87500	12879	140331.28830739792
3	2437	438	1430	444	3.8015	169100	12877	193648.69453086433
4	1573	272	142	55	2.1719	420000	12876	107897.97096123580
5	3344	531	1768	541	5.8305	245600	12871	283984.82727439349
6	3030	574	1623	589	5.1356	218700	12869	226073.7
7	1340	235	1336	270	4.2361	179500	12867	221551.5
8	1834	377	1450	347	3.7188	161500	12857	207064.00545748157

2a.5 Optional: Configure live inference

Optionally, this model can be consumed as a REST API via a direct deployment. [Learn how to configure a direct deployment.](#)

2b.6 How to view a model and submit it to a modeling objective

After your model is built you can open the model either by selecting `linear_regression_model` in the `model_training/model_training.py` file or by navigating to the model in the folder structure we created earlier.

```
1  from transforms.api import transform, Input
2  from palantir_models.transforms import ModelOutput
3  from main.model_adapters.adapter import SklearnRegressionAdapter
4
5
6  Replace paths with RIDs
7  @transform(
8      training_data_input=Input("/Public/Model Training Tutorial/data/housing_training_data"),
9      model_output=ModelOutput("linear_regression_model 58b-8ab3-4e5c-b9a0-6f5f7f88f813"),
10     )
11  def compute(
12      trainin
13      model
14      # Wrap the trained model in a ModelAdapter
15      foundry_model = SklearnRegressionAdapter(model)
16
17      # Publish and write the trained model to Foundry
18      model_output.publish(
19          model_adapter=foundry_model,
20      )
21
```

The model view has the source of where the model was trained, the training datasets used to produce this model, the model API, and the model adapter this model was published as. Importantly, you can publish many different versions to the same model; these model versions are available in the dropdown menu on the left sidebar.

As the model version is connected to the specific model adapter used during training, you need to republish and build your model training process to apply any changes to the model adapter logic.

The screenshot shows the Foundry interface for the 'linear_regression_model' asset. The left sidebar shows the asset's name and a dropdown menu. The main area is divided into sections: 'Version 1.0.0', 'Created from', 'Version notes', 'Model version API', and 'Configuration'. The 'Model version API' section is expanded, showing 'Inputs (1)' and 'Outputs (1)'. The 'Inputs (1)' section lists 'input_df' as a required dataset containing 'median_income', 'housing_median_age', and 'total_rooms'. The 'Outputs (1)' section lists 'output_df' as a required dataset containing 'median_income', 'housing_median_age', 'total_rooms', and 'prediction'. The 'Configuration' section shows the adapter module as 'main.model_adapters.adapter', the adapter class as 'SklearnRegressionAdapter', and the adapter code repository as 'median_house_price_model_repo'. A 'Dependencies' section is also present.

Now that we have a model, we can submit that model to our modeling objective for management, evaluation, and release to operational applications.

Action: Select **linear_regression_model** in the code to navigate to the model asset you have created, select **Submit to a Modeling Objective** and submit that model to the modeling objective you created in [step 1 of this tutorial](#). You will be asked to provide a submission name and submission owner. This is metadata that is used to track the model uniquely inside the modeling objective. Name the model `linear_regression_model` and mark yourself as the submission owner.

The screenshot shows the Foundry interface for managing a machine learning model. On the left, there's a sidebar with a tree view under 'models' for 'linear_regression_model'. A selected node shows 'Version 1.0.0 Latest' from 'Aug 15, 2023, 12:39 PM by Foundry'. Below it, there's a note about 'Add a description...'. The main area is titled 'Version 1.0.0' and contains a 'Modeling Objectives' section with a 'Submit to a modeling objective to evaluate and deploy' button. A modal window is open, titled 'Submit to a modeling objective', showing two steps: 'Select modeling objective' (with 'linear_regression_model' selected) and 'Add metadata' (with 'name (name)' entered). At the bottom of the modal are 'Back' and 'Submit' buttons. The background shows other tabs like 'Data', 'Builds', and 'Logs'.

Next step

Now that you have trained a model in Foundry, you can move onto model management, testing, and model evaluation. Here are some examples of additional steps you can take in Modeling Objectives:

- [Automatic model evaluation](#)
- [Configuring checks for model submissions](#)
- [Live and batch inference](#) can also be configured from the modeling objective.

Optionally, you can also [train a model in a Jupyter® notebook with the Code Workspaces application](#) for fast and iterative model development.

© 2025 Palantir Technologies Inc. All rights reserved.

[Cookies Statement ↗](#)

[Privacy Statement ↗](#)

—Do Not Sell or Share My Personal Information