Multitask Recommender Systems for Cancer Drug Response

Release 9/23/2020

Sep 26, 2020

Contents:

Hello, this is intended to be technical documentation that will make it easier to use data pipeline and the methods for recommender systems in this project. So far, this proejct consists of wrappers around open source software purposed for multitask learning between datasets containing drug/cancer interactions. For now, these docs aren't intended for developers trying to use this as an API but just collaborators who might want to use the code directly.

If you are reading this and have any suggestions for features to be added / changes to be made email me at ladd12@llnl.gov

1 Introduction

1.1 Background

The goal of this code is to accurately model the interactions between cancer drugs and cancer cell lines across multiple datasets. Datasets can be from:

- National Cancer Institute (NCI)[1] [Sch11]
- Broad Institute Cancer and Cell Line Encyclopedia[2] [eal19]
- Genomics and Drug Sensitivity in Cancer[3]
- National Institute of Health Clincial Trials Reporting Program[4][Sch11]

1.2 Problem Statement

We need to have each of the following things from each dataset: a sparse ratings/interaction matrix indicating the (IC50) effective concentrations of each drug for each type of cancer and features for each respective drug and cell line. Here are some examples of services that generate theses features:

- Dragon7 (discontinued)[5]
- MOE[6]
- Lincs Cell Features[7]

Then we try to fit machine learning models to this data **to create valid predictors what the outcome of future drug and cancer line interactions will be** in order to inform future experiments and clinical doctors. Finally, in this introduction it will be useful to outline what ML methods we are using and some basic properties.

1 https://www.cancer.gov/
2 https://portals.broadinstitute.org/ccle
3 https://www.cancerrxgene.org/
4 https://www.cancer.gov/about-nci/organization/ccct/ctrp
5 https://chm.kode-solutions.net/products_dragon.php
6 https://www.chemcomp.com/Products.htm
7 http://www.lincsproject.org/

1.3 Methods

Table 1: ML Methods Used

Method	MTL or STL	Feature Based?	Source
Collaborative Filtering Matrix Factorization	STL and MTL	Feature Based	Surprise [Hug20]
K Nearest Neighbors	STL	Not Feature Based	Surprise [Hug20]
Nonnegative Matrix Factorization	STL	Feature Based	Surprise [Hug20]
Feedforward Neural Net	STL	Feature Based	Custom w/ Pytorch [PGM+19]
Gaussian Process	STL and MTL	Feature Based	Gpytorch [GPB+18]
Neural Collaborative Filtering	STL and MTL	Both	Author Github [HLZ+17]

2 Installation

2.1 Using Conda

1. Navigate to root directory, where environment.yml is located
2. conda env create -f environment.yml
3. conda activate mtl4c_env
4. Verify that environment installed correctly with conda env list

Open to any other suggestions for env sharing, like docker.

3 Data

3.1 Synthetic Data

We can't publish real data, so instead there is a file that will load synthetic data. This file and it's attributes are important for adapting this software. If you want to use this code, the best way would be to create a loader class with similar methods to the methods outlined below.

`class SyntheticDataCreator`(*num_tasks=1*, *cellsPerTask=300*, *drugsPerTask=10*, *function='gaussian'*, *normalize=True*, *noise=0.1*, *graph=False*, *test_split=0.3*, ***kwargs*)

create synthetic data

Args:

- **`num_tasks` (int):** number of tasks to create
- **`cellsPerTask` (int):** number of cells to make for each task, this is dimension n of our ratings matrix
- **`drugsPerTask` (int):** number of drugs to make for each task, this is dimension m of our ratings matrix
- **`sparsityPct` (int (0,100)):** this is the amount of sparsity to put on the ratings matrix, the a higher percentage corresponds to a more sparse prediction matrix. Must be between 0 and 100.
- **`function` (string):** gaussian or cosine indicating recipe for synthetic data
- **`normalize` (boolean):** Boolean, whether data should be normalized
- **`test_split` (float [0,1]):** determines size of training and testing data
- **`noise` (boolean):** amount of noise to use in creating synthetic data, the higher this value is the less correlation between generated tasks

Returns object with:

- **`self.datasets` (list):** list of strings indicating dataset names
- **`self.data` (dict):** multilevel dictionary with keys for train/test, then keys for x,y, then finally keys for dataset name. ie: self.data['train']['x'][name1] gives training data for task 1. The models are built correspondingly.

self.trainRatings (np.array): array of ratings with shape (n,m) where n is the number of training cells / task and m is the number training of drugs / task

self.testRating (np.array): array of ratings with shape (n,m) where n is the number of test cells / task and m is the number training of drugs / task

create_x_and_y()
Wrapper for prepare data

generateCosSynthData(*num_tasks=1*, *ptsPerTask=1000*, *noise=0.1*, *graph=False*)
Method used to generate synthetic data with the cosine function. This function selects set of uniform points on interval 0 to 1 and scales them each on intervals of 2c*pi Where c is in range [1,nfeatures] user can set nfeatures in optional_params.txt. Each feature maps to the same y value because they are shfited by one period. Then finally, some noise is added to each y, in order to control the correlation between tasks. The more noise -> the less correlation.

generateSynthData(*num_tasks=1*, *ptsPerTask=1000*, *noise=0.1*, *graph=False*)
Generates gaussian synthetic data. Coefficients apply common linear transformation to multivariate gaussian vectors. Number of gaussian vectors = nfeatures and cna be changed in optional_params.txt to add/remove features. Some noise added to coefficients to control correlation, similar to cosine function more noise -> less correlation.

prepare_data()
Run this public method to prepare data

set_test_split(*new_test_split*)
Update the dict of test split.

shuffle_and_split()
As stated in the name, shuffles and splits data again. Will fail if data has not been initialized

3.2 Example

```
from datasets import SyntheticData as SD

dataset = SD.SyntheticDataCreator(num_tasks=3,cellsPerTask=400, drugsPerTask=10,␣
↪function="cosine",
             normalize=False, noise=1, graph=False, test_split=0.3)

dataset.prepare_data()
```

Now that we have instantiated dataset object, we can use dataset.data as a dictionary to access all the data

```
task0_train_x = dataset.data['train']['x']['0']
task0_train_y = dataset.data['train']['y']['0']
task0_test_x = dataset.data['test']['x']['0']
task0_test_y = dataset.data['test']['y']['0']
```

4 Base Classes

class BaseEstimator(*name*, *type_met*, *paradigm*, *output_shape*)
Abstract class representing a generic STL Method.

abstract evaluate()
Perform prediction.

Args `x` (np.array): np.array w/shape (nsamples,nfeatures) `y` (np.array): np.array w/shape (nsamples,1)

Return `results` (np.array): np.array of errors

abstract fit()
fit model parameters

Args `x` (np.array): np.array w/shape (nsamples,nfeatures) `y` (np.array): np.array w/shape (nsamples,1)

abstract predict()
Perform prediction.

Args `x` (np.array): np.array w/shape (nsamples,nfeatures)

set_output_directory(*output_dir*)
Set output folder path.

Args: `output_dir` (str): path to output directory.

abstract set_params()
Set method's parameters for optuna

class BaseMTLEstimator(*name*, *type_met*)
Base class for multitask learning estimators

fit(*x*, ***kwargs*)
fit model parameters

Args `x` (dict): dictionary with keys corresponding to feature vectors for each task eg: {"CCLE": np.array w/shape (nsamples,nfeatures)} `y` (dict): dictionary with keys corresponding to output vectors for each task eg: {"CCLE": np.array w/shape (nsamples,1)}

predict(*x*, ***kwargs*)
predict model parameters

Args x (dict): dictionary with keys corresponding to feature vectors for each task
eg: {“CCLE”: np.array w/shape (nsamples,nfeatures)}

5 Single Task Learning

These methods, mainly from Surprise (citation), offer clear recommender system benchmarks.

class `SVD_MF`(*n_factors*, *n_epochs=50*, *name='SVD_MF'*)
Bases: `methods.base.BaseSurpriseSTLEstimator`

Matrix Factorization

Args:

`n_factors` (int): number of latent vectors/factors for matrix factorization

`n_epochs` (int): Integer, The number of iteration of the SGD procedure. Default is 20

see https://surprise.readthedocs.io/en/stable/matrix_factorization.html for more info

class `NonNegative_MF`(*n_factors*, *n_epochs=50*, *name='NonNegative_MF'*)
Bases: `methods.base.BaseSurpriseSTLEstimator`

Nonnegative Matrix Factorization

Args:

`n_factors` (int): number of latent vectors/factors for matrix factorization

`n_epochs` (int): Integer, The number of iteration of the SGD procedure. Default is 20

see https://surprise.readthedocs.io/en/stable/matrix_factorization.html for more info

class `KNN_Basic`(*k*, *name='KNN_Basic'*, *sim_options=None*)
Bases: `methods.base.BaseSurpriseSTLEstimator`

Args:

`k` (int): number of neighbors

`sim_options` (optional): option from surprise for a similarity metric

class `NN`(*input_dim*, *arch*, *activation*)
Bases: `torch.nn.modules.module.Module`

Vanilla Neural Network implementation

Args:

`input` (int): dimension of input data

`arch` (list): list specifying architecture for each layer

`activation` (string): string specifying what activation to use ie: “ReLU” or “Sigmoid” or “TanH”

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

6 Neural Collaborative Filtering

class Neural_Collaborative_Filtering_Features(*hyperparams, name='Neural_Collaborative_Filtering_Features', type_met='feature_based', paradigm='stl', output_shape=None, warm_start=False, learner=None, learning_rate=None, reg_mf=None, num_factors=None*)

Bases: *methods.base.BaseEstimator*

Neural Collaborative Filtering adapted from https://github.com/hexiangnan/neural_collaborative_filtering

Combines matrix factorization and Multilayer Perceptron. **Uses cell and drug features**

Args:

hyperparams (dict):

dictionary containing keys for each hyperparameter.

num_epochs (int): number of epochs to train for

batch_size (int): size of each batch in training epochs

mf_dim (int): number of factors to be used by matrix factorization

layers (list): list describing architecure for multilayer perceptron. ie: [32,16,8]

reg_mf (float): regularization penalty for matrix factorization

reg_layer (list): list describing architecure for regularizing multilayer perceptron. ie: [32,16,8]. Must match length of layers

num_negatives : ignore this, deprecated

learning_rate (int): learning rate for gradient descent weight optimization

learner (string): name of learner to use. Options are sgd, adam, rmsprop, decayed sgd, scheduled sgd, adagrad

`warmstart` (boolean): whether to instantiate a model for each task or keep training the same one.

Ignore the other arguments, they are there to pass in hyperparameters with optuna.

class `Neural_Collaborative_Filtering`(*hyperparams*, *name='Neural_Collaborative_Filtering'*, *type_met='non_feature_based'*, *paradigm='stl'*, *output_shape=None*, *warm_start=False*, *learner=None*, *learning_rate=None*, *reg_mf=None*, *num_factors=None*)

Bases: *`methods.base.BaseEstimator`*

Neural Collaborative Filtering adapted for regression from https://github.com/hexiangnan/neural_collaborative_filtering

Combines matrix factorization and Multilayer Perceptron. **Does not use cell and drug features**

Args:

`hyperparams` (dict):

dictionary containing keys for each hyperparameter.

`num_epochs` (int): number of epochs to train for

`batch_size` (int): size of each batch in training epochs

`mf_dim` (int): number of factors to be used by matrix factorization

`layers` (list): list describing architecure for multilayer perceptron. ie: [32,16,8]

`reg_mf` (float): regularization penalty for matrix factorization

`reg_layer` (list): list describing architecure for regularizing multilayer perceptron. ie: [32,16,8]. Must match length of layers

`num_negatives` : ignore this, deprecated

`learning_rate` (int): learning rate for gradient descent weight optimization

`learner` (string): name of learner to use. Options are sgd, adam, rmsprop, decayed sgd, scheduled sgd, adagrad

`warmstart` (boolean): whether to instantiate a model for each task or keep training the same one.

Ignore the other arguments, they are there to pass in hyperparameters with optuna.

class `Neural_Collaborative_Filtering_FeaturesMTLMLP`(*hyperparams*, *name='Neural_Collaborative_Filtering_Featu* *type_met='feature_based'*, *paradigm='mtl'*, *output_shape=None*, *warm_start=False*, *learner=None*, *learning_rate=None*, *reg_mf=None*, *num_factors=None*)

Bases: *methods.base.BaseMTLEstimator*

NCF adapted for multitask model that first trains shared MLP on pooled data, then trains a seperate GMF model for each task.

Combines matrix factorization and Multilayer Perceptron. **Uses cell and drug features**

Args:

hyperparams (dict):

dictionary containing keys for each hyperparameter. `num_epochs` (int):

number of epochs to train for

batch_size (int): size of each batch in training epochs

mf_dim (int): number of factors to be used by matrix factorization

layers (list): list describing architecure for multilayer perceptron. ie: [32,16,8]

reg_mf (float): regularization penalty for matrix factorization

reg_layer (list): list describing architecure for regularizing multilayer perceptron. ie: [32,16,8]. Must match length of layers

learning_rate (int): learning rate for gradient descent weight optimization

learner (string): name of learner to use. Options are sgd, adam, rmsprop, decayed sgd, scheduled sgd, adagrad

mlp_lr (float): (0,1) float for learning rate for pooled MLP model.

warmstart (boolean): whether to instantiate a model for each task or keep training the same one.

Ignore the other arguments, they are there to pass in hyperparameters with optuna.

class Neural_Collaborative_Filtering_FeaturesMTLMF(*hyperparams, name='Neural_Collaborative_Filtering_Featur type_met='feature_based', paradigm='mtl', output_shape=None, warm_start=False, learner=None, learning_rate=None, reg_mf=None, num_factors=None*)

Bases: *methods.base.BaseMTLEstimator*

NCF adapted for multitask model that first trains shared MF on pooled data, then trains a seperate MLP model for each task.

Combines matrix factorization and Multilayer Perceptron. **Uses cell and drug features**

Args:

hyperparams (dict):

dictionary containing keys for each hyperparameter.

num_epochs (int): number of epochs to train for

`batch_size` (int): size of each batch in training epochs

`mf_dim` (int): number of factors to be used by matrix factorization

`layers` (list): list describing architecure for multilayer perceptron. ie: [32,16,8]

`reg_mf` (float): regularization penalty for matrix factorization

`reg_layer` (list): list describing architecure for regularizing multilayer perceptron. ie: [32,16,8]. Must match length of layers

`learning_rate` (int): learning rate for gradient descent weight optimization

`learner` (string): name of learner to use. Options are sgd, adam, rmsprop, decayed sgd, scheduled sgd, adagrad

`mf_lr` (float): (0,1) float for learning rate for pooled MF model.

`warmstart` (boolean): whether to instantiate a model for each task or keep training the same one.

Ignore the other arguments, they are there to pass in hyperparameters with optuna.

7 Gaussian Processes

7.1 Single Task GPs

class ExactGPRegression(*name='ExactGP', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale=100.0, output_scale=1.0*)

Bases: `methods.base.BaseOwnSTLEstimator`

Exact GP, Gaussian Process evaluated at all training points

Args:

`name` (optional, string): model name

`num_iters` (int): number of iterations for Gaussian Process

`learning_rate` (int): learning rate for conjugate gradient. recommended around .1 or .01

`noise_covar` (float): hyperparamter, noise assumed in the data

`lengthscale` (float): hyperparameter, magnitude relative to assumed correlation in data

`output_scale` (optional, float): scaling parameter

class ExactGPCompositeKernelRegression(*name='ExactGPCompositeKernel', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale_cell=100.0, output_scale_cell=1.0, length_scale_drug=100.0, output_scale_drug=1.0*)

Bases: `methods.base.BaseOwnSTLEstimator`

Exact GP where seperate Kernels are evaluated for drugs and cells and then muliplied or added to make a shared kernel, Gaussian Process evaluated at all training points

Args:

`name` (optional, string): model name

`num_iters` (int): number of iterations for Gaussian Process

`learning_rate` (int): learning rate for conjugate gradient. recommended around .1 or .01

`noise_covar` (float): hyperparamter, noise assumed in the data

`length_scale_cell` (float): hyperparameter, magnitude relative to assumed correlation in **cell** data

`length_scale_drug` (float): hyperparameter, magnitude relative to assumed correlation in **drug** data

`output_scale_drug` (optional, float): scaling parameter for **drug** data

`output_scale_cell` (optional, float): scaling parameter for **cell** data

`class SparseGPRegression`(*name='SparseGP', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale=100.0, output_scale=1.0, n_inducing_points=500, use_initial=True*)

Bases: `methods.base.BaseOwnSTLEstimator`

Sparse GP, Gaussian Process evaluated only at N inducing points sampled from training points

Args:

`name` (optional, string): model name

`num_iters` (int): number of iterations for Gaussian Process

`learning_rate` (int): learning rate for conjugate gradient. recommended around .1 or .01

`noise_covar` (float): hyperparamter, noise assumed in the data

`lengthscale` (float): hyperparameter, magnitude relative to assumed correlation in data

`output_scale` (optional, float): scaling parameter

`n_inducing_points` (optional, int): number of training points to sample from for Gaussian Process

`class SparseGPCompositeKernelRegression`(*name='SparseGPCompositeKernel', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale_cell=100.0, output_scale_cell=1.0, length_scale_drug=100.0, output_scale_drug=1.0, n_inducing_points=500*)

Bases: `methods.base.BaseOwnSTLEstimator`

Sparse GP where seperate Kernels are evaluated for drugs and cells and then muliplied or added to make a shared kernel, Gaussian Process evaluated at only n_inducing training points

Args:

`name` (optional, string): model name

`num_iters` (int): number of iterations for Gaussian Process

`learning_rate` (int): learning rate for conjugate gradient. recommended around .1 or .01

`noise_covar` (float): hyperparamter, noise assumed in the data

`length_scale_cell` (float): hyperparameter, magnitude relative to assumed correlation in **cell** data

`length_scale_drug` (float): hyperparameter, magnitude relative to assumed correlation in **drug** data

`output_scale_drug` (optional, float): scaling parameter for **drug** data

`output_scale_cell` (optional, float): scaling parameter for **cell** data

`n_inducing_points` (optional, int): number of training points to sample from for Gaussian Process

7.2 MultiTask GPs

class HadamardMTL(*name='HadamardMTL', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale=100.0, output_scale=1.0, n_inducing_points=500, composite=False, validate=False, bias=False, stabilize=False, use_initial=True*)

Bases: `methods.base.BaseMTLEstimator`

pipeline suited implementation of https://docs.gpytorch.ai/en/v1.1.1/examples/03_Multitask_Exact_GPs/Hadamard_Multitask_GP_Regression.html

Args:

`name` (optional, string): model name

`num_iters` (int): number of iterations for Gaussian Process

`learning_rate` (int): learning rate for conjugate gradient. recommended around .1 or .01

`noise_covar` (float): hyperparamter, noise assumed in the data

`lengthscale` (float): hyperparameter, magnitude relative to assumed correlation in data

`output_scale` (optional, float): scaling parameter

`n_inducing_points` (optional, int): number of training points to sample from for Gaussian Process

`composite` (bool): whether to use composite kernel or not

`validate` (bool): whether to produce validation curve data as well during training

`bias` (bool): whether to add bias term for each dataset

`stabilize` (bool): whether to stabilize loss at the end

`use_initial` (bool): whether to even use initial parameters

class GPyFullMTL(*name='fullGP', num_iters=50, learning_rate=0.1, noise_covar=1.0, length_scale=100.0, output_scale=1.0, n_inducing_points=500, use_initial=True, num_tasks=1, validate=False*)

Bases: *methods.base.BaseMTLEstimator*

Adapted of Full MultiTask GP model from GpyTorch

Args:

name (optional, string): model name

num_iters (int): number of iterations for Gaussian Process

learning_rate (int): learning rate for conjugate gradient. recommended around .1 or .01

noise_covar (float): hyperparamter, noise assumed in the data

lengthscale (float): hyperparameter, magnitude relative to assumed correlation in data

output_scale (optional, float): scaling parameter

n_inducing_points (optional, int): number of training points to sample from for Gaussian Process

bias_only (bool): deprecated. Do not use.

num_tasks (int): number of tasks you are giving model, should be equal to number of datasets

8 Optuna Example Hyperparameter Optimization KNN, SVD, NNMF

In order for the code below to work for a different method/model, the model should have member functions like this:

```
def get_hyper_params(self):
    hparams = {'num_factors': {'type': 'integer', 'values': [2, 10]},
               'rho_1': {'type': 'loguniform', 'values': [1e-3, 100]},
               'rho_2': {'type': 'loguniform', 'values': [1e-3, 100]}}
    return hparams

def set_hyper_params(self, **kwargs):
    self.num_factors = kwargs['num_factors']
    self.rho1 = kwargs['rho_1']
    self.rho2 = kwargs['rho_2']
```

[2]:
```
import os, sys
sys.path.append('../')
import hp_optimization as hopt
from optuna.visualization import plot_optimization_history, \
                                 plot_intermediate_values, \
                                 plot_contour

from design import ModelTraining
from methods.matrix_factorization.MF_STL import MF_STL
from methods.matrix_factorization.MF import SVD_MF, NonNegative_MF
from methods.knn.KNN import KNN_Normalized
from shutil import copyfile
from UTILS.utils import datasetParams2str
from datasets import SyntheticData as SD

outdir = '../outputs/experiment_004x' # make sure that it lines up with the experiment
↪'s filename
if not os.path.exists(outdir):
    os.makedirs(outdir)
```

[3]:
```
dataset = SD.SyntheticDataCreator(num_tasks=3,cellsPerTask=400, drugsPerTask=10,␣
↪function="cosine",
             normalize=False, noise=1, graph=False, test_split=0.3)
```

(continues on next page)

(continued from previous page)

```
dataset.prepare_data()
```

[ ]:
```
methods  = [KNN_Normalized(k=10), SVD_MF(n_factors=100), NonNegative_MF(n_
↪factors=100)]
i = 0
for method in methods:
    if i == 0:
        study = hopt.optimize_hyper_params(method, dataset,n_trials=5)
        i += 1
    else:
        study = hopt.optimize_hyper_params(method, dataset,n_trials=50)
    plot_optimization_history(study)
    plot_intermediate_values(study)
    plot_contour(study)
    print("best params for "+ method.name + " : ",study.best_params)
    # copy the study, i.e. hyperparam trials
    dataset_str = datasetParams2str(dataset.__dict__)
    study_name = '{}_{}'.format(method.name,dataset_str)
    storage='hyperparam_experiments/{}.db'.format(study_name)
    copyfile(storage, os.path.join(outdir,study_name + '.db'))
```

[ ]:
```

```

9 Visualizing High-dimensional Parameter Relationships

This notebook demonstrates various visualizations of studies in Optuna. The hyperparameters of a neural network trained to classify images are optimized and the resulting study is then visualized using these features.

Note: If a parameter contains missing values, a trial with missing values is not plotted.

```
[ ]: # If you run this notebook on Google Colaboratory, uncomment the below to install␣
     ↪Optuna.
     #! pip install --quiet optuna
```

SOURCE: https://github.com/optuna/optuna/blob/master/examples/visualization/plot_study.ipynb

9.1 Preparing the Dataset

```
[ ]: from sklearn.datasets import fetch_openml
     from sklearn.model_selection import train_test_split

     mnist = fetch_openml(name='Fashion-MNIST', version=1)
     classes = list(set(mnist.target))

     # For demonstrational purpose, only use a subset of the dataset.
     n_samples = 4000
     data = mnist.data[:n_samples]
     target = mnist.target[:n_samples]

     x_train, x_valid, y_train, y_valid = train_test_split(data, target)
```

9.2 Defining the Objective Function

```
[ ]: from sklearn.neural_network import MLPClassifier

     def objective(trial):

         clf = MLPClassifier(
             hidden_layer_sizes=tuple([trial.suggest_int('n_units_l{}'.format(i), 32, 64)␣
     ↪for i in range(3)]),
             learning_rate_init=trial.suggest_float('lr_init', 1e-5, 1e-1, log=True),
         )

         for step in range(100):
             clf.partial_fit(x_train, y_train, classes=classes)
             value = clf.score(x_valid, y_valid)

             # Report intermediate objective value.
             trial.report(value, step)

             # Handle pruning based on the intermediate value.
             if trial.should_prune():
                 raise optuna.TrialPruned()

         return value
```

9.3 Running the Optimization

```
[ ]: import optuna

     optuna.logging.set_verbosity(optuna.logging.WARNING)  # This verbosity change is just␣
     ↪to simplify the notebook output.

     study = optuna.create_study(direction='maximize', pruner=optuna.pruners.
     ↪MedianPruner())
     study.optimize(objective, n_trials=100)
```

9.4 Visualizing the Optimization History

```
[ ]: from optuna.visualization import plot_optimization_history

     plot_optimization_history(study)
```

9.5 Visualizing the Learning Curves of the Trials

```
[ ]: from optuna.visualization import plot_intermediate_values

     plot_intermediate_values(study)
```

9.6 Visualizing High-dimensional Parameter Relationships

```
[ ]: from optuna.visualization import plot_parallel_coordinate

     plot_parallel_coordinate(study)
```

```
[ ]: plot_parallel_coordinate(study, params=['lr_init', 'n_units_l0'])
```

9.7 Visualizing Parameter Relationships

```
[ ]: from optuna.visualization import plot_contour

     plot_contour(study)
```

```
[ ]: plot_contour(study, params=['n_units_l0', 'n_units_l1'])
```

9.8 Visualizing Individual Parameters

```
[ ]: from optuna.visualization import plot_slice

     plot_slice(study)
```

```
[ ]: plot_slice(study, params=['n_units_l0', 'n_units_l1'])
```

9.9 Visualizing Parameter Importances

```
[ ]: from optuna.visualization import plot_param_importances

     plot_param_importances(study)
```

10 STL and MTL GP Regression Walkthrough

[3]:
```
import sys
sys.path.append('../')
#from design import ModelTraining
from datasets import SyntheticData as SD
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
from time import time
import matplotlib.pyplot as plt
import methods.mtl.MTL_GP as MtlGP
import os
import numpy as np
import matplotlib
import seaborn as sns
```

10.1 Setting Up Datasets

The very first step to running through these Gaussian Process Tutorials is retrieving some data to train our models on. Here we are using the CTRP, GDSC and CCLE datasets mentioned in the introduction.

[4]:
```
import importlib
importlib.reload(MtlGP)
dataset = SD.SyntheticDataCreator(num_tasks=3,cellsPerTask=400, drugsPerTask=10,␣
↪function="cosine",
             normalize=False, noise=1, graph=False, test_split=0.3)
dataset.prepare_data()
```

10.2 Single Task Gaussian Process Example

below is an exaple of training and testing a basic Sparse Gaussian Process from gpytorch with our data.

```
[ ]: import methods.regressor.SparseGP as SGP
importlib.reload(SGP)

y_pred = {}
sparsegp = SGP.SparseGPRegression(num_iters=50, length_scale=50, noise_covar=1.5, n_
↪inducing_points=250)
for k in dataset.datasets:
    sparsegp.fit(dataset.data['train']['x'][k],
                 y=dataset.data['train']['y'][k],
                 cat_point=dataset.cat_point)
    y_pred[k] = sparsegp.predict(dataset.data['test']['x'][k])

for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name] - dataset.data['test']['y'][name]) ** 2) /␣
↪len(y_pred[name])))
    print(rmse, name)
```

Next, we have a more complex method, composite kernel Gaussian Process Regression

```
[ ]: import methods.regressor.SparseGPCompositeKernel as sgpc
importlib.reload(sgpc)
y_pred = {}
sparsegpcomp = sgpc.SparseGPCompositeKernelRegression(num_iters=10, length_scale_
↪cell=100, length_scale_drug=100, noise_covar=1.5, n_inducing_points=500, learning_
↪rate=.1)
for k in dataset.datasets:
    sparsegpcomp.fit(dataset.data['train']['x'][k],
                 y=dataset.data['train']['y'][k],
                 cat_point=dataset.cat_point)
    y_pred[k] = sparsegpcomp.predict(dataset.data['test']['x'][k])

for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name] - dataset.data['test']['y'][name]) ** 2) /␣
↪len(y_pred[name])))
    print(rmse, name)
```

10.3 Multitask Background

Given a set of observations y_0 we wish to learn parameters θ_x and k^x of the matrix K_f. k^x is a covariance function over the inputs and θ_x are the parameters for that specific covariance function

10.4 Hadamard Product MTL

A clear limitation of the last method is that although it is technically multitask, it will fail to capture most task relationships. In order to do this I'll introduce another spin on vanilla GP Regression.

Now we just have one model parameterized as:

$$\begin{aligned} y_i &= f(x_i) + \varepsilon_i \\ f &\sim \mathcal{GP}(C_t, K_\theta) \\ \theta &\sim p(\theta) \\ \varepsilon_i &\overset{iid}{\sim} \mathcal{N}(0, \sigma^2) \end{aligned}$$

With one key difference. Our kernel is now defined as: $K([x,i],[x',j]) = k_{inputs}(x,x') * k_{tasks}(i,j)$ where $ k_{tasks} $ is an "index kernel", essentially a lookup table for inter-task covariance. This lookup table is defined $\forall$ $i, j \in$ the set of tasks T. Here's a basic example with 4 datapoints and 2 tasks.

```
[ ]: importlib.reload(MtlGP)

     hadamardMTL = MtlGP.HadamardMTL(num_iters=300, length_scale=20, noise_covar=.24, n_
     ↪inducing_points=500, \
                                     composite=False, learning_rate=.07, validate=False,
     ↪bias=False,stabilize=False)


     hadamardMTL.fit(dataset.data['train']['x'],
                                        y=dataset.data['train']['y'],
                                        catpt=dataset.cat_point)
```

```
[ ]: y_pred = hadamardMTL.predict(dataset.data['test']['x'])
     for name in y_pred.keys():
         rmse = np.sqrt(np.sum(((y_pred[name].numpy() - dataset.data['test']['y'][name])␣
     ↪** 2) / len(y_pred[name])))
         print(rmse, name)
```

10.5 Example Visualizing Covariance Using Getter

```
[ ]: full_covar = hadamardMTL.model.getCovar().numpy()
     plt.imshow(full_covar)
     plt.imshow(hadamardMTL.model.getCovar().numpy())
```

```
[ ]: from mpl_toolkits.axes_grid1 import make_axes_locatable
     fig, ax = plt.subplots()
     task_covar = hadamardMTL.model.getTaskCovar().numpy() # cast from torch to numpy
     im = ax.imshow(task_covar, cmap="Reds")
     ax.set_xticks([200,800,1300])
     ax.set_xticklabels(dataset.datasets)
     ax.set_yticks([200,800,1300])
     ax.set_yticklabels(dataset.datasets)
     divider = make_axes_locatable(ax)
     cax = divider.append_axes("right", size="10%", pad=0.5)
     cbar = plt.colorbar(im, cax = cax)
```

10.6 Full Multitask GP with Multitask Kernel

```
[ ]: importlib.reload(MtlGP)

     gpymtl = MtlGP.GPyFullMTL(num_iters=300, length_scale=15, noise_covar=1, n_inducing_
     ↪points=200,  num_tasks=3, learning_rate=.05)


     gpymtl.fit(dataset.data['train']['x'],
                                        y=dataset.data['train']['y'],
                                        cat_point=dataset.cat_point)
```

```
[10]: y_pred = gpymtl.predict(dataset.data['test']['x'])
      i = 0
      for name in y_pred.keys():
          rmse = np.sqrt(np.sum(((y_pred[name] - dataset.data['test']['y'][name]) ** 2) /␣
      ↪len(y_pred[name])))
          i +=  1
          print(rmse, name)
```

```
0.5526002760370554 0
0.721262580126851 1
0.7105683397091712 2
```

10.7 Example Find Initial Conditions

In order to understand what parameters to start at, we can test different configurations of initial conditions

```
[ ]: import importlib
     importlib.reload(MtlGP)
     multiBias = MtlGP.HadamardMTL(num_iters=10, noise_covar=1.5, n_inducing_points=500,␣
     ↪multitask_kernel=False)   #testing #0)

     multiBias._find_initial_conditions(dataset.data['train']['x'], dataset.data['train'][
     ↪'y'], \
                                        n_restarts=800,n_iters=50, n_inducing_points=500)
```

(tensor(1.2674, grad_fn=), {'likelihood.noise_covar.noise': 0.7006388902664185, 'covar_module.lengthscale': 10.444199562072754})

```
[ ]:
```

```
[2]: import sys

     sys.path.append('../')
     from design import ModelTraining
     import matplotlib.pyplot as plt
     import methods.matrix_factorization.FeaturizedNCF as NCF_feat
     import methods.matrix_factorization.CustomInputNCF as NCF
     from datasets import SyntheticData as SD
     import numpy as np
     from sklearn.model_selection import train_test_split
     import pandas as pd
     from time import time
     from UTILS import utils
     import methods.mtl.NCF_MTL as NCF_MTL
```

# NonFeaturized NCF

```
[3]: #%%capture
     import importlib
     importlib.reload(NCF)

     dataset = SD.SyntheticDataCreator(num_tasks=3,cellsPerTask=400, drugsPerTask=10,␣
     ↪function="cosine",
                  normalize=False, noise=1, graph=False, test_split=0.3)
     dataset.prepare_data()

     hyperparams = {'batch_size': 32, 'epochs': 200, 'layers': '[64,32,16,8]', \
                        'learner': 'rmsprop', 'lr': 0.001, 'num_factors': 8, 'num_neg': 4,␣
     ↪\
                        'reg_layers': '[0,0,0,0]', 'reg_mf': 0.0, 'verbose': 1, 'warm_start
     ↪':False}

     NCF1  = NCF.Neural_Collaborative_Filtering(hyperparams, 'name', 'non_feature_based')
     ↪# can be feature based
             # it needs to be non bc models does feature transform
```

(continues on next page)

(continued from previous page)

```
# iterate through datasets in single task learning paradigm
y_pred = {}
for name in dataset.datasets:
    NCF1.fit(x=dataset.trainRatings[name])
    y_pred[name] = NCF1.predict(dataset.testRatings[name])

#rmse
for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name][:,0] - dataset.data['test']['y'][name]) **␣
↪2) / len(y_pred[name])))
    print(rmse, name)
```

```
MODEL HAS BEEN DEFINED
TRAINING…
NCF reinitialized
0
PREDICTING…
TRAINING…
NCF reinitialized
0
75
150
PREDICTING…
TRAINING…
NCF reinitialized
0
75
150
PREDICTING…
0.8048341314316828 0
0.9302123806595449 1
0.9556996063227641 2
```

11 Featurized NCF Example

```
[4]: #%%capture
import importlib
#reload python import so we don't have to start and restart kernel
importlib.reload(NCF_feat)




hyperparams = {'batch_size': 64, 'epochs': 100, 'layers': '[64,32,16,8]', 'learner':
↪'adam', 'lr': 0.001, \
               'num_factors': 16, 'num_neg': 4, 'reg_layers': '[0.01,0,0,0.01]', 'reg_
↪mf': 0.01,\
               'verbose': 1, 'warm_start':False}
NCF2  = NCF_feat.Neural_Collaborative_Filtering_Features(hyperparams,'name', 'feature_
↪based') # can be feature based
        # it needs to be non bc models does feature transform

# iterate through datasets in single task learning paradigm
y_pred = {}
for k in dataset.datasets:
    NCF2.fit(x=dataset.data['train']['x'][k], y=dataset.data['train']['y'][k])
    y_pred[k] = NCF2.predict(dataset.data['test']['x'][k])

#rmse
for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name][:,0] - dataset.data['test']['y'][name]) **␣
↪2) / len(y_pred[name])))
    print(rmse, name)
```

```
model has been defined
TRAINING…
NCF reinitialized
break… model converged
PREDICTING…
(1200, 20)
TRAINING…
NCF reinitialized
break… model converged
PREDICTING…
```

(continues on next page)

(continued from previous page)

```
(1200, 20)
TRAINING…
NCF reinitialized
break… model converged
PREDICTING…
(1200, 20)
0.25058302922958736 0
0.24961774086704824 1
0.25494508025729135 2
```

12

MTL NCF with Pooled MLP Example

```
[5]: #%%capture
import importlib
importlib.reload(NCF_MTL)


hyperparams_mtlmlp = {'batch_size': 64, 'epochs': 150, 'layers': '[64,32,16,8]', \
                     'learner': 'adam', 'lr': .001,'mlp_lr': .001, 'num_factors': 10, \
                     'reg_layers': '[0,0,0,.01]', 'reg_mf': 0.01, 'verbose': 1}


NCF3 = NCF_MTL.Neural_Collaborative_Filtering_FeaturesMTLMLP(hyperparams_mtlmlp,'name
↪', 'feature_based')

NCF3.fit(x=dataset.data['train']['x'],
                                   y=dataset.data['train']['y'],
                                   cat_point=dataset.cat_point)

y_pred = NCF3.predict(dataset.data['test']['x'], dataset.data['test']['y'])

#rmse
for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name][:,0] - dataset.data['test']['y'][name]) **␣
↪2) / len(y_pred[name])))
    print(rmse, name)
```

```
model has been defined
PREDICTING…
0.8660533164359661 0
0.6996997536437438 1
0.624245763614103 2
```

13 MTL NCF with Pooled MF

```
[6]: #%%capture
import importlib
importlib.reload(NCF_MTL)

hyperparams_mtlmlp = {'batch_size': 64, 'epochs': 150, 'layers': '[64,32,16,8]', \
                   'learner': 'adam', 'lr': .001,'mlp_lr': .001, 'num_factors': 10, \
                   'reg_layers': '[0,0,0,.01]', 'reg_mf': 0.01, 'verbose': 1}


NCF3 = NCF_MTL.Neural_Collaborative_Filtering_FeaturesMTLMLP(hyperparams_mtlmlp,'name
↪', 'feature_based')

NCF3.fit(x=dataset.data['train']['x'],
                                    y=dataset.data['train']['y'],
                                    cat_point=dataset.cat_point)

y_pred = NCF3.predict(dataset.data['test']['x'], dataset.data['test']['y'])

#rmse
for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name][:,0] - dataset.data['test']['y'][name]) **␣
↪2) / len(y_pred[name])))
    print(rmse, name)
```

```
model has been defined
PREDICTING…
0.7123474321731185 0
1.7743642422838766 1
0.4966135018616864 2
```

```
[32]: #%%capture
import importlib
importlib.reload(NCF_MTL)

hyperparams_mtlmf = {'batch_size': 64, 'epochs': 150, 'layers': '[64,32,16,8]', \
                   'learner': 'adam', 'lr': .001,'mf_lr': .001, 'num_factors': 10, \
                   'reg_layers': '[0,0,0,.01]', 'reg_mf': 0.01, 'verbose': 1}
```

(continues on next page)

(continued from previous page)

```
NCF4 = NCF_MTL.Neural_Collaborative_Filtering_FeaturesMTLMF(hyperparams_mtlmf,'name',
↪'feature_based')

NCF4.fit(x=dataset.data['train']['x'],
                                   y=dataset.data['train']['y'],
                                   cat_point=dataset.cat_point)

y_pred = NCF4.predict(dataset.data['test']['x'], dataset.data['test']['y'])

#rmse
for name in y_pred.keys():
    rmse = np.sqrt(np.sum(((y_pred[name][:,0] - dataset.data['test']['y'][name]) **␣
↪2) / len(y_pred[name])))
    print(rmse, name)
```

```
PREDICTING…
0.6981196213543954 CCLE
0.5048938257444706 GDSC
0.7784718127828185 CTRP
```

14 Featurized NCF Train Test Curve Example

```
[7]: #%%capture
     import importlib
     #reload python import so we don't have to start and restart kernel
     importlib.reload(NCF_feat)

     hyperparams = {'batch_size': 64, 'epochs': 1, 'layers': '[64,32,16,8]', \
                    'learner': 'adam', 'lr': 0.001, 'mf_pretrain': '', 'mlp_pretrain':
     ↪'', \
                    'num_factors': 16, 'num_neg': 4, 'out': 1, 'path': 'Data/', \
                    'reg_layers': '[0.01,0,0,0.01]', 'reg_mf': 0.01, 'verbose': 1,
     ↪'warm_start':False}
     model  = NCF_feat.Neural_Collaborative_Filtering_Features(hyperparams,'name',
     ↪'feature_based') # can be feature based
             # it needs to be non bc models does feature transform
     epochs = 600
     batch_size = 64
     plot_counter = 0
     fig, axs = plt.subplots(4, figsize=(8,20))
     for k in dataset.datasets:
         print(k)
         train_rmses = []
         test_rmses = []

         for epoch in range(epochs):
             if epoch % 10 == 0:
                 print("epoch : " , epoch)
             t1 = time()
             # Generate training instances
             train_x = dataset.data['train']['x'][k][:30000]
             train_y = dataset.data['train']['y'][k][:30000]
             test_x = dataset.data['test']['x'][k][:10000]
             test_y = dataset.data['test']['y'][k][:10000]
     #         overlap = 0
     #          for y in test_y:
     #              if y in train_y:
     #                  overlap += 1
     #          print(overlap, "Overlap")

             train_hist = model.model.fit({'user_inputs':np.array(train_x[:,:10]), 'item_
     ↪inputs':np.array(train_x[:,10:])} \
```

(continues on next page)

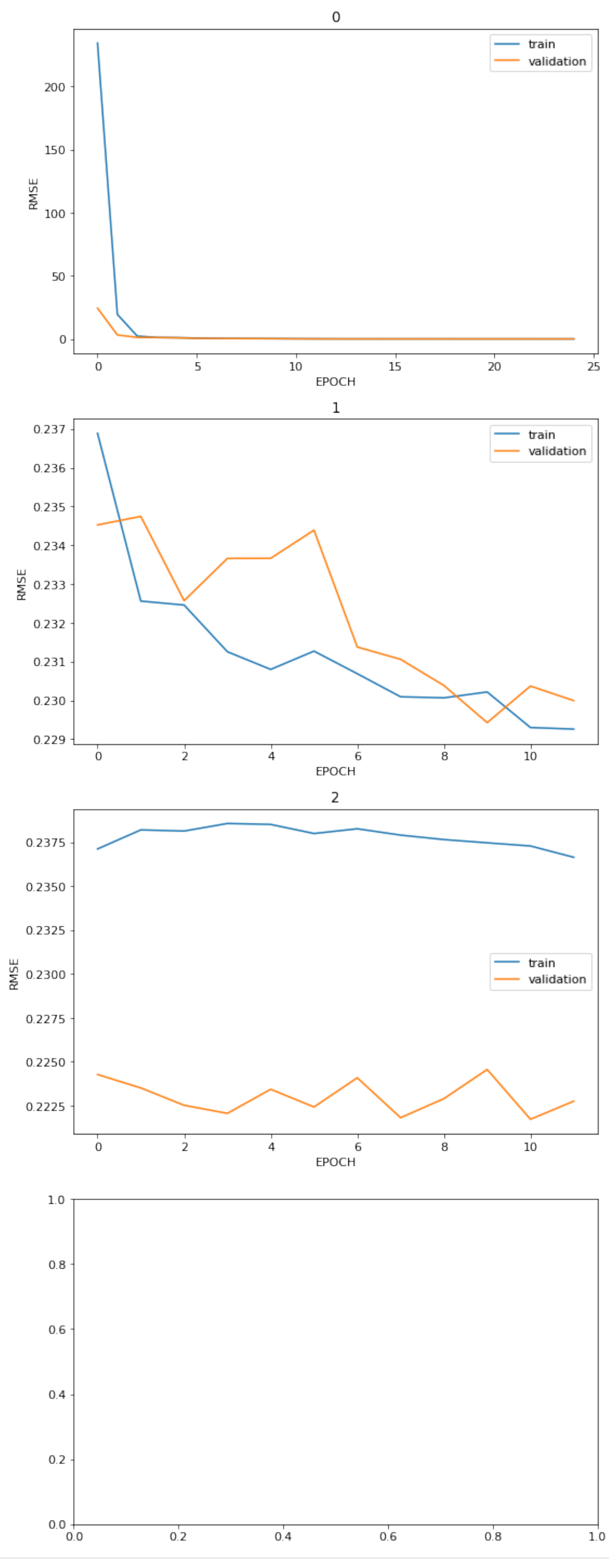
(continued from previous page)

```
                                            , np.array(train_y), batch_size=batch_size,␣
↪epochs=1, verbose=0, shuffle=False)
        test_hist = model.model.evaluate({'user_inputs':np.array(test_x[:,:10]),
↪'item_inputs':np.array(test_x[:,10:])}, np.array(test_y),
                                    batch_size=batch_size, verbose=0,return_dict=True)
        t2 = time()
        #print("train: ",train_hist.history['root_mean_squared_error'], "test: ",␣
↪test_hist['root_mean_squared_error'])

        train_rmses.append(train_hist.history['root_mean_squared_error'])
        test_rmses.append(test_hist['root_mean_squared_error'])
        if epoch > 10 and np.max(train_rmses[epoch-10:epoch] - np.min(train_
↪rmses[epoch-10:epoch])) < .008:
            print("BREAK")
            break
    axs[plot_counter].plot(train_rmses)
    axs[plot_counter].plot(test_rmses)
    axs[plot_counter].set_title(k)
    axs[plot_counter].legend(['train', 'validation'])
    axs[plot_counter].set_ylabel('RMSE')
    axs[plot_counter].set_xlabel('EPOCH')
    plot_counter += 1
    print('min train err: ',min(train_rmses), "min test err: ", min(test_rmses) )
```

```
model has been defined
0
epoch :  0
epoch :  10
epoch :  20
BREAK
min train err:  [0.2359551042318344] min test err:  0.23274201154708862
1
epoch :  0
epoch :  10
BREAK
min train err:  [0.22925961017608643] min test err:  0.22942525148391724
2
epoch :  0
epoch :  10
BREAK
min train err:  [0.23665441572666168] min test err:  0.2217150777578354
```

```
/usr/tce/packages/python/python-3.7.2/lib/python3.7/site-packages/matplotlib/figure.
↪py:2366: UserWarning: This figure includes Axes that are not compatible with tight_
↪layout, so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```


0
train
validation
RMSE
EPOCH
1
2

15 Python Script Example

15.1 Python Script Example

```python
import sys
sys.path.append('../')
from design import ModelTraining
from methods.mtl.MF_MTL import MF_MTL
from methods.matrix_factorization.MF_STL import MF_STL

# from methods.regressor.FFNN import FeedForwardNN
from methods.matrix_factorization.MF import SVD_MF, NonNegative_MF
from methods.knn.KNN import KNN_Normalized
from datasets.DrugCellLines import DrugCellLinesMTL


if __name__ == '__main__':

    drug_transform = {'type': 'pca', 'num_comp': 10}
    cell_transform = {'type': 'pca', 'num_comp': 10}
    dataset = DrugCellLinesMTL(['CCLE', 'GDSC', 'CTRP', 'NCI60'], common=True,
                               unseen_cells=False, normalize=True,
                               test_split=0.2, drug_transform=drug_transform,
                               cell_transform=cell_transform)
    dataset.prepare_data()



    methods = [SVD_MF(n_factors=100),
               KNN_Normalized(k=10)]

    metrics = ['rmse', 'explained_variance_score', 'mae']

    exp_folder = __file__.strip('.py')
    exp = ModelTraining(exp_folder)
    exp.execute(dataset, methods, metrics, nruns=1)
    exp.generate_report()
```

16 Indices and tables

- genindex
- modindex
- search

Bibliography

[eal19] Ghandi et al. Next-generation characterization of the cancer cell line encyclopedia. *Nature*, 569:1–6, 05 2019. doi:10.1038/s41586-019-1186-3[8].

[GPB+18] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. Gpytorch: blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*. 2018.

[HLZ+17] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, 173–182. Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. URL: https://doi.org/10.1145/3038912.3052569, doi:10.1145/3038912.3052569[9].

[Hug20] Nicolas Hug. Surprise: a python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020. URL: https://doi.org/10.21105/joss.02174, doi:10.21105/joss.02174[10].

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Sch11] Manfred Schwab, editor. *NCI 60 Cell Line Screen*, pages 2468–2468. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: https://doi.org/10.1007/978-3-642-16483-5_3987, doi:10.1007/978-3-642-16483-5_3987[11].

[8] https://doi.org/10.1038/s41586-019-1186-3
[9] https://doi.org/10.1145/3038912.3052569
[10] https://doi.org/10.21105/joss.02174
[11] https://doi.org/10.1007/978-3-642-16483-5_3987

Python Module Index

Index