```
import os
os.environ["OMP_NUM_THREADS"] = "1"
import flexynesis
import torch
torch.set_num_threads(4)

# parameters cell (required to pass arguments to the notebook) (see
View -> show right sidebar -> add tag -> parameters)
HPO_ITER = 5 # number of HPO iterations for final modeling run
```

Modeling Breast Cancer Subtypes

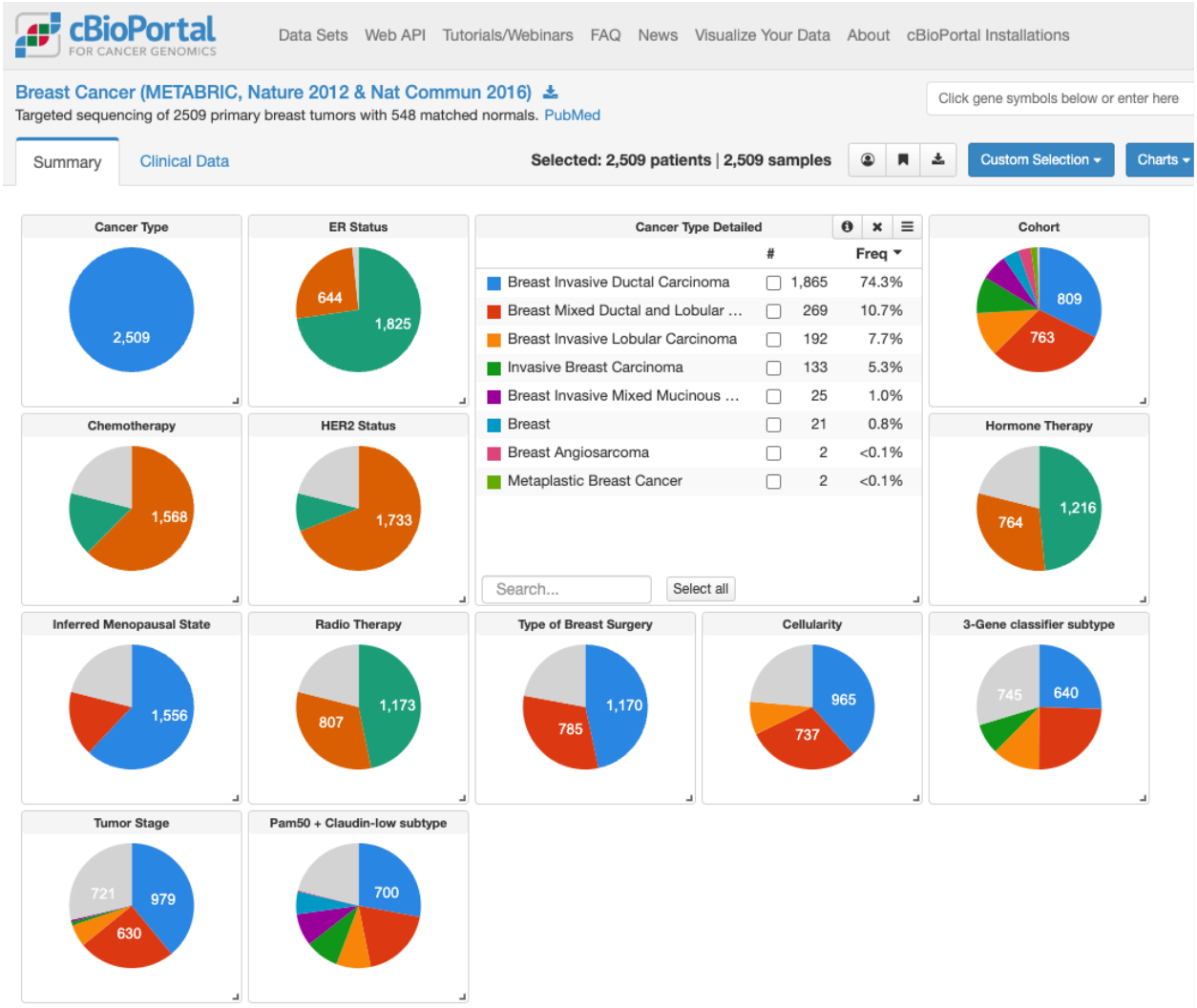
Here, we demonstrate the capabilities of `flexynesis` on a multi-omic dataset of Breast Cancer samples from the METABRIC consortium. The data was downloaded from Cbioportal and randomly split into `train` (70% of the samples) and `test` (30% of the samples) data folders. The data files were processed to follow the same nomenclature.

- `gex.csv` contains "gene expression" data
- `cna.csv` contains "copy number alteration" data
- `mut.csv` contains "mutation" data, which is a binary matrix of genes versus samples.
- `clin.csv` contains "clinical/sample metatada", which is a table of clinical parameters such as age, gender, therapy, subtypes.

Data Download

The data can be downloaded as follows:

```
if not os.path.exists("brca_metabric_processed"):
    !wget -O brca_metabric.tgz
"https://bimsbstatic.mdc-berlin.de/akalin/buyar/flexynesis-benchmark-
datasets/brca_metabric_processed.tgz" && tar -xzvf brca_metabric.tgz
```



Let's check the number of samples and number of features in the corresponding files under train and test folders:

```
# train data
!wc -l ./brca_metabric_processed/train/*

# test data
!wc -l ./brca_metabric_processed/test/*
```

Importing Multiomics Data Into Flexynesis

Procedure

We use the `flexynesis.DataImporter` class to import multiomics data from the data folders. Data importing includes:

1. Validation of the data folders

2. Reading data matrices
3. Data processing, which includes:
   - Cleaning up the data matrices to:
     - remove uninformative features (e.g. features with near-zero-variation)
     - remove samples with too many NA values
     - remove features with too many NA values and impute NA values for the rest with few NA values
4. Feature selection **only on training data** for each omics layer separately:
   - Features are sorted by Laplacian score
   - Features that make it in the `top_percentile`
   - Highly redundant features are further removed (for a pair of highly correlated features, keep the one with the higher laplacian score).
5. Harmonize the training data with the test data.
   - Subset the test data features to those that are kept for training data
6. Normalize the datasets
   - Normalize training data (standard scaling) and apply the same scaling factors to the test data.
7. (Optional): Log transform the final matrices.
8. Distinguish numerical and categorical variables in the "clin.csv" file. For categorical variables, create a numerical encoding of the labels for training data. Use the same encoders to map the test samples to the same numerical encodings.

Usage

- Here, we import both train/test datasets from the data folder we downloaded and unpacked before.
- We choose which omic layers to import
- We choose whether we want to concatenate the data matrices (early integration) or not (intermediate integration) before running them through the neural networks.
- We want to apply feature selection and keep only top 10% of the features. In the end, we want to keep at least 1000 features per omics layer.
- We apply a variance threshold (for simplicity of demonstration, we want to keep a small number of most variable features). Setting this to 80, will remove 80% of the features with lowest variation from each modality.

```
data_importer = flexynesis.DataImporter(path
='./brca_metabric_processed/',
                                        data_types = ['gex', 'cna'],
                                        concatenate=False,
                                        top_percentile=10,
                                        min_features=100,
                                        variance_threshold=0.8, # set
to 0.8 for 80%
                                       )
train_dataset, test_dataset = data_importer.import_data()
```

- **dataset.dat** contains the data matrices

```
train_dataset.dat

train_dataset.dat['gex'].shape, train_dataset.dat['cna'].shape
```

- dataset.ann contains the sample annotation data (from clin.csv), where the keys are variable names and values are tensors.

```
train_dataset.ann
```

- A mapping of the sample labels for categorical variables can be found in dataset.label_mappings

```
train_dataset.label_mappings
```

- As the data matrices are stored as tensors, the row and column names cannot be stored as tensors. These are stored in the same dataset object as:`dataset.samples` and `dataset.features`

```
train_dataset.samples[1:10], train_dataset.features
```

- We can get a summary of sample metadata using `print_summary_stats`. For categorical variables, we can the sample counts per label and for numerical variables, we get mean/median statistics.

```
flexynesis.print_summary_stats(train_dataset)
```

Training flexynesis models

We create a `tuner` object by specifying:

1. `dataset`: the training dataset (as we constructed above)
2. `model_class`: which model architecture to use:
   a) DirectPred: a fully connected network (standard multilayer perceptron) with supervisor heads (one MLP for each target variable)
   b) Supervised Variational Autoencoder: A variational autoencoder (MMD-loss) with supervisor heads (one MLP for each target variable)
   c) MultiTripletNetwork: A network structured in triplets to enable contrastive learning (using triplet loss) and additiona supervisor heads (one MLP for each target variable)
3. `target_variables`: A comma separated list of target variables (specify the column headers from the clin.csv).
   - One MLP per each target variable will be created.
   - The target variables may contain NA values
4. `config_name`: which hyperparameter search space configuration to use.
5. `n_iter`: How many hyperparameter search steps to implement.

- This example runs 1 hyperparameter search step using DirectPred architecture and a hyperparameter configuration space defined for "DirectPred" with a supervisor head for "CLAUDIN_SUBTYPE" variable:

- We use `perform_tuning` function to run the hyperparameter optimisation procedure. At the end of the parameter optimisation, best model will be selected and returned.

Early Stopping

Training a model longer than needed causes the model to overfit, yield worse validation performance, and also it takes a longer time to train the models, considering if we have to run a long hyperparameter optimisation routine, not just for 1 step, but say more than 100 steps.

It is possible to set `early stopping` criteria in flexynesis, which is basically a simple callback that is handled by `Pytorch Lightning`. This is regulated using the `early_stop_patience`. When set to e.g. 10, the training will stop if the validation loss has not been improved in the last 10 epochs.

One can also visualize the training setting `plot_losses` to `True`. This will print the loss values training/validation splits and also the individual loss values for each target variable. In this case, the total loss value for the training equals the loss value of the single variable we chose.

```
tuner = flexynesis.HyperparameterTuning(dataset = train_dataset,
                                        model_class =
flexynesis.DirectPred,
                                        target_variables =
["CLAUDIN_SUBTYPE"],
                                        config_name = "DirectPred",
                                        n_iter=1,  plot_losses=True,
early_stop_patience=10)
model, best_params = tuner.perform_tuning()
```

- One can also provide own parameter optimisation spaces via a `yaml` file as input:
- We can also provide multiple target variables as input. This will create multiple MLP heads (one per variable) and the network will be trained to learn to predict both variables.
- We can mix numerical and categorical variables. The relevant network structure and evaluation procedures will be applied depending on the type of variable

Longer Training

In reality, hyperparameter optimisation should run for multiple steps so that the parameter search space is large enough to find a good set. However, for demonstration purposes, we only run it for 5 steps here.

```
tuner = flexynesis.HyperparameterTuning(dataset = train_dataset,
                                        model_class =
flexynesis.DirectPred,
                                        target_variables =
["CLAUDIN_SUBTYPE"],
                                        config_name = "DirectPred",
```

```
                                                n_iter=HPO_ITER,
plot_losses=True,
                                                early_stop_patience=10)
model, best_params = tuner.perform_tuning()

model

best_params
```

Prediction and Model Evaluation

We can use the best model (chosen based on the hyperparameter optimisation procedure) to make predictions on the test dataset

```
y_pred_dict = model.predict(test_dataset)

y_pred_dict
```

- The predictions are class labels for both variables. Now, we can run `evaluate_wrapper` to evaluate all predictions. The wrapper goes through each variable and figures out which type of evaluation to apply to the corresponding variable (whether to report metrics relevant to regression tasks or classification tasks)

```
metrics_df = flexynesis.evaluate_wrapper(method = 'DirectPred',
y_pred_dict=y_pred_dict, dataset = test_dataset)
metrics_df
```

Extracting the sample embeddings

All models trained within `flexynesis` comes with a `transform` method, which extracts the sample embeddings that are generated by the encoding networks (whether it is an MLP or a variational autoencoder). The embeddings reflect a merged representation of multiple omic layers.

```
ds = test_dataset
E = model.transform(ds)

E.head()
```

Visualizing the sample embeddings

Let's visualize the embeddings in a reduced space and color by the target variables.

```
f = 'CLAUDIN_SUBTYPE'
labels = [ds.label_mappings[f][x] for x in ds.ann[f].numpy()] #map the
sample labels from numeric vector to initial labels.

flexynesis.plot_dim_reduced(E, labels, color_type = 'categorical',
method='pca')
```

We can also use UMAP visualisation

```
flexynesis.plot_dim_reduced(E, labels, color_type = 'categorical',
method='umap')
```