Transfer Learning CLI/API Design

Revision History

Date	Modified by	Description
April 22, 2022	Dina Jones	Initial version of the design doc
April 28, 2022	Dina Jones	Updates based on doc review feedback after meeting with Tran, Melanie, Kam, and Tyler
May 9, 2022	Dina Jones	Updated the API Design section with high level class diagrams and sample API usages. Updated sequence diagram to change “data scientist” to user and update commands to closer match the implementation.
May 23, 2022	Dina Jones	Updated the dataset class diagram to match class names in our implementation.

1. Introduction

1.1. Overview

Transfer learning uses pretrained model weights as a starting point to train the model on a different dataset for a new task. This allows for a smaller amount of training data and reduced training time, compared to training from scratch. Transfer learning has been increasing in popularity (particularly for computer vision and natural language processing) and there are many pretrained models available in public model hubs like TensorFlow Hub, torchvision, and HuggingFace. As Intel, we want to utilize transfer learning on XPU to demonstrate that the training times can be reasonable with transfer learning and fine tuning. To help do this, we will provide a CLI and API to reduce the complexity that can be involved with transfer learning.

1.2. Problem Statement

Transfer learning involves several steps including preprocessing the dataset, finding and downloading an appropriate pretrained model, manipulating the model’s layers, retraining, evaluation, and exporting the final model. There are many transfer learning tutorials online, but those tutorials often cover a single use case, may gloss over certain details like how the dataset needs to be formatted, and do not always explain how to apply the same method to other tasks. These tutorials also do not utilize technologies that give the best performance on Intel hardware (like Intel-optimized frameworks/extensions and pruning/quantization using the Intel Neural Compressor).

A transfer learning CLI and API will provide a consistent interface for users to apply transfer learning to their own tasks across various CV and NLP use cases. The tool will also abstract out differences between frameworks and different model hubs. The tools can be run on the command line, interactively in a low code environment with sample Jupyter notebooks, and deployed as part of MLOps pipelines in containers.

1.3. Scope

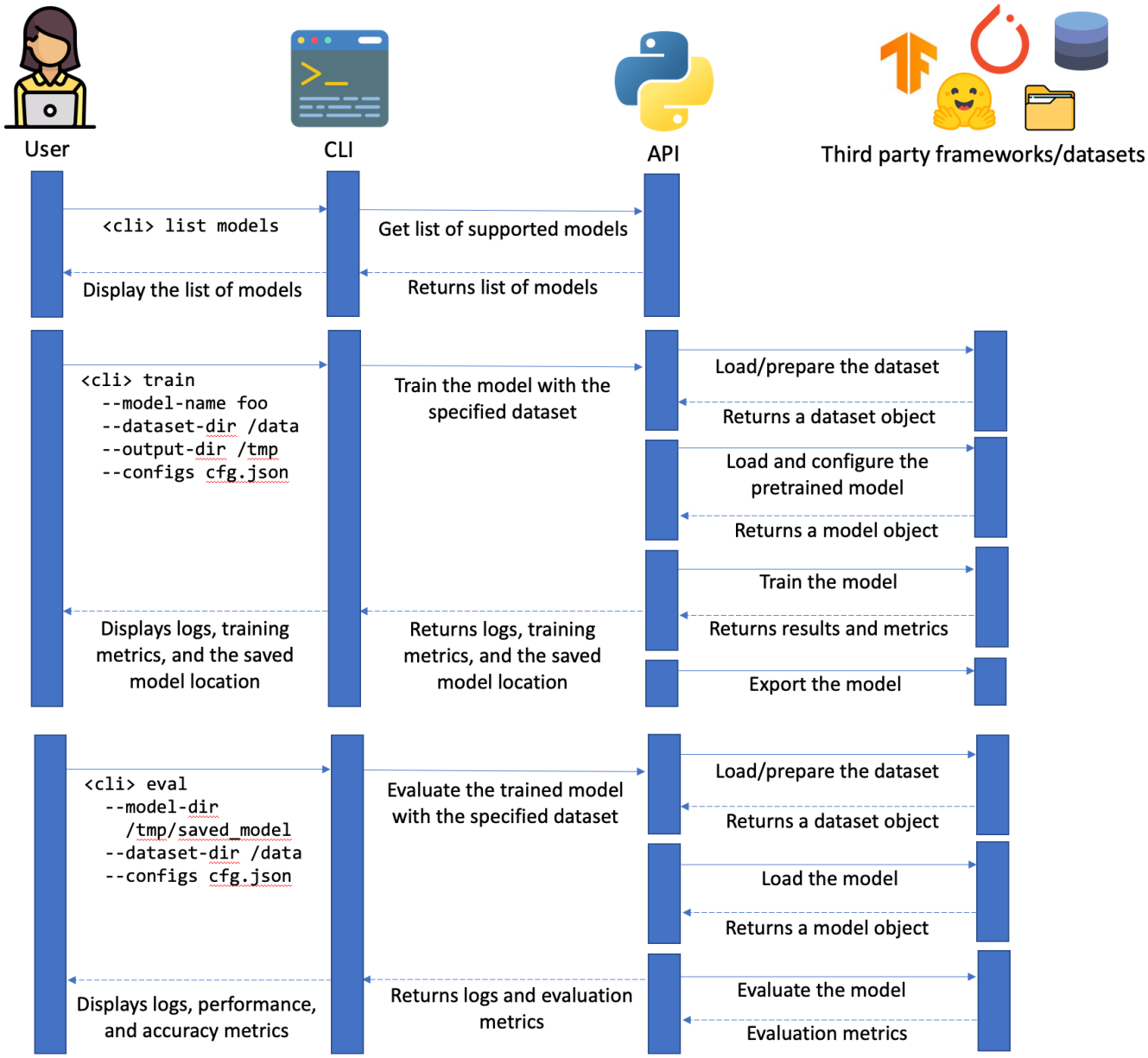
This document provides an overview of the software design for the transfer learning CLI and API. Separate documents will be used to track requirements, the product concept, and user stories.

1.4. Definitions and Acronyms

API (application programming interface)	A python library consisting of classes/methods that the user can import and use as part of a python script (or jupyter notebook).
CLI (command line interface)	Terminal tool consisting of commands, subcommands, and arguments
CV (computer vision)	Models that have images or video as input. Computer vision use cases include image classification, image segmentation, and object detection.
HuggingFace	HuggingFace is a company that has tools, models, and datasets for natural language processing with PyTorch and TensorFlow. Their API (transformers) allows for downloading and fine-tuning models from their hub. They also have a datasets API for using datasets from their catalog and custom datasets. The HuggingFace Datasets documentation says that it was forked from TensorFlow Datasets, so their API is similar.
NLP (natural language processing)	Models that are used for processing and analyzing text input. NLP models can be used for a variety of tasks including text classification (like sentiment analysis), question answering (find the answer to a question in a paragraph of text), sentence similarity (determining how similar two sentences are) and fill mask (predicting which word is used to fill in a blank/mask). HuggingFace has a good list of NLP tasks here (click on a task to get the description).
TensorFlow Datasets	TensorFlow Datasets is a collection of datasets that can be loaded through their API and returned as tf.data.Datasets objects, so that they are ready to use with TF models.
TensorFlow Hub	TensorFlow Hub (TFHub) is a collection of pretrained TensorFlow models. From the website, they have filters for looking at models by problem domain, TF1/TF2, and a toggle for finding “fine tunable” models. There is an API to load models from TFHub.
torchvision	torchvision is a PyTorch package used for working with computer vision models. torchvision has pretrained model weights and a collection of datasets (similar to TF datasets).

2. Sequence Diagram

The diagram below shows a high-level overview of the workflow for the CLI being used for selecting a pretrained model, retraining the model on a new dataset, and evaluating the trained model. The user (like a data scientist) submits a request from a terminal using the CLI. The CLI makes backend calls into the API. The API will make calls out to third-party libraries like PyTorch, torchvision, TensorFlow, TFHub, TensorFlow Datasets, and HuggingFace for loading datasets and models, training, and evaluation.



3. CLI Design

The CLI is a tool built on top of the API where the commands and subcommands make calls into the backend python library. The intent of the CLI is to provide a simple way to run transfer learning workloads without code. The CLI will have config files passed in as arguments, giving detailed control over model training options. The CLI will consist of commands, subcommands, and arguments. There may be multiple levels of subcommands, depending on the call.

```
<cli-name> <command> <optional: subcommand(s)> --optional: argument(s)
```

Environment variables are sometimes also used for setting arguments. For example, the KUBECONFIG setting in kubectl. These are often used for configs that someone wants to set and have applied to

multiple commands. In the case of this transfer learning tool, if someone knows that they are only going to want to look at TensorFlow models, an environment variable setting could be useful so that they do not need to instead use the `--framework` argument with every CLI command. To have environment variable settings as an option, there needs to be some hierarchy defined to determine which config is used in the case where both an environment variable and a CLI argument are given (normally, the CLI argument would win). There is also the possibility that the user forgets that they have the environment variable set, and then is confused by the results that they see. For simplicity, the initial implementation of the CLI will not have the option for environment variable settings, but we can revisit it in the future if we find the need to have sticky arguments.

3.1. CLI Frameworks

Three python CLI frameworks were evaluated:

- ***Argparse*** is the standard argument parser that is built-in to python.
- ***Docopt*** is a third-party CLI library that we have previous experience with from MLT (machine learning container templating tool).
- ***Click*** is another third-party python library. It uses decorators to define the CLI commands and arguments. This is the framework we selected to use for the CLI.

Proof-of-concepts (POCs) of simple CLIs have been created in the scratch-pad repo. The POCs have README.md documents for argparse, docopt, and click that show how the CLIs are different from a user's perspective (the help output, commands, and error handling). The CLI POC directories have code needed to create equivalent CLIs, so that the implementation from a developer's perspective can be evaluated.

Docopt was eliminated as an option because the help output is not as intuitive as argparse and click, the error handling is not good when invalid arguments are given, and the project also seems to be abandoned.

From a user's perspective, argparse and click are nearly identical, so picking a CLI framework came down to the implementation. Click was selected, because the decorators make it easy to define commands, subcommands, and arguments, and there are less lines of code needed for parsing. The argparse API also gets more confusing when there are multiple levels of subcommands.

3.2. CLI Commands

Top-level commands represent the general functions of the command line tool:

- ***list***: lists the available models/use cases. The list should have information about the model like the dataset it was trained on, framework, image size, etc. so that the user can select a model that is appropriate to their task.
- ***train***: specify the name of pretrained model, the dataset that you want to use, the framework (optional?), output directory, and a config file for more specific parameters like epochs and learning rate. After training is completed, export the saved model (saved_model.pb for TensorFlow or .pt file for PyTorch) to the output directory. Along with the saved model, the tool should store configs/info

about the model so that it can be reloaded or reproduced.
Other considerations:

- We may need to add the option to export to ONNX at some point in the future.
- How can we make training runs completely reproducible. Are there any seeds or randomization involved in the training?
- Multi-instance/multi-device (like multi-tile GPU)/multi-node support will also need to be considered.

- ***eval***: given a trained model, evaluate the inference performance and accuracy with a validation dataset. Evaluate is a separate step because people may want to evaluate the same trained model multiple times with different sets of data. Maybe they have new data coming in daily that they want to evaluate against their saved model to determine whether it needs to be retrained.
- ***predict***: (maybe?) given a trained model, make predictions on a set of data. The data does not need to be labeled (since we aren't calculating accuracy like we do with the evaluation command). This also has potential for being combined with the eval command (with different flags/args to determine if we are evaluating or just predicting).
- ***quantization*** (maybe?): post-training quantization using INC
- ***deploy*** (maybe?): serve a saved model

3.3. Environment

The CLI will be run on bare metal since our target users may not have docker installed. Since there will be many dependencies required to run the CLI, we can recommend that users install the tool in a virtual environment. We can also separate out dependencies by framework in the `setup.py` file so that users who know that they only want to work with a single framework can do a framework-specific install like `pip install <tool>[tensorflow]` or `pip install <tool>[pytorch]`. `pip install <tool>` would install the dependencies for both TensorFlow and PyTorch.

3.3.1. Caveats

We may hit a point where we have conflicting dependencies in our environment. One model may require numpy version x, while another model may require numpy version y. This could mean that the tool would need to be smart enough to do dependency version checking at runtime per model, which could get complicated.

3.3.2. Alternatives

The way that Nvidia's TAO tool works is that it runs a docker container in the background with each CLI command. We could do something similar, which would allow us to use a different container depending on which model is being run, which would allow us to have a known working set of dependencies. We are not initially taking this route, because there are users like data scientists who do not already have

docker installed and might immediately navigate away when seeing that it is a requirement.

If the dependency conflict ends up being between TensorFlow and PyTorch the tool could be split out into two separate installs, but we initially avoided this because there are users who do not care what framework their workload is run on.

3.4. Code structure

The repo will have a directory for the cli tool, with a main.py file (which defines the CLI click group), and a subdirectory for the "commands". Each of the top-level commands will have its own .py file in that folder that defines the click command (and any subcommands) function with the decorators to define any arguments, etc. for the command. The command function will call out to the API.

```
cli
├── commands
│   ├── eval.py
│   ├── list.py
│   └── train.py
└── main.py
```

The command files with the click decorators will look something like:

```
@click.command()
@click.option("--framework", "-f",
              required=False,
              default="tensorflow",
              help="Deep learning framework [default: tensorflow]")
@click.option("--model-name", "--model_name",
              required=True,
              type=str,
              help="Name of the model to use")
@click.option("--output-dir", "--output_dir",
              required=True,
              type=str,
              help="Output directory for saved models, logs, etc")
@click.option("--dataset-name", "--dataset_name",
              required=False,
              type=str,
              help="Name of a dataset to use")
@click.option("--dataset-dir", "--dataset_dir",
              required=True,
              type=str,
              help="Dataset directory")
def train(framework, model_name, output_dir, dataset_name, dataset_dir):
    """
    Trains the model
    """

    # TODO: Call API to train
    print("\n<call the train API>\n")
```

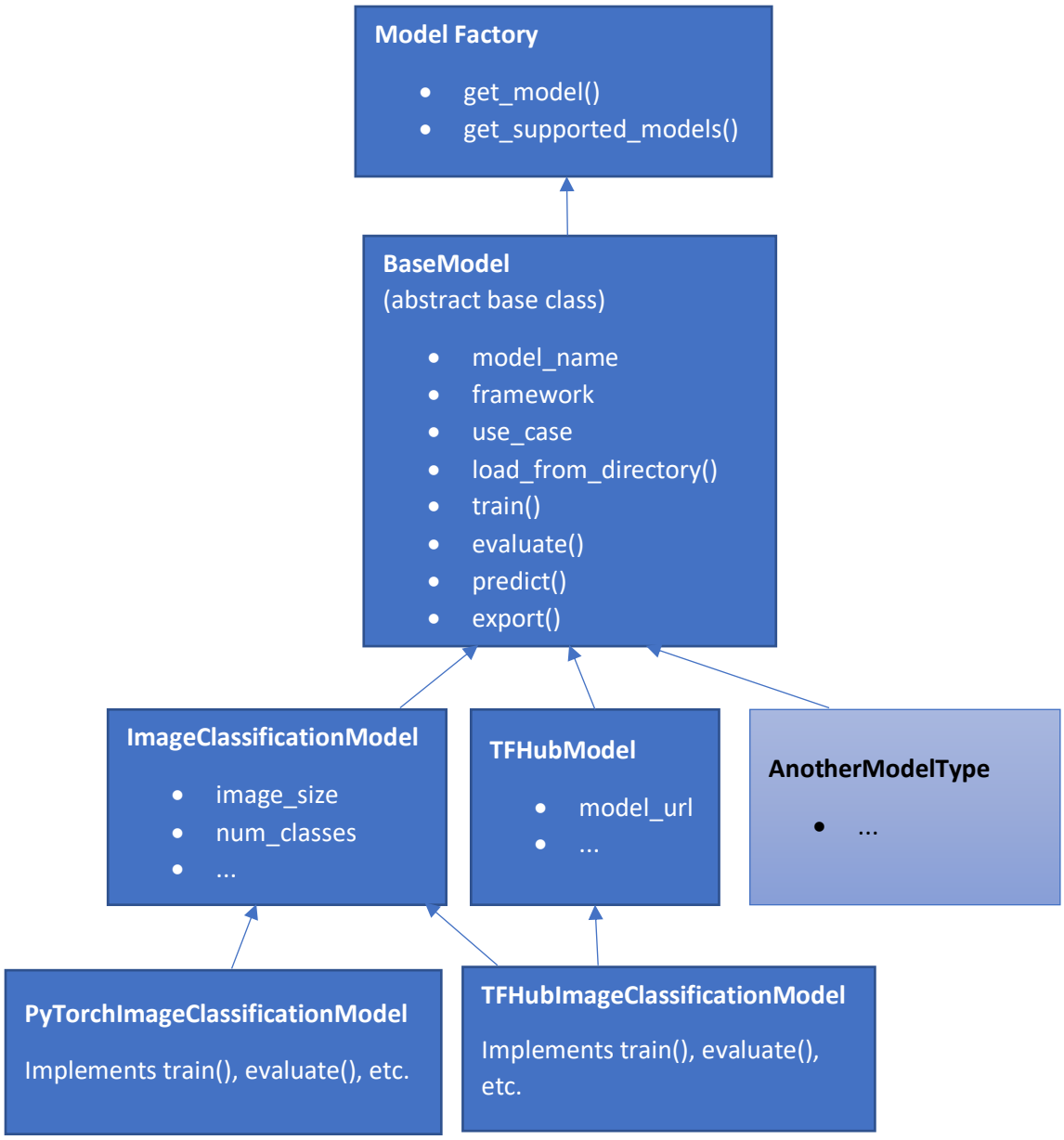
4. API Design

The API will follow an object-oriented design to define classes for different datasets and models. There will be abstract base classes for Models and Datasets to define common properties and methods and subclasses. In most cases, a subclass will be able to cover a group of datasets or models. For example, there could be a subclass for image classification models from the TF Hub, and another subclass for image classification models from torchvision. In some cases, a specific model or dataset may need its own subclass (for example, the PennFudan pedestrian dataset for PyTorch object detection).

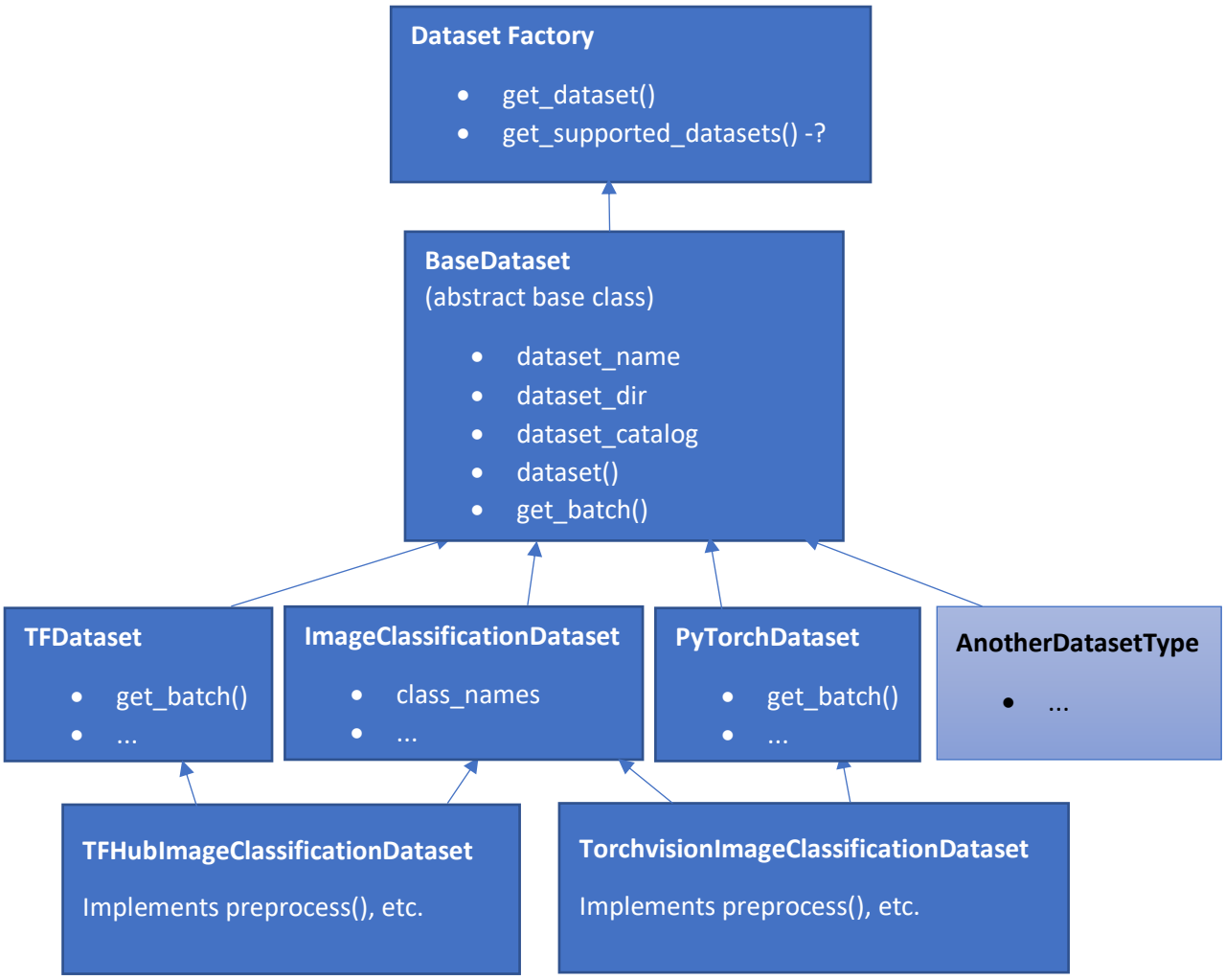
Factory methods will be used to instantiate model and dataset objects as they are requested.

4.1. Class Diagrams

The class diagram below shows a high-level example of the model factory and classes used for TensorFlow image classification. Model support for other frameworks and use cases would follow a similar pattern.



The class diagram below shows a high-level example of the dataset factory and classes used for TensorFlow image classification datasets. Support for other types of datasets would follow the same pattern.



4.2. API Usage

The API is a necessary component for our CLI to function, but it can also be used as a stand-alone library to simplify the transfer learning process. The API can be used by someone who someone who is comfortable with python and wants more control or access to advanced usages that the CLI does not provide. The API will provide the user with access to the actual framework objects (for example, `dataset.dataset()` would return a `tf.data.Dataset` for a TensorFlow dataset) for models and datasets, so if the user wants to directly manipulate the dataset or graph, they have that option.

An example of the API being used would look something like this:

```
from tlk.datasets import dataset_factory
from tlk.models import model_factory

# Get the model
model = model_factory.get_model(model_name="efficientnet_b0",
                                framework="tensorflow")

# Get and preprocess a dataset
dataset = dataset_factory.get_dataset(dataset_dir="/tmp/data",
                                      use_case="image_classification",
                                      framework="tensorflow",
                                      dataset_name="tf_flowers")
dataset.preprocess(image_size=model.image_size, batch_size=32)

# Train the model
model.train(dataset, output_dir="/tmp/output", epochs=1)

# Evaluate the model
metrics = model.evaluate(dataset)

# Get a single batch from the dataset
images, labels = dataset.get_batch()
labels = [dataset.class_names[id] for id in labels]

# Predict using the batch
predictions = model.predict(images)
predicted_labels = [dataset.class_names[id] for id in predictions]

# Export the model
model.export(output_dir="/tmp/output")
```

5. Repository Structure

The tree below has an example of the repository directory structure. Note that the .py names are placeholders that serve as an example of what types of files we might see and are not an exact representation of what will exist.

```
frameworks.ai.transfer-learning/

├── README.md
├── docs
│   ├── images
│   ├── Makefile
│   └── *.rst
├── examples
│   └── example_config_files
├── notebooks
│   ├── README.md
│   ├── image_classification
```

```
│   ├── object_detection
│   ├── question_answering
│   └── text_classification
├── setup.py
├── tests
└── <tool name>
    ├── dataset
    │   ├── dataset.py
    │   ├── dataset_factory.py
    │   ├── image_classification
    │   │   └── tf_flowers.py
    │   ├── object_detection
    │   │   └── pennfudan_pedestrian.py
    │   ├── text_classification
    │   │   ├── imdb.py
    │   │   └── sms_spam_collection.py
    │   └── tf_dataset.py
    ├── models
    │   ├── huggingface_model.py
    │   ├── model.py
    │   ├── model_factory.py
    │   ├── object_detection
    │   │   ├── tf_object_detection.py
    │   │   └── torchvision_object_detection.py
    │   ├── tfhub_model.py
    │   ├── tfhub_nlp_model.py
    │   └── torchvision_model.py
    ├── tools
    │   ├── cli
    │   │   ├── commands
    │   │   │   ├── eval.py
    │   │   │   ├── list.py
    │   │   │   └── train.py
    │   │   └── main.py
    │   └── docker
    │       └── Dockerfiles
    └── utils
        ├── config_utils.py
        ├── dataset_utils.py
        ├── file_utils.py
        └── model_utils.py
```