Intro

Welcome to **Jaqpot**, your platform for deploying machine learning models. Jaqpot allows machine learning modellers to upload, manage, and make predictions using machine learning models, ensuring easy access and seamless integration with various tools and services.

This documentation will help you understand how to get started, use the Python client `jaqpotpy`, and interact with the **Jaqpot API**.

Getting Started

Ready to dive in? Start by learning the basics of **Jaqpot** and how you can upload your models and use the platform for predictions. Whether you're a developer or a researcher, we have resources to guide you through each step.

Jaqpotpy

If you prefer working in Python, you'll love **Jaqpotpy**, our Python client for interacting with Jaqpot. It enables you to upload models, make predictions, and automate processes using familiar Python tools.

Jaqpot API

For those who want more control, the **Jaqpot API** provides programmatic access to the platform. This is perfect for integrating Jaqpot functionality into your own applications or pipelines.

Explore the docs to learn how to interact with the API, authenticate requests, and manage your models and data.

Getting started

Installation

You can install JaqpotPy using pip:

```
pip install jaqpotpy
```

Deploy your first model

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.models import SklearnModel
from jaqpotpy import Jaqpot
# Sample data
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4, 5],
    'feature2': [2.1, 3.2, 4.3, 5.4, 6.5],
    'target': [0, 1, 0, 1, 0]
})
# Create dataset for binary classification
dataset = JaqpotpyDataset(
    df=data,
    x_cols=['feature1', 'feature2'],  # Feature columns
    y_cols=['target'],                # Target column
    task='binary_classification'      # Specify the task type
)
# Train a model
model = SklearnModel(
    model =LogisticRegression(),
    dataset = dataset
)
model.fit()
# Upload the pretrained model on Jaqpot
jaqpot = Jaqpot()
jaqpot.login()
model.deploy_on_jaqpot(
    jaqpot=jaqpot,
    name="My first Jaqpot Model",
```

```
    description="This is my first attempt to train and upload a Jaqpot model.",
    visibility="PRIVATE",
)
```

Success!

Voila! Your first model is now online on Jaqpot! Kudos to you, awesome modeler, for taking the first step toward your modeling journey! 🚀🌟

Jaqpotpy

jaqpotpy is a Python library that helps you build models and use the Jaqpot platform. You can use it to create, train and deploy machine learning models.

Key Features:

- **Build Models**: Create and train models with preprocessing and evaluation tools
- **Upload Models**: Upload your trained models to Jaqpot
- **Custom Tools**: Add your own preprocessing steps, featurization and metrics
- **Use Jaqpot**: Connect to Jaqpot API to deploy models and get predictions
- **Deploy**: Get your models ready for production use

Jaqpotpy helps you take your models from your computer to the Jaqpot platform, where others can use them.

📄 ONNX Integration

In this section, we explain Jaqpot's use of ONNX to ensure compatibility across various machine learning mod...

🗃️ Scikit-learn models

6 items

🗃️ PyTorch Geometric models

4 items

🗃️ Full API Reference

27 items

📄 Examples

GitHub Examples

Edit this page

ONNX Integration

In this section, we explain Jaqpot's use of ONNX to ensure compatibility across various machine learning models and highlight the benefits and limitations of this integration.

Overview

Jaqpot uses **ONNX** **(Open Neural Network Exchange)** as a standardized format to allow models from different machine learning libraries to be compatible with our platform. This format helps bridge the compatibility gap, especially for Scikit-Learn and PyTorch models, enabling seamless model deployment through Jaqpot’s API.

Supported Libraries

Currently, Jaqpot fully supports the conversion and deployment of:

- **Scikit-learn Models**: These models work directly and reliably with ONNX, making Scikit-Learn a primary library for Jaqpot.
- **PyTorch Geometric Models**: Support for PyTorch-Geometric models is being developed (specifically for molecular modelling), and users can start to experiment with some Pytorch Graph Neural Network models using ONNX.

ONNX allows users to convert models into a single, standardized format, providing cross-platform compatibility and a smoother integration experience.

Pros and Cons

Pros

- **Compatibility across libraries**: ONNX allows Jaqpot to handle multiple model types by standardizing them into one format.
- **Ease of use**: Once a model is converted to ONNX, users experience minimal hassle, as ONNX abstracts many library-specific differences.
- **Scikit-Learn as first-class citizen**: Jaqpot fully supports Scikit-Learn models with ONNX, making it the preferred library for seamless integration.

- **Backward compatibility**: ONNX ensures that models remain compatible with Jaqpot's API, even as the platform evolves. Even if Jaqpot or scikit-learn updates its API, ONNX models will remain compatible.

Cons

- **Dependency on ONNX features**: Jaqpot's capabilities are limited to the functions ONNX currently supports. If ONNX doesn't support a specific function from a library, Jaqpot may not be able to support it either.
- **Ongoing support for Torch Models**: Jaqpot's compatibility with PyTorch Geometric models is a work in progress, and some advanced Torch features may not yet be supported. Currently, Graph Neural Networks architectures for molecular modelling are supported.

Future directions

Jaqpot aims to enhance ONNX integration further, focusing on extending support for PyTorch and exploring compatibility with other machine learning frameworks as ONNX evolves. Future updates will address additional Torch functionalities and possibly other libraries that can be adapted into Jaqpot's API.

Limitations and alternatives

- **Limitations**: ONNX does not support every function across all libraries, so certain advanced or custom model components may not yet be compatible.
- **Alternatives**: If ONNX does not support a specific feature you need, consider adjusting the model or checking Jaqpot's compatibility updates for future support.

📄 Create a Dataset

This guide demonstrates how to create and work with datasets using jaqpotpy. The JaqpotpyDataset class is ...

📄 Create a Model

This example demonstrates how to create a model using jaqpotpy with a scikit-learn model. The following co...

📄 Upload a Model on Jaqpot

This example demonstrates how to upload a trained model on Jaqpot using the jaqpotpy library. The code bel...

📄 Feature Preprocessing

Using Multiple Featurizers

📄 Evaluate a Model

In this example, we will demonstrate how to evaluate the robustness of a model using jaqpotpy. We will use ...

📄 Domain of Applicability

To demonstrate how to use the domain of applicability (DOA) with Jaqpotpy models, we will create a regressi...

Create a Dataset

This guide demonstrates how to create and work with datasets using `jaqpotpy`. The `JaqpotpyDataset` class is versatile and can handle various types of data, including molecular representations (SMILES) and their descriptors.

Basic Setup

First, import the necessary libraries:

```
import pandas as pd
import numpy as np
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors, MordredDescriptors,
TopologicalFingerprint, MACCSKeysFingerprint
```

Creating a Basic Dataset

For a simple dataset without molecular descriptors:

```
# Sample data
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4, 5],
    'feature2': [2.1, 3.2, 4.3, 5.4, 6.5],
    'target': [0, 1, 0, 1, 0]
})
# Create dataset for binary classification
dataset = JaqpotpyDataset(
    df=data,
    x_cols=['feature1', 'feature2'],  # Feature columns
    y_cols=['target'],                # Target column
    task='binary_classification'      # Specify the task type
)
```

Creating a Dataset with SMILES and Molecular Descriptors

When working with molecular data, the column containing the SMILES, as well as the featurizer should be:

```
# Sample data with SMILES
mol_data = pd.DataFrame({
    'smiles': ['CC', 'CCO', 'CCC', 'CCCl'],
    'temperature': [25, 30, 35, 40],
    'activity': [0.5, 0.7, 0.3, 0.9]
})
# Initialize a molecular descriptor calculator
rdkit_desc = RDKitDescriptors()
# Create dataset with molecular descriptors
mol_dataset = JaqpotpyDataset(
    df=mol_data,
    x_cols=['temperature'],     # Additional feature columns
    y_cols=['activity'],        # Target column
    smiles_cols=['smiles'],     # SMILES column
    task='regression',          # Regression task
    featurizer=rdkit_desc       # Specify the descriptor calculator
)
```

Available Task Types

JaqpotpyDataset supports three types of machine learning tasks:

- `regression`: For predicting continuous values
- `binary_classification`: For two-class classification problems
- `multiclass_classification`: For classification with more than two classes

Available Molecular Descriptors

Jaqpotpy offers four different molecular descriptor calculators:

```
# RDKit descriptors
rdkit_desc = RDKitDescriptors()
# Mordred descriptors
mordred_desc = MordredDescriptors()
# Topological fingerprints
topo_fp = TopologicalFingerprint()
# MACCS keys fingerprints
maccs_fp = MACCSKeysFingerprint()
```

Creating a Multiclass Classification Dataset

```
# Sample data for multiclass classification
multi_data = pd.DataFrame({
    'smiles': ['CC', 'CCO', 'CCC', 'CCCl'],
    'feature1': [1, 2, 3, 4],
    'class': ['A', 'B', 'C', 'A']
})
# Using MACCS keys fingerprints
maccs_fp = MACCSKeysFingerprint()
multi_dataset = JaqpotpyDataset(
    df=multi_data,
    x_cols=['feature1'],
    y_cols=['class'],
    smiles_cols=['smiles'],
    task='multiclass_classification',
    featurizer=maccs_fp
)
```

Important Notes

1. The `smiles_cols` parameter is optional. If not provided, no molecular descriptors will be generated.
2. When using `smiles_cols`, a `featurizer` must be specified.
3. The `task` parameter must match your problem type:
   - Use `regression` for continuous targets
   - Use `binary_classification` for two-class problems
   - Use `multiclass_classification` for multiple classes
4. Feature columns (`x_cols`) can include both molecular and non-molecular features.
5. Target columns (`y_cols`) specify the variable(s) to be predicted.

This dataset object can then be used with the `Sklearn()` Jaqpotpy model classes for training and prediction tasks.

Create a Model

This example demonstrates how to create a model using `jaqpotpy` with a scikit-learn model. The following code will guide you through generating a dataset, training a logistic regression model, and making predictions.

First, we import the necessary libraries:

```
import pandas as pd
from sklearn.datasets import make_classification
from jaqpotpy.datasets import JaqpotpyDataset
from sklearn.linear_model import LogisticRegression
from jaqpotpy.models import SklearnModel
```

Next, we generate a small binary classification dataset:

```
X, y = make_classification(n_samples=100, n_features=4, random_state=42)
```

We then create a DataFrame with the features and target:

```
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Now, we initialize a `JaqpotpyDataset` with the DataFrame:

```
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="binary_classification",
)
```

We wrap the scikit-learn model with Jaqpotpy's `SklearnModel`:

```
jaqpot_model = SklearnModel(dataset=dataset, model=LogisticRegression())
```

Next, we fit the model to the dataset:

```
jaqpot_model.fit()
```

We generate a small prediction dataset:

```
X_test, _ = make_classification(n_samples=5, n_features=4, random_state=42)
```

We create a DataFrame with the features:

```
df_test = pd.DataFrame(X_test, columns=["X1", "X2", "X3", "X4"])
```

We initialize a `JaqpotpyDataset` for prediction:

```
test_dataset = JaqpotpyDataset(
    df=df_test,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=None,
    task="binary_classification",
)
```

Finally, we use the trained model to predict the classes of the new data and the estimate their claissification probabilities and print the predictions:

```
predictions = jaqpot_model.predict(test_dataset)
probabilities = jaqpot_model.predict_proba(test_dataset)
print(predictions)
```

This code snippet covers the entire process from dataset creation to model training and prediction using `jaqpotpy` and scikit-learn's `LogisticRegression`.

Upload a Model on Jaqpot

This example demonstrates how to upload a trained model on Jaqpot using the `jaqpotpy` library. The code below shows the complete process, from generating a dataset to deploying the model on Jaqpot.

First, we generate a small regression dataset and create a DataFrame with the features and target:

```
import pandas as pd
from sklearn.datasets import make_regression
# Generate a small regression dataset
X, y = make_regression(n_samples=100, n_features=4, noise=0.2, random_state=42)
# Create a DataFrame with the generated data
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Next, we initialize a `JaqpotpyDataset` with the DataFrame, specifying the feature columns, target column, and task type:

```
from jaqpotpy.datasets import JaqpotpyDataset
# Initialize a JaqpotpyDataset with the DataFrame
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="regression",
)
```

We then wrap the scikit-learn model with `Jaqpotpy`'s `SklearnModel` and fit the model to the dataset:

```
from sklearn.linear_model import LinearRegression
from jaqpotpy.models import SklearnModel
# Wrap the scikit-learn model with Jaqpotpy's SklearnModel
jaqpot_model = SklearnModel(dataset=dataset, model=LinearRegression())
# Fit the model to the dataset
jaqpot_model.fit()
```

Finally, we upload the trained model to Jaqpot. To upload a model, a Jaqpot account is required. You can create one here. After logging in to Jaqpot, we use the `deploy_on_jaqpot` method to upload the model, providing the model name, description, and visibility settings (PUBLIC or PRIVATE):

```
from jaqpotpy import Jaqpot
# Upload the pretrained model on Jaqpot
jaqpot = Jaqpot()
jaqpot.login()
jaqpot_model.deploy_on_jaqpot(
    jaqpot=jaqpot,
    name="My first Jaqpot Model",
    description="This is my first attempt to train and upload a Jaqpot model.",
    visibility="PRIVATE",
)
```

In this final step, we first create an instance of the `Jaqpot` class and log in. Then, we call the `deploy_on_jaqpot` method on our model, passing in the `Jaqpot` instance along with the model name, description, and visibility settings. This process allows you to easily deploy your trained models on Jaqpot for further use and sharing.

Feature Preprocessing

Using Multiple Featurizers

This guide is about using multiple featurizers and performing feature selection.

First, we import necessary libraries.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.feature_selection import VarianceThreshold
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder, StandardScaler
from sklearn.ensemble import RandomForestRegressor
from jaqpotpy.models import SklearnModel
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors, MACCSKeysFingerprint
```

Create a dataframe with SMILES strings, a categorical variable, temperature, and activity values. SMILES is a unified method to represent chemical structures in the form of a line notation. For more info about SMILES check this Wikipedia page.

```
data = pd.read_csv("https://github.com/ntua-unit-of-control-and-
informatics/jaqpot-google-colab-examples/raw/doc/JAQPOT-
425/Sklearn_jupyter_examples/datasets/regression_smiles_categorical.csv")
```

Define a list of desired featurizers.

```
from jaqpotpy.descriptors import RDKitDescriptors, MACCSKeysFingerprint
featurizers = [RDKitDescriptors(), MACCSKeysFingerprint()]
```

We then pass this list of featurizers to the `JaqpotpyDataset` object when creating the training dataset:

```
train_dataset = JaqpotpyDataset(
    df=data,
    x_cols=["cat_col", "temperature"],
```

```
    y_cols=["activity"],
    smiles_cols=["smiles"],
    task="regression",
    featurizer=featurizers,
)
```

By providing a list of featurizers, the dataset will generate both RDKit descriptors and MACCS keys fingerprints for the SMILES data, resulting in a more comprehensive set of molecular features.

Feature Selection

In the second script, we demonstrate the use of feature selection. After creating the `JaqpotpyDataset` object, we apply a feature selection technique using the `select_features()` method:

```
# Use VarianceThreshold to select features with a minimum variance of 0.1
FeatureSelector = VarianceThreshold(threshold=0.1)
train_dataset.select_features(
    FeatureSelector,
    ExcludeColumns=["cat_col"],  # Explicitly exclude the categorical variable
)
```

This will apply the VarianceThreshold feature selector to the dataset, excluding the "cat_col" variable, which is a categorical feature that cannot be included in the selection process.

Alternatively, you can directly select specific columns by name using the `SelectColumns` argument:

```
myList = [
    "temperature",
    "cat_col",
    "MaxAbsEStateIndex",
    "MaxEStateIndex",
    "MinAbsEStateIndex",
    "MinEStateIndex",
    "SPS",
    "MolWt",
    "HeavyAtomMolWt",
]
train_dataset.select_features(SelectColumns=myList)
```

This method allows you to manually choose the features you want to include in the model, which can be useful if you have domain knowledge about the most relevant variables.

Feature Preprocessing

In the first script, we define a preprocessing pipeline for the feature columns and the target column:

```
# Preprocessing for the feature columns
double_preprocessing = [
    ColumnTransformer(
        transformers=[
            ("OneHotEncoder", OneHotEncoder(), ["cat_col"]),
        ],
        remainder="passthrough",
        force_int_remainder_cols=False,
    ),
    StandardScaler(),  # Standard scaling for numerical features after encoding
]
# Preprocessing for the target column
single_preprocessing = MinMaxScaler()
```

The `double_preprocessing` pipeline first applies OneHotEncoder to the categorical "cat_col" feature, then applies StandardScaler to the numerical features (including the encoded categorical variable).

The `single_preprocessing` pipeline applies MinMaxScaler to the target variable "activity".

We then pass these preprocessing pipelines to the `SklearnModel` object:

```
jaqpot_model = SklearnModel(
    dataset=train_dataset,
    model=RandomForestRegressor(random_state=42),
    preprocess_x=double_preprocessing,
    preprocess_y=single_preprocessing,
)
jaqpot_model.fit()
```

This ensures that the feature and target variables are properly preprocessed before being used to train the machine learning model.

By using multiple featurizers, feature selection, and feature preprocessing, you can create more robust and effective machine learning models with JaqpotPy.

Evaluate a Model

In this example, we will demonstrate how to evaluate the robustness of a model using `jaqpotpy`. We will use a RandomForestRegressor model and perform various evaluations including cross-validation, external evaluation, and a randomization test.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from jaqpotpy.models import SklearnModel
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors
```

We start by creating a sample dataset with molecular structures represented as SMILES strings, along with temperature and activity values.

```
# Create sample data
data = pd.DataFrame(
    {
        "smiles": ["CC", "CCO", "CCC", "CCCl",
        "CCBr", "COC", "CCOCC", "CCCO",
        "CCCC", "CCCCCC",
    ],
        "temperature": np.random.randint(20, 37, size=10),

        "activity": [80, 81, 81, 84, 83.5,
        83, 89, 90, 91, 97,
    ],
    }
)
```

Next, we prepare the dataset for training using `JaqpotpyDataset` and `RDKitDescriptors` for feature extraction.

```
featurizer = RDKitDescriptors()
# Prepare the dataset for training with Jaqpotpy
train_dataset = JaqpotpyDataset(
    df=data,
    x_cols=["temperature"],
    y_cols=["activity"],
```

```
    smiles_cols=["smiles"],
    task="regression",
    featurizer=featurizer,
)
```

We then initialize a RandomForestRegressor model and wrap it with `SklearnModel` from `jaqpotpy`. The model is trained on the prepared dataset.

```
model = RandomForestRegressor(random_state=42)
jaqpot_model = SklearnModel(dataset=train_dataset, model=model)
jaqpot_model.random_seed = 1231
jaqpot_model.fit()
```

To estimate the model's performance, we perform cross-validation on the training data.

```
# Perform cross-validation on the training data
jaqpot_model.cross_validate(train_dataset, n_splits=10)
```

We define a test dataset for external evaluation and prepare it using `JaqpotpyDataset`.

```
# Define test data for external evaluation
X_test = pd.DataFrame(
    {
        "smiles": ["CCCOC", "CO"],
        "temperature": [27.0, 22.0],
        "activity": [89.0, 86.0],
    }
)
# Prepare the test dataset with Jaqpotpy
test_dataset = JaqpotpyDataset(
    df=X_test,
    smiles_cols="smiles",
    x_cols=["temperature"],
    y_cols=["activity"],
    task="regression",
    featurizer=featurizer,
)
```

We evaluate the model on the test dataset to assess its performance on new/unseen data.

```
# Evaluate the model on the test dataset
jaqpot_model.evaluate(test_dataset)
```

```
predictions = jaqpot_model.predict(test_dataset)
print(predictions)
```

Finally, we conduct a randomization test to assess the model's robustness against randomization of target labels.

```
# Conducts a randomization test to assess the model's robustness
jaqpot_model.randomization_test(
    train_dataset=train_dataset,
    test_dataset=test_dataset,
    n_iters=10,
)
```

Domain of Applicability

To demonstrate how to use the domain of applicability (DOA) with Jaqpotpy models, we will create a regression model using scikit-learn's Linear Regression and evaluate the DOA using Jaqpotpy's Leverage, BoundingBox, and MeanVar methods.

First, we generate a small regression dataset with 100 samples, each having 4 features and some noise. We then create a DataFrame with the features stored in columns "X1", "X2", "X3", "X4" and the target in column "y".

```
import pandas as pd
from sklearn.datasets import make_regression
from jaqpotpy.datasets import JaqpotpyDataset
from sklearn.linear_model import LinearRegression
from jaqpotpy.models.sklearn import SklearnModel
from jaqpotpy.doa import Leverage, BoundingBox, MeanVar
X, y = make_regression(n_samples=100, n_features=4, noise=0.2, random_state=42)
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Next, we initialize a `JaqpotpyDataset` with the DataFrame, specifying the feature columns and the target column, and define the task as regression.

```
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="regression",
)
```

We then wrap the scikit-learn model with Jaqpotpy's `SklearnModel`, using Linear Regression as the regression model and specifying the DOA methods: Leverage, BoundingBox, and MeanVar.

```
jaqpot_model = SklearnModel(
    dataset=dataset,
    model=LinearRegression(),
    doa=[Leverage(), BoundingBox(), MeanVar()],
)
```

After fitting the model to the dataset, we generate a small prediction dataset with 5 samples, each having 4 features, and create a DataFrame with the features.

```
jaqpot_model.fit()
X_test, _ = make_regression(n_samples=5, n_features=4, noise=0.2, random_state=42)
df_test = pd.DataFrame(X_test, columns=["X1", "X2", "X3", "X4"])
```

We initialize a `JaqpotpyDataset` for prediction, specifying the feature columns and setting `y_cols` to `None` since we are predicting.

```
test_dataset = JaqpotpyDataset(
    df=df_test,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=None,
    task="regression",
)
```

Finally, we use the trained model to check if the test data are in or out of the domain of applicability, using `predict_doa` method.

```
doa_predictions = jaqpot_model.predict_doa(test_dataset)
print(doa_predictions)
```

This example demonstrates how to use Jaqpotpy to evaluate the domain of applicability for a regression model, ensuring that predictions are made within the reliable range of the model.

📄 Create a Dataset

This example demonstrates how to create a Molecular Graph Neural Network dataset with jaqpotpy. SmilesGr...

📄 Create a Model

This document demonstrates how to create a Graph Neural Network model using JaqpotPy with a specific arc...

📄 Train and Evaluate a Model

In this section, we continue building on the previous example by defining the optimizer, loss function, and tra...

📄 Upload a Model

Import Required Libraries

Create a Dataset

This example demonstrates how to create a Molecular Graph Neural Network dataset with `jaqpotpy`. `SmilesGraphFeaturizer` is used to create graph based features on SMILES input, while `SmilesGraphDataset` is the dataset class for Graph Neural Networks training.

Basic Setup

We first import the necessary libraries:

```
import torch
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
```

Demo Data

We create a small dataset for demonstration purposes with SMILES strings and their corresponding target values. This example uses a classification target.

```
smiles_list = [
    "CC(=O)OC1=CC=CC=C1C(=O)O",  # Aspirin
    "CN1C=NC2=C1C(=O)N(C(=O)N2C)C",  # Caffeine
    "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O",  # Ibuprofen
    "CC(C)CCCC(C)C1CCC2C1(CCC3C2CCC4=C3CC=C4)O",  # Cholesterol
    "OC[C@H]1OC(=O)C(O)=C1O"  # Vitamin C (Ascorbic acid)
]
y = [0, 1, 0, 1, 1] # Binary Activity
```

Featurization

To convert the SMILES strings into graph-based features, we use the `SmilesGraphFeaturizer`. This featurizer can include edge features and supports various atom and bond features.

Default featurizer

We create an instance of `SmilesGraphFeaturizer` and configure it with default settings for graph features (nodes and edges).

```
featurizer = SmilesGraphFeaturizer(include_edge_features=True)
featurizer.set_default_config()
```

Custom hardcoded features

Node features and their coresponding values can be hardcoded by user if it is supported by the featurizer. Name of features and feature values should be provided for one hot encoding Names of features and their RDKit functions can be obtained with:

```
featurizer.SUPPORTED_ATOM_FEATURES()
featurizer.SUPPORTED_BOND_FEATURES()
```

Example of custom graph features

```
featurizer = SmilesGraphFeaturizer(include_edge_features=True)
# First argument is the feature name
# Secnd argument is the feature possible values
featurizer.add_atom_feature("symbol", ["C", "O", "N", "F", "Cl", "Br", "I"])
featurizer.add_atom_feature("total_num_hs", [0, 1, 2, 3, 4])
# If possible values are not provided, the feature is not one one-hot encoded
featurizer.add_atom_feature("formal_charge")
```

Dataset Preparation

We can create datasets for training, validation, and testing using the featurizer. The datasets include the SMILES strings, target values, and the configured featurizer.

```
dataset = SmilesGraphDataset(
    smiles=smiles, y=y, featurizer=featurizer
)
```

Precompute Featurization

For small datasets, it is recommended to precompute the featurization to save time during training.

```
```

```
dataset.precompute_featurization()
```

Edit this page

Create a Model

This document demonstrates how to create a Graph Neural Network model using JaqpotPy with a specific architecture. The example assumes you have already preprocessed your data and have a dataset ready to use.

Basic Setup

We first import the necessary libraries:

```
import torch
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
from jaqpotpy.models.torch_geometric_models.graph_neural_network import
GraphSageNetwork, GraphAttentionNetwork
```

Define the Graph Neural Network Architecture

In this example, we use a GraphSageNetwork architecture. The network takes node features from a featurizer and creates a model with specified input dimensions, hidden layers, and output dimensions. Currently Jaqpotpy library supports the following architectures:

- GraphConvolutionNetork
- GraphSageNetwork
- GraphAttentionNetwork
- GraphTransformerNetwork

Graph Convolution and Graph Sage architectures proccess only node attributes. You can easily create a model like the example below:

```
node_features = featurizer.get_num_node_features()
model = GraphSageNetwork(
    input_dim=node_features,
    hidden_layers=1,
    hidden_dim=4,
```

```
    output_dim=1,
    activation=torch.nn.ReLU(),
    dropout_proba=0.1,
    batch_norm = False,
    seed = 42,
    pooling="mean",
)
```

- input_dim: Number of input neurons, determined by the number of node features.
- hidden_layers: Number of hidden layers in the network.
- hidden_dim: Number of neurons in each hidden layer.
- output_dim: Number of output neurons (default set to 1 for binary classification or regression).
- activation: Activation function, specified using PyTorch (e.g., ReLU).
- pooling: Graph pooling method (options: mean, add, max).
- dropout_proba: Dropout probability for regularization.

Graph Attention and Graph Transformer architectures can proccess both node and edge attributes (optionally)

```
node_features = featurizer.get_num_node_features()
edge_features = featurizer.get_edge_features()
model = GraphAttentionNetwork(
    input_dim=node_features,
    hidden_layers=1,
    hidden_dim=4,
    output_dim=1,
    activation=torch.nn.ReLU(),
    dropout_proba=0.1,
    batch_norm = False,
    seed = 42,
    pooling="mean",
    edge_dim = edge_features,
    heads = 2
)
```

Extra arguments:

- edge_dim: Number of edge features or None
- heads: Attention heads for each neuron in hidden layers

Train and Evaluate a Model

In this section, we continue building on the previous example by defining the optimizer, loss function, and training procedure for our graph neural network (GNN) model using `jaqpotpy`. We utilize PyTorch-based components to configure the training process.

Import Required Libraries

We first import the necessary libraries:

```
import torch
from torch_geometric.loader import DataLoader
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
from jaqpotpy.models.torch_geometric_models.graph_neural_network import
GraphSageNetwork, GraphAttentionNetwork
from jaqpotpy.models.trainers.graph_trainers import BinaryGraphModelTrainer,
RegressionGraphModelTrainer
```

Binary Classification

PyTorch Configuration

We begin by defining the optimizer and loss function from Pytorch. In this example, we use the Adam optimizer and a binary cross-entropy loss function with logits, as the task is a binary classification problem. You can choose any type of optimizer that comes with PyTorch.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss = torch.nn.BCEWithLogitsLoss()
```

Trainer Configuration

We use the BinaryGraphModelTrainer from jaqpotpy to handle the training process. This trainer requires the model, number of epochs, optimizer, and loss function as inputs. Optionally, a learning rate scheduler can also be provided.

```
trainer = BinaryGraphModelTrainer(
    model=model, # Jaqpotpy Graph Model
    n_epochs=20,
    optimizer=optimizer,
    loss_fn=loss,
    scheduler=None
)
```

Data Loaders

We use PyTorch Geometric's DataLoader to load the training, validation, and test datasets in mini-batches. This helps in efficiently feeding data into the model during training and evaluation. The datasets are defined in `Create a dataset` page.

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Train the Model

The model is trained using the training and validation data loaders.

```
trainer.train(train_loader, val_loader)
```

Evaluate the Model

Finally, the model is evaluated on the test dataset to obtain the loss, metrics, and confusion matrix.

```
loss, metrics, conf_matrix = trainer.evaluate(test_loader)
```

Regression

PyTorch Configuration

The main difference here is only the loss function that needs to be for regression specific tasks.

```
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss = torch.nn.MSELoss()
```

Trainer Configuration

We use the RegressionModelTrainer from jaqpotpy to handle the training process. This trainer requires the same arguments as the BinaryGraphModelTrainer

```
trainer = RegressionGraphModelTrainer(
    model=model, # Jaqpotpy Graph Model
    n_epochs=20,
    optimizer=optimizer,
    loss_fn=loss,
    scheduler=None
)
```

DataLoaders

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Train the Model

The model is trained using the training and validation data loaders.

```
trainer.train(train_loader, val_loader)
```

Evaluate the Model

Finally, the model is evaluated on the test dataset to obtain the loss and metrics.

```
loss, metrics = trainer.evaluate(test_loader)
```

jaqpotpy.utils package

Submodules

jaqpotpy.utils.data_utils module

jaqpotpy.utils.fragment_utils module

jaqpotpy.utils.geometry_utils module

jaqpotpy.utils.installed_packages module

jaqpotpy.utils.molecule_feature_utils module

jaqpotpy.utils.pdbqt_utils module

jaqpotpy.utils.pytorch_utils module

jaqpotpy.utils.rdkit_utils module

jaqpotpy.utils.seeding module

jaqpotpy.utils.types module

jaqpotpy.utils.url_utils module

Module contents

Examples

GitHub Examples

Find a comprehensive collection of usage examples in our main repository:
https://github.com/ntua-unit-of-control-and-informatics/jaqpotpy/tree/main/examples

Google Colab Examples

For interactive, runnable examples, check out our Google Colab repository:
https://github.com/ntua-unit-of-control-and-informatics/jaqpot-google-colab-examples

These Colab notebooks provide step-by-step tutorials and practical demonstrations of Jaqpotpy features in an interactive environment. You can run the examples directly in your browser without any local setup, making it perfect for quick experimentation and learning.

Jaqpot API

The Jaqpot API is a service that handles model management, deployment and predictions. It receives models through its endpoints, manages them, and can run predictions for them via the API.

Key Features:

- **Model Storage**: Store and version your machine learning models
- **Deployment:** Deploy models for production use
- **Predictions:** Get predictions through HTTP endpoints
- **User Management**: Handle users, organizations and model sharing
- **SDKs**: Use our Python SDK (jaqpotpy) to interact with the API - more SDKs coming soon

We maintain SDKs to help you use the API directly in your code. Currently available in Python, with more languages planned.

🗃️ Authentication

1 item

🗃️ SDKs

3 items

🗃️ Documentation

3 items

📄 Create an API key

API keys are used to authenticate requests to the Jaqpot API. You can create an API key for your account and ...

Create an API key

API keys are used to authenticate requests to the **Jaqpot API**. You can create an API key for your account and use it to access Jaqpot programmatically by visiting the API Key dashboard page.

Once an API key is created, you will receive a **client key** and a **client secret**.

Keep these credentials secure and do not share them with anyone.

How to use the API Key

To authenticate API requests, include the **client key** as the username and the **client secret** as the password in the headers `X-Api-Key` and `X-Api-Secret` of your requests.

Example Request

```
curl -X GET "https://api.jaqpot.org/v1/models" \
  -H "X-Api-Key: <client_key>" \
  -H "X-Api-Secret: <client_secret>"
```

or use one of the SDKs

Edit this page

📄 Python SDK

The jaqpotpy Python SDK provides a streamlined interface for interacting with Jaqpot's predictive models and...

📄 Kotlin/Java SDK

The SDK lets you use Jaqpot models and predictions in your Java or Kotlin applications.

📄 QSAR Toolbox Python SDK

The jaqpotpy SDK provides seamless access to QSAR Toolbox, a comprehensive software application designe...

Python SDK

The jaqpotpy Python SDK provides a streamlined interface for interacting with Jaqpot's predictive models and services within Python environments. With this SDK, users can easily retrieve models, make predictions (both synchronously and asynchronously), manage datasets, and handle model sharing across organizations. The SDK simplifies the process of integrating Jaqpot's machine learning capabilities into Python applications, supporting both individual predictions and batch processing through CSV files.

Authentication

Before using the SDK, you'll need to obtain API credentials from Jaqpot. To get your API keys:

1. Log in to app.jaqpot.org
2. Click on the account icon in the top right corner
3. Select "API keys" from the dropdown menu

The API keys are valid for 6 months from their generation date. Please store these keys securely, as they cannot be retrieved from Jaqpot after initial generation. If your keys are lost or expired, you will need to generate new ones by following the same steps above.

There are multiple ways to set up your API credentials as environment variables for Jaqpotpy. Here are presented 2 common methods. Using these methods, your API credentials will be automatically loaded and set as HTTP headers (`X-Api-Key` and `X-Api-Secret`) required for authentication with the Jaqpot service.

Method 1: Using Environment Variables Directly

You can set environment variables before running your script:

```
# Linux/MacOS
export JAQPOT_API_KEY='your_api_key_here'
export JAQPOT_API_SECRET='your_api_secret_here'
python your_script.py
# Windows (Command Prompt)
set JAQPOT_API_KEY=your_api_key_here
```

```
set JAQPOT_API_SECRET=your_api_secret_here
python your_script.py
```

Method 2: Using a .env file

Alternatively, you can create a `.env` file in your project directory with the following content:

```
JAQPOT_API_KEY='your_api_key_here'
JAQPOT_API_SECRET='your_api_secret_here'
```

Then load these variables in your Python code:

```
from dotenv import load_dotenv
load_dotenv(".env") # Returns True if successful
```

Note 1: If `load_dotenv()` returns False, verify that your `.env` file is properly formatted and in the correct location.

Note 2: Remember to add `.env` to your .gitignore file to keep your credentials secure.

Initializing the Client

After setting up your environment variables, initialize the Jaqpot client:

```
from jaqpotpy.api.jaqpot_api_client import JaqpotApiClient
jaqpot = JaqpotApiClient()
```

Retrieve a model by ID

To retrieve a model by its ID, you can use the `get_model_by_id` method. Here is an example:

```
model = jaqpot.get_model_by_id(model_id=1886)
print(model)
```

Get shared models with organizations

To get the models that you share with Jaqpot organizations, you can use the `get_shared_models` method. Here is an example:

```
shared_models = jaqpot.get_shared_models()
print(shared_models)
```

Synchronous prediction

To take a synchronous prediction with a model, you can use the `predict_sync` method. In synchronous prediction, the http call waits until the prediction is complete before moving on. This is useful for quick predictions where you need an immediate result. Use the predict_sync method as shown below:

```
input_data = [{"SMILES": "CC", "X1": 1, "X2": 2, "Cat_col": "CAT_1"}]
prediction = jaqpot.predict_sync(model_id=1886, dataset=input_data)
print(prediction)
```

In this case, the program will wait until predict_sync returns the prediction results, which are then printed.

Asynchronous prediction

To take an asynchronous prediction with a model, you can use the `predict_async` method. In asynchronous prediction, the http call doesn't wait for the prediction to finish, allowing it to continue running other tasks. This is useful for longer predictions or when handling multiple requests. Use the predict_async method as follows:

```
input_data = [{"SMILES": "CC", "X1": 1, "X2": 2, "Cat_col": "CAT_1"}]
prediction_dataset_id = jaqpot.predict_async(model_id=1886, dataset=input_data)
```

Get a dataset by ID

To retrieve a dataset by its ID, you can use the `get_dataset_by_id` method. Here is an example:

```
results = jaqpot.get_dataset_by_id(dataset_id=prediction_dataset_id)
print(results)
```

Prediction with a model and a CSV File

To take a prediction with a model using a CSV file, you can use the `predict_with_csv_sync` method. Here is an example:

```
csv_path = "ADD_A_CSV_PATH.csv"
prediction = jaqpot.predict_with_csv_sync(model_id=1886, csv_path=csv_path)
print(prediction)
```

Kotlin/Java SDK

The SDK lets you use Jaqpot models and predictions in your Java or Kotlin applications.

Get your API Keys

You'll need an API key and secret from your Jaqpot account. Create them by following the API keys guide.

The API keys are valid for 6 months from their generation date. Please store these keys securely, as they cannot be retrieved from Jaqpot after initial generation. If your keys are lost or expired, you will need to generate new ones by following the same steps above.

Add to your project

Maven central page: https://central.sonatype.com/artifact/org.jaqpot/kotlin-sdk

For Gradle (Kotlin DSL):

```
implementation("org.jaqpot:kotlin-sdk:0.4.0") // or the latest version of the SDK
```

For Maven:

```
<dependency>
    <groupId>org.jaqpot</groupId>
    <artifactId>kotlin-sdk</artifactId>
    <version>0.4.0</version>
</dependency>
```

Basic usage

Make predictions in Java:

```
JaqpotApiClient jaqpotApiClient =
    new JaqpotApiClient(System.getenv("JAQPOT_API_KEY"),
System.getenv("JAQPOT_API_SECRET"));
Dataset dataset = jaqpotApiClient
        .predictSync(
                modelId,
                List.of(
                        Map.of("X1", "1", "X2", "2", "X3", "3", "X4", "4")
                )
        );
System.out.println(dataset)
```

or in Kotlin:

```
val jaqpotApiClient =
    JaqpotApiClient(System.getenv("JAQPOT_API_KEY"),
System.getenv("JAQPOT_API_SECRET"))
val dataset = jaqpotApiClient.predictSync(
    modelId,
    listOf(
        mapOf("X1" to "1", "X2" to "2", "X3" to "3", "X4" to "4")
    )
)
println(dataset)
```

Setting API Keys

Use environment variables:

```
export JAQPOT_API_KEY=your_key_here
export JAQPOT_API_SECRET=your_secret_here
```

Or pass them directly to the client:

```
val client = JaqpotApiClient("your-key", "your-secret")
```

Error handling

The SDK throws JaqpotSDKException on errors like:

- Wrong API credentials
- Model not found
- Failed predictions
- Network issues

QSAR Toolbox Python SDK

The jaqpotpy SDK provides seamless access to QSAR Toolbox, a comprehensive software application designed for filling data gaps in toxicity data. Through the SDK, users can interact with three main components of QSAR Toolbox:

1. Calculators: Tools that compute various molecular descriptors and properties
2. Models: QSAR predictive models for different endpoints
3. Profilers: Components that analyze chemical structures and identify structural features

This integration allows users to programmatically access QSAR Toolbox functionalities, perform calculations, make predictions, and analyze chemical structures directly from their Python environment. All components are accessible through simple API calls, requiring only SMILES notation as input for predictions.

QSAR Toolbox

To retrieve a component (model, profiler or calculator) of QSAR Toolbox, you can retrieve a model by its ID using the `get_model_by_id` method. Here is an example:

```
# To get all calculators of QSAR Toolbox:
calculators = jaqpot.get_model_by_id(model_id=6)
# To get all models of QSAR Toolbox:
models = jaqpot.get_model_by_id(model_id=1837)
# To get all profilers of QSAR Toolbox
profilers = jaqpot.get_model_by_id(model_id=1842)
```

QSAR Toolbox Calculator

To take a prediction with any QSAR Toolbox calculator, you can use the `qsartoolbox_calculator_predict_sync` method. Here is an example:

```
prediction = jaqpot.qsartoolbox_calculator_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    calculator_guid="1804a854-9041-4495-9931-7414c22a5e49"
)
print(prediction)
```

QSAR Toolbox Model

To take a prediction with any QSAR Toolbox model, you can use the `qsartoolbox_qsar_model_predict_sync` method as:

```
prediction = jaqpot.qsartoolbox_qsar_model_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    qsar_guid="c377150b-77ae-4f99-be14-357b85dd8d1f",
)
print(prediction)
```

QSAR Toolbox Profiler

To take a prediction with any QSAR Toolbox profiler, you can use the `qsartoolbox_profiler_predict_sync` method. Here is an example:

```
prediction = jaqpot.qsartoolbox_profiler_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    profiler_guid="723eb011-3e5b-4565-9358-4c3d8620ba5d"
)
print(prediction)
```

Documentation

Browse our complete API documentation through Swagger UI. Here you'll find all available endpoints, request/response examples, and authentication details. The API follows REST principles and returns JSON responses.

Visit the Jaqpot API Swagger documentation to try out the endpoints directly in your browser.

For easier integration, check out our SDKs that wrap these API calls into your favorite programming language.

📄 Full API reference

📄 Swagger API Documentation

Browse our complete API documentation through Swagger UI. Here you'll find all available endpoints, request...

📄 Prediction Workflow

When making predictions with Jaqpot models, the process is asynchronous. Here's how it works:

Getting started

Installation

You can install JaqpotPy using pip:

```
pip install jaqpotpy
```

Deploy your first model

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.models import SklearnModel
from jaqpotpy import Jaqpot
# Sample data
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4, 5],
    'feature2': [2.1, 3.2, 4.3, 5.4, 6.5],
    'target': [0, 1, 0, 1, 0]
})
# Create dataset for binary classification
dataset = JaqpotpyDataset(
    df=data,
    x_cols=['feature1', 'feature2'],  # Feature columns
    y_cols=['target'],                # Target column
    task='binary_classification'      # Specify the task type
)
# Train a model
model = SklearnModel(
    model =LogisticRegression(),
    dataset = dataset
)
model.fit()
# Upload the pretrained model on Jaqpot
jaqpot = Jaqpot()
jaqpot.login()
model.deploy_on_jaqpot(
    jaqpot=jaqpot,
    name="My first Jaqpot Model",
```

```
    description="This is my first attempt to train and upload a Jaqpot model.",
    visibility="PRIVATE",
)
```

Success!

Voila! Your first model is now online on Jaqpot! Kudos to you, awesome modeler, for taking the first step toward your modeling journey! 🚀🌟

Jaqpotpy

jaqpotpy is a Python library that helps you build models and use the Jaqpot platform. You can use it to create, train and deploy machine learning models.

Key Features:

- **Build Models**: Create and train models with preprocessing and evaluation tools
- **Upload Models**: Upload your trained models to Jaqpot
- **Custom Tools**: Add your own preprocessing steps, featurization and metrics
- **Use Jaqpot**: Connect to Jaqpot API to deploy models and get predictions
- **Deploy**: Get your models ready for production use

Jaqpotpy helps you take your models from your computer to the Jaqpot platform, where others can use them.

📄 ONNX Integration

In this section, we explain Jaqpot's use of ONNX to ensure compatibility across various machine learning mod...

🗃️ Scikit-learn models

6 items

🗃️ PyTorch Geometric models

4 items

🗃️ Full API Reference

27 items

📄 Examples

GitHub Examples

Edit this page

ONNX Integration

In this section, we explain Jaqpot's use of ONNX to ensure compatibility across various machine learning models and highlight the benefits and limitations of this integration.

Overview

Jaqpot uses **ONNX** **(Open Neural Network Exchange)** as a standardized format to allow models from different machine learning libraries to be compatible with our platform. This format helps bridge the compatibility gap, especially for Scikit-Learn and PyTorch models, enabling seamless model deployment through Jaqpot’s API.

Supported Libraries

Currently, Jaqpot fully supports the conversion and deployment of:

- **Scikit-learn Models**: These models work directly and reliably with ONNX, making Scikit-Learn a primary library for Jaqpot.
- **PyTorch Geometric Models**: Support for PyTorch-Geometric models is being developed (specifically for molecular modelling), and users can start to experiment with some Pytorch Graph Neural Network models using ONNX.

ONNX allows users to convert models into a single, standardized format, providing cross-platform compatibility and a smoother integration experience.

Pros and Cons

Pros

- **Compatibility across libraries**: ONNX allows Jaqpot to handle multiple model types by standardizing them into one format.
- **Ease of use**: Once a model is converted to ONNX, users experience minimal hassle, as ONNX abstracts many library-specific differences.
- **Scikit-Learn as first-class citizen**: Jaqpot fully supports Scikit-Learn models with ONNX, making it the preferred library for seamless integration.

- **Backward compatibility**: ONNX ensures that models remain compatible with Jaqpot's API, even as the platform evolves. Even if Jaqpot or scikit-learn updates its API, ONNX models will remain compatible.

Cons

- **Dependency on ONNX features**: Jaqpot's capabilities are limited to the functions ONNX currently supports. If ONNX doesn't support a specific function from a library, Jaqpot may not be able to support it either.
- **Ongoing support for Torch Models**: Jaqpot's compatibility with PyTorch Geometric models is a work in progress, and some advanced Torch features may not yet be supported. Currently, Graph Neural Networks architectures for molecular modelling are supported.

Future directions

Jaqpot aims to enhance ONNX integration further, focusing on extending support for PyTorch and exploring compatibility with other machine learning frameworks as ONNX evolves. Future updates will address additional Torch functionalities and possibly other libraries that can be adapted into Jaqpot's API.

Limitations and alternatives

- **Limitations**: ONNX does not support every function across all libraries, so certain advanced or custom model components may not yet be compatible.
- **Alternatives**: If ONNX does not support a specific feature you need, consider adjusting the model or checking Jaqpot's compatibility updates for future support.

📄 Create a Dataset

This guide demonstrates how to create and work with datasets using jaqpotpy. The JaqpotpyDataset class is ...

📄 Create a Model

This example demonstrates how to create a model using jaqpotpy with a scikit-learn model. The following co...

📄 Upload a Model on Jaqpot

This example demonstrates how to upload a trained model on Jaqpot using the jaqpotpy library. The code bel...

📄 Feature Preprocessing

Using Multiple Featurizers

📄 Evaluate a Model

In this example, we will demonstrate how to evaluate the robustness of a model using jaqpotpy. We will use ...

📄 Domain of Applicability

To demonstrate how to use the domain of applicability (DOA) with Jaqpotpy models, we will create a regressi...

Create a Dataset

This guide demonstrates how to create and work with datasets using `jaqpotpy`. The `JaqpotpyDataset` class is versatile and can handle various types of data, including molecular representations (SMILES) and their descriptors.

Basic Setup

First, import the necessary libraries:

```
import pandas as pd
import numpy as np
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors, MordredDescriptors,
TopologicalFingerprint, MACCSKeysFingerprint
```

Creating a Basic Dataset

For a simple dataset without molecular descriptors:

```
# Sample data
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4, 5],
    'feature2': [2.1, 3.2, 4.3, 5.4, 6.5],
    'target': [0, 1, 0, 1, 0]
})
# Create dataset for binary classification
dataset = JaqpotpyDataset(
    df=data,
    x_cols=['feature1', 'feature2'],  # Feature columns
    y_cols=['target'],                # Target column
    task='binary_classification'      # Specify the task type
)
```

Creating a Dataset with SMILES and Molecular Descriptors

When working with molecular data, the column containing the SMILES, as well as the featurizer should be:

```
# Sample data with SMILES
mol_data = pd.DataFrame({
    'smiles': ['CC', 'CCO', 'CCC', 'CCCl'],
    'temperature': [25, 30, 35, 40],
    'activity': [0.5, 0.7, 0.3, 0.9]
})
# Initialize a molecular descriptor calculator
rdkit_desc = RDKitDescriptors()
# Create dataset with molecular descriptors
mol_dataset = JaqpotpyDataset(
    df=mol_data,
    x_cols=['temperature'],     # Additional feature columns
    y_cols=['activity'],        # Target column
    smiles_cols=['smiles'],     # SMILES column
    task='regression',          # Regression task
    featurizer=rdkit_desc       # Specify the descriptor calculator
)
```

Available Task Types

JaqpotpyDataset supports three types of machine learning tasks:

- `regression`: For predicting continuous values
- `binary_classification`: For two-class classification problems
- `multiclass_classification`: For classification with more than two classes

Available Molecular Descriptors

Jaqpotpy offers four different molecular descriptor calculators:

```
# RDKit descriptors
rdkit_desc = RDKitDescriptors()
# Mordred descriptors
mordred_desc = MordredDescriptors()
# Topological fingerprints
topo_fp = TopologicalFingerprint()
# MACCS keys fingerprints
maccs_fp = MACCSKeysFingerprint()
```

Creating a Multiclass Classification Dataset

```
# Sample data for multiclass classification
multi_data = pd.DataFrame({
    'smiles': ['CC', 'CCO', 'CCC', 'CCCl'],
    'feature1': [1, 2, 3, 4],
    'class': ['A', 'B', 'C', 'A']
})
# Using MACCS keys fingerprints
maccs_fp = MACCSKeysFingerprint()
multi_dataset = JaqpotpyDataset(
    df=multi_data,
    x_cols=['feature1'],
    y_cols=['class'],
    smiles_cols=['smiles'],
    task='multiclass_classification',
    featurizer=maccs_fp
)
```

Important Notes

1. The `smiles_cols` parameter is optional. If not provided, no molecular descriptors will be generated.
2. When using `smiles_cols`, a `featurizer` must be specified.
3. The `task` parameter must match your problem type:
   - Use `regression` for continuous targets
   - Use `binary_classification` for two-class problems
   - Use `multiclass_classification` for multiple classes
4. Feature columns (`x_cols`) can include both molecular and non-molecular features.
5. Target columns (`y_cols`) specify the variable(s) to be predicted.

This dataset object can then be used with the `Sklearn()` Jaqpotpy model classes for training and prediction tasks.

Create a Model

This example demonstrates how to create a model using `jaqpotpy` with a scikit-learn model. The following code will guide you through generating a dataset, training a logistic regression model, and making predictions.

First, we import the necessary libraries:

```
import pandas as pd
from sklearn.datasets import make_classification
from jaqpotpy.datasets import JaqpotpyDataset
from sklearn.linear_model import LogisticRegression
from jaqpotpy.models import SklearnModel
```

Next, we generate a small binary classification dataset:

```
X, y = make_classification(n_samples=100, n_features=4, random_state=42)
```

We then create a DataFrame with the features and target:

```
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Now, we initialize a `JaqpotpyDataset` with the DataFrame:

```
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="binary_classification",
)
```

We wrap the scikit-learn model with Jaqpotpy's `SklearnModel`:

```
jaqpot_model = SklearnModel(dataset=dataset, model=LogisticRegression())
```

Next, we fit the model to the dataset:

```
jaqpot_model.fit()
```

We generate a small prediction dataset:

```
X_test, _ = make_classification(n_samples=5, n_features=4, random_state=42)
```

We create a DataFrame with the features:

```
df_test = pd.DataFrame(X_test, columns=["X1", "X2", "X3", "X4"])
```

We initialize a `JaqpotpyDataset` for prediction:

```
test_dataset = JaqpotpyDataset(
    df=df_test,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=None,
    task="binary_classification",
)
```

Finally, we use the trained model to predict the classes of the new data and the estimate their claissification probabilities and print the predictions:

```
predictions = jaqpot_model.predict(test_dataset)
probabilities = jaqpot_model.predict_proba(test_dataset)
print(predictions)
```

This code snippet covers the entire process from dataset creation to model training and prediction using `jaqpotpy` and scikit-learn's `LogisticRegression`.

Upload a Model on Jaqpot

This example demonstrates how to upload a trained model on Jaqpot using the `jaqpotpy` library. The code below shows the complete process, from generating a dataset to deploying the model on Jaqpot.

First, we generate a small regression dataset and create a DataFrame with the features and target:

```
import pandas as pd
from sklearn.datasets import make_regression
# Generate a small regression dataset
X, y = make_regression(n_samples=100, n_features=4, noise=0.2, random_state=42)
# Create a DataFrame with the generated data
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Next, we initialize a `JaqpotpyDataset` with the DataFrame, specifying the feature columns, target column, and task type:

```
from jaqpotpy.datasets import JaqpotpyDataset
# Initialize a JaqpotpyDataset with the DataFrame
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="regression",
)
```

We then wrap the scikit-learn model with `Jaqpotpy`'s `SklearnModel` and fit the model to the dataset:

```
from sklearn.linear_model import LinearRegression
from jaqpotpy.models import SklearnModel
# Wrap the scikit-learn model with Jaqpotpy's SklearnModel
jaqpot_model = SklearnModel(dataset=dataset, model=LinearRegression())
# Fit the model to the dataset
jaqpot_model.fit()
```

Finally, we upload the trained model to Jaqpot. To upload a model, a Jaqpot account is required. You can create one here. After logging in to Jaqpot, we use the `deploy_on_jaqpot` method to upload the model, providing the model name, description, and visibility settings (PUBLIC or PRIVATE):

```
from jaqpotpy import Jaqpot
# Upload the pretrained model on Jaqpot
jaqpot = Jaqpot()
jaqpot.login()
jaqpot_model.deploy_on_jaqpot(
    jaqpot=jaqpot,
    name="My first Jaqpot Model",
    description="This is my first attempt to train and upload a Jaqpot model.",
    visibility="PRIVATE",
)
```

In this final step, we first create an instance of the `Jaqpot` class and log in. Then, we call the `deploy_on_jaqpot` method on our model, passing in the `Jaqpot` instance along with the model name, description, and visibility settings. This process allows you to easily deploy your trained models on Jaqpot for further use and sharing.

Feature Preprocessing

Using Multiple Featurizers

This guide is about using multiple featurizers and performing feature selection.

First, we import necessary libraries.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.feature_selection import VarianceThreshold
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder, StandardScaler
from sklearn.ensemble import RandomForestRegressor
from jaqpotpy.models import SklearnModel
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors, MACCSKeysFingerprint
```

Create a dataframe with SMILES strings, a categorical variable, temperature, and activity values. SMILES is a unified method to represent chemical structures in the form of a line notation. For more info about SMILES check this Wikipedia page.

```
data = pd.read_csv("https://github.com/ntua-unit-of-control-and-
informatics/jaqpot-google-colab-examples/raw/doc/JAQPOT-
425/Sklearn_jupyter_examples/datasets/regression_smiles_categorical.csv")
```

Define a list of desired featurizers.

```
from jaqpotpy.descriptors import RDKitDescriptors, MACCSKeysFingerprint
featurizers = [RDKitDescriptors(), MACCSKeysFingerprint()]
```

We then pass this list of featurizers to the `JaqpotpyDataset` object when creating the training dataset:

```
train_dataset = JaqpotpyDataset(
    df=data,
    x_cols=["cat_col", "temperature"],
```

```
    y_cols=["activity"],
    smiles_cols=["smiles"],
    task="regression",
    featurizer=featurizers,
)
```

By providing a list of featurizers, the dataset will generate both RDKit descriptors and MACCS keys fingerprints for the SMILES data, resulting in a more comprehensive set of molecular features.

Feature Selection

In the second script, we demonstrate the use of feature selection. After creating the `JaqpotpyDataset` object, we apply a feature selection technique using the `select_features()` method:

```
# Use VarianceThreshold to select features with a minimum variance of 0.1
FeatureSelector = VarianceThreshold(threshold=0.1)
train_dataset.select_features(
    FeatureSelector,
    ExcludeColumns=["cat_col"],  # Explicitly exclude the categorical variable
)
```

This will apply the VarianceThreshold feature selector to the dataset, excluding the "cat_col" variable, which is a categorical feature that cannot be included in the selection process.

Alternatively, you can directly select specific columns by name using the `SelectColumns` argument:

```
myList = [
    "temperature",
    "cat_col",
    "MaxAbsEStateIndex",
    "MaxEStateIndex",
    "MinAbsEStateIndex",
    "MinEStateIndex",
    "SPS",
    "MolWt",
    "HeavyAtomMolWt",
]
train_dataset.select_features(SelectColumns=myList)
```

This method allows you to manually choose the features you want to include in the model, which can be useful if you have domain knowledge about the most relevant variables.

Feature Preprocessing

In the first script, we define a preprocessing pipeline for the feature columns and the target column:

```
# Preprocessing for the feature columns
double_preprocessing = [
    ColumnTransformer(
        transformers=[
            ("OneHotEncoder", OneHotEncoder(), ["cat_col"]),
        ],
        remainder="passthrough",
        force_int_remainder_cols=False,
    ),
    StandardScaler(),  # Standard scaling for numerical features after encoding
]
# Preprocessing for the target column
single_preprocessing = MinMaxScaler()
```

The `double_preprocessing` pipeline first applies OneHotEncoder to the categorical "cat_col" feature, then applies StandardScaler to the numerical features (including the encoded categorical variable).

The `single_preprocessing` pipeline applies MinMaxScaler to the target variable "activity".

We then pass these preprocessing pipelines to the `SklearnModel` object:

```
jaqpot_model = SklearnModel(
    dataset=train_dataset,
    model=RandomForestRegressor(random_state=42),
    preprocess_x=double_preprocessing,
    preprocess_y=single_preprocessing,
)
jaqpot_model.fit()
```

This ensures that the feature and target variables are properly preprocessed before being used to train the machine learning model.

By using multiple featurizers, feature selection, and feature preprocessing, you can create more robust and effective machine learning models with JaqpotPy.

Evaluate a Model

In this example, we will demonstrate how to evaluate the robustness of a model using `jaqpotpy`. We will use a RandomForestRegressor model and perform various evaluations including cross-validation, external evaluation, and a randomization test.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from jaqpotpy.models import SklearnModel
from jaqpotpy.datasets import JaqpotpyDataset
from jaqpotpy.descriptors import RDKitDescriptors
```

We start by creating a sample dataset with molecular structures represented as SMILES strings, along with temperature and activity values.

```
# Create sample data
data = pd.DataFrame(
    {
        "smiles": ["CC", "CCO", "CCC", "CCCl",
        "CCBr", "COC", "CCOCC", "CCCO",
        "CCCC", "CCCCCC",
    ],
        "temperature": np.random.randint(20, 37, size=10),

        "activity": [80, 81, 81, 84, 83.5,
        83, 89, 90, 91, 97,
    ],
    }
)
```

Next, we prepare the dataset for training using `JaqpotpyDataset` and `RDKitDescriptors` for feature extraction.

```
featurizer = RDKitDescriptors()
# Prepare the dataset for training with Jaqpotpy
train_dataset = JaqpotpyDataset(
    df=data,
    x_cols=["temperature"],
    y_cols=["activity"],
```

```
    smiles_cols=["smiles"],
    task="regression",
    featurizer=featurizer,
)
```

We then initialize a RandomForestRegressor model and wrap it with `SklearnModel` from `jaqpotpy`. The model is trained on the prepared dataset.

```
model = RandomForestRegressor(random_state=42)
jaqpot_model = SklearnModel(dataset=train_dataset, model=model)
jaqpot_model.random_seed = 1231
jaqpot_model.fit()
```

To estimate the model's performance, we perform cross-validation on the training data.

```
# Perform cross-validation on the training data
jaqpot_model.cross_validate(train_dataset, n_splits=10)
```

We define a test dataset for external evaluation and prepare it using `JaqpotpyDataset`.

```
# Define test data for external evaluation
X_test = pd.DataFrame(
    {
        "smiles": ["CCCOC", "CO"],
        "temperature": [27.0, 22.0],
        "activity": [89.0, 86.0],
    }
)
# Prepare the test dataset with Jaqpotpy
test_dataset = JaqpotpyDataset(
    df=X_test,
    smiles_cols="smiles",
    x_cols=["temperature"],
    y_cols=["activity"],
    task="regression",
    featurizer=featurizer,
)
```

We evaluate the model on the test dataset to assess its performance on new/unseen data.

```
# Evaluate the model on the test dataset
jaqpot_model.evaluate(test_dataset)
```

```
predictions = jaqpot_model.predict(test_dataset)
print(predictions)
```

Finally, we conduct a randomization test to assess the model's robustness against randomization of target labels.

```
# Conducts a randomization test to assess the model's robustness
jaqpot_model.randomization_test(
    train_dataset=train_dataset,
    test_dataset=test_dataset,
    n_iters=10,
)
```

Domain of Applicability

To demonstrate how to use the domain of applicability (DOA) with Jaqpotpy models, we will create a regression model using scikit-learn's Linear Regression and evaluate the DOA using Jaqpotpy's Leverage, BoundingBox, and MeanVar methods.

First, we generate a small regression dataset with 100 samples, each having 4 features and some noise. We then create a DataFrame with the features stored in columns "X1", "X2", "X3", "X4" and the target in column "y".

```
import pandas as pd
from sklearn.datasets import make_regression
from jaqpotpy.datasets import JaqpotpyDataset
from sklearn.linear_model import LinearRegression
from jaqpotpy.models.sklearn import SklearnModel
from jaqpotpy.doa import Leverage, BoundingBox, MeanVar
X, y = make_regression(n_samples=100, n_features=4, noise=0.2, random_state=42)
df = pd.DataFrame(X, columns=["X1", "X2", "X3", "X4"])
df["y"] = y
```

Next, we initialize a `JaqpotpyDataset` with the DataFrame, specifying the feature columns and the target column, and define the task as regression.

```
dataset = JaqpotpyDataset(
    df=df,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=["y"],
    task="regression",
)
```

We then wrap the scikit-learn model with Jaqpotpy's `SklearnModel`, using Linear Regression as the regression model and specifying the DOA methods: Leverage, BoundingBox, and MeanVar.

```
jaqpot_model = SklearnModel(
    dataset=dataset,
    model=LinearRegression(),
    doa=[Leverage(), BoundingBox(), MeanVar()],
)
```

After fitting the model to the dataset, we generate a small prediction dataset with 5 samples, each having 4 features, and create a DataFrame with the features.

```
jaqpot_model.fit()
X_test, _ = make_regression(n_samples=5, n_features=4, noise=0.2, random_state=42)
df_test = pd.DataFrame(X_test, columns=["X1", "X2", "X3", "X4"])
```

We initialize a `JaqpotpyDataset` for prediction, specifying the feature columns and setting `y_cols` to `None` since we are predicting.

```
test_dataset = JaqpotpyDataset(
    df=df_test,
    x_cols=["X1", "X2", "X3", "X4"],
    y_cols=None,
    task="regression",
)
```

Finally, we use the trained model to check if the test data are in or out of the domain of applicability, using `predict_doa` method.

```
doa_predictions = jaqpot_model.predict_doa(test_dataset)
print(doa_predictions)
```

This example demonstrates how to use Jaqpotpy to evaluate the domain of applicability for a regression model, ensuring that predictions are made within the reliable range of the model.

📄 Create a Dataset

This example demonstrates how to create a Molecular Graph Neural Network dataset with jaqpotpy. SmilesGr...

📄 Create a Model

This document demonstrates how to create a Graph Neural Network model using JaqpotPy with a specific arc...

📄 Train and Evaluate a Model

In this section, we continue building on the previous example by defining the optimizer, loss function, and tra...

📄 Upload a Model

Import Required Libraries

Create a Dataset

This example demonstrates how to create a Molecular Graph Neural Network dataset with `jaqpotpy`. `SmilesGraphFeaturizer` is used to create graph based features on SMILES input, while `SmilesGraphDataset` is the dataset class for Graph Neural Networks training.

Basic Setup

We first import the necessary libraries:

```
import torch
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
```

Demo Data

We create a small dataset for demonstration purposes with SMILES strings and their corresponding target values. This example uses a classification target.

```
smiles_list = [
    "CC(=O)OC1=CC=CC=C1C(=O)O",  # Aspirin
    "CN1C=NC2=C1C(=O)N(C(=O)N2C)C",  # Caffeine
    "CC(C)CC1=CC=C(C=C1)C(C)C(=O)O",  # Ibuprofen
    "CC(C)CCCC(C)C1CCC2C1(CCC3C2CCC4=C3CC=C4)O",  # Cholesterol
    "OC[C@H]1OC(=O)C(O)=C1O"  # Vitamin C (Ascorbic acid)
]
y = [0, 1, 0, 1, 1] # Binary Activity
```

Featurization

To convert the SMILES strings into graph-based features, we use the `SmilesGraphFeaturizer`. This featurizer can include edge features and supports various atom and bond features.

Default featurizer

We create an instance of `SmilesGraphFeaturizer` and configure it with default settings for graph features (nodes and edges).

```
featurizer = SmilesGraphFeaturizer(include_edge_features=True)
featurizer.set_default_config()
```

Custom hardcoded features

Node features and their coresponding values can be hardcoded by user if it is supported by the featurizer. Name of features and feature values should be provided for one hot encoding Names of features and their RDKit functions can be obtained with:

```
featurizer.SUPPORTED_ATOM_FEATURES()
featurizer.SUPPORTED_BOND_FEATURES()
```

Example of custom graph features

```
featurizer = SmilesGraphFeaturizer(include_edge_features=True)
# First argument is the feature name
# Secnd argument is the feature possible values
featurizer.add_atom_feature("symbol", ["C", "O", "N", "F", "Cl", "Br", "I"])
featurizer.add_atom_feature("total_num_hs", [0, 1, 2, 3, 4])
# If possible values are not provided, the feature is not one one-hot encoded
featurizer.add_atom_feature("formal_charge")
```

Dataset Preparation

We can create datasets for training, validation, and testing using the featurizer. The datasets include the SMILES strings, target values, and the configured featurizer.

```
dataset = SmilesGraphDataset(
    smiles=smiles, y=y, featurizer=featurizer
)
```

Precompute Featurization

For small datasets, it is recommended to precompute the featurization to save time during training.

```
```

```
dataset.precompute_featurization()
```

Edit this page

Create a Model

This document demonstrates how to create a Graph Neural Network model using JaqpotPy with a specific architecture. The example assumes you have already preprocessed your data and have a dataset ready to use.

Basic Setup

We first import the necessary libraries:

```
import torch
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
from jaqpotpy.models.torch_geometric_models.graph_neural_network import
GraphSageNetwork, GraphAttentionNetwork
```

Define the Graph Neural Network Architecture

In this example, we use a GraphSageNetwork architecture. The network takes node features from a featurizer and creates a model with specified input dimensions, hidden layers, and output dimensions. Currently Jaqpotpy library supports the following architectures:

- GraphConvolutionNetork
- GraphSageNetwork
- GraphAttentionNetwork
- GraphTransformerNetwork

Graph Convolution and Graph Sage architectures proccess only node attributes. You can easily create a model like the example below:

```
node_features = featurizer.get_num_node_features()
model = GraphSageNetwork(
    input_dim=node_features,
    hidden_layers=1,
    hidden_dim=4,
```

```
    output_dim=1,
    activation=torch.nn.ReLU(),
    dropout_proba=0.1,
    batch_norm = False,
    seed = 42,
    pooling="mean",
)
```

- input_dim: Number of input neurons, determined by the number of node features.
- hidden_layers: Number of hidden layers in the network.
- hidden_dim: Number of neurons in each hidden layer.
- output_dim: Number of output neurons (default set to 1 for binary classification or regression).
- activation: Activation function, specified using PyTorch (e.g., ReLU).
- pooling: Graph pooling method (options: mean, add, max).
- dropout_proba: Dropout probability for regularization.

Graph Attention and Graph Transformer architectures can proccess both node and edge attributes (optionally)

```
node_features = featurizer.get_num_node_features()
edge_features = featurizer.get_edge_features()
model = GraphAttentionNetwork(
    input_dim=node_features,
    hidden_layers=1,
    hidden_dim=4,
    output_dim=1,
    activation=torch.nn.ReLU(),
    dropout_proba=0.1,
    batch_norm = False,
    seed = 42,
    pooling="mean",
    edge_dim = edge_features,
    heads = 2
)
```

Extra arguments:

- edge_dim: Number of edge features or None
- heads: Attention heads for each neuron in hidden layers

Train and Evaluate a Model

In this section, we continue building on the previous example by defining the optimizer, loss function, and training procedure for our graph neural network (GNN) model using `jaqpotpy`. We utilize PyTorch-based components to configure the training process.

Import Required Libraries

We first import the necessary libraries:

```
import torch
from torch_geometric.loader import DataLoader
from jaqpotpy.descriptors.graph import SmilesGraphFeaturizer
from jaqpotpy.datasets import SmilesGraphDataset
from jaqpotpy.models.torch_geometric_models.graph_neural_network import
GraphSageNetwork, GraphAttentionNetwork
from jaqpotpy.models.trainers.graph_trainers import BinaryGraphModelTrainer,
RegressionGraphModelTrainer
```

Binary Classification

PyTorch Configuration

We begin by defining the optimizer and loss function from Pytorch. In this example, we use the Adam optimizer and a binary cross-entropy loss function with logits, as the task is a binary classification problem. You can choose any type of optimizer that comes with PyTorch.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss = torch.nn.BCEWithLogitsLoss()
```

Trainer Configuration

We use the BinaryGraphModelTrainer from jaqpotpy to handle the training process. This trainer requires the model, number of epochs, optimizer, and loss function as inputs. Optionally, a learning rate scheduler can also be provided.

```
trainer = BinaryGraphModelTrainer(
    model=model, # Jaqpotpy Graph Model
    n_epochs=20,
    optimizer=optimizer,
    loss_fn=loss,
    scheduler=None
)
```

Data Loaders

We use PyTorch Geometric's DataLoader to load the training, validation, and test datasets in mini-batches. This helps in efficiently feeding data into the model during training and evaluation. The datasets are defined in `Create a dataset` page.

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Train the Model

The model is trained using the training and validation data loaders.

```
trainer.train(train_loader, val_loader)
```

Evaluate the Model

Finally, the model is evaluated on the test dataset to obtain the loss, metrics, and confusion matrix.

```
loss, metrics, conf_matrix = trainer.evaluate(test_loader)
```

Regression

PyTorch Configuration

The main difference here is only the loss function that needs to be for regression specific tasks.

```
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss = torch.nn.MSELoss()
```

Trainer Configuration

We use the RegressionModelTrainer from jaqpotpy to handle the training process. This trainer requires the same arguments as the BinaryGraphModelTrainer

```
trainer = RegressionGraphModelTrainer(
    model=model, # Jaqpotpy Graph Model
    n_epochs=20,
    optimizer=optimizer,
    loss_fn=loss,
    scheduler=None
)
```

DataLoaders

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Train the Model

The model is trained using the training and validation data loaders.

```
trainer.train(train_loader, val_loader)
```

Evaluate the Model

Finally, the model is evaluated on the test dataset to obtain the loss and metrics.

```
loss, metrics = trainer.evaluate(test_loader)
```

jaqpotpy.utils package

Submodules

jaqpotpy.utils.data_utils module

jaqpotpy.utils.fragment_utils module

jaqpotpy.utils.geometry_utils module

jaqpotpy.utils.installed_packages module

jaqpotpy.utils.molecule_feature_utils module

jaqpotpy.utils.pdbqt_utils module

jaqpotpy.utils.pytorch_utils module

jaqpotpy.utils.rdkit_utils module

jaqpotpy.utils.seeding module

jaqpotpy.utils.types module

jaqpotpy.utils.url_utils module

Module contents

Examples

GitHub Examples

Find a comprehensive collection of usage examples in our main repository:
https://github.com/ntua-unit-of-control-and-informatics/jaqpotpy/tree/main/examples

Google Colab Examples

For interactive, runnable examples, check out our Google Colab repository:
https://github.com/ntua-unit-of-control-and-informatics/jaqpot-google-colab-examples

These Colab notebooks provide step-by-step tutorials and practical demonstrations of Jaqpotpy features in an interactive environment. You can run the examples directly in your browser without any local setup, making it perfect for quick experimentation and learning.

Jaqpot API

The Jaqpot API is a service that handles model management, deployment and predictions. It receives models through its endpoints, manages them, and can run predictions for them via the API.

Key Features:

- **Model Storage**: Store and version your machine learning models
- **Deployment:** Deploy models for production use
- **Predictions:** Get predictions through HTTP endpoints
- **User Management**: Handle users, organizations and model sharing
- **SDKs**: Use our Python SDK (jaqpotpy) to interact with the API - more SDKs coming soon

We maintain SDKs to help you use the API directly in your code. Currently available in Python, with more languages planned.

🗃️ Authentication

1 item

🗃️ SDKs

3 items

🗃️ Documentation

3 items

📄 Create an API key

API keys are used to authenticate requests to the Jaqpot API. You can create an API key for your account and ...

Create an API key

API keys are used to authenticate requests to the **Jaqpot API**. You can create an API key for your account and use it to access Jaqpot programmatically by visiting the API Key dashboard page.

Once an API key is created, you will receive a **client key** and a **client secret**.

Keep these credentials secure and do not share them with anyone.

How to use the API Key

To authenticate API requests, include the **client key** as the username and the **client secret** as the password in the headers `X-Api-Key` and `X-Api-Secret` of your requests.

Example Request

```
curl -X GET "https://api.jaqpot.org/v1/models" \
  -H "X-Api-Key: <client_key>" \
  -H "X-Api-Secret: <client_secret>"
```

or use one of the SDKs

Edit this page

📄 Python SDK

The jaqpotpy Python SDK provides a streamlined interface for interacting with Jaqpot's predictive models and...

📄 Kotlin/Java SDK

The SDK lets you use Jaqpot models and predictions in your Java or Kotlin applications.

📄 QSAR Toolbox Python SDK

The jaqpotpy SDK provides seamless access to QSAR Toolbox, a comprehensive software application designe...

Python SDK

The jaqpotpy Python SDK provides a streamlined interface for interacting with Jaqpot's predictive models and services within Python environments. With this SDK, users can easily retrieve models, make predictions (both synchronously and asynchronously), manage datasets, and handle model sharing across organizations. The SDK simplifies the process of integrating Jaqpot's machine learning capabilities into Python applications, supporting both individual predictions and batch processing through CSV files.

Authentication

Before using the SDK, you'll need to obtain API credentials from Jaqpot. To get your API keys:

1. Log in to app.jaqpot.org
2. Click on the account icon in the top right corner
3. Select "API keys" from the dropdown menu

The API keys are valid for 6 months from their generation date. Please store these keys securely, as they cannot be retrieved from Jaqpot after initial generation. If your keys are lost or expired, you will need to generate new ones by following the same steps above.

There are multiple ways to set up your API credentials as environment variables for Jaqpotpy. Here are presented 2 common methods. Using these methods, your API credentials will be automatically loaded and set as HTTP headers (`X-Api-Key` and `X-Api-Secret`) required for authentication with the Jaqpot service.

Method 1: Using Environment Variables Directly

You can set environment variables before running your script:

```
# Linux/MacOS
export JAQPOT_API_KEY='your_api_key_here'
export JAQPOT_API_SECRET='your_api_secret_here'
python your_script.py
# Windows (Command Prompt)
set JAQPOT_API_KEY=your_api_key_here
```

```
set JAQPOT_API_SECRET=your_api_secret_here
python your_script.py
```

Method 2: Using a .env file

Alternatively, you can create a `.env` file in your project directory with the following content:

```
JAQPOT_API_KEY='your_api_key_here'
JAQPOT_API_SECRET='your_api_secret_here'
```

Then load these variables in your Python code:

```
from dotenv import load_dotenv
load_dotenv(".env") # Returns True if successful
```

Note 1: If `load_dotenv()` returns False, verify that your `.env` file is properly formatted and in the correct location.

Note 2: Remember to add `.env` to your .gitignore file to keep your credentials secure.

Initializing the Client

After setting up your environment variables, initialize the Jaqpot client:

```
from jaqpotpy.api.jaqpot_api_client import JaqpotApiClient
jaqpot = JaqpotApiClient()
```

Retrieve a model by ID

To retrieve a model by its ID, you can use the `get_model_by_id` method. Here is an example:

```
model = jaqpot.get_model_by_id(model_id=1886)
print(model)
```

Get shared models with organizations

To get the models that you share with Jaqpot organizations, you can use the `get_shared_models` method. Here is an example:

```
shared_models = jaqpot.get_shared_models()
print(shared_models)
```

Synchronous prediction

To take a synchronous prediction with a model, you can use the `predict_sync` method. In synchronous prediction, the http call waits until the prediction is complete before moving on. This is useful for quick predictions where you need an immediate result. Use the predict_sync method as shown below:

```
input_data = [{"SMILES": "CC", "X1": 1, "X2": 2, "Cat_col": "CAT_1"}]
prediction = jaqpot.predict_sync(model_id=1886, dataset=input_data)
print(prediction)
```

In this case, the program will wait until predict_sync returns the prediction results, which are then printed.

Asynchronous prediction

To take an asynchronous prediction with a model, you can use the `predict_async` method. In asynchronous prediction, the http call doesn't wait for the prediction to finish, allowing it to continue running other tasks. This is useful for longer predictions or when handling multiple requests. Use the predict_async method as follows:

```
input_data = [{"SMILES": "CC", "X1": 1, "X2": 2, "Cat_col": "CAT_1"}]
prediction_dataset_id = jaqpot.predict_async(model_id=1886, dataset=input_data)
```

Get a dataset by ID

To retrieve a dataset by its ID, you can use the `get_dataset_by_id` method. Here is an example:

```
results = jaqpot.get_dataset_by_id(dataset_id=prediction_dataset_id)
print(results)
```

Prediction with a model and a CSV File

To take a prediction with a model using a CSV file, you can use the `predict_with_csv_sync` method. Here is an example:

```
csv_path = "ADD_A_CSV_PATH.csv"
prediction = jaqpot.predict_with_csv_sync(model_id=1886, csv_path=csv_path)
print(prediction)
```

Kotlin/Java SDK

The SDK lets you use Jaqpot models and predictions in your Java or Kotlin applications.

Get your API Keys

You'll need an API key and secret from your Jaqpot account. Create them by following the API keys guide.

The API keys are valid for 6 months from their generation date. Please store these keys securely, as they cannot be retrieved from Jaqpot after initial generation. If your keys are lost or expired, you will need to generate new ones by following the same steps above.

Add to your project

Maven central page: https://central.sonatype.com/artifact/org.jaqpot/kotlin-sdk

For Gradle (Kotlin DSL):

```
implementation("org.jaqpot:kotlin-sdk:0.4.0") // or the latest version of the SDK
```

For Maven:

```
<dependency>
    <groupId>org.jaqpot</groupId>
    <artifactId>kotlin-sdk</artifactId>
    <version>0.4.0</version>
</dependency>
```

Basic usage

Make predictions in Java:

```
JaqpotApiClient jaqpotApiClient =
    new JaqpotApiClient(System.getenv("JAQPOT_API_KEY"),
System.getenv("JAQPOT_API_SECRET"));
Dataset dataset = jaqpotApiClient
        .predictSync(
                modelId,
                List.of(
                        Map.of("X1", "1", "X2", "2", "X3", "3", "X4", "4")
                )
        );
System.out.println(dataset)
```

or in Kotlin:

```
val jaqpotApiClient =
    JaqpotApiClient(System.getenv("JAQPOT_API_KEY"),
System.getenv("JAQPOT_API_SECRET"))
val dataset = jaqpotApiClient.predictSync(
    modelId,
    listOf(
        mapOf("X1" to "1", "X2" to "2", "X3" to "3", "X4" to "4")
    )
)
println(dataset)
```

Setting API Keys

Use environment variables:

```
export JAQPOT_API_KEY=your_key_here
export JAQPOT_API_SECRET=your_secret_here
```

Or pass them directly to the client:

```
val client = JaqpotApiClient("your-key", "your-secret")
```

Error handling

The SDK throws JaqpotSDKException on errors like:

- Wrong API credentials
- Model not found
- Failed predictions
- Network issues

QSAR Toolbox Python SDK

The jaqpotpy SDK provides seamless access to QSAR Toolbox, a comprehensive software application designed for filling data gaps in toxicity data. Through the SDK, users can interact with three main components of QSAR Toolbox:

1. Calculators: Tools that compute various molecular descriptors and properties
2. Models: QSAR predictive models for different endpoints
3. Profilers: Components that analyze chemical structures and identify structural features

This integration allows users to programmatically access QSAR Toolbox functionalities, perform calculations, make predictions, and analyze chemical structures directly from their Python environment. All components are accessible through simple API calls, requiring only SMILES notation as input for predictions.

QSAR Toolbox

To retrieve a component (model, profiler or calculator) of QSAR Toolbox, you can retrieve a model by its ID using the `get_model_by_id` method. Here is an example:

```
# To get all calculators of QSAR Toolbox:
calculators = jaqpot.get_model_by_id(model_id=6)
# To get all models of QSAR Toolbox:
models = jaqpot.get_model_by_id(model_id=1837)
# To get all profilers of QSAR Toolbox
profilers = jaqpot.get_model_by_id(model_id=1842)
```

QSAR Toolbox Calculator

To take a prediction with any QSAR Toolbox calculator, you can use the `qsartoolbox_calculator_predict_sync` method. Here is an example:

```
prediction = jaqpot.qsartoolbox_calculator_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    calculator_guid="1804a854-9041-4495-9931-7414c22a5e49"
)
print(prediction)
```

QSAR Toolbox Model

To take a prediction with any QSAR Toolbox model, you can use the `qsartoolbox_qsar_model_predict_sync` method as:

```
prediction = jaqpot.qsartoolbox_qsar_model_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    qsar_guid="c377150b-77ae-4f99-be14-357b85dd8d1f",
)
print(prediction)
```

QSAR Toolbox Profiler

To take a prediction with any QSAR Toolbox profiler, you can use the `qsartoolbox_profiler_predict_sync` method. Here is an example:

```
prediction = jaqpot.qsartoolbox_profiler_predict_sync(
    smiles="CCCCC(CC)COC(=O)C(=C(C1=CC=CC=C1)C2=CC=CC=C2)C#N",
    profiler_guid="723eb011-3e5b-4565-9358-4c3d8620ba5d"
)
print(prediction)
```