Combination of change models and graph embeddings to predict alignment adjustments

Janina Dujaka

September 11, 2020

Abstract

Contents

1 Introduction

Ontology alignment is the task of identifying semantically equivalent entities from two given ontologies. Different ontologies have different representations of the same entity, resulting in a need to de-duplicate entities when merging ontologies. Changes in the aligned ontologies are accompanied by changes in the alignments. Especially for large ontologies this is a laborius task often accompanied by manual tasks. The question arises as to which parts of the alignment need to be changed. Depending on the size of the ontologies, this can quickly become confusing. Finding these changed alignments is therefore a time-consuming task.

In this paper, we test different approaches to predict alignment changes automatically using different machine learning techniques. There are approaches using embeddings as a representation for classifying changes. In our approach, we propose a method for enriching the graph structure with the integration of change features, and use this additional information for ontology alignment. So we use embeddings and changes for this task. Since the current approaches only use the graph structure to reason about possible changes, the integration of node and change features seems promising. Change types as another input to the classification process could lead to improve classification accuracy. Therefore we compare embedding-based approaches to a neural network architecture built for node classification in custom knowledge graphs. We compare its performance regarding established classification methods. We adapt the embedding approach by normalizing the training data in one variant and coding the change types as vector addition in the other. The approach about the custom neural network is that we build MLP Neural Classifier and Classifier for separated networks.

1.1 Objective

Ontologies change with the knowledge they represent, not only the ontologies have to be adjusted, but the alignments as well. Ontology alignments (sometimes referred to as ontology mapping) describe an interface that is used to connect concepts in different ontologies. When they change, neighboring alignments also change. These changes must also be recognized There is still no automatic process that really delivers clear results. The automatisms already found tend to raise new questions.

This work intends to find an intelligent way to find these changed alignments. A time-saving variant to predict such alignment changes can be implemented by graph node embedding approaches in combination with classification.

In our approach we have a preprocessed dataset in form of embeddings which embed the multi-relational knowledge into low-dimensional representations of entities and relations via tensor and matrix factorization but also building a custom neural network.

Considering the entities and relationships between them, we use translation-based and multiplication-based models. The models represent entities as low-dimensional vectors and represent relations as operators that combine the representations of two entities. They are based on knowledge base completion approaches, where entity and relation embeddings are learned to provide some kind of scoring function to predict whether a triple should be part of the graph. The main difference among these techniques lies in the parametrization of the relation operators. Embeddings have shown to be powerful tools for multi-relational learning and inference due to their high scalability and strong generalization abilities [19].

For the approach with the custom neural network, we build a feedforward network once with embeddings and changes and once embeddings and changes separated into subnetworks. Both procedures are based on a principle of layering, the information moves in only one direction – forward – from the input nodes, through the hidden nodes and to the output nodes. The difference when creating subnetworks is that the subnetworks are only merged after separate training.

The following experiments aim to improve the prediction of alignment adjustments. An embedding can be learned and reused across models. There are already procedures in this context doing this with embeddings. We deviate from this by applying selected procedures to embeddings and changes. We have the information about the changes as well as the embeddings.

1.2 Approach

To achieve a fair comparison, we evaluate several embedding-based methods on our dataset and compare the results to new approaches. To do this, we examine two main approaches. First we consider the influence of vector manipulations. Since the transformation from the knowledge graph to the n-dimensional vector space leads to inconsistencies in the distances between the individual vectors, the normalization of the vectors can bring us more precise results [17]. Another approach is to code the change types differently. The change types available to us are: *Deleted, Added, Superclass, Annotation, Renamed*.

As a second approach, we use a custom neural network as a further possibility to improve the results. Therefore we build a kind of Multilayer Perceptron (MLP). The process by which a MLP learns is called the Backpropagation algorithm. It is a supervised training scheme, which means, it learns from labeled training data. In supervised learning, the training set is labeled. This means, for some given inputs, we know the desired/expected output (label).

1.3 Overview

The remainder of this paper is structured as follows: Section 2 presents foundations and related work and identifies a research gap. Our approach is discussed in Section 3. In Section 4, the implementation will be presented. In Section 5, the procedure of the experiments will be described. In Section 6, an evaluation of different classification approaches is presented and results of this evaluation are discussed. Section 7 closes this paper with a conclusion and an outlook.

2 Foundations and related work

Knowledge graphs model information in the form of entities and relationships between them. Relational learning methods are used to scale to large knowledge bases. The following embedding models help us to embed knowledge graphs into continuous low-dimensional spaces. These embedding models are based on the technique of knowledge base completion, where entity and relation embeddings are learned to provide some kind of scoring function to predict whether a triple should be part of the graph. The scoring function is mostly based on some kind of translation [1,8,15] or multiplication [14,18]. Embeddings from both categories can theoretically be useful when predicting alignment changes [6].

2.1 Translation-based models

The translation-based models represent entities and relations as the embedding vectors by regarding relations as translations from head entities to tail entities, such as TransE [1], TransH [15] and TransR [8]. The translation-based training procedure is time consuming, since they all employ stochastic gradient descent (SGD) to optimize a translation-based loss function, which may require days to converge for large knowledge graphs [21].

We consider TransE here as an example to describe translation-based models. In TransE, relationships are represented as translations in the embedding space: if (h, r, t) holds, then the embedding of the tail entity t should be close to the embedding of the head entity h plus some vector that depends on the relationship r. The score function of TransE is defined as:

$$s(h, r, t) = \|h + r - t\|$$

.

The approach relies on a reduced set of parameters in comparison to RESCAL or neural tensor networks as it learns only one low-dimensional (100 to 1000 dimensions) vector for each entity and each relationship [1].

2.2 Multiplication-based models

Multiplication-based models [14,18] represent the entities and sometimes even the relations as N-dimensional real vectors (equal to TransE), these vectors are denoted by bold font, $h, r \in \mathbb{R}^{\mathbb{N}}$. Complex models also depicts reflexive relationships, which is why this could also be interesting. These models [14] represent the entities e_k and relations w_r as N-dimensional complex-valued vectors.

DistMult model computes a score for each triplet (h, r, t) as

$$s(h, r, t) = h^T \cdot W_r \cdot t = \sum_{i=1}^{N} h_i \cdot r_i \cdot t_i \tag{1}$$

where W_r is a diagonal matrix with elements of vector r on its diagonal. As seen above, DistMult uses a trilinear dot product to compute the score for each triplet.

The Complex model involves real vectors corresponding to the real and imaginary parts of the embeddings and relations [14].

$$\begin{aligned} s(h, r, t; \vartheta) &= Re(< w_h, e_r, \bar{e}_t >) \\ &= \langle Re(w_h), Re(e_r), Re(e_t) \rangle \\ &+ \langle Re(w_h), Im(e_r), Im(e_t) \rangle \\ &+ \langle Im(w_h), Re(e_r), Im(e_t) \rangle \\ &- \langle Im(w_h), Im(e_r), Re(e_t) \rangle \end{aligned} \tag{2}$$

Trouillon [14] uses the Hadamard product here to compute the score for each triplet, where ϑ denotes the parameters of the corresponding model. The advantage is that this procedure is commutative. This enables us to accurately describe both symmetric and antisymmetric relations between pairs of entities, while still using joint representations of entities, whether they appear as subject or object of relations.

2.3 Custom neural networks with Keras

In our second approach we want to build Neural Classifier and Classifier for separated networks. Both Multilayer Perceptrons (MLP) and Convolutional Neural Networks (CNN) are suitable practices. In the following we use MLP because of its property of being fully connected as opposed to the local connectivity of the CNN. This aspect led to a high accuracy and robustness in other investigations. A multilayer perceptron (MLP) is a class of feedforward artificial neural network. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable [20].

Tensorflow uses high level Keras API to give developers an easy-to-use deep learning framework. Keras is a high-level neural networks library, written in Python that can be run on either TensorFlow or Theano, but simple for constructing a neural network. The following properties make keras also a useful tool [7]:

- Easy and fast prototyping (through total modularity, minimalism, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.

- Supports arbitrary connectivity schemes (including multi-input and multi-output training).
- Runs seamlessly on CPU and GPU.

Since multi-input training is possible, keras lends itself to be useful for subnetworking. We will use this property later in the implementation.

Keras provides a sequential and a functional model API. Both present ways of creating deep learning models where an instance class is created and model layers are created and added to it [3]. The sequential model has its limitations when you want to use multiple different input sources, produce multiple output destinations or models that re-use layers. The functional model offers these possiblities, it allows you to define ad hoc acyclic network graphs. Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a model that specifies the layers to act as the input and output to the model.

3 Approach

In this section we are looking for a procedure that enables us to make better predictions about alignment adjustments. The classification task considered so far is the same as in [6]: For every changed entity c that is close to an alignment statement, we make a prediction whether the alignment statement near c needs to be changed. For the classification itself, we have no straightforward path to walk.

Four approaches are considered in this work:

1. *Vector normalization*: Before training, embedding and change vectors of the training data set are normalized.
2. *Coding change types as vector addition*: Change types are coded as vector addition before they are added to the training data.
3. *Multi Layer Perceptron (MLP) Neural Classifier*: Embedding and change vectors learn in stacked layers.
4. *Multi Layer Perceptron (MLP) Classifier with separated subnetworks*: Embeddings and changes learn in separated MLP and are then merged.

In order to evaluate our methods, we conducted a series of experiments that are described in the following subsections. For the vector manipulation part we investigate the approach with vector normalization on the one hand and coding change types as vector addition on the other hand. In the case of the custom neural network, an important contemplation is the separation of change and embedding data in subnetworks. The difficulty here is to find the number and size of the individual layers.

3.1 General overview

Embeddings are low-dimensional, learned continuous vector representations of discrete variables. Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space. We use embeddings as input to a machine learning model for a supervised task. Here we use following embedding models, which have been described in section 2: TransE, TransH, DistMult and Complex model.

In this section, the architecture illustrated in figure 1 is presented. Processing steps are shown as boxes and data as ovals. This also applies to the following figures in this section. We have as a basis knowledge graph embeddings of our data set and the associated change representations. The architecture is divided into two main approaches, model tuning with two different vector manipulations and the construction of a custom neural network with and without separated subnetworks.

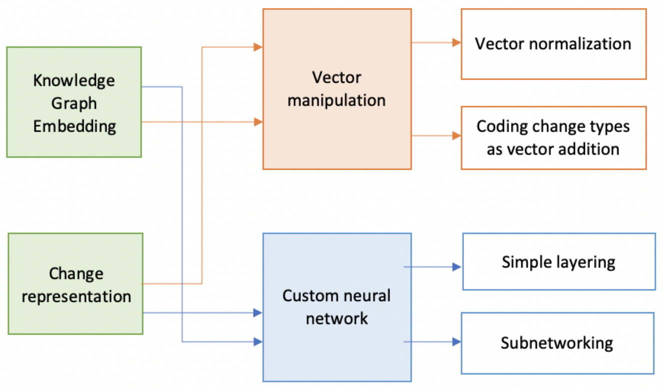


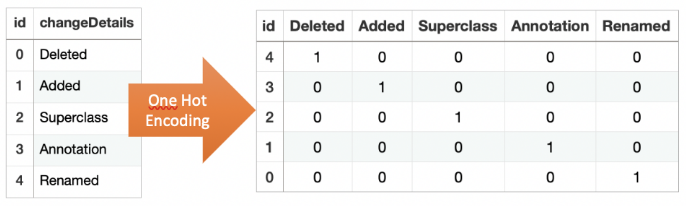
Figure 1: Overview

We consider the data sets separately as changes and embeddings and embeddings and changes together (in the following *both*). This is so that subnetworks can be built. Changes are defined as changes in the state of alignments in the ontologies. These are filtered out in the first step, one hot encoding. The following changes can occur in the alignment system: *Deleted*, *Added*, *Superclass*, *Annotation*, *Renamed*. We need to encode the strings to numbers for the different change options.

The operation of one-hot encoding categorical variables is actually a simple embedding where each category is mapped to a different vector. This process takes discrete entities and maps each observation to a vector of 0s and a single 1 signaling the specific category.

The great advantage with one-hot encoding is that the transformation does not rely on any supervision. We can greatly improve embeddings by learning them using a neural network on a supervised task. The embeddings form the weights of the network which are adjusted to minimize loss on the task. The resulting embedded vectors are representations of categories where similar categories, relative to the task, are closer to one another [16].

It is a common way of preprocessing categorical features for machine learning models [9]. This type of encoding creates a new binary feature for each possible category and assigns a value of 1 to the feature of each sample that corresponds to its original category. It is easier to understand visually (see figure 2): we One Hot Encode a change feature which consists of the previous categories.


One Hot Encoding

id	changeDetails
0	Deleted
1	Added
2	Superclass
3	Annotation
4	Renamed

id	Deleted	Added	Superclass	Annotation	Renamed
4	1	0	0	0	0
3	0	1	0	0	0
2	0	0	1	0	0
1	0	0	0	1	0
0	0	0	0	0	1

Figure 2: One Hot Encoding

After the one-hot encoding, we bring the data created there with the embeddings together to form a common training data set. For the evaluation task we split the data into training and test data. Training data is the data on which we will train our graph node embedding. Test data is used to check our trained graph node embedding. This data is totally new for our graph node embedding and if it performs well on this dataset, it shows that there is no overfitting.

The general process is basically similar for all approaches and can be traced back to the steps described in figure 3.

3.2 Vector manipulation

In this subsection we describe two ways we have manipulated the input data. On the one hand we use feature scaling through normalization and on the other hand we manipulate the input vector while we code change types as the addition of vectors.

The goal of normalization is to change the values of numeric columns in the dataset

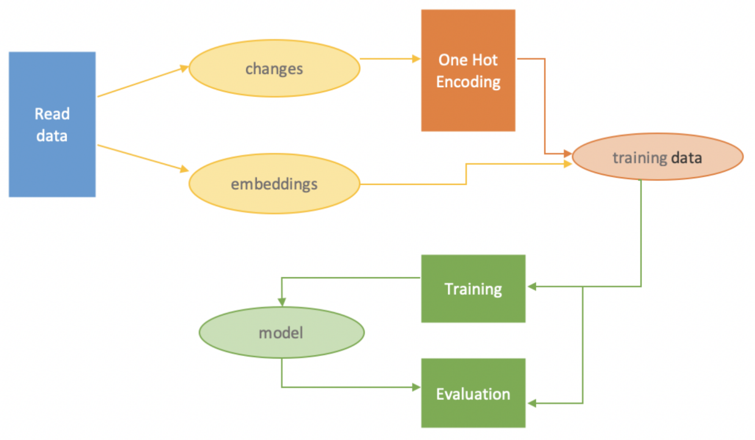


Figure 3: Pre-test without changing the training input vectors

to a common scale, without distorting differences in the ranges of values. Divide an array by its norm to normalize the array. The input data is centered but not scaled for each feature. That is the reason why a normalization of the vectors should lead to an improvement of the results. The rescaling of the features involves that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one. In comparison to the basic sequence after one hot encoding, the vectors are standardized in the cutout in figure 4.

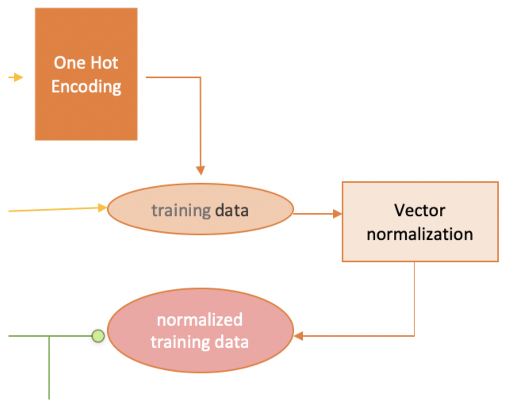


Figure 4: Vector Normalization

Another way to manipulate our training data is to code the change types as vector addition, so that the change representation is different. Compared to the basic sequence, the vectors are merged and manipulated in one step (see figure 5).

To each of the different kinds of changes will be added a vector with fixed scalars (see figure 6).

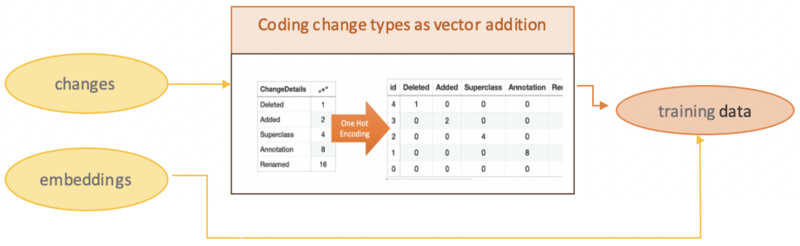
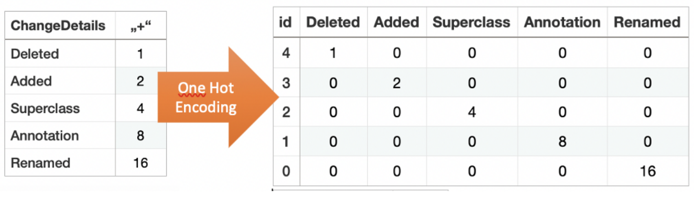


Figure 5: Vector Manipulation



ChangeDetails	„+“
Deleted	1
Added	2
Superclass	4
Annotation	8
Renamed	16

One Hot Encoding

id	Deleted	Added	Superclass	Annotation	Renamed
4	1	0	0	0	0
3	0	2	0	0	0
2	0	0	4	0	0
1	0	0	0	8	0
0	0	0	0	0	16

Figure 6: Vector Manipulation Details

3.3 MLP Neural Classifier and Classifier for separated networks

For building our custom neural network we use the concept of a Multi Layer Perceptron (MLP). With this we want to take advantage of the fact that non-linear classification is also possible.

First we build a MLP, in which we do not yet form subnetworks. We are creating the model sequentially and the output of each layer we push forward is input to the next layer we specify (see figure 7).

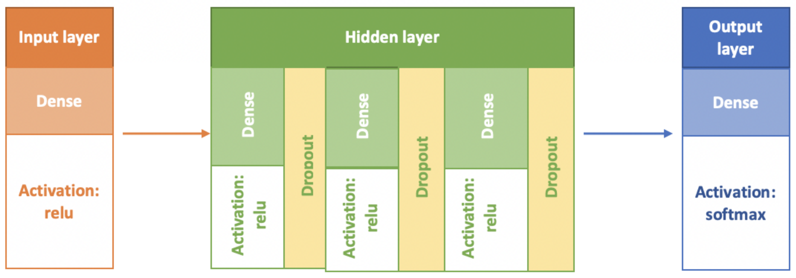


Figure 7: Multi Layer Perceptron Classifier structure

So that our layers are completely connected, we use Dense Layer. The activation function to be used in this case is relu. The layers between the input and output layers are called hidden layers. These are separated by dropout layers and also have relu as an activation function. At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left. This should prevent overfitting. The output layer takes different activation functions and for the case of multiclass classification, it is softmax. Other options would be tanh and logistic sigmoid, both activation functions are used in feed-forward nets. The logistic sigmoid function can cause a neural network to get stuck at the training time. The tanh function is mainly used classification between two classes [12]. For optimizing the performance of our neural network we try out layers of different sizes and numbers of layers.

The last plan is to build subnetworks for changes and embeddings (see figure 8). Each subnetwork will take its own input and its own training. The output of both

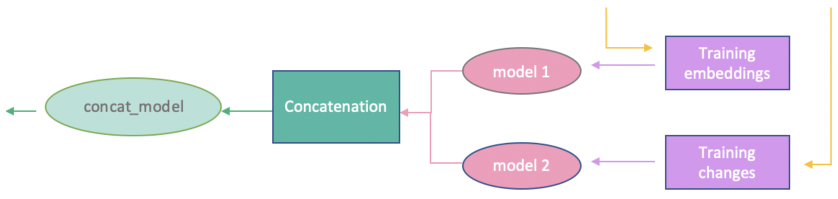


Figure 8: Subnetworking

interpretation models will be concatenated into one long vector that is passed to the output layer used to make a binary prediction. Output layer predicts the class.

Further we build a more complex architecture, which allows to build arbitrary graphs of layers. In contrast to the sequential model, you have to create and define a separate input level that specifies the form of the input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality of the input data [3]. We define a Multilayer Perceptron model for binary classification. Since we want to divide the input into embeddings and changes and let them learn separately, we are building two independent MLPs (see figure 9). The two sub-networks that are formed for changes and embeddings to learn separately from each other and are then concatenated (see figure 8).

Embeddings and changes have different input layers as can be seen in figure 9. The hidden layers of the two subnetworks can be structured differently. We tested different approaches here. After concatenating the two outputs, another Dense layer follows, so that the learning successes from both subnetworks are also combined in the learning process.

As with the sequential model, we test here the performance of our MLP we try out layers of different sizes, numbers of layers and training time.

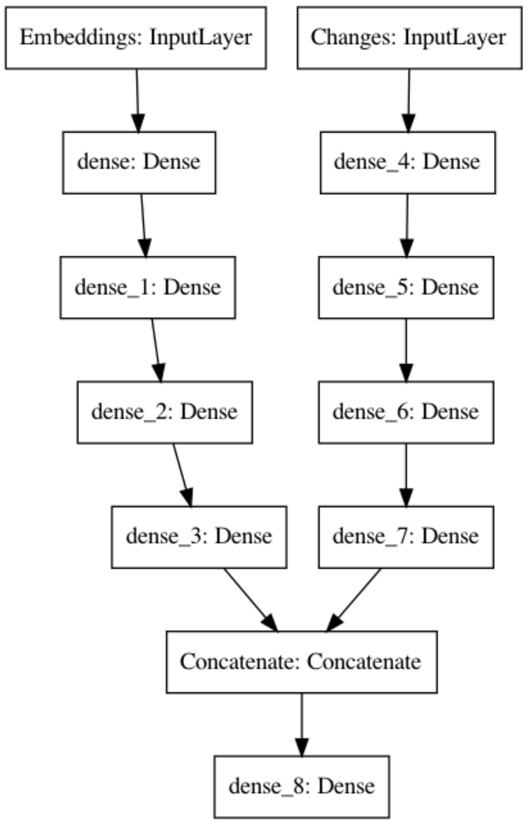


Figure 9: Separated networks

There is still no satisfactory recipe for solving this task other than by trying it out [11].

3.4 Evaluation approach

For comparing the different embedding methods as a part of the representation learning process, we first compare the performance of some classifiers. The classification approaches we apply on top of our data, are from the area of Regression, Naive Bayes, Tree-Based Algorithms as well as Support Vector Machines and Multilayer Perceptrons (see table 1). In the classification step all possible combinations were evaluated.

The evaluation metrics we are looking at are standard classification metrics, namely **precision**, **recall**, **f1**-measure, **accuracy** and **roc_auc**. We consider these for changes, embeddings separated and then both together. For each model (Complex, DistMult, TransE, TransH) we examine whether the results of vector manipulation improve or worsen and which combination of classifier and model works well. For each case we look for the two best result per metric and use these results to judge which experiment is the most suitable. We evaluated each classification method on *changes*,

Table 1: Classifiers

Category	Method
Regression	Logistic Regression (LR)
Naive Bayes	Gaussian Naive Bayes (NB)
Nearest Neighbour	KNN
Tree-Based Algorithms	CART, Random Forest (RF)
Support Vector Machines	RBF-Kernel, Linear Kernel
Multilayer Perceptron	MLP

embedding and *both* to figure out if our approaches improved the learning process.

4 Implementation

Before feeding data to our neural network we need it in a specific way so we need to process it accordingly. The preprocessing of data depends on the type of data. Since our input files contain numbers and letters, we have to encode them first so that we can use them directly in binary form. This leads to the first step.

The data is read in, then transformed in the One Hot Encoding step to prepare for training. Then the data is trained and finally evaluated considering the selected success metrics. So that we can better compare the results, we evaluate the training once without an inserted change step. This evaluation is intended to serve as a basis for comparison for the various approaches.

4.1 Implementation approach

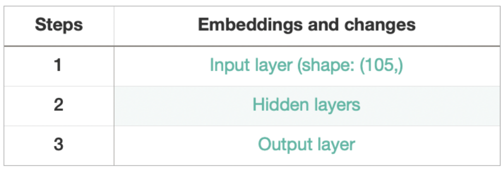
The dataset used in our experiments is a real-word dataset from the domain of biomedical ontologies. The given embeddings and change alignments have been extracted in a previous research [5]. The implementation of the Vector Manipulation approaches the embeddings and changes were trained using DistMult, Complex, TransH and TransE model in a python environment. For the implementation of the MLP Neural Classifier and Classifier for separated networks we us Keras Sequential and Functional API. The classification itself was implemented using scikit-learn [10].

4.2 Vector manipulation and normalization

The manipulation of the change types will be defined in the function "embedding_to_manipulate(actualChangeDetails, embedd_vec)". The normalization of the vectors has been done with the numpy function *linalg.norm()*.

4.3 Keras for MLP Neural Classifier and Classifier for separated networks

The core data structures of Keras are layers and models. The simplest type of model is the Sequential model, a linear stack of layers like described in figure 10. We create a Sequential model incrementally via the add() method. We start our model by passing an Input object to it with the first layer. In the hidden layer we add alternately dense and dropout layers. For activation we use relu and layer with 50, 25, 10, 2 neurons on top of each other. We test 0.2, 0.1, 0.01, 0.001 as a dropout.



Steps	Embeddings and changes
1	Input layer (shape: (105,)
2	Hidden layers
3	Output layer

Figure 10: Keras sequential model

The arguments of Dense are output dimension which is 50 in the first case, input dimension which is 105 for input dimension and the activation function to be used which is relu in this case:

```
model.add(Dense(50, activation='relu', input_shape=(105,)))
```

The second layer is similar, we dont need to specify the input dimension as we have defined the model to be sequential so keras will automatically consider the input dimension to be same as the output of the last layer i.e 50. In the third layer(output layer) the output dimension is 25(number of classes). The number of hidden layers and number of neurons in hidden layers is not fixed and so we tested different numbers and tinker to get the best results.

Output layer predicts the class. The output layer takes different activation functions and for the case of multiclass classification, it is softmax. We add a softmax layer to the network, so that it is possible to translate the numbers into a probability distribution [13]. As final step we need to specify the loss function and the optimizer.

```
    model.compile(loss='binary_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

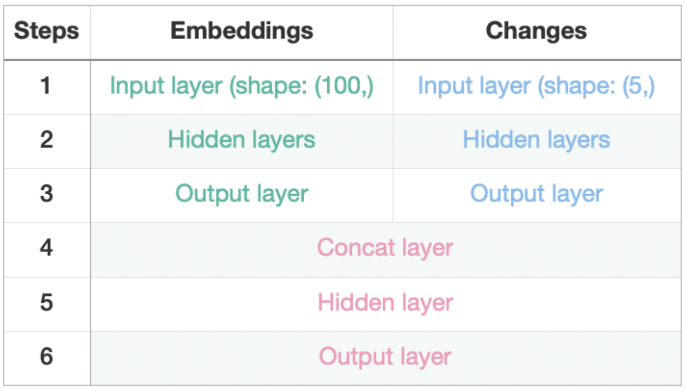
Here loss is cross entropy loss. Categorical_crossentropy specifies that we have multiple classes. The optimizer is Adam. Metrics is used to specify the way we want to judge the performance of our neural network. Here we have specified it to accuracy.

The structure for subnetworking is somewhat different than for simple layering. The shape tuple is defined with a hanging last dimension because the input is one-dimensional (100,). This is because space needs to be left for the shape of the mini-batch size used in dividing the data when training the network.

```
            inp1 = Input(shape=(100,))
```

The model of the embeddings has 100 inputs, 3 hidden layers with 100, 200, and 100 neurons, and an output layer with 1 output. Rectified linear activation functions are used in each hidden layer. A softmax activation function is used in the output layer, for binary classification.

The model of the changes has 5 inputs, 3 hidden layers with 5, 10, and 5 neurons, and an output layer with 1 output. Rectified linear activation functions are used in each hidden layer. A softmax activation function is used in the output layer, for binary classification.



Steps	Embeddings	Changes
1	Input layer (shape: (100,)	Input layer (shape: (5,)
2	Hidden layers	Hidden layers
3	Output layer	Output layer
4	Concat layer	
5	Hidden layer	
6	Output layer	

Figure 11: Keras subnetworking

The result are two MLP feature extraction submodels. The outputs from these feature extraction submodels are flattened into vectors and concatenated into one long vector and passed on to a fully connected layer for interpretation before a final output layer makes a binary classification.

The Layer $keras.layers.Concatenate(axis = -1)$ concatenates a list of inputs. It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor, the concatenation of all inputs. The model has already been build with functional model objects. The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket notation is used, such that after the layer is created, the layer from which the input to the current layer comes from is specified [3].

Table 2: Dataset

	Version	Entities	Triples	#Cchange	#Cnochange
FMA-NCI	2009-2011	2M	7M	725	984

5 Evaluation

The main research question of our evaluation is whether the approach with vector manipulation is better than keras subnetworking. What process can we use to better index new changes? The impact of the choice between vector normalization and coding change types as vector addition also needs to be examined. In the case of keras layering, an important contemplation is the separation of change and embedding data in subnetworks. In order to evaluate our methods, we conducted a series of experiments that are described in the previous subsections. We examine the following approaches: Vector manipulation, normalization, Keras layering and Keras subnetworking.

5.1 Data

The dataset we use has been extracted from biomedical ontologies by Jiménez-Ruiz et al. [4] and has been made publically available by Groß et al. [2]. This dataset consists of three biomedical ontologies – SNOMED, FMA and the NCI Thesaurus, with versions from 2009-2012 and mappings between all ontologies for each of those versions. In the previous work of [6] the change of alignment statements has been examined. For each changed entity close to an alignment statement in a new version of the ontology, we examine, if an alignment statement has been changed. If this was the cause, the entity was assigned to the class C_{change}, else, to the class $C_{nochange}$. Table 2 gives an overview of the datasets we use. Overall we have more entities that are assigned to the class $C_{nochange}$.

The input data consists of this ontology that have been aligned. A neural network model cannot take strings of characters as direct input. Therefore it has beem transformed into a set of numerical vectors that a model can use as input. These embeddings provide a representation vector for each possible alignment and is of dimension 100. The alignment is a set of comparisons between each class of both ontologies to be aligned. Since the preprocessing steps are already done, the input data can be used to train a neural network model. Therefore we divide the data divide into training and test data. We use established embedding approaches TransE [1], TransH [15], Distmult [18] and Complex [17] from the area of language-modeling based approaches. These models are the basis from which the embedding

Table 3: Change types

Change Type	
Deleted	0.603225806451613
Added	0.615023474178404
Superclass	0.722222222222222
Annotation	0.590909090909091
Renamed	0
Overall:	0.628947368421053

files are, from which we build the training and test data.

In the experiment we consider the change types "Deleted", "Added", "Superclass", "Renamed" and "Overall". In table 3 the percentage of the change is displayed. This part is important if certain success metrics do not change and, in case of doubt, help to find out whether the model has learned.

5.2 Baselines

For comparing different embedding methods as a part of the representation learning process, we first train embeddings and then compare the performance of classifiers that use these embeddings as features on our dataset. As evaluation metrics we use standard classification metrics, namely precision, recall, f1-measure, accuracy and roc_auc. Since f1, precision and roc_auc are the most interesting metrics here, only these are shown below. The complete results can be viewed in the appendix. We consider the results separately in **changes**, **embeddings** and **both**. The results for the different models without any vector manipulation or normalization is shown in the tables 17 to 20. For readability and clarity purposes, we have only included the best result per metric and embedding method.

The *changes* behave the same for all embedding models. For changes we have the best **f1** value with **Naive Bayes**. The **precision** value is best for the **Logistic Regression**, the **Support Vector Machines (RBF kernel)** and the **Multilayer Perceptron 250**. The **roc_auc** value is highest for the **Support Vector Machines (linear kernel)**.

Overall, it is noticeable that the best results are achieved for the ComPlex embedding for one classifier. The **ComPlex** model has the best results for the tree-based algorithm CART (see table 17).

For a better overview of the combination of model and classifier in which the best results were achieved, we first consider the metrics separately. the two best values for changes, embeddings and both are shown in the tables 4, 5 and 6. The best **f1** values 4are achieved with **Naive Bayes**. Only the best **f1** value for *both* was achieved

Table 4: Baseline: f1 value

	classifier	f1	model
changes	NB	0.74	all
embeddings	NB	0.77	TransH
embeddings	NB	0.75	DistMult
both	SVM-linear	0.77	TransE
both	NB	0.77	TransH

Table 5: Baseline: precision value

	classifier	precision	model
changes	LR/SVM-rbf/MLP250	0.76	all
embeddings	MLP250	0.813	DistMult
embeddings	CART	0.809	Complex
both	RandomForest	0.810	TransE
both	MLP250	0.807	DistMult

with a different classifier: **SVM-linear**. Overall, the translation-based models had better **f1** values than the others. The best **f1** values range between 0.74 and 0.77.

The **precision** 5 values are very much influenced by the classifiers, which work well with DistMult and Complex: **MLP250** (DistMult) and **CART** (Complex). Only the best precision value for both was achieved with a different classifier: RandomForest (TransE). The best precision values range between 0.807 and 0.813.

The best **roc_auc** 6 values are even more clearly characterized by Complex and DistMult. These range between 0.69 and 0.70.

5.3 Results: Vector manipulation and normalization

Now that we have considered the results for each metric in the baselines, we will examine the individual metrics in the degenerated cases. So in this part we look at the results of the metrics for the normalized (7, 8, 9) and for the manipulated

Table 6: Baseline: roc_auc value

	classifier	roc_auc	model
changes	MLP250	0.53	DistMult
changes	SVM-linear	0.53	all except DistMult
embeddings	MLP250	0.69	DistMult
embeddings	CART	0.69	Complex
both	CART	0.70	Complex
both	MLP250	0.70	DistMult

vectors(10, 11, 12).

We consider these in parallel in comparison to the baselines, initially for embeddings and both, then across the board.

With the normalized vectors the **f1** value is a little better with the complex model if embeddings and changes are separated. The **f1** value for **both** is slightly worse than before. If you look at both, embeddings and changes together, the values are much better in both procedures. If you consider embeddings separately from the changes, the values are at the same level as before.

The **precision** value for changes has not changed either. The other precision values have improved slightly. The models are the same as in the baseline, TransE, DistMult and Complex, only the ranking and some classifiers have changed. DistMult reached the best value with SVM-rbf for both and TransE the best value with Naive Bayes for embeddings. The best precision values range between 0.811 and 0.814.

The **roc_auc** values have not changed for changes. The ranking has changed for **embeddings**, but **roc_auc** values were very close to one another without and with normalization of the vectors. The vector normalization has not really changed anything here either. The best **roc_auc** value for both is slightly higher than before (DistMult with MLP250). The **roc_auc** values range between 0.69 and 0.71.

Manipulating the vectors by coding the change in vector addition did not noticeably improve the results either. The **f1** values for changes have improved minimally in the decimal places. The **translation-based** methods were able to improve somewhat and are dominant in the results for embeddings and both. DistMult also improved slightly. The classifier SVM-rbf was the best performing for **embeddings**. SVM-linear and Naive Bayes had the highest values for **both**.

The **precision** value for changes has gone down significantly. The **translation-based** methods performed worse than before for embeddings and both. Thus DistMult and Complex are ahead here, and their values have not changed at all. The range of the best **precision** values is between 0.805 und 0.809.

The **roc_auc** values for changes went down slightly. The DistMult model dominates this metric and has one of the best value everywhere. DistMult increased slightly for embeddings and Complex has deteriorated for both. The range of the best **roc_auc** values is between 0.68 and 0.703.

So that we can better assess whether our approaches have led to improvements, we compare the best values per metric for changes, embeddings and both. For embeddings (see table 13), the **f1** value for vector manipulation has improved by 0.2 compared to the baseline. On the other hand the precision has decreased by 0.2 here. The normalization has only resulted in very minimal changes, which can also

Table 7: Normalized vectors: f1 value

	classifier	**f1**	**model**
changes	NB	0.74	all
embeddings	NB	0.765	TransH
embeddings	NB	0.745	DistMult
both	SVM-linear	0.767	TransE
both	NB	0.76	TransH

Table 8: Normalized vectors: precision value

	classifier	**precision**	**model**
changes	LR/SVM-rbf/MLP250	0.76	all
embeddings	NB	0.812	TransE
embeddings	CART	0.811	Complex
both	SVM-rbf	0.813	DistMult
both	CART	0.811	Complex

Table 9: Normalized vectors: roc_auc value

	classifier	**roc_auc**	**model**
changes	SVM-linear	0.53	all
embeddings	CART	0.696	Complex
embeddings	RandomForest	0.692	TransE
both	MLP250	0.71	DistMult
both	CART	0.699	Complex

Table 10: Manipulated vectors: f1 value

	classifier	**f1**	**model**
changes	NB/SVM-linear	0.75	TransH
changes	NB/SVM-linear	0.76	all except TransH
embeddings	SVM-rbf	0.79	TransH
embeddings	SVM-rbf	0.78	DistMult
both	SVM-linear	0.77	TransE
both	NB	0.77	TransH

Table 11: Manipulated vectors: precision value

	classifier	**precision**	**model**
changes	KNN	0.64	DistMult
changes	NB/SVM-linear	0.63	Complex/TransE
embeddings	CART	0.809	Complex
embeddings	RandomForest	0.807	TransE
both	MLP250	0.807	DistMult
both	RandomForest	0.805	TransE

Table 12: Manipulated vectors: roc_auc

	classifier	**roc_auc**	**model**
changes	KNN	0.51	DistMult
changes	NB/SVM-linear	0.50	Complex/TransE
embeddings	MLP250	0.70	DistMult
embeddings	CART	0.69	TransH
both	MLP250	0.70	DistMult
both	CART	0.69	Complex

Table 13: Results for embeddings

	f1	**precision**	**roc_auc**
baseline	0.77	0.813	0.69
normalization	0.765	0.812	0.696
manipulation	0.79	0.79	0.7

be attributed to calculation inaccuracies. Overall, it looks similar for both (see table 14). The normalization also leads to very little improvement when considering the best value. The vector manipulation also leads to a worse precision value here.

5.4 Results: MLP Neural Classifier and Classifier for separated networks

The results from both models reflect the majority class. The accuracy and precision for the sequential model and for subnetworking are 0.628947. Further values for subnetworking are the f1 value with 0.772213, roc_auc with 0.5 and recall with 1.0 (see table 15).

5.5 Discussion

The procedures with vector normalization and coding changes as vector addition had slightly positive effects on the learning behavior of the embedding models. When comparing the approaches that combine embeddings with DistMult and MLP250 shows the best results. DistMult performs best before the optimization attempts. The translationbased embedding methods seem also very promissing.

Table 14: Results for both

	f1	**precision**	**roc_auc**
baseline	0.77	0.81	0.7
normalization	0.767	0.813	0.71
manipulation	0.77	0.77	0.7

Table 15: Keras subnetworking results

Embedding model	f1	accuracy	precision	recall	roc_auc
TransH:	0.772213	0.628947	0.628947	1.0	0.5
TransE:	0.772213	0.628947	0.628947	1.0	0.5
Complex:	0.772213	0.628947	0.628947	1.0	0.5
DistMult:	0.772213	0.628947	0.628947	1.0	0.5

Table 16: Comparison of the vector approaches

	f1	**precision**	**roc_auc**
baseline	0.7	0.813	0.7
normalization	0.767	0.813	0.71
manipulation	0.79	0.809	0.7

The values for embeddings and both are just as good as those for DistMult. With both, Complex hangs behind the others when it comes to precision.

The other embedding models are catching up with the optimizations, but they can only catch up and not overtake. Overall, only improvements or minimal deteriorations can be observed in the second decimal place. Complex comes close to the other embedding models and now performs similarly well for both.

When comparing the baseline approaches that combine embeddings with traditional classification methods, we can observe that DistMult in combination with MLP250 seems to show the best results where the computational effort allows it. Another embedding method that seems very promising is Complex in combination with CART. By normalizing the training vectors or coding the change types as vector addition TransE and TransH achieve promising results, too. Due to the manipulation, the embedding approaches are close to each other. One success is that neither with embeddings nor both, there is no combination that is lagging behind the best values. However, one should keep in mind that we are only looking at the best f1, precision and roc_auc values here. Because when comparing the best values for embeddings and both, no major changes can be seen. For a further approach, it might be useful to tune well-functioning combinations of embedding models and classifiers.

The custom neural network has not learned in either the sequential or the subnetworking version. One reason for this could be the activation function in the hidden layers. Negative values cannot be processed correctly with ReLu. The issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately [12].

6 Conclusion

In this paper, we presented two main approaches to predicting alignment changes after an ontology update: a two-step approach that consists of representation learning as a first step and established classification methods as a second step and an end-to-end approach that uses MLP Neural Classifier and Classifier for separated networks. In our evaluation, we could show that on the dataset we used, the best-performing representation learning approaches where DistMult and Complex. The translation-based models performed slightly better after vector manipulation. On the dataset, best-performing classifiers had a precision of 0.81. For changes, the performance was significantly worse with 0.74 to 0.76. Adding the changes to the embeddings did not have a positive effect on training. The metrics have hardly changed here. Keras for MLP Neural Classifier and Classifier for separated networks have been very promising, as the MLP250 also worked well with conventional embedding models. But there was no learning effect here.

As future work, an expansion of both main approaches would be conceivable. Further evaluations could be performed on different datasets. Datasets with more change types could be a possible refinement. In this way, the change types could achieve greater added value in improving the classification. Also, a combination of our approaches with existing mapping adaption approaches could be examined. In order to improve the second approach with custom neural networks, we could use Leaky Relu as an activation function instead of Relu. This offers the attempt to compensate for the elimination of negative values. Another approach would be to use a CNN instead of the MLP classifier. The layers are sparsely connected or partially connected rather than fully connected. Every node does not connect to every other node. The lack of success in learning could also be due to the fact that MLP include too many parameters because it is fully connected. Each node is connected to another in a very dense web — resulting in redundancy and inefficiency.

A Evaluation appendix: Vector manipulation

A.1 Baselines

With the **DistMult** model, the best results barely differ with the **f1** value. **Embeddings** have the best value for **Naive Bayes** and **embeddings and changes** have the best value for **Support Vector Machines (RBF kernel)**. The best values for both **precision** and **roc_auc** are achieved here with the **MLP250** (see table 18).

Table 17: Complex model

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250	0.76	SVM-linear:	0.53
embeddings	CART:	0.72	CART:	0.809	CART:	0.69
both	CART:	0.73	CART:	0.69	CART:	0.70

Table 18: DistMult model

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250	0.76	SVM-linear:	0.53
embeddings	NB:	0.75	MLP250	0.81	MLP250:	0.698
both	SVM-rbf:	0.75	MLP250	0.81	MLP250:	0.70

The **TransE** model has classifiers similar to DistMult for the **f1** value. The only difference is that for *embeddings and changes* the best value is achieved for **SVM-lin** and not for SVM-rbf. For the metrics **precision** and **roc_auc**, **CART** achieved the best results (see table 19).

The **TransH** model has the best results with the **f1** value with the **Naive Bayes** method. The results differ for *embeddings only* and *embeddings and changes*. **Random-Forest** performed best with *embeddings only* and **CART** with *embeddings and changes* (see table 20).

A.2 Vector manipulation and normalization

With the **DistMult** model, the best results for the **f1** value behave similarly as before. With the manipulated vectors the best results are with slightly different models. **Changes** have the best **f1** value with **SVM-linear**, **embeddings** with **SVM-rbf** and **both** with **SVM-linear**.

The best values with **embeddings** and **both**, for both **precision** and **roc_auc**, are achieved here with the **MLP250**. The values for both DistMult models are at a similar level as before.

The results of the **TransE** model are slightly better with the manipulated vectors than with the normalized vectors. With the manipulated vectors the best results for

Table 19: TransE model

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	MLP250:	0.53
embeddings	NB:	0.74	RandomForest:	0.80	RandomForest:	0.68
both	SVM-lin:	0.77	RandomForest:	0.81	RandomForest:	0.69

Table 20: TransH model

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	SVM-linear:	0.53
embeddings	NB:	0.76	RandomForest:	0.795	RandomForest:	0.68
both	NB:	0.77	CART:	0.80	CART:	0.69

Table 21: Complex model manipulated vectors

	f1		precision		roc_auc	
changes	SVM-linear:	0.76	SVM-linear:	0.63	SVM-linear:	0.502359
embeddings	SVM-rbf:	0.74	CART:	0.81	CART:	0.69
both	CART:	0.72	CART:	0.80	CART:	0.69

changes have been gained with **Naive Bayes** and **SVM-linear**, **embeddings** with **SVM-rbf** and **both** for **SVM-linear**, the remainind models for do not differ from the results before. The **f1** value for **embeddings** has increased slightly. With the normalized vectors the **precision** value has increased minimally with the CART model. The best **f1** values for **changes** have been gained with **Naive Bayes** model in the case of separated **changes** and **embeddings**.

The results of the **TransH** model are slightly better with the normalized vectors than with the manipulated vectors. With the manipulated vectors the best results for **changes** have been gained with **Naive Bayes** and **SVM-linear**, **embeddings** with **SVM-rbf** and **both** for **Naive Bayes**. The **f1** value for **embeddings** has increased slightly. With the normalized vectors the **precision** value has increased minimally with the CART model.

Table 22: Complex model normalized vectors

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	SVM-linear:	0.53
embeddings	CART:	0.72	CART:	0.81	CART:	0.696
both	CART:	0.73	CART:	0.81	CART:	0.699

Table 23: DistMult model manipulated vectors

	f1		precision		roc_auc	
changes	SVM-linear:	0.76	KNN:	0.64	KNN:	0.51
embeddings	SVM-rbf:	0.78	MLP250:	0.80	MLP250:	0.70
both	SVM-rbf:	0.75	MLP250:	0.81	MLP250:	0.70

Table 24: DistMult model normalized vectors

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	SVM-linear:	0.53
embeddings	NB:	0.75	MLP250:	0.81	MLP250:	0.69
both	SVM-rbf:	0.75	MLP250:	0.81	MLP250:	0.71

Table 25: TransE model manipulated vectors

	f1		precision		roc_auc	
changes	NB/SVM-linear:	0.76	NB/SVM-linear:	0.63	NB/SVM-linear:	0.50
embeddings	SVM-rbf:	0.78	RandomForest:	0.81	RandomForest:	0.69
both	SVM-linear:	0.77	RandomForest:	0.81	RandomForest:	0.69

Table 26: TransE model normalized vectors

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	SVM-linear:	0.53
embeddings	NB:	0.74	CART:	0.81	RandomForest:	0.69
both	SVM-linear:	0.77	CART:	0.81	RandomForest:	0.69

Table 27: TransH model manipulated vectors

	f1		precision		roc_auc	
changes	NB/SVM-linear:	0.75	alle:	~ 0.62	alle:	0.5
embeddings	SVM-rbf:	0.79	CART/RandomForest:	0.799	CART:	0.689
both	NB:	0.77	CART:	0.80	CART:	0.69

Table 28: TransH model normalized vectors

	f1		precision		roc_auc	
changes	NB:	0.74	LR/SVM-rbf/MLP250:	0.76	SVM-linear:	0.53
embeddings	NB:	0.77	CART:	0.80	CART:	0.69
both	NB:	0.77	CART:	0.80	CART:	0.69

References

[1] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2787–2795, USA, 2013. Curran Associates Inc.

[2] Anika Groß, Julio Cesar Dos Reis, Michael Hartung, Cédric Pruski, and Erhard Rahm. Semi-automatic adaptation of mappings between life science ontologies. In Christopher J. O. Baker, Greg Butler, and Igor Jurisica, editors, *Data Integration in the Life Sciences*, pages 90–104, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] Jason Brownlee. How to use the keras functional api for deep learning. 2020.

[4] Ernesto Jiménez-Ruiz, Bernardo Grau, Ian Horrocks, and Rafael Berlanga. Logic-based assessment of the compatibility of umls ontology sources. *Journal of biomedical semantics*, 2 Suppl 1:S2, 03 2011.

[5] Matthias Jurisch and Bodo Igler. Rdf2vec-based classification of ontology alignment changes. *CoRR*, abs/1805.09145, 2018.

[6] Matthias Jurisch and Bodo Igler. Graph-convolution-based classification for ontology alignment change prediction. In Mehwish Alam, Davide Buscaldi, Michael Cochez, Francesco Osborne, Diego Reforgiato Recupero, and Harald Sack, editors, *Proceedings of the Workshop on Deep Learning for Knowledge Graphs (DL4KG2019) Co-located with the 16th Extended Semantic Web Conference 2019 (ESWC 2019), Portoroz, Slovenia, June 2, 2019*, volume 2377 of *CEUR Workshop Proceedings*, pages 11–20. CEUR-WS.org, 2019.

[7] Keras. Keras documentation. 2020.

[8] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2181–2187. AAAI Press, 2015.

[9] George Novack. Building a One Hot Encoding Layer with TensorFlow, Juni 7, 2020.

[10] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[11] Hassan Ramchoun, Mohammed Amine, Mohammed Amine Janati Idrissi, Youssef Ghanou, and Mohamed Ettaouil. Multilayer perceptron: Architecture optimization and training. *International Journal of Interactive Multimedia and Artificial Inteligence*, 4:26–30, 01 2016.

[12] Sagar Sharma. Activation functions in neural networks. 2020.

[13] Thomas Wood. What is the softmax function? 2020.

[14] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction, 2016.

[15] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1112–1119. AAAI Press, 2014.

[16] Will Koehrsen. Neural network embeddings explained - how deep learning can represent war and peace as a vector. 2020.

[17] Chao Xing, Dong Wang, Chao Liu, and Yiye Lin. Normalized word embedding and orthogonal transform for bilingual word translation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1006–1011, Denver, Colorado, May–June 2015. Association for Computational Linguistics.

[18] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and li Deng. Embedding entities and relations for learning and inference in knowledge bases. 12 2014.

[19] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and li Deng. Learning multi-relational semantics using neural-embedding models. 11 2014.

[20] Hui Yu, David C. Samuels, Ying-yong Zhao, and Yan Guo. Architectures and accuracy of artificial neural network for disease classification from omics data. *BMC Genomics*, 20(1):167, 2019.

[21] Denghui Zhang, Manling Li, Yantao Jia, Yuanzhuo Wang, and Xueqi Cheng. Efficient parallel translating embedding for knowledge graphs. In *Proceedings of the International Conference on Web Intelligence*, WI '17, pages 460–468, New York, NY, USA, 2017. ACM.