NXTfusion

Release 0.0.1

Daniele Raimondi

Feb 15, 2021

CONTENTS

NXTfusion is a pytorch-based library for non-linear data fusion over Entity-Relation graph.

CHAPTER

ONE

WHAT IS DATA FUSION?

As summarized by Wikipedia.: Data fusion is the process of integrating multiple data sources to produce more consistent, accurate, and useful information than that provided by any individual data source.

In the era of big data , many scientific disciplines are producing enormous amounts of heterogeneous data from which we want to infer reliable predictive or descriptive models. We are thus in a pressing need for powerful, scalable algorithms that integrate multiple sources of information and learn complex patterns from this multi-faceted and interconnected data. To face this challenge we propose a novel data fusion approach for nonlinear inference over arbitrary entity-relation graphs.

1.1 WHAT IS NXTfusion

`NXTfusion` is a Neural Network based data fusion method that extends the classical Matrix Factorization paradigm by allowing non-linear inference over arbitrariy connected Entity-Relation Graphs (ER graphs).

1.2 What is this an Entity-Relation graph?

An ER graph is an abstract data structure, similar to a relational database, that allows to model classes of objects (Entities) and relations between them (Relations).

The ER formalism is a generalization of the well known Matrix Factorization formalism, and indeed we can describe every data fusion problem in terms of entity-relation (ER) models, where entities are classes of objects belonging to a particular domain and relations describe the interactions between entities. Such an arbitrary data fusion model is completely general and could allow inference on an extremely broad class of problems . Moreover, the ease in which entities can be connected through relations would allow the inclusion of data sets that are only loosely related with the problem under investigation.

1.3 NXTfusion approach generalizes existing data fusion methods

In general, we each relation corresponds to a possibly sparsely observed matrix and the entities are the objects represented as rows and columns on that matrix.

In the classical Matrix Factorization paradigm, usually only a single matrix `Y = UV` is factorized into two latent matrices `U` and `V`, meaning that a single interaction (`Y`) between two entities (of which `U` and `V` are the latent representation) is considered. An extension to this is the Tensor Factorization (e.g. https://arxiv.org/abs/1512.00315), where multiple matrices/relations between two entities are factorized at the same time.

Real world data is nevertheless richer than this, and a problem might be characterized by many relations between many pairs of objects, thus forming a complex graph of entities (the nodes) connected by relations (the edges).

Here we further extend the field of data fusion by building a Neural Network-based data fusion framework for non-linear inference over completely arbitrary ER graphs, as we showed here https://doi.org/10.1093/bioinformatics/btab092.

1.4 Examples from the scientific world

Few examples from the scientific world are listed in this non-exhaustive list:

- **drug-protein interaction** predictor, in which Protein and Drugs are entities and the relation between them indicate which drugs interact with which proteins (https://arxiv.org/abs/1512.00315).
- **gene prioritization** (where Gene and Disease are the entity and the relation "gene u is involved in disease v" between them is modeled) (https://doi.org/10.1093/bioinformatics/bty289)
- **protein-protein interaction predictor**, including tensor factorization and inference over arbitrary Entity-Relation graph (https://doi.org/10.1093/bioinformatics/btab092)

1.5 What is this repository for?

The code here contains a pytorch-based `python3` library taht should allow anyone to use our Entity-Relation data fusion framework on your data science problem of choice. An example of its application, on protein-protein interaction is available here: https://bitbucket.org/eddiewrc/nxtppi/src/master/, and it has been published here: https://doi.org/10.1093/bioinformatics/btab092 .

CHAPTER

TWO

INSTALLATION GUIDE

2.1 How do I set it up?

You can install NXTfusion either from the bitbucket (https://bitbucket.org/eddiewrc/nxtfusion/src/master/) repo or from our pypi package (https://pypi.org/project/nxtfusion/).

In either ways, we recommend to follow these steps:

First, set up a dedicated conda environment, to avoid problems with existing softwares.

- Download and install miniconda from https://docs.conda.io/en/latest/miniconda.html
- Create a new conda environment by typing: `conda create -n nxtfusion -python=3.6`

Enter the environment by typing: `conda activate nxtfusion`

2.2 Install from git repo

If you want to install NXTfusion from this repo, you need to install the dependencies first.

Install `pytorch >= 1.0` with the command: `conda install pytorch -c pytorch` or refer to `pytorch` website https://pytorch.org Install the remaining requirements with the command: `conda install scipy numpy multipledispatch` You can remove this environment at any time by typing: `conda remove -n nxtfusion --all`

2.3 Install from pip

If you want to install it via Pypi, just add pip to your conda environment: `conda install pip`

and then just type: `pip install nxtfusion`

and all the required dependencies will be installed automatically. Congrats, you're ready to rock!

2.4 Install from conda

TODO

CHAPTER

THREE

QUICKSTART

The `examples` folder contains some scripts showing in an incremental way how the `NXTfusion` library can be used, on both synthetic and real data.

3.1 Example1: single (nonlinear) matrix factorization

The file `examples/example1.py` contains the simplest example of how NXTfusion can be used. We use numpy to randomly generate a (100,1000) real valued matrix and we assume it represents the affinity between proteins (represented by the protein entity protEnt) and compounds/drugs (drugEnt).

```
protDrugMat = np.random.rand(100, 1000)
protEnt = NX.Entity("proteins", list(range(0,100)), np.int16)
drugEnt = NX.Entity("compounds", list(range(0,1000)), np.int16)
```

We thus transform the numpy.ndarray matrix into a *`NXTfusion.DataMatrix.DataMatrix`* object which stores the matrix/relation data in a way suitable for minibatching in a Neural Network (NN). As you can see from the module details, there are many constructors for the *`NXTfusion.DataMatrix.DataMatrix`* object, in this case the one that processes a numpy.ndarray matrix will be automatically called.

```
protDrugMat = DM.DataMatrix("protDrugMatrix", protEnt, drugEnt, protDrugMat)
```

Next, we define a loss function suitable for this relation. Since we generated real values, we the task of factorizing this relation will be a regression.

```
protDrugLoss = L.LossWrapper(t.nn.MSELoss(), type="regression", ignore_index = IGNORE_
↪INDEX)
```

The ignore_index is used to tell the NN which values should be ignored during the computation of the loss. It allows to train on partially observed matrices (sparse).

After that we just need to build the Entity-Relation graph (ERgraph) as we intend it using the APIs provided by NXTfusion. To fo so, we first define a *`NXTfusion.NXTfusion.MetaRelation`* “prot-drug” that will contain all the relations between those entities.

We then append the actual *`NXTfusion.NXTfusion.Relation`* object (represented by the protDrugMat object) to this *`NXTfusion.NXTfusion.MetaRelation`* with the *`NXTfusion.NXTfusion.MetaRelation.append()`* method. In the classic Matrix Factorization settings, only one metrix is considered, meaning that there will be only one relation between two entities.

```
protDrugRel = NX.MetaRelation("prot-drug", protEnt, drugEnt, None, None)
protDrugRel.append(NX.Relation("drugInteraction", protEnt, drugEnt, protDrugMat,
↪"regression", protDrugLoss, relationWeight=1))
ERgraph = NX.ERgraph([protDrugRel])
```

In this case the `NXTfusion.NXTfusion.ERgraph` will thus be formed by a single `NXTfusion.NXTfusion.MetaRelation` containing only a `NXTfusion.NXTfusion.Relation`. We create such object as shown.

The next step is to define a NN model able to perform inference over this simple graph. We provide such a pytorch NN as example1Model. We input this model to the `NXTfusion.NXmultiRelSide.NNwrapper` object, which will mediate the interaction between the NN object and the `NXTfusion.NXTfusion.ERgraph`, in a transparent way to the user.

```
model = example1Model(ERgraph, "mod1")
wrapper = NNwrapper(model, dev = DEVICE, ignore_index = IGNORE_INDEX)
wrapper.fit(ERgraph, epochs=50)
```

The NNwrapper has the scikit-learn-inspired `NXTfusion.NXmultiRelSide.NNwrapper.fit()` and `NXTfusion.NXmultiRelSide.NNwrapper.predict()` methods, that are the only way in which the user is required to interact with it. The `NXTfusion.NXmultiRelSide.NNwrapper.fit()` model will train the example1Model NN to factorize the `NXTfusion.NXTfusion.ERgraph`.

In order to obtain predictions from the trained model, we will use the `NXTfusion.NXmultiRelSide.NNwrapper.predict()` method. In order to tell the `NXTfusion.NXmultiRelSide.NNwrapper` which cells in the matrix/Relation we are interested in, we need to build a special "input vector" X. In this case we want to predict the entire matrix, to make sure that the training converged, and we thus use the buildPytorchFeats function to transform the entire matrix into a `NXTfusion.NXmultiRelSide.NNwrapper.predict()`-understandable format.

```
X, Y, corresp = buildPytorchFeats(protDrugMat)
Yp = wrapper.predict(ERgraph, X, "prot-drug", "drugInteraction", None, None)
```

We thus use the predict to obtain the model's predictions for the requested positions (X) of the `NXTfusion.NXTfusion.Relation` "drugInteraction" within the `NXTfusion.NXTfusion.MetaRelation` "prot-drug" in the `NXTfusion.NXTfusion.ERgraph` . This specification of which `NXTfusion.NXTfusion.Relation` and:py:class:*NXTfusion.NXTfusion.MetaRelation* should be predicted seems unnecessary here, where only one `NXTfusion.NXTfusion.Relation` exists, but becomes important when you want to predict a specific relation in larger ER graphs.

CHAPTER

FOUR

ADVANCED EXAMPLES

4.1 Example2: tensor factorization

The `examples/example2.py` file contains a simple script performing tensor factorization, namely inference over multiple *NXTfusion.NXTfusion.Relation* between two *NXTfusion.NXTfusion.Entity* .

We start by defining the same entities used in examples/example1.

```
protEnt = NX.Entity("proteins", list(range(0,100)), np.int16)
drugEnt = NX.Entity("compounds", list(range(0,1000)), np.int16)
```

Then we create three random matrices that will define the 3 different relations between protEnt and drugEnt, and we put them into the *NXTfusion.DataMatrix.DataMatrix* format, which allows optimized mini-batching during training.

```
protDrugMat1 = np.random.rand(100, 1000)
protDrugMat2 = np.random.rand(100, 1000)
protDrugMat3 = np.random.rand(100, 1000)
protDrugMat1 = DM.DataMatrix("protDrugMatrix1", protEnt, drugEnt, protDrugMat1)
protDrugMat2 = DM.DataMatrix("protDrugMatrix2", protEnt, drugEnt, protDrugMat2)
protDrugMat3 = DM.DataMatrix("protDrugMatrix3", protEnt, drugEnt, protDrugMat3)
```

Since we have three relations, and that they might constitute different prediction tasks (e.g. regression, prediction), we define one loss function for each *NXTfusion.NXTfusion.Relation*. As an example, here we use 3 different losses for regression that are provided by pytorch.

We encapsulate each of them with the *NXTfusion.NXLosses.LossWrapper* class: this will allow the losses to ignore the ignore_index values, thus allowing fast (batched) inference over sparsely observed matrices (matrices/Relations with missing values).

```
protDrugLoss1 = L.LossWrapper(t.nn.MSELoss(), type="regression", ignore_index =␣
↪IGNORE_INDEX)
protDrugLoss2 = L.LossWrapper(t.nn.L1Loss(), type="regression", ignore_index = IGNORE_
↪INDEX)
protDrugLoss3 = L.LossWrapper(t.nn.SmoothL1Loss(), type="regression", ignore_index =␣
↪IGNORE_INDEX)
```

We then build the ER graph using the NXTfusion API. We thus define the *NXTfusion.NXTfusion.Relation* that will contain all the relations between the protEnt and drugEnt entities, and we add the relations one by one. Finally, we instantiate the *NXTfusion.NXTfusion.ERgraph* object, which will contain the MetaRelation.

```
protDrugRel = NX.MetaRelation("prot-drug", protEnt, drugEnt, None, None)
protDrugRel.append(NX.Relation("drugInteraction1", protEnt, drugEnt, protDrugMat1,
↪"regression", protDrugLoss1, relationWeight=1))
```

(continues on next page)

(continued from previous page)

```
protDrugRel.append(NX.Relation("drugInteraction2", protEnt, drugEnt, protDrugMat2,
↪"regression", protDrugLoss2, relationWeight=1))
protDrugRel.append(NX.Relation("drugInteraction3", protEnt, drugEnt, protDrugMat3,
↪"regression", protDrugLoss3, relationWeight=1))
ERgraph = NX.ERgraph([protDrugRel])
```

We perform training as usual, defining a t.nn.Module suitable for the target ERgraph and we incapsulate it into the NNwrapper. We can then use the .fit() and .predict() methods to train and test the model.

```
model = example2Model(ERgraph, "mod2")
wrapper = NNwrapper(model, dev = DEVICE, ignore_index = IGNORE_INDEX)
wrapper.fit(ERgraph, epochs=5)
```

Since the ERgraph contains multiple relations, we can predict separately each of them. The following code shows how to do it. First, we compute the X values for the *NXTfusion.NXTfusion.Relation* we want to predict, and then we specify to the .predict function the name of the target MetaRelation and the *NXTfusion.NXTfusion.Relation* in it. The *NXTfusion.NXmultiRelSide.NNwrapper.predict()* method will return the predictions for the specified relation, or an error if it is not present.

```
X, Y, corresp = buildPytorchFeats(protDrugMat2)
Yp1 = wrapper.predict(ERgraph, X, "prot-drug", "drugInteraction2", None, None)
print("Final MSE: ", (np.sum((np.array(Yp) - np.array(Y))**2))/float(len(Yp)))

X, Y, corresp = buildPytorchFeats(protDrugMat3)
Yp1 = wrapper.predict(ERgraph, X, "prot-drug", "drugInteraction3", None, None)
print("Final MSE: ", (np.sum((np.array(Yp) - np.array(Y))**2))/float(len(Yp)))
```

4.2 Example3: Two relations among 3 entities

The `examples/example3.py` script shows how to use `NXTfusion` to perform inference over 3 *NXTfusion.NXTfusion.Entity* connected by 2 *NXTfusion.NXTfusion.Relation*.

As usual (see previous examples) we create the random matrices representing our relations.In this case we define also the protein-domain *NXTfusion.NXTfusion.Relation*, creating a binary (0/1) matrix. The protein-domain *NXTfusion.NXTfusion.Relation* mimicks the presence or absence of protein domains (e.g. PFAM) in each protein.

```
protEnt = NX.Entity("proteins", list(range(0,100)), np.int16)
drugEnt = NX.Entity("compounds", list(range(0,1000)), np.int16)
domainEnt = NX.Entity("protein", list(range(0,700)), np.int16)

protDrugMat = np.random.rand(100, 1000)
protDomainMat = np.random.randint(2, size=(100, 700))
protDrugMat = DM.DataMatrix("protDrugMatrix", protEnt, drugEnt, protDrugMat)
protDomainMat = DM.DataMatrix("protDomainMatrix", protEnt, domainEnt, protDomainMat)
```

We transformed the raw data in *NXTfusion.DataMatrix.DataMatrix* objects, as usual.

We then define the losses. In this case, the protein-domain *NXTfusion.NXTfusion.Relation* constitutes a binary prediction (discrimination) task, and so we use the `t.nn.BCEWithLogitsLoss` loss from pytorch and we specify "binary" as type for this loss. Ignore_index works as usual.

```
protDrugLoss = L.LossWrapper(t.nn.MSELoss(), type="regression", ignore_index = IGNORE_
↪INDEX)
protDomainLoss = L.LossWrapper(t.nn.BCEWithLogitsLoss(), type="binary", ignore_index␣
↪= IGNORE_INDEX)
```

This time we will define two `NXTfusion.NXTfusion.MetaRelation`, one for the prot-drug and one for the prot-domain relations. We append the corresponding `NXTfusion.NXTfusion.Relation` to each MetaRelation.

Here we build the prot-drug MetaRelation: .. code-block:: python

> protDrugRel = NX.MetaRelation("prot-drug", protEnt, drugEnt, None, None) protDrugRel.append(NX.Relation("drugInteraction", protEnt, drugEnt, protDrugMat, "regression", protDrugLoss, relationWeight=1))

And here we build the prot-domain `NXTfusion.NXTfusion.Relation`. Finally, we add BOTH MetaRelations to the ERgraph.

```
protDomainRel = NX.MetaRelation("prot-domain", protEnt, domainEnt, None, None)
protDomainRel.append(NX.Relation("pfamDomains", protEnt, domainEnt, protDomainMat,
↪"binary", protDomainLoss, relationWeight=1))
ERgraph = NX.ERgraph([protDrugRel, protDomainRel])
```

Using the NNwrapper object, we can perform training and testing as usual.

Please pay attention to the fact that `BCEWithLogitsLoss` does not use a Sigmoid activation in the NN. If, after prediction, you want to compute the prediction scores, you will have to apply `t.nn.Sigmoid` by yourself! (This is a `pytorch` good practice, not NXTfusion.)

4.3 Example4: Using side information

Latent data fusion methods, such as Matrix Factorization or Entity-Relation learn a latent representation for for the entities representing the objects described by the rows and the columns of each matrix/Relation.

Clearly, in these settings, if a row or a column of the matrix is completely empty, no optimization of the corresponding latent variables can be performed. A possible solution to overcome this problem is to add to the model some *explicit* variables, which are analogous to the conventional features used in regular ML methods. These *feature vecotrs* are called **side information** in the MF/ER data fusion context.

In `examples/exampleSide.py` we show and example of how side information can be introduced into a `NXTfusion` model.

In this example we will use the following datasets:

```
wget http://homes.esat.kuleuven.be/~jsimm/chembl-IC50-346targets.mm
wget http://homes.esat.kuleuven.be/~jsimm/chembl-IC50-compound-feat.mm
```

First we read the datasets and we transpose the traget matrix to make sure that the matrix is in the prot-drug format.

WARNING the ECFP side information is quite large and due to some missing support for sparse side information, it will required 12Gb of RAM. For this reason we are reading it but running the example on a smaller dataset.

Please note that for matrices/Relations the sparsity support IS present in NXTfusion, and so the library can scale quite well to very large matrices.

```
ic50 = mmread("chembl-IC50-346targets.mm").transpose()
shape = ic50.shape
#read the side information (features)
```

(continues on next page)

(continued from previous page)

```
#requires 12Gb of ram, so we propose a smaller (randomly generated) alternative
#(Sparse support for side information is currentyl missing)
ecfp = mmread("chembl-IC50-compound-feat.mm")
ecfp = np.random.rand(ecfp.shape[0], 50)
```

We define the Entities as usual, and we transform the input data into DataMatrix format. In this case we transform also the SideInfo raw data into a NXTfusion-understandable format using the SideInfo class.

```
protEnt = NX.Entity("proteins", list(range(0,shape[0])), np.int16)
drugEnt = NX.Entity("compounds", list(range(0,shape[1])), np.int16)
ic50DrugMat = DM.DataMatrix("ic50", protEnt, drugEnt, ic50)
ecfpSideMat = DM.SideInfo("drugSide", drugEnt, ecfp)
```

We build the MetaRelation and Relation as usual. The only diference is that the ecfpSideMat containing the side information is passed as argument to the MetaRelation, in order to specify that the side information for the drugEnt *`NXTfusion.NXTfusion.Entity`* (ent2) is available.

```
protDrugRel = NX.MetaRelation("prot-drug", protEnt, drugEnt, None, ecfpSideMat)
protDrugRel.append(NX.Relation("drugInteraction", protEnt, drugEnt, ic50DrugMat,
↪"regression", protDrugLoss, relationWeight=1))
ERgraph = NX.ERgraph([protDrugRel])
```

Training and testing is performed as usual.

```
model = example1Model(ERgraph, "mod1")
wrapper = NNwrapper(model, dev = DEVICE, ignore_index = IGNORE_INDEX)
wrapper.fit(ERgraph, epochs=5)
```

For prediction, we need to specify the side information again. This is done by just passing it to the .predict() method.

```
X, Y, corresp = buildPytorchFeats(ic50DrugMat)
Yp = wrapper.predict(ERgraph, X, "prot-drug", "drugInteraction", None, ecfpSideMat)
print("Final MSE: ", (np.sum((np.array(Yp) - np.array(Y))**2))/float(len(Yp)))

#we do the same but taking as input the coo_matrix instead
X, Y, corresp = buildPytorchFeats(ic50, protEnt, drugEnt)
Yp = wrapper.predict(ERgraph, X, "prot-drug", "drugInteraction", None, ecfpSideMat)
print("Final MSE: ", (np.sum((np.array(Yp) - np.array(Y))**2))/float(len(Yp)))
```

In this example we compute the predictions twice to show that the buildPytorchFeats function can build the input X vector starting from both DataMatrix objects or other formats like scipy.sparse.coo_matrix objects thanks to method overloading.

CHAPTER

FIVE

HOW TO BUILD A NN MODEL TO BE USED IN NXTFUSION

As you can see from the examples in the `examples/` folder, in order to perform inference over an ERgraph it is necessary to pass a NN object (t.nn.Module) to *`NXTfusion.NXmultiRelSide.NNwrapper`*.fit.

Our original idea was to automatically build a model suitable for each specific *`NXTfusion.NXTfusion.ERgraph`*, but, while developing the library, some considerations made us realize that this was not the best solution. First, NN are designed to be customizable and flexible, why restricting the users to our choices? Second, the entire idea of NXTfusion is to allow inference over totally arbitrary ER graphs, why restricting the most important part of the inference, namely the NN model that is actually trained to factorize the graph?

We thus opted for providing a skeleton class *`NXTfusion.NXmodels.NXmodelProto`* that contains a prototypical model that could be used in the *`NXTfusion.NXmultiRelSide.NNwrapper`*. It is barely an interface, but, alongside with this explanation and the NN models inherited from it in the examples folder we hope it's enough.

5.1 NN model for single matrix factorization

In `examples/example1.py` and we perform inference over an ERgraph with 1 Relation between 2 Entities (matrix factorization problem).

In order to do so we propose the following simple model.

```
class example1Model(NXmodelProto):
   def __init__(self, ERG, name):
       super(example1Model, self).__init__()
       self.name = name
       #########DEFINE NN HERE##############
       protEmbLen = ERG["prot-drug"]["lenDomain1"]
       drugEmbLen = ERG["prot-drug"]["lenDomain2"]
       PROT_LATENT_SIZE = 10
       DRUG_LATENT_SIZE = 20
       ACTIVATION = t.nn.Tanh
       self.protEmb = t.nn.Embedding(protEmbLen, PROT_LATENT_SIZE)
       self.protHid = t.nn.Sequential(t.nn.Linear(PROT_LATENT_SIZE, 10), t.nn.
↪LayerNorm(10), ACTIVATION())

       self.drugEmb = t.nn.Embedding(drugEmbLen, DRUG_LATENT_SIZE)
       self.drugHid = t.nn.Sequential(t.nn.Linear(DRUG_LATENT_SIZE, 20), t.nn.
↪LayerNorm(20), ACTIVATION())
       self.biProtDrug = t.nn.Bilinear(10, 20, 10)
       self.outProtDrug = t.nn.Sequential( t.nn.LayerNorm(10), ACTIVATION(), t.nn.
↪Dropout(0.1), t.nn.Linear(10,1))
       self.apply(self.init_weights)
```

The trainable latent variables are represented by the `protEmb` and `drugEmb`, which are `t.nn.Embedding` objects. The embeddings are processed by the specific `protHid` and `drugHid` hidden layer. These layers are then joined (effectively performing the factorization), by the `biProtDrug` bilinear layer, which is followed by the `outProtDrug` final layer, which outputs the final prediction.

The names of these submodules are intended to be as familiar as possible with respect to the Entities and Relations initialized in the main of `examples/example1.py`.

The forward method helps understanding how these submodules are arranged. They basically connect the protEmb and drugEmb latent variables (embeddings) into making a non-linear final prediction of the cells of the target matrix.

```
def forward(self, relName, i1, i2, s1=None, s2=None):
    if relName == "prot-drug":
            u = self.protEmb(i1)
            v = self.drugEmb(i2)
            u = self.protHid(u).squeeze()
            v = self.drugHid(v).squeeze()
            o = self.biProtDrug(u, v)
            o = self.outProtDrug(o)
            return o
```

In order to make the parameters of the models (e.g. latent sizes, etc.) less dependent on magic numbers, since the *NXTfusion.NXmodels.NXmodelProto* class takes as input the entire ERgraph, it is possible to call by name every *NXTfusion.NXTfusion.Relation* and *NXTfusion.NXTfusion.MetaRelation* in order to automatically fetch information such as the expected number of objects in each *NXTfusion.NXTfusion.Entity*, as shown here.

```
protEmbLen = ERG["prot-drug"]["lenDomain1"]
drugEmbLen = ERG["prot-drug"]["lenDomain2"]
```

5.2 A NN for tensor factorization

As shown in `examples/example2.py`, if the model needs to model multiple *NXTfusion.NXTfusion.Relation* between two *NXTfusion.NXTfusion.Entity*, once the submodules are defined for a single relation, is sufficient to increase the number of output neuronsin the `outProtDrug` final layer. In this case there are 3 relations to be reconstructed (predicted) and indeed there are 3 output neurons.

```
self.outProtDrug = t.nn.Sequential( t.nn.LayerNorm(10), ACTIVATION(), t.nn.Dropout(0.
↪1), t.nn.Linear(10,3))
def forward(self, relName, i1, i2, s1=None, s2=None):
       if relName == "prot-drug":
               u = self.protEmb(i1)
               v = self.drugEmb(i2)
               u = self.protHid(u).squeeze()
               v = self.drugHid(v).squeeze()
               o = self.biProtDrug(u, v)
               o = self.outProtDrug(o)
               return o
```

5.3 A NN for inference over arbitrary ER graphs

When the NN model must be able to predict mutiple *NXTfusion.NXTfusion.MetaRelation* involving multiple *NXTfusion.NXTfusion.Entity* (an arbitrarily connected ERgraph).

In `examples/example3.py` we show such a NN model. We define the embedding, entity-specific hidden (hid) and bilinear+output layer for 2 *NXTfusion.NXTfusion.MetaRelation* among 3 *NXTfusion.NXTfusion.Entity*.

```
class example3Model(NXmodelProto):
     def __init__(self, ERG, name):
             super(example3Model, self).__init__()
             self.name = name
             #########DEFINE NN HERE##############
             protEmbLen = ERG["prot-drug"]["lenDomain1"]
             drugEmbLen = ERG["prot-drug"]["lenDomain2"]
             domainEmbLen = ERG["prot-domain"]["lenDomain2"]
             PROT_LATENT_SIZE = 10
             DOMAIN_LATENT_SIZE = 10
             DRUG_LATENT_SIZE = 20
             ACTIVATION = t.nn.Tanh
             self.protEmb = t.nn.Embedding(protEmbLen, PROT_LATENT_SIZE)
             self.protHid = t.nn.Sequential(t.nn.Linear(PROT_LATENT_SIZE, 10), t.nn.
↪LayerNorm(10), ACTIVATION())

             self.drugEmb = t.nn.Embedding(drugEmbLen, DRUG_LATENT_SIZE)
             self.drugHid = t.nn.Sequential(t.nn.Linear(DRUG_LATENT_SIZE, 20), t.nn.
↪LayerNorm(20), ACTIVATION())
             self.biProtDrug = t.nn.Bilinear(10, 20, 10)
             self.outProtDrug = t.nn.Sequential( t.nn.LayerNorm(10), ACTIVATION(), t.
↪nn.Dropout(0.1), t.nn.Linear(10,1))

             self.domainEmb = t.nn.Embedding(domainEmbLen, DOMAIN_LATENT_SIZE)
             self.domainHid = t.nn.Sequential(t.nn.Linear(DOMAIN_LATENT_SIZE, 20), t.
↪nn.LayerNorm(20), ACTIVATION())
             self.biProtDomain = t.nn.Bilinear(10, 20, 10)
             self.outProtDomain = t.nn.Sequential( t.nn.LayerNorm(10), ACTIVATION(),␣
↪t.nn.Dropout(0.1), t.nn.Linear(10,1))

             self.apply(self.init_weights)
```

Besides the initializations, the most important part to understand is in the forward method. The *NXTfusion.NXmultiRelSide.NNwrapper* class will call **by name** the forward to predict each *NXTfusion.NXTfusion.MetaRelation* in the *NXTfusion.NXTfusion.ERgraph*, and to do wo it will use the argument `relName`.

The NNwrapper thus uses the specific `name` of each *NXTfusion.NXTfusion.MetaRelation* to **tell the forward** which branch of the NN must be run (each branch corresponds to a *NXTfusion.NXTfusion.MetaRelation*, as explained here https://doi.org/10.1093/bioinformatics/btab09).

```
def forward(self, relName, i1, i2, s1=None, s2=None):
     if relName == "prot-drug":
             u = self.protEmb(i1)
             v = self.drugEmb(i2)
             u = self.protHid(u).squeeze()
             v = self.drugHid(v).squeeze()
             o = self.biProtDrug(u, v)
             o = self.outProtDrug(o)
```

(continues on next page)

(continued from previous page)

```
        if relName == "prot-domain":
                u = self.protEmb(i1)
                v = self.domainEmb(i2)
                u = self.protHid(u).squeeze()
                v = self.domainHid(v).squeeze()
                o = self.biProtDomain(u, v)
                o = self.outProtDomain(o)
        return o
```

It is thus crucial to build a forward specifying the different branches that the computation of each *NXTfusion.NXTfusion.MetaRelation* needs to run in order to obtain the final predictions.

5.4 Further reading

A more rigorous and theoretical description of the intuitiion behind the models shown in the examples/ scripts can be found in the original publication https://doi.org/10.1093/bioinformatics/btab09.

CHAPTER

SIX

LIST OF MODULES AND FUNCTIONS IN NXTFUSION

6.1 NXTfusion.DataMatrix module

class NXTfusion.DataMatrix.**DataMatrix**

Bases: `object`

The input "data" format should be: {(ent1, ent2): value} for all the observed elements in the matrix.

The format in which the data is stored in the DataMatrix object is the following: featsHT = {domain1Name_numeric : [numpy16_domain2Names_numeric, numpyX_labels]}

__init__(*self*, *name: str*, *ent1: Entity*, *ent2: Entity*, *data: numpy.ndarray*)

One of the alternative constructors for the DataMatrix class.

Parameters

- **name** (*str*) – Name of the data matrix
- **ent1** (`Entity`) – Entity object representing the object on the dimension 0
- **ent2** (`Entity`) – Entity object representing the object on the dimension 1
- **data** (*dict*) – Hash table containing the (sparse) elements and in the matrix describing the relation. The input "data" format should be: {(ent1, ent2): value} for all the observed elements in the matrix.
- **dtype** (*numpy.dtype*) – The smallest possible type that could be used to store the elements of the matrix (e.g. np.int16 can represent up to 2^16 unique objects in the entity)

__init__(*self*, *name: str*, *ent1: NX.Entity*, *ent2: NX.Entity*, *data: dict*, *dtype: type*)

One of the alternative constructors for the DataMatrix class.

Parameters

- **name** (*str*) – Name of the data matrix
- **ent1** (`Entity`) – Entity object representing the object on the dimension 0
- **ent2** (`Entity`) – Entity object representing the object on the dimension 1
- **data** (*numpy.ndarray*) – Numpy matrix containing the (dense) describing the relation between ent1 and en2.
- **dtype** (*numpy.dtype*) – The smallest possible type that could be used to store the elements of the matrix (e.g. np.int16)

Returns the message id

__init__(*self*, *name: str*, *data: numpy.ndarray*, *dtype: numpy.dtype*)

Simplified constructor for the DataMatrix class. Entities are inferred from the dimensionality of the np.ndarray.

Parameters

- **name** (*str*) – Name of the data matrix
- **data** (*numpy.ndarray*) – Numpy matrix containing the (dense) describing the relation between ent1 and en2.

- **`dtype`** (*`numpy.dtype`*) – The smallest possible type that could be used to store the elements of the matrix (e.g. np.int16 can represent up to 2^16 unique objects in the entity)

`__init__`(*self*, *path: str*)
Constructor that reads the DataMatrix from a previously serialized DataMatrix object.
Parameters **`path`** (*`str`*) – Path of the serialized DataMatrix

`size`()
Function that return the size of the relation (number of elements in the matrix).

Returns

Return type Size of the relation in the DataMatrix object

`standardize`()
Method that standardizes the matrix with the formula x' = (x - mu)/s, where mu is the mean and s is the standard deviation.

Returns

Return type None

`toHashTable`() → dict
Method that returns an hash table (dict) containing the DataMatrix data.

Returns

Return type dict

class `NXTfusion.DataMatrix.`**`SideInfo`**
Bases: `object`

Class that encapsulated the side information raw data in order to be efficiently processed by NXTfusion. You can use this class to wrap side information vectors analogously to how DataMatrix wraps matrix/relations.

`__init__`(*self*, *name: str*, *ent1: Entity*, *ent2: Entity*, *data: dict*)
One of the alternative constructors for the SideInfo class.
Parameters
- **`name`** (*`str`*) – Name of the data matrix
- **`ent1`** (*`Entity`*) – Entity object representing the object on the dimension 0
- **`data`** (*`dict`*) – Dict containing ent1 objects as keys and feature vectors (side information) as values.

`__init__`(*self*, *name: str*, *ent1: Entity*, *ent2: Entity*, *data: numpy.ndarray*)
One of the alternative constructors for the SideInfo class.
Parameters
- **`name`** (*`str`*) – Name of the data matrix
- **`ent1`** (*`Entity`*) – Entity object representing the object on the dimension 0
- **`data`** (*`numpy.ndarray`*) – Numpy array that contains the side information. It has shape (ent1 obj, feature length), similarly to a scikit-learn feature vector.

`__init__`(*self*, *name: str*, *ent1: Entity*, *ent2: Entity*, *data: scipy.sparse.coo_matrix*)
One of the alternative constructors for the SideInfo class.
Parameters
- **`name`** (*`str`*) – Name of the data matrix
- **`ent1`** (*`Entity`*) – Entity object representing the object on the dimension 0
- **`data`** (*`scipy.sparse.coo_matrix`*) – Scipy coo_matrix that contains the side information. It has shape (ent1 obj, feature length), similarly to a scikit-learn feature vector. It can be sparse, but currently the sparsity during mini batching is NOT supported.

__init__(*self*, *path: str*)
This constructor reads a serialized (SideInfo.dump()) SideInfo object. :param str path: Path to the serialized SideInfo object.

dump(*path=None*)
Method that serializes the SideInfo storing it at the selected path.

path: str Destination path for the serialized file

Returns

Return type None

normalize()
Method that standardizes the matrix with the formula x' = (x - mu)/s, where mu is the mean and s is the standard deviation.

Returns

Return type None

6.2 NXTfusion.NXFeaturesConstruction module

`NXTfusion.NXFeaturesConstruction.`**buildPytorchSide**(*data*, *domain*, *expectedLen=20*, *sideDtype=<class 'numpy.float32'>*)
This function builds the data structure containing the side information

The data structure is a {} indicized with the domain numeric names.

sideX = {domainName_numeric : numpy32_feats}

`NXTfusion.NXFeaturesConstruction.`**buildPytorchFeats**(*data: numpy.ndarray*, *domain1: Entity*, *domain2: Entity*, *side1=None*, *side2=None*)
This function is used to produce the input for the NNwrapper.predict() method, at prediction time. It produces the inputs necessary to predict the output for certain cells (or the entire matrix) for a given relation.

This version of the method takes as prediction target a (dense) numpy matrix.

Parameters

- **data** (*numpy.ndarray*) – Numpy matrix representing the target. This form of the method is more useful when the entire matrix needs to be predicted. The actual values in the matrix are provided as "labels" in output, but are ignored during prediction.
- **domain1** (`Entity`) – Entity representing the objects on the dimension 0 of the data matrix
- **domain2** (`Entity`) – Entity representing the objects on the dimension 1 of the data matrix

Returns Returns 3 lists `(x, y, corresp)` when used without side information. The first (x) is a list of tuples `[(i,j),(k,j),...]` containing the pairs of of objects belonging to domain1 and domain2 that needs to be predicted.

The second (`y`) is a list containing the values of the input data matrix corresponding to the pairs of object in `x`. The third (corresp) is a list of tuples containing the corresponding names of the pairs of objects listed in `x`.

`NXTfusion.NXFeaturesConstruction.`**`buildPytorchFeats`**(*data: dict*, *domain1: Entity*, *domain2: Entity*, *side1=None*, *side2=None*)

This function is used to produce the input for the NNwrapper.predict() method, at prediction time. It produces the inputs necessary to predict the output for certain cells (or the entire matrix) for a given relation.

This version of the method takes as prediction target a dict containing the pairs of ojects that need to be predicted. This is useful when only relatively few cells of the matrix need to be predicted (sparse prediction).

Parameters

- **`data`** (*`dict`*) – Dict in the form {(obj[i],obj[j]):value1, (obj[i],obj[k]):value2, . . . }. It represents the target cells of the matrix that need to be predicted. The actual values in the matrix are provided as value associated to each pair of objects in the dict, but are ignored during prediction. Used to represent sparse matrices.
- **`domain1`** (`Entity`) – Entity representing the objects on the dimension 0 of the data matrix
- **`domain2`** (`Entity`) – Entity representing the objects on the dimension 1 of the data matrix

Returns Returns 3 lists `(x, y, corresp)` when used without side information. The first (x) is a list of tuples `[(i,j),(k,j),...]` containing the pairs of of objects belonging to domain1 and domain2 that needs to be predicted.

The second (`y`) is a list containing the values of the input data matrix corresponding to the pairs of object in `x`. The third (`corresp`) is a list of tuples containing the corresponding names of the pairs of objects listed in `x`.

`NXTfusion.NXFeaturesConstruction.`**`buildPytorchFeats`**(*datam:DataMatrix*, *= None*, *side2 = None*)

This function is used to produce the input for the NNwrapper.predict() method, at prediction time. It produces the inputs necessary to predict the output for certain cells (or the entire matrix) for a given relation. This version of the method takes as prediction target a DataMatrix object.

Parameters **`datam`** (`DataMatrix`) – DataMatrix object containing the matrix representing the predidction target. The actual values in the observed cells in the DataMatrix are provided as "labels" y in output, but are ignored during prediction.

Returns Returns 3 lists `(x, y, corresp)` when used without side information. The first (x) is a list of tuples `[(i,j),(k,j),...]` containing the pairs of of objects belonging to domain1 and domain2 that needs to be predicted.

The second (`y`) is a list containing the values of the input data matrix corresponding to the pairs of object in `x`. The third (`corresp`) is a list of tuples containing the corresponding names of the pairs of objects listed in x.

6.3 NXTfusion.NXLosses module

`class` `NXTfusion.NXLosses.`**`FocalLoss`**(*alpha=1*, *gamma=2*, *logits=True*, *reduction='sum'*)

Bases: `torch.nn.modules.module.Module`

Implementation of the FocalLoss, which is used as loss for heavily unbalanced binary predictions. It is bult as a convetional pytorch module.

`__init__`(*alpha=1*, *gamma=2*, *logits=True*, *reduction='sum'*)

Constructor for the FocalLoss.

Parameters

- **`alpha`** (*`int`*) – Parameter of the loss

- **gamma** (*int*) – Parameter of the loss
- **logits** (*bool*) – Uses logits if True

class NXTfusion.NXLosses.**LossWrapper**(*loss: torch.nn.modules.module.Module*, *type: str*, *ignore_index: int*)

Bases: `torch.nn.modules.module.Module`

Class that wraps any pytorch loss allowing for ignore index. In the Matrix Factorization context it may be useful to define a value indicating missing values even when performing a regression, for example if the goal is to predict a sparsely observed real-valued matrix.

__call__(*input*, *target*)

Function defining the forward pass for this wrapper. It implements the ignore_index filtering and then it calls the actual self.loss on the remaining values.

Parameters

- **input** (*t.nn.Tensor*) – Pytorch tensor containing the predicted values
- **target** (*t.nn.Tensor*) – Pytorch tensor containing the target values

Returns

Return type Loss score computed only for the target values that are not equal to self.ignore_index.

__init__(*loss: torch.nn.modules.module.Module*, *type: str*, *ignore_index: int*)

Constructor for the wrapper.

Parameters

- **loss** (*t.nn.Module*) – The argument can be any pytorch compatible loss functioni
- **type** – Specifies wheter is a regression or a binay prediction (deprecate?)
- **ignore_index** (*int*) – Specifies which value should be ignored while computing the loss, to allow for the presence of missing values in the matrix/relation.

6.4 NXTfusion.NXTfusion module

class NXTfusion.NXTfusion.**Entity**(*name*, *domain*, *dtype=<class 'numpy.int32'>*)

Bases: `object`

Class representing the Entity concept.

__getitem__(*self*, *x: str*)

Method that returns the numeric value internally associated to each object in the Entity class.

Parameters **x** (*str*) – String name of a specific object in the Entity.

Returns primary key int

__getitem__(*self*, *x: int*)

Method that returns the str name of the object with primary key x.

Parameters **x** (*int*) – Primary key (unique id) of an object in the domain represented by the Entity.

Returns name of the object : str

__init__(*name*, *domain*, *dtype=<class 'numpy.int32'>*)

Constructor for the Entity class.

Parameters

- **name** (*str*) – Name of the Entity (use a mnemonic name describing the class of objects represented by the Entity)
- **domain** (*iterable (list) containing str*) – List of the possible objects belonging to this class (e.g. patients IDs, proteins Uniprot identifiers, . . .). It is an unique identifier naming (with a string) all the objects composing the domain of the Entity.
- **dtype** (*np.dtype*) – Smallest possible numpy type able to uniquely enumerate all the objects. len(domain) < max_number_representable(dtype).

class NXTfusion.NXTfusion.**Relation**(*name: str*, *domain1: NXTfusion.NXTfusion.Entity*, *domain2: NXTfusion.NXTfusion.Entity*, *data*, *task: str*, *loss: torch.nn.modules.module.Module*, *relationWeight: float*, *side1=None*, *side2=None*, *path=None*)

Bases: `dict`

Class that represent a relation (matrix in MF terms) with all its parameters and functions.

__init__(*name: str*, *domain1: NXTfusion.NXTfusion.Entity*, *domain2: NXTfusion.NXTfusion.Entity*, *data*, *task: str*, *loss: torch.nn.modules.module.Module*, *relationWeight: float*, *side1=None*, *side2=None*, *path=None*)

Constructor for the Relation class..

Parameters

- **name** (*str*) – Mnemonic name of the specific relation/matrix.
- **domain1** (*Entity*) – Entity1 involved in the relation (on dimension 0)
- **domain2** (*Entity*) – Entity2 involved in the relation (on dimension 1)
- **data** (*DataMatrix*) – DataMatrix object containing the matrix describing this relation
- **task** (*str ["regression", "binary"]*) – Type of prediction task associated to this relation. "Regression" for real valued predictions, "binary" for binary classification.
- **loss** (*NX.NXLosses or t.nn.Module*) – Pytorch-like loss module corresponding to the loss that must be used to compute the reconstruction error for this relation.
- **relationWeight** (*float*) – A relation-specific weight that will multiply the loss score during training.

class NXTfusion.NXTfusion.**MetaRelation**(*name*, *domain1*, *domain2*, *side1=None*, *side2=None*, *relations=[]*, *prediction=False*)

Bases: `object`

Constructor for the Relation class. The Meta Relation represents multi-relations between the same entities (used for example in tensor factorization).

As a convention, we recommend to use names in the form ENT1-ENT2.

The domains must be the same for each relation in it, since the MetaRelation defines a tensor where the dimension 0 and 1 represent the same entities for all the matrices involved in the tensor. Can allow side info (common to all relations in it).

__getitem__(*self*, *x: str*)

Getitem method that searches by Relation.name

Parameters **x** (*str*) – The name of a Relation in this MetaRelation

Returns The target Relation or None

__getitem__(*self*, *x: int*)

Getitem method that searches by position of the target Relation in the tensor/MetaRelation.

Parameters **x** (*str*) – The position (index) of a Relation in this MetaRelation

Returns The target Relation or None

`__init__`(*name*, *domain1*, *domain2*, *side1=None*, *side2=None*, *relations=[]*, *prediction=False*)
Constructor for the MetaRelation class.

Parameters

- **`name`** (*`str`*) – Mnemonic name of the specific relation/matrix.
- **`domain1`** (`Entity`) – Entity1 involved in the relation (on dimension 0)
- **`domain2`** (`Entity`) – Entity2 involved in the relation (on dimension 1)
- **`relations`** (*`list of NX.Relation objects`*) – List of the relations involved in this MetaRelation (tensor)

`append`(*r*)
Method that adds a Relation object to an existing MetaRelation :param r: :type r: Relation

`getPos`(*x*)

`next`()

`pop`(*pos*)

class `NXTfusion.NXTfusion.`**`ERgraph`**(*entityList: list*, *name=''*)
Bases: `list`

Class that represents the entire Entity-Relation graphs, namely a list of MetaRelations. Each MetaRelation might contain multiple Relations.

`__contains__`(*self*, *x: int*)
Function that determines whether a specific MetaRelation object is present in the graph.
Parameters **`x`** (`MetaRelation`) – A MetaRelation object.
Returns bool (Is x present?)

`__contains__`(*self*, *x: str*)
Function that determines whether a specific MetaRelation.name is present in the graph.
Parameters **`x`** (*`str`*) – A MetaRelation str name.
Returns bool (Is x present?)

`__init__`(*entityList: list*, *name=''*)
Constructor for the ERgraph (Entity-Relation Graph) object.

Parameters **`entityList`** (*`list of MetaRelations`*) – List of MetaRelations that describe the full Entity Relation graph.

`__str__`()
Function that expresses the ERgraph as string

6.5 NXTfusion.NXmodels module

class `NXTfusion.NXmodels.`**`NXmodelProto`**
Bases: `torch.nn.modules.module.Module`

This class is the father of the pytorch modules used in the ER datafusion wrapper. It implements basic functions, leaving only the init and the forward empty

6.6 NXTfusion.NXmultiRelSide module

class `NXTfusion.NXmultiRelSide.`**`NNwrapper`**(*model*, *dev*, *ignore_index*, *initialEpoch=0*, *nworkers=0*)

Bases: `object`

Class that wraps a t.nn.Module (pytorch module) and uses scikit-learn-like methods such as .fit() and .predict() to train and test it.

`__init__`(*model*, *dev*, *ignore_index*, *initialEpoch=0*, *nworkers=0*)

Constructor for the NNWrapper class, which facilitates and standardizes the training of pytorch neural networks.

Parameters

- **`model`** (*`t.nn.Module`*) – The pytorch Neural Network that should be trained or tested.
- **`def`** (*`t.device`*) – The device on which the model should run. E.g. t.device("cuda") or t.device("cpu:0")
- **`ignore_index`** (*`int`*) – The ignore index value that will be used to mark "missing values" and "N/A" on partially observed matrices, in order to let the corresponding loss ignore those instances.

`computeLosses`(*y*, *yp*, *losses*, *relationData*, *weightRelations*)

This function computes the losses for the entire ER graph, by iterating through them. Used internally.

Parameters

- **`y`** (*`t.tensor`*) – Pytorch tensor containing the labels
- **`yp`** (*`t.tensor`*) – Pytorch tensor containing the predictions
- **`losses`** (*`list`*) – list of losses (LossWrapper or t.nn.Module)
- **`relationData`** (*`list`*) – list of MetaRelations
- **`weightRelations`** (*`list`*) – list of weights associated to each loss

Returns

- **loss** (*real*) – total loss
- **tmpLoss** (*list*) – list containing the losses associated to each Relation
- *meta private:*

`countParams`(*parameters: list*) → int

Method that counts the number of trainable parameters in the model.

Parameters **`parameters`** (*`iterable`*) – The iterable containtaining the pytorch model parameters.

Returns

Return type Number of parameters (int)

`fit`(*relationList*, *epochs=100*, *batch_size=500*, *save_model_every=10*, *LOG=False*, *MUTE=True*)

Function that performs the training of the wrapped pytorch model. It is analogous to scikit-learn .fit() method.

Parameters

- **`relationList`** (`ERgraph`) –
- **`epochs`** (*`int`*) – Number of epochs

- **`batch_size`** (`int`) – batch size during training
- **`save_model_every`** (`int`) – Stores the model every int epochs

`predict`(*ERgraph*, *X*, *metaRelationName*, *relationName*, *sidex1=None*, *sidex2=None*, *batch_size=500*, *plotGraph=False*)

Function that performs the training of the wrapped pytorch model. It is analogous to scikit-learn .predict() method.

Parameters

- **`ERgraph`** (`ERgraph`) –
- **`X`** (`list`) – List containing the 2D coordinates of the positions that should be predicted in the ERgraph.metaRelationName.relationName Relation.
- **`metaRelationName`** (`str`) – Name of the MetaRelation that contains the target relation
- **`relationName`** (`str`) – Name of the relation that you want to predict
- **`batch_size`** (`int`) – batch size during prediction

Returns **yp** – List containing the predictions for the target Relation

Return type list

`printBatchesLog`(*rel*, *e*, *bi*, *errTotOld*, *errTot*, *totLen*, *epochTime*, *loadTime*, *forwTime*, *LossTime*, *start*, *batch_size*, *mute=True*)

This class simplifies the live logging of the batches. If muted, it will only signal excessively long loading times.

Parameters **`TODO`** –

Returns

Return type meta private:

`processDatasets`(*DS: list*)

This method takes the external Entity Relation graph representation, in the form of one MetaRelation at a time and converts it into lower level data structures to be used within the wrapper, creating a MetaDataset structure from the ER representation passed as input. This structure mimics the ERgraph, but it's suitable for efficient multi-task mini batching during training. This function is used internally by the NNwrapper and does not need to be called by the user.

Parameters **`DS`** (`MetaRelation`) –

Returns

- **DS** (*MetaRelation*) – The original MetaRelation without the data matrices, in an attempt to save space. (still have to run benchmarks on it)
- **datasets** (*list of SubDataset*)
- **losses** (*list of losses*)
- **refSize** (*size of the target matrix (to be removed)*)
- *meta private:*

`saveModel`(*e: int*)

Method that stores the trained model at a certain iteration. Used internally.

Parameters **`e`** (`int`) – Epoch number. The model is automatically saved using the model name and the epoch number using t.save function.

CHAPTER SEVEN

HOW TO CITE

If you find this library useful, please cite: https://doi.org/10.1093/bioinformatics/btab092.

CHAPTER EIGHT

ABOUT US

This library has been developed during my stay at KU Leuven (ESAT-STADIUS), funded by KU Leuven and FWO grants.

CHAPTER

NINE

DISCLAIMER

I did my best effort to make this library available to anyone, but bugs might be present. Should you experience problems in using or installing it, or just to share any comment, please contact daniele [dot] raimondi [At] kuleuven [dOt] be.

Please note that:

THIS SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX