Support ML_PREDICT for Python based model providers

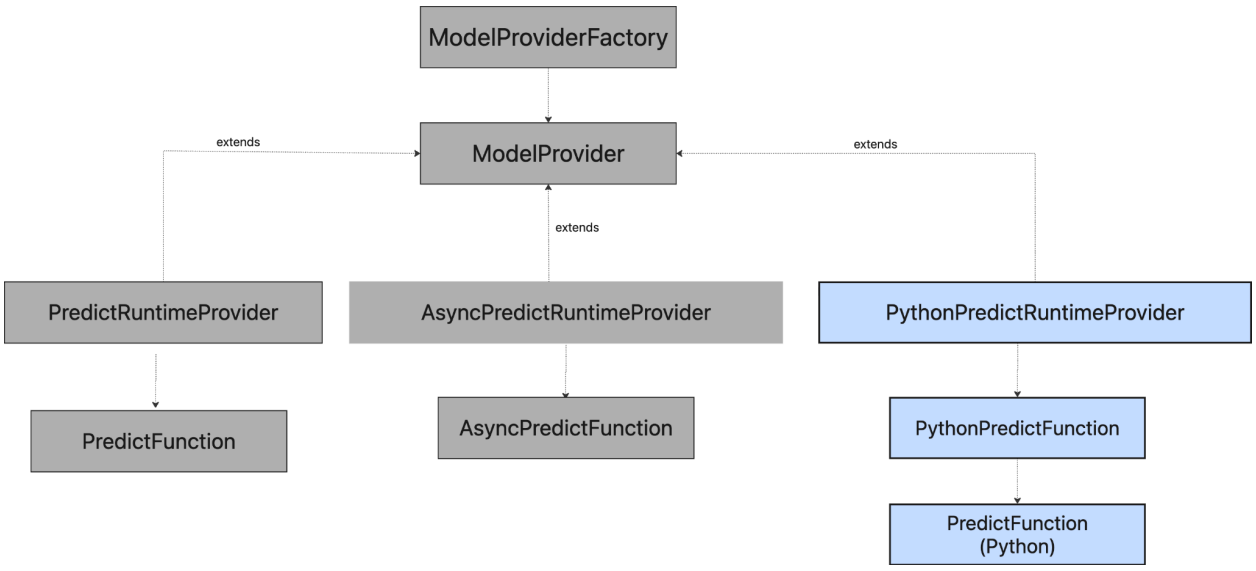
Motivation

[FLIP-525: Model ML_PREDICT, ML_EVALUATE Implementation Design]
introduced interfaces that allow any model provider to be implemented and discovered through SPI. However, these interfaces are difficult to integrate with a model provider in Python. This proposal outlines the design for extending the ML_PREDICT functionality to support Python-based models. The design extends the existing provider-based architecture and adds new interfaces to shield users from the complexities of Java-to-Python inter-process communication.

Public Interfaces

Model runtime

Classes in blue are proposed to be added to support python model providers.



Interface Definitions

```
public interface PythonPredictRuntimeProvider extends ModelProvider {

   PythonPredictFunction getPythonPredictFunction();
}
```

```
public abstract class PythonPredictFunction {
   /**
    * Get the Python class name for this prediction function. This python class
should be an
    * implementation of PredictFunction abstract class in Python
    *
    * @return Fully qualified Python class name
    */
   public abstract String getPythonClass();

   /**
    * Get the model configuration serialized string from the context. This method
should be
    * implemented by the concrete provider.
    *
    * <p>This will be passed to PredictFunction.py, where it's deserialized and
required
    * properties, schema are extracted.
    *
    * @return model configuration string including the properties and serialized
schema
    */
   public abstract String getModelConfig();

   /**
    * Create PythonFunctionInfo for this prediction function.
    *
    * @return PythonFunctionInfo configured for this prediction function
    */
   @Internal
   public PythonFunction createPythonFunction() {
       return PythonFunctionUtils.getPythonFunction(
               getPythonClass(),
               modelContext.getConfiguration(),
```

```
                this.getClass().getClassLoader());
    }
```

```
class PredictFunction(TableFunction):

    """
    Base class for Python-based prediction functions used in the advanced provider
path.
    """
    def open(self, context):
        """
        Initialization method for the function. It is called before the predict
method
        and can be used for one-time setup tasks.

        :param context: A context object that provides access to model properties
and other runtime information.
        """
        if self.model_config_json is None:
            raise RuntimeError("Model config not set")

    def predict(self, data: Row) -> List[Row]:
        """
        Performs prediction on the input data.

        :param data: The input data for prediction.
        :return: A list of rows containing the prediction results.
        """
        raise NotImplementedError

    def set_model_config(self, model_config_json: str):
        """
        Set model configuration programmatically.
        This method will be called by the Java integration layer.
        Parses the JSON configuration and sets the properties directly.
        """
        self.model_config_json = model_config_json
```

Usage

Generic Python Model Provider can be one of the provider implementations packaged with Flink , to support running any custom predict function implemented by the user.

```
CREATE MODEL my_python_model
INPUT (text STRING)
OUTPUT (prediction STRING, confidence FLOAT)
WITH (
   'provider' = 'generic-python',
   'model-directory-path' = '/path/to/your/model',
   'python-predict-class' = 'mymodule.MyCustomModel',
   'properties.model_version' = '1'
);
```

The generic-python provider enables users to deploy and execute custom Python models within Flink. Users package their model files into a directory and specify the path and a primary Python class.

The implementation follows a factory pattern to configure the Python execution environment:

- A GenericPythonModelProviderFactory identifies the 'generic-python' provider. It parses the DDL configuration, including the required **model-directory-path** and **python-predict-class**, as well as optional parameters and custom **properties**.
- The factory creates a GenericPythonModelProvider, which holds the parsed configuration.
- The provider, in turn, instantiates a GenericPythonPredictFunction, which acts as the configuration bridge to the Python runtime. It serializes all necessary details—including the model path, class name, custom properties, and I/O schemas—into a serializable object that is passed to the Python UDF, enabling it to dynamically load and execute the user's model.

Example:

HuggingFace Sentence Transformer

```
  CREATE MODEL simple_python_model
  INPUT (text STRING)
  OUTPUT (sentiment STRING, confidence FLOAT)
  WITH (
      'provider' = 'generic-python',
      'model-directory-path' =
'/path/to/python-models/sentiment-roberta-large-english',
      'properties.model_version' = '1',
      'python-predict-class' =
'models.HuggingFaceSentimentPredictFunction'
  )
```

HuggingFaceSentimentPredictFunction is user-provided predict function implementation.

```
class HuggingFaceSentimentPredictFunction(PyPredictFunction):
   """
   Hugging Face transformers-based sentiment analysis prediction function.
   """

   def open(self, context):
       """
       Initialize the function and load the Hugging Face model.

       :param context: A context object that provides access to model properties
and other runtime information.
       """
       # Initialize the sentiment analysis pipeline
       self.pipeline = pipeline("sentiment-analysis", model=self.model_directory,
return_all_scores=True)

   def predict(self, data: Row) -> List[Row]:
       """
       Perform sentiment prediction using Hugging Face transformers.

       :param data: The input data row for prediction (expects text field)
```

```
        :return: A list of rows containing the prediction results
        """
        text = data[0] if len(data) > 0 else ''
        results = self.pipeline(text)
```

Future Improvements

- Record batching for efficient model prediction. Also, Vectorized table functions for efficient communication of batched records.
- Model download, cache directory setup support.
- Support for more model providers.

Rejected Alternatives:

Test Plan

- All existing tests pass
- Add new unit tests and integration tests for any new code changes