UTRECHT UNIVERSITY

MASTER'S THESIS

Leveraging Static Models for Temporal Knowledge Graph Completion

Author:
Wessel Radstok (5692024)
wradstok@gmail.com

Supervisor:
Dr. Melisachew Wudage Chekol
m.w.chekol@uu.nl

Secondary supervisor:
Prof. Dr. Yannis Velegrakis
i.velegrakis@uu.nl

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computing Science

Department of Information and Computing Science
January 11, 2021

Abstract

The inclusion of temporal information in knowledge graph embedding (KGE) has remained relatively unexplored, even though it stands to argue that this would result in better embeddings. Additionally, models that do include the temporal component perform only marginally better than those that do not (static models). Noting this, we introduce SpliMe, a model-agnostic pre-processor for temporal knowledge graphs that makes it possible to embed them with static models. We show that SpliMe achieves state-of-the-art performance on two datasets commonly used for temporal KGE method evaluation and increases performance with regards to our baseline on another dataset. Furthermore, we uncover problems with existing evaluation procedures for static KGE models on temporal graphs and propose a simple method to fix these issues. Finally, we redefine the link prediction metric for temporal knowledge graph embedding to better suit temporal scope prediction.

Contents

1. Introduction

A knowledge graph (KG) is a graph used for representing structured information about the world, which can then be utilized in other tasks such as question answering, predicate extraction and recommender systems. Many large KGs have been created in recent years, such as DBPedia, Google Knowledge Graph and FreeBase. Using the Resource Description Framework (RDF) language created by the World Wide Web Consortium, these facts are stored as *(subject, predicate, object)* (*s,p,o*) triples. For instance, *(Trump, BornIn, New York).* The subject and object are also referred to as *entities*, and the predicate as the *relationship* between them. In the graph representation, the entities are nodes, and predicates are labeled directed edges connecting them.

Existing knowledge graphs are very large: DBPedia contains around 10 billion triples (Lehmann et al., 2015). Yet, much information is still missing. For example, only 30% of the people in FreeBase have place of birth information (West et al., 2014). However, due to redundancy in the data, it is often possible to recover this information. For instance, we could deduce someones country of birth by looking at his city of birth. Predicting the most likely connection between entities is called *link prediction.* Using link prediction to fill in missing links in a KG is called *knowledge graph completion.*

Another problem for KGs is dealing with evolving or historic information. Facts that may be true at one point in time might be invalidated later. As an example, *(Obama, PresidentOf, USA)* was only true between 2009-2017. One approach to solving this issue is by including temporal information with the facts. A temporal knowledge graph (TKG) is a KG where (a subset of) the facts have a temporal scope denoting the time during which they were valid, or the time at which they occurred. An example of the latter is (*Obama, awarded, Nobel Peace Prize, 2015*).

Over the past few years there has been much research into how to perform knowledge graph completion. Much research has focussed on *latent variable models* (Nickel et al., 2011; Bordes et al., 2013; Trouillon et al., 2016). In these models, entities and predicates are represented as vectors, called *embeddings.* Link prediction is then done through transformations on these embeddings.

It stands to reason that including temporal information inside these embeddings would result in improved performance. Yet, few embedding models have been created for this purpose (e.g. (Jiang et al., 2016; García-Durán et al., 2018)), and their results are only marginally better than static KG embedding models. Additionally, many TKG embedding models (e.g. (Xu et al., 2019; Goel et al., 2019)) require that all facts in the KG have temporal scopes, even though many KGs are only partially temporal.

Noting this, this thesis will aim for the following goals:

- Develop a model-agnostic TKG embedding method which utilizes feature engineering to incorporate temporal information at the level of entities and predicates.
- Reformulate the link prediction task to suit temporal scope prediction.
- Provide a runnable implementation of the system in Python for reproducibility.

2. Preliminaries

2.1. Knowledge Graphs and the RDF Data Model

The first specification of the Resource Description Framework (RDF) was published by the World Wide Web Consortium in 2004, with a followup 1.1 specification made official in 2014. RDF is a general method for modelling information, specifically aimed to be used in web resources. Information is stored as triples which consists of a subject s, a predicate p and an object o, in that order i.e. $(s{,}p{,}o)$. Examples of RDF triples are (*Utrecht, provinceIn, Netherlands*) and (*Penrose, winnerOf, Nobel prize*). In some literature, a triple is said to be composed of a *head*, *relation* and *tail*. In this thesis we use the terms relation and predicate interchangeably, but stick to the $(s{,}p{,}o)$ notation.

More formally, we consider a set of entities $\mathcal{E} = \{e_1, e_2, \dots, e_n\}$ and a set of predicates $\mathcal{P} = \{r_1, r_2, \dots, r_m\}$. Let $|X|$ denote the cardinality of set X. Then $|\mathcal{E}| = n$ and $|\mathcal{P}| = m$. A knowledge graph $\mathcal{D}$ is a set of observed RDF triples, i.e. $\mathcal{D} = \{(s, p, o)|s, o \in \mathcal{E}, p \in \mathcal{P}\}$.

Predicates have different cardinality types. That is, a relation can be a one-to-one (1-1), a one-to-many (1-n), a many-to-one (n-1) or a many-to-many (n-n) relation. This distinction will become useful later. Furthermore, as defined in Sun et al. (2019), we recognize four different types of predicate patterns: *(anti-)symmetry*, *inversion* and *composition*. We will now give a formal definition for all four.

- A predicate p is **symmetric** when pair $\forall x, y \in \mathcal{E},\ (x, p, y) \implies (y, p, x)$. An example of this is the predicate *marriedTo*.
- A predicate is **anti-symmetric** when $\forall x, y \in \mathcal{E},\ (x, p, y) \implies \neg(y, p, x)$. An example of an asymmetric relationship is the *childOf* predicate.
- A predicates $p_1 \in \mathcal{P}$ is **inverse** to $p_2 \in \mathcal{P}$ if $\forall x, y \in \mathcal{E},\ (x, p_2, y) \implies (y, p_1, x)$. An example of this are the *childOf* and *parentOf* relations.
- Lastly, a predicate $p_1 \in \mathcal{P}$ is said to be **composed** of predicates $p_2, p_3 \in \mathcal{P}$ if $\forall x, y, z \in \mathcal{E},\ (x, p_2, y) \wedge (y, p_3, z) \implies (x, p_1, z)$. An example of this is how the relation *nieceOf* is composed of the relations *childOf* and *siblingOf*

RDF has no official semantics to specify temporal information for triples, but extensions do exist. Our notation is based on the formalization of Gutierrez et al. (2005). Specifically, we consider time as a as point based, discrete, linearly ordered domain, i.e. $\mathcal{T} = \{t_1, t_2, \dots, t_l\}$ s.t. $|\mathcal{T}| = l$. We consider two types of temporal knowledge graphs. The first variety is an event KG. Here, each triple is annotated with a timestamp $h \in \mathcal{T}$ which denotes when the fact occurred, i.e. an event KG is a set of observed quadruples: $T = \{(s, p, o, h)\ |s, o \in \mathcal{E}, p \in \mathcal{P},\ h \in \mathcal{T}\}$. The second variety is the *valid time* representation. Each fact is annotated with a begin time $b \in \mathcal{T}$ and end time $e \in \mathcal{T}$, denoting when the fact held true. This is represented as a set of quintuples: $T = \{(s, p, o, b, e)\ |s, o \in \mathcal{E}, p \in \mathcal{P},\ b, e \in \mathcal{T}, b \leq e\}$.

Whether a fact is modelled as an event can depend on the granularity at which we consider the data. To illustrate, if we were to take the time granularity of a year we would say that Mt. Everest was first successfully ascended by Hillary and Norgay in 1953. Yet, at the granularity of a month we would say that it was first successfully ascended between March 1593 and May 1953.

Additionally, we use superscript to define a subset of a (temporal) knowledge graph: let $T^{p=r}$ denotes all facts in the TKG that contain predicate $r \in \mathcal{P}$, i.e. $T^{p=r} = \{(s,p,o,b,e) \in T \mid p = r\}$. Analogously, we define $T^{s=e}$ as the TKG of all facts containing entity $e \in \mathcal{E}$ as subject, $T^{b=t}$ as the TKG containing all facts with start time $t \in \mathcal{T}$ and so on. This indexation can be chained: $T^{p=r,\ e=t}$ is the TKG of all facts which contain predicate r and end at timestamp t.

2.2. Knowledge Graph Construction and Applications

Propietary: Knowledge graphs have seen a wide variety of uses in recent years. The most well known use is the enhancement of search, for which the Google Knowledge Graph was created in 2013. Containing around 500 million triples and 18 billion facts at the time, the Google KG is used to discern entities in search and provide the user with structured information (Dong et al., 2014). Additionally, companies like Microsoft and Yahoo have also created KGs for this purpose. These are general purpose knowledge graphs, which model general information about the world. E.g., who is married to who, why a certain person is famous or that a city lies in a certain country.

Another purpose of knowledge graphs is in the case of recommender systems. An example of this would be the Facebook Social Graph, which models which people are friends which each other, what people link and how they interact with each other. This information can then be used to suggest pages that one may be interested in, or people that one may want to become friends with. However, just like the search examples these graphs are proprietary and as a result little information is known about how they are structured and how they were created.

Open: As a counterpart to these proprietary KGs, there also also open source knowledge graphs. For instance, WordNet is a knowledge graph denoting the semantic relations between words, using relations such as ”synonym”, ”hypernym” and ”hyponym” (Miller, 1995). Originally constructed for the english language by a group of experts, new WordNets have since been created for other languages and are linked to the original WordNet (e.g. (Vossen, 1998), (Vossen et al., 1999)). The resulting KG is widely used in the natural language processing community.

One of the first academic projects to automatically construct a knowledge graph was YAGO, which has recently seen the release of its fourth installment (Pellissier Tanon et al., 2020). YAGO takes the semi-structured information present in Wikipedia infoboxes and combines this with lexical information from WordNet to create a comprehensive general purpose KG. Through manual inspecting of sampled facts the accuracy of the previous version, YAGO3, has been estimated at 95%.

Another example of an open source KG is Wikidata (Erxleben et al., 2014). Wikidata is

Knowledge Graph	**# Entities**	**# Predicates**	**# Triples**	**Temporal**
YAGO3 (2015)	5M	77	17M	Partly
FreeBase (2013)*	40M	35,000	637M	No
YAGO4 (2020)$^+$	67M	151	343M	Partly
Wikidata (2020)$^+$	78M	7,947	974M	Partly
Google KG (2013)*	570M	35,000	18000M	Unknown

Table 1: An overview of several general purpose knowledge graphs, and whether they include temporal information. *: data taken from Nickel et al. (2015), $^+$: data taken from Pellissier Tanon et al. (2020). YAGO3 information was taken from (Mahdisoltani et al., 2013).

collaborative, meaning that anyone can contribute to it just like in Wikipedia. Currently, around 40.000 people contribute to Wikidata at least once a month. However, to make this collaboration possible Wikidata does not enforce semantic constraints (i.e. a person can have more than one father). Consequently, its accuracy is generally lower than that of curated KGs. Additionally, there exists an effort called Linked Open Data Cloud which aims to interlink many different KGs by mapping related entities and predicates to each other. As of May 2020, this effort contained around 1250 datasets[1]

Finally, there exist many knowledge graphs specialized to a specific domain, which mostly useful for question answering. For instance, Bio2RDF is a comprehensive repository of bioinformatics data. It can be used to investigate questions such as how different genes interact in the face of a disease like Parkinson (Belleau et al., 2008).

Temporal aspect Most knowledge graphs are atemporal (also called static) or only partially temporal, i.e. they only contain temporal information for a subset of the facts contained in it. This is to be expected, since a general purpose knowledge graph has no use for temporal information in many cases. For instance, the *parentOf* relation generally either holds between two people or it does not hold between them, regardless of time. A knowledge graph containing both temporal and atemporal information is called a *hybrid* graph.

Additionally, event based knowledge bases are constructed through automatic analysis of news articles. From these articles, timestamped records are extracted and the temporal information is thus complete. However, these data sets generally require some post-processing to be transformed into graph format. Examples of event KGs are the Integrated Crisis Early Warning System (ICEWS) created by Boschee et al. (2015) and the Global Database of Events, Language and Tone (GDELT), created by Leetaru and Schrodt (2013).

[1]https://lod-cloud.net/#about (accessed on 2020/09/16).

2.3. KG Completion

In the last section, we discussed some examples of knowledge graphs and also noted how large they are. Yet, despite containing millions of nodes and sometimes billions of edges, much information is still missing. To some degree this is expected; capturing all information in the world is impossible. But also much basic information is missing. For instance, over 70% of the in persons in Freebase are missing place of birth information, despite this being a required field (Nickel et al., 2015).

Yet at the same time, KGs contain many redundancies and patterns. Consider for instance those discussed in section 2.1, there exist also higher order patterns such as autocorrelation and block structure. Autocorrelation is the tendency for entities which have similar properties to be related to each other. Block structure is the property that entities can be grouped into blocks in such a way that each entity in the block has similar relations as every other member in the block.

Exploiting these patterns and redundancies to automatically infer new facts is called *knowledge graph completion* (KGC) or *link prediction*. Specifically, we are tasked with predicting (the probability of) edges (links) between nodes in a graph. In the context of social networks like Facebook, this could be predicting which people (nodes) are likely to become friends (have an edge between them). In knowledge graphs, link prediction is more involved because we are modelling multi-dimensional data. For example, we wish to predict not simply whether there exists a relationship between two people but also what that relationship is: are they dating or are they just friends?

In link prediction, a model is presented with a triple where one component is missing and asked to present the most likely entity or predicate to be inserted into the missing component such that the result is a true triple. In entity prediction, a triple of the form $(?, p, o)$ or $(s, p, ?)$ is given. In predicate prediction, a triple of the form $(s, ?, o)$ is given.

A TKG alters the setting for link prediction as facts now contain also time information. The model can now be given an (s, p, o, h) quadruple or (s, p, o, b, e) quintuple where one component is missing. If the entity or predicate component is missing, the model should predict the most likely answer at the given time (interval). If a h, b or e is missing, the model should predict the most likely timestamp. The latter is called *temporal (scope) prediction.*

2.3.1. Approaches

Roughly speaking, a KGC algorithm has two requirements. Firstly, it must be linear in both space and time complexity (with regards to entities, predicates, triples) in order to deal with the sizes of current KGs. Secondly, it must be expressive enough to capture and exploit the patterns in a KG.

Given the above constraints, KGC research has focussed on two areas, *pattern mining approaches* and *embedding models*, with the latter being the most popular. In either approach, a model is learned on the KG, which can be used to predict the plausibility of any given triple. From this, the most likely triples can then be selected and accepted into the KG. Furthermore, the model can also be used during KG construction as a form of bootstrapping: given a triple extracted from a (possibly unreliable) source, its likelihood can be evaluated given the current knowledge in the KG before the fact is accepted.

Pattern mining approaches operate by mining patterns from the graph and converting these into logical rules. For instance, we could imagine a finding a rule of the form $\textit{MarriedTo}(X, Y) \wedge \textit{livesIn}(Y, Z) \implies \textit{livesIn}(X, Z)$, which captures the assumption that married people live together. The approach is based on the idea that every observed triple is an application of a specific rule, which can then be generalized to encompass multiple observations.

Embedding models convert a knowledge graph into a low-dimensional vector space through learning a vector representation for each entity and predicate. The vector representations should model the latent features of the entities and predicates, i.e. the features that explain the entities and predicates. KG embedding can therefore be considered a form of representation learning. In this thesis, we will focus on this approach.

2.3.2. Knowledge Graph Embeddings

A knowledge graph embedding is the transformation of a KG into a continuous, low-dimensional vector representation. The idea is that the semantics of the KG are encoded into a simpler representation, and that the vectors represent the *latent features* of entities and relationships. A good embedding model is able to capture all earlier predicate patterns discussed earlier.

A high level overview of a KG embedding model is displayed in Figure 1. Taking a knowledge graph as input and a random initialization of the vectors, a vector representation of the KG is gradually learned (calibrated) using a scoring function $\phi(s, p, o)$. The scoring function should reflect how well the embedding captures the semantics of the KG, i.e. a poor embedding should receive a low score and a good embedding should receive a high score

This training process eventually outputs a vector representation of the original knowledge graph, which can then be utilized for downstream applications. A number of different scoring functions have been proposed both for both static and temporal KG embeddings. We will discuss a number of these in section 3.

We will denote vectors in **boldface**. Formally, a KG embedding is a set of vectors, where each vector represent the embedding of a specific entity or predicate: $\{\mathbf{e}^k \ \forall e \in \mathcal{E}, \mathbf{r}^k \ \forall r \in \mathcal{P}\}$, where k denotes the dimensionality or size of the embedding. The larger k, the more expressive the model can be. However, this comes at the cost of increased training time and the chance of overfitting. The chosen value for k thus requires careful consideration. Additionally, given a vector $\mathbf{v}$, $\mathbf{v}_j$ refers to the j'th element of that

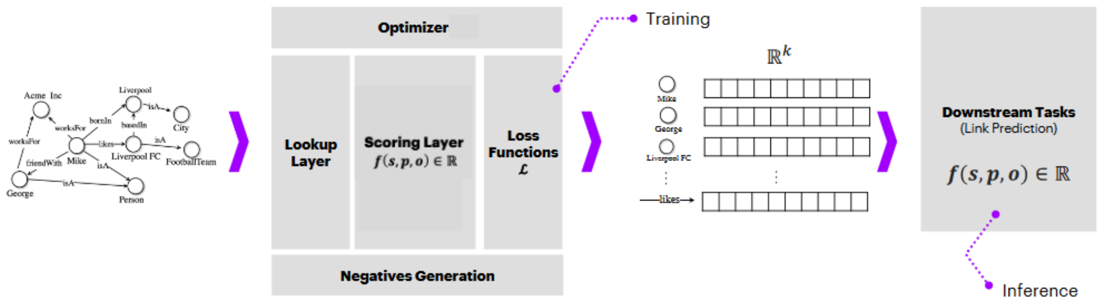


Figure 1: Anatomy of a KG embedding mode (Costabello et al., 2020).

vector. When discussing the embedding of a specific triple, we will use $\mathbf{s}, \mathbf{p}, \mathbf{o}$ to refer to the embeddings of the subject, predicate and object respectively.

Updating the vectors is done using (stochastic) gradient descent. In gradient descent, the derivative of a loss function is calculated with respect to each parameter. Embeddings are updated proportional to the negative of the difference between two steps. In practice, calculating the loss over the entire dataset is too computationally expensive. To combat this, stochastic gradient descent (SGD) is used instead. Under SGD, a fixed number of samples is taken (a batch) a a time, and the gradient is calculated only on that batch instead of on the entire dataset. Once the entire training set has been processed, an epoch has passed.

2.4. Model Evaluation

2.4.1. Method

A KG embedding can be evaluated through its performance on the link prediction task. One method to evaluate how well an embedding performs at link prediction is through the ranking test laid out in Bordes et al. (2011). For a single triple (s, p, o), the tasks composes of removing one of its elements, resulting in either $(?, p, o)$, $(s, ?, o)$ or $(s, p, ?)$. The missing element is then substituted with every entity or predicate resulting in a set of corrupted triplets, e.g. $\{(e, p, o) | e \in \mathcal{E}\}$. Each of these triples is then scored according the model and sorted. The rank of the original triple is recorded. This is repeated for every triple in the test set.

From these ranks multiple metrics are computed. Let R denote the ranks obtained for each triple in the test set. The mean rank (MR) is calculated as $\frac{1}{|R|} \sum_{r \in R} r$. While useful, MR is susceptible to outliers. To combat this, we also calculate the mean reciprocal rank (MRR) as $\frac{1}{|R|} \sum_{r \in R} \frac{1}{r}$. Lastly, an additional useful concept is the *Hits@x* metric, which is the fraction of queries where the correct answer was in the top x, where usually

$x \in \{1, 3, 10\}$. Hits@x is calculated as $\frac{1}{|R|}\sum_{r\in R}\begin{cases}1 \text{ if } r \leq x\\ 0 \text{ otherwise}\end{cases}$.

However, note that these metrics can be flawed in cases where other variations of the triple that is being ranked are also true. I.e. when creating corruptions, we may create a true triple. Come evaluation time, this triple might score higher than the original triple, lowering the score of embedding. To combat this problem Bordes et al. (2013) propose removing all triples that are contained in the train, validation, or test set from the corrupted triples. This is called the *filtered* setting.

2.4.2. Datasets

To help with the turnaround time associated with evaluating a new KGC method, both pattern mining approaches and embedding models are not tested on full-sized KGs. Instead, a number of benchmark KGs have been produced. In this section, we will provide an overview of commonly used benchmark knowledge graphs for both static and temporal KG completion.

Static For static model evaluation, the most commonly used datasets are WordNet18 and Freebase15k. WordNet18 is a subset of WordNet containing 18 relations, Freebase15k is a subset of Freebase containing 15,000 entities. Both were extracted by Bordes et al. (2013). Recently, both have been criticized as poor benchmarks datasets because many of the test triples can be obtained by simply reversing examples from the training set. I.e. they suffer from test leakage. This problem was first identified in Toutanova and Chen (2015).

To show that this is a problem, Dettmers et al. (2017) introduce a simple rule based model which obtained very high test scores on FB15k. Working towards solving this issue, they introduce two new KGs: Freebase-237 and WordNet18-RR, which have the reciprocal relations removed and thus not suffer from these issues. Furthermore, Akrami et al. (2018) evaluate an extensive set of KG embedding models on both the original and fixed data sets. Their results show a significant decrease in performance on all metrics, for all models. As a result, Freebase-237 and WordNet18-RR are becoming the new default benchmark datasets for static embedding models.

Temporal Evaluation of temporal models requires temporal knowledge graphs. We previously noted in section 2.2 that most general purpose KGs are *hybrid*. However, as we will see in section 3.3, most TKG embedding models can only embed fully temporal graphs. Therefore, subsets of YAGO3 and Wikidata have been created which contain only temporal facts. However, different papers have extracted different subsets suitable to their likings, making it difficult to compare results.

Another approach is to operate on event based graphs, i.e. those containing (s,p,o,h) quadruples. Examples of Event KGs discussed previously were ICEWS and GDELT.

From the former, García-Durán et al. (2018) have extracted two subsets. ICEWS14 which contains all events in the year 2014 and ICEWS05-15 which contains all events between 2005 and 2015. Furthermore, Trivedi et al. (2017) have created a subset of GDELT containing events between April 1st 2015 and March 31st 2016 at a granularity of 15 minutes.

2.5. Loss functions

The method described in the previous section is a *ranking measure*, used to evaluate the performance between different KGE models. However, in addition to not being differentiable and thus not being apply to apply gradient descent, it requires comparing each triple with all of its possible permutations. As a consequence it is too computationally expensive to perform at every step during fitting. Therefore, during the learning phase a loss function is utilized. In KG literature, generally two types of loss functions are applied: pointwise loss (e.g Nickel et al. (2011); Trouillon et al. (2016)) and pairwise loss functions (e.g. Bordes et al. (2013), (Mohamed et al., 2019)). Both require a set of positive examples and a set of negative examples, which can be obtained through the procedures laid out in section 2.6.

In pointwise loss functions, the objective is to minimize the scores of negative samples, and maximize the scores of positive samples individually. That is, each sample is scored by itself and only compared to the output from the true labelling function l which is generally defined to return *-1* if the triple is negative, and *1* if it is true. An example of a pointwise loss function called the pointwise hinge loss is displayed in equation 1. Note that here, the scoring function ϕ must output a value in the range $[-1, 1]$.

$$\mathcal{L} = \sum_{x \in \mathcal{D}} max(0, \gamma - l(x) * \phi(x)) \tag{1}$$

In pairwise loss functions, the objective is to maximize the difference between scores of positive samples and negative samples. An example of a pairwise loss function is the pairwise hinge loss function given in equation 2. Here, γ denotes the margin hyperparameter. In the context of knowledge graphs, this is equivalent to maximizing the difference between the score of a true triple and a false triple. An explanation of how false triples can be obtained is given in section 2.6.

$$\mathcal{L} = \sum_{x^+ \in \mathcal{D}^+} \sum_{x^- \in \mathcal{D}^-} max(0, \gamma + \phi(x^+) - \phi(x^-)) \tag{2}$$

Apart from having a positive impact regarding computational complexity, Trouillon et al. (2016) have shown that using the negative log-likelihood of logistic model will have a positive impact on the performance of most KGE models. This can be applied to both hinge and pairwise loss functions.

2.6. Sampling

Let $\mathcal{D}^+$denote the observed (positive) triples in the KG and let $\mathcal{D}^-$ denote all false triples. The assumption that a triple not in the KG is always false is called the closed world assumption (CWA) i.e. $\mathcal{D}^- = \{(s,p,o) \notin \mathcal{D}^+\}$. Due to the sparsity of a KG, this set will be very large. Furthermore, many triples in this set will be nonsensical. The opposite to the CWA is the open world assumptions (OWA), which states that the fact can either missing or false, but that we do not know which.

A more useful middle ground is the local-closed world assumption (LCWA). This states that the KG is locally complete, meaning that if we observe an (s,p) pair in the KG then any $(s,p,o) \notin D$ is false, i.e. the CWA. However, if (s,p) is not observed in the KG, then we take the OWA. Specifically, we can generate $\mathcal{D}^-$ under the LCWA by perturbing positive triples to generate corrupted triples. Corrupted triples are created by taking a positive triple, modifying one of its components to obtain (s',p,o), (s,p',o) or (s,p,o'), and checking whether the result is not already in $\mathcal{D}^+$. By only modifying one component at a time, the result is much more likely to be sensical triple (Nickel et al., 2015).

Another way to increase the likelihood of obtaining a sensical triple is through the use of self-adversarial negative sampling, introduced by Sun et al. (2019). Here, negative samples are generated according to the current model. Given a triple (s,p,o), the probability of sampling the j'th negative triple (s_j,p,o_j) can be seen in equation 3. Here, $\alpha \in [0,1]$ is the sampling temperature and ϕ is the model specific scoring function.

$$p((s'_j,p,o'_j)|\{(s_i,p,o_i)\}) = \frac{\exp \alpha\phi(\mathbf{s}_j,\mathbf{p},\mathbf{o}_j)}{\exp \sum_i \alpha\phi(\mathbf{s}_i,\mathbf{p},\mathbf{o}_i)} \tag{3}$$

When generating negative samples from a TKG the temporal aspect needs to be taken into account also. Let T_t^+ and T_t^- denote the set of triples that are valid and invalid at time t respectively. Dasgupta et al. (2018) propose two sampling methods: time agnostic negative sampling (TANS) and time dependent negative sampling (TDNS).

In TANS the temporal aspect is simply ignored, and T_t^- is constructed using the same procedure as in a static KG. That is, the triple may not occur in the data set. Formally, negative samples for the subject entity are $\{(s',p,o,h)|s' \in \mathcal{E}, (s',p,o,h) \notin T^+\}$. Negative object samples are defined analogously, and T_t^- is the union of both.

With TDNS, extra negative samples are added by selecting samples which are present in the KG, but not at that timestamp, e.g. for negative samples of the subject entity: $\{(s',p,o,h)|s' \in \mathcal{E},\ (s',p,o) \in \mathcal{D}, (s',p,o,h) \notin T_t^+\}$. Again, T_t^- can be created by taking the union of the corruptions of both subject and object entities.

2.7. Entity Evolution and Concept Drift

In machine learning, concept drift refers to a change in properties of the target variable which the model is trying to predict. It hints at hidden variables which influence the prediction accuracy of a model (Gama et al., 2014). For example, because the meaning of words change over time, a sentiment analyser trained now would not be able to provide accurate results in the future. A specific instance of concept drift, *entity evolution* refers to the changing of an entity over time. For example, a portable music player would have been called a Walkman in the eighties but an iPod in the zeroes (additionally, the concept may have now have been subsumed entirely by mobile phones). It is not just the name of the entity that can change; any of its attributes can. Entity evolution is a major obstacle to the automatic construction of knowledge graphs and by extension its applications.

2.7.1. Record Linkage

Given a set of records, each of which describing attributes of a singular entity, the process of matching which records refer to which entities is called record linkage. E.g. figuring out that '*Donald Trump*' and '*President of the USA*' are the same person, because both records indicate that they are male, life in the White House and have five children.

Formally, record linkage is the defined as follows. Given a set of entities $\mathcal{E}$ and a set of attributes $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$. $\mathcal{R}$ is a set of records, each of the form $(x_1, x_2, \dots, x_n, t)$ with $x_i, i \in [1, n]$ denoting the value of attribute a_i, and t denoting the records timestamp. Traditionally, each attribute is associated with a proximity function $sim_a(r_1, r_2)$ which outputs the proximity of attribute a between records $r_1, r_2 \in \mathcal{R}$ in the range $[0, 1]$.

Li et al. (2011) were the first to utilize the temporal aspect of records in record linkage. In order to capture the effect of time on entity evolution they introduce the concept of time decay. Consider two records with associated attributes at two points in time. The attribute values can agree or disagree with each other, pointing to the records referring to the same entity or not. However, the strength of that (dis)agreement decays with time: the larger the gap between the time steps, the weaker the (dis)proximity should be. The probability that an entity changes attribute value a within Δt timesteps is denoted as $d_{=}(a, \Delta t)$. The probability that two entities share the same value for attribute a within Δt timesteps is denoted $d_{\neq}(a, \Delta t)$.

$$w_a(s, \Delta t) = 1 - s \cdot d_{=}(a, \Delta t) - (1 - s) \cdot d_{\neq}(a, \Delta t) \tag{4}$$

The decay functions are used in the weighting function seen in equation 4, which returns the weight that should be associated with a given attribute with its calculated proximity. Here, $0 \leq s \leq 1$ is the proximity between the two values. The combination of the weighing function and proximity function to evaluate the total proximity of two records can be seen in equation 5. Here, sim_a refers to the attribute specific proximity function and t_1 and t_2 refer to the timestamps associated with r_1 and r_2 respectively.

$$sim(r_1, r_2) = \frac{\sum_{a \in \mathcal{A}} w_a(sim_a(r_1, r_2), \mathrm{abs}(t_2 - t_1)) * sim_a(r_1, r_2))}{\sum_{a \in \mathcal{A}} w_a(sim_a(r_1, r_2), \mathrm{abs}(t_2 - t_1))} \tag{5}$$

The (dis)agreement decay functions need to be specified for every attribute and time span length. Li et al. (2011) note two ways of doing so. The first method is to simply let a domain expert set the decay rates. The second method is learning them from a labeled dataset. The learning algorithm proposed operates under the CWA and requires that each attribute only has a singular value at any point in time.

If these assumptions hold, one can calculate the life span of each value of an attribute: the number of timesteps between its appearance and its disappearance. If the value does not disappear (because it does not occur in any future records), it is called a partial life span. Otherwise it is called a full life span. The sets containing all these life spans are denoted $\bar{L}_p$ and $\bar{L}_f$. Additionally, for any two entities we can calculate the length of the time period between when they had the same value: the span distance. If two entities do not share a value, that span distance is set to infinity. The set of all span distances is denoted as $\bar{L}$.

The disagreement and agreement decay are calculated from the life spans and the span distances functions. The formulas can be seen in equations 6 and 7 respectively.

$$d_{\neq}(a, \Delta t) = \frac{|\{l \in \bar{L}_f | l \leq \Delta t\}|}{|\bar{L}_f| + |\{l \in \bar{L}_p | l \geq \Delta t\}|} \quad (6)$$

$$d_{=}(a, \Delta t) = \frac{|\{l \in \bar{L} | l \leq \Delta t\}|}{|\bar{L}|} \quad (7)$$

2.7.2. Evolution Summaries

A formal concept is a pair (A, B) where A is a set of objects and B is a set of properties, such that all objects in A share all attributes in B and B consists of all attributes shared by objects in A. A and B are also called the extent and the intent respectively. In the context of KGs, we take A to be the set of entities $\mathcal{E}$ and B as the combination of predicate and object. E.g. a property could be "Lives in, New York". Formal concepts are useful because they represent a lattice, which when visualized may provide insights into the data.

Given multiple revisions of a knowledge graph, Tasnim et al. (2019) propose a way to create evolution summaries: a compact representation of all the object and data properties that an entity was connected to over time. This enables the modelling of temporal evolution without explicit temporal scoping of facts. The first step is to create RDF molecules from every revision. An RDF molecule for entity e is the subgraph containing all triples with e as the subject entity. All molecules representing the same entity M_e are then combined into an $|M_e| \times |N|$ matrix, where N is the set of unique object or data properties in M_e, i.e. the object-predicate combinations. Each element in the matrix represents whether the property is present in the molecule.

This matrix can then be input into a formal concept analyser. The output will be a set of formal concept pairs, each combination of molecules with its shared set of properties. These properties are properties that have remained the same over the different revisions of the knowledge graph. From this output, a temporal summary graph is created by first selecting for all molecules the properties that are unique to that molecule, i.e., they do not occur in another formal concept.

2.7.3. Change Point Detection

Change point are (abrupt) variations in time series data which may represent transitions that occur between states (Aminikhanghahi and Cook, 2017). Change point detection (CPD) is the problem of finding change points in time series data. E.g., given a time series where each data point measures the height of the individual, the beginning and end of puberty could be change points.

Broadly speaking, CPD algorithms can be divided into two categories: online CPD and offline CPD. Online CPD regards analyzing a time series of unknown (possibly infinite) length, and the decision on whether a change point has occurred is made every time a new data point arrives. In contrast, offline CPD considers the entire data set at once, and can thus look back in time to see when a change occurred. Here, we will focus on offline CPD as it is the most relevant to this thesis.

Another division of CPD methods concerns the parametric versus nonparametric assumption. Parametric methods rely on assumptions about the underlying data distribution and its parameters. For instance, that it is a normal distribution with a specific mean and standard deviation. Nonparametric methods use less such assumptions, or none at all (Garreau, 2017).

Formally, CPD considers a time series S of length n, i.e. $S = \{x_0, x_1, \ldots, x_n\}$, where x_i is a d-dimensional data vector designating the value at timestamp i. At every timestamp $0 \leq i \leq n$, S can be separated into two sets, one containing all data before i and one containing all after i, i.e. $S_{0..i}$ and $S_{i..n}$. Note that there exists 2^{n-1} different ways to segment the data, making this a very difficult problem on large series.

Change point detection can be considered a hypothesis testing problem where the null hypothesis is that the samples in $S_{0..i}$ and $S_{i..n}$ are from the same distribution, which can be evaluated using a statistical test. Rejection of the null hypothesis implies a change point (Dries and Rückert, 2009). Furthermore, Truong et al. (2020) define CPD as a model selection problem which consists of selecting the best possible segmentation of S s.t. an associated cost function is minimized.

In the latter definition, a CPD algorithm consist of three components: a cost function, a search method and a constraint (on the number of change points). The cost function is a measure of homogeneity, i.e. how similar data points in a given segment are. The search method concerns locating possible segment boundaries and comes in both optimal and approximate forms. If the number of change points is not known a priori, a complexity penalty is used instead.

Examples of (simple) cost functions are the $l1$ and $l2$ norms. However, these can only successfully capture the proximity between data points if the data actually represents a structure. If this is not the case, a kernel function has to be used instead. An explanation of kernel functions is outside the scope of this thesis. Instead we refer to section 1.4 in Garreau (2017) for an introduction to kernel methods with regards to change point detection.

In our experiments, we use the bottom-up search function. This procedure starts with many change points and then deletes those that are less significant. This method has two parameters. The first is *minimum size*, which specifies the minimum length of a segment. The second is *jump*, which considers how many samples apart the initial change point possibilities are distributed.

2.8. Network Proximity

Network proximity measures capture the proximity between a pair of nodes in a graph. To do this, two sources of information can be used: the attributes of the nodes, and the structure of the graph itself.

Drawing from the many existing network proximity measures, Liben-Nowell and Kleinberg (2007) apply several to the problem of link prediction in homogenous graphs (e.g. social networks). This is done by calculating the network proximity scores for (a selection of) pairs of nodes in the graph. The pairs with the highest scores are the most similar and are intuitively the most likely to form a new link. We will now briefly cover a selection of network proximity measures from the aforementioned paper, that are relevant to this thesis based on their performance and complexity.

The methods we will discuss are based on *node-neighborhoods*. The node-neighborhood of a node x, denoted $\Gamma(x)$, is the set of nodes that are direct neighbors of x. The idea is that two nodes are more likely to form a link if they have many overlapping neighbors. Hence, the well known Jaccard coefficient between two nodes can be written as

$$score(x, y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|} \tag{8}$$

A slightly more advanced model was introduced in Adamic and Adar (2003) in the context of identifying social networks from personal homepages and mailing list subscriptions. The idea is that common features (nodes) should be weighted less heavily than uncommon ones. The measure is referred to as *Adamic/Adar* after their creators and is defined as

$$score(x, y) = \sum_{z \,\in\, \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|} \tag{9}$$

The last approach based on node-neighborhoods that we discuss is *preferential attachment*. It is based on the notion that the probability that a node x obtains a new edge is proportional to its current number of edges, i.e. $|\Gamma(x)|$. The probability that a new edge is created between two nodes is equal to the product of the number of edges between them, leading to the scoring measure

$$score(x, y) = |\Gamma(x)| * |\Gamma(y)| \tag{10}$$

3. Overview of Knowledge Graph Embedding Models

Roughly speaking, KG embedding approaches using latent features can be defined in two groups, *translational* approaches and *tensor-factorization* approaches. Models of the first type apply meaning to latent representation, namely that entities that are similar must be close together according to some distance measure. Models of the second type do not apply any meaning to the embedding themselves, but try to capture the underlying interactions directly. We will now give an overview of several important KG embedding models, including an overview of some temporal KG embedding models.

3.1. Translational models

First introduced by Bordes et al. (2013), and perhaps the type of model with the easiest intuition, translational, or distance models represent predicates as translations in the vector space. Intuitively, entities that are similar should be close together, and dissimilar ones should be far apart. Phrased in the context of vectors, combining subject and predicate should result in the object vector. For example, the vectors representing entities *Donald Trump* and *Barack Obama* should both be approximately equal to the vector representing the entity *USA* after applying the predicate vector *PresidentOf* as a transformation.

3.1.1. TransE

As stated earlier, the first distance based model was introduced by Bordes et al. (2013) under the name TransE (*Translating Embeddings*). This model learns unique representations for all entities (**s** and **o**), and for all predicates (**p**) in the same space. Furthermore, the entity vectors must at most unit length according to the L2 norm. If this were not the case, the loss could simply be minimized by setting all entity vectors to high values, maximizing the distance between corrupted and true triples. There are no constraints on the predicate vectors. The scoring function for TransE is displayed in equation 11.

$$\phi(s,p,o) = ||\mathbf{s} + \mathbf{p} - \mathbf{o}||_{1,2} \tag{11}$$

The largest benefit of TransE is it low number of parameters, namely $\Theta(nk + mk)$, with k denoting the length of the embedding vector, i.e. the number of parameters scales linearly with the number of entities and predicates in the database, a property which is required in order for a model to be able to scale to the massive sizes of existing KB's. Furthermore, the approach works well for one-to-one and asymmetric relations (e.g. *parentOf*). However, it performs poorly in one-to-many or many-to-many relationships (e.g. *bornIn*, *playsFor*), and cannot model symmetric relationships at all (e.g. *spouse*).

To understand why TransE fails under symmetric predicates, consider that both ($\mathbf{s}+\mathbf{p} \approx \mathbf{o}$) and ($\mathbf{o}+\mathbf{p} \approx \mathbf{s}$), therefore, $\mathbf{p}$ must be zero, as any other value would increase the loss. To see why TransE cannot model one-to-many or many-to-many relationships, note that

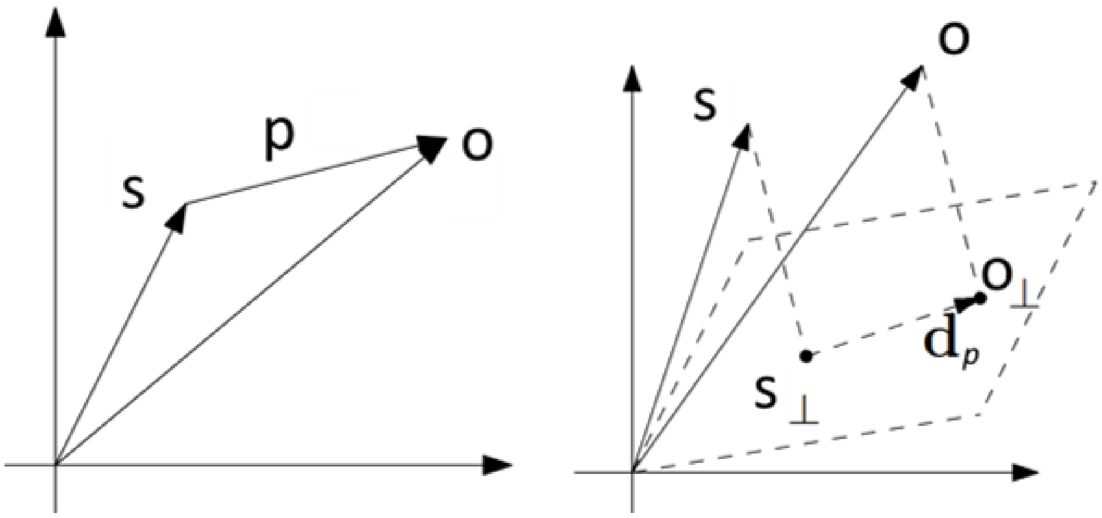


Figure 2: Visual comparison between the scoring functions of TransE (left) and TransH (right). Adapted from (Wang et al., 2014).

a set of entities which share a predicate to the same object, will also receive the same embedding, even though they are not the same entities. For example, all people who live in New York should have an equivalent embedding.

3.1.2. TransH

To extend TransE to work on more types of predicates, Wang et al. (2014) introduce TransH. TransH allows entities to have a distributed representation. That is, the representation differs depending on the predicate in which the entity is involved. This is achieved by characterizing each predicate with an additional normal vector which defines a hyperplane. Thus, every predicate embedding consists of a pair of vectors: $(\mathbf{r}, \mathbf{w}_r)$. $\mathbf{r}$ represents the standard predicate vector, and $||\mathbf{w}_r||_2 = 1$ $\mathbf{r}$ represents the hyperplane normal vector. Recall that $|\mathcal{E}| = n$ and $|\mathcal{P}| = m$, the total number of parameters for TransH is thus $\Theta(nk + 2mk)$.

Distance is calculated by first projecting the entities to the predicate specific hyperplane before adding the predicate vector. This can be done using the dot product. The full scoring function can be seen in equation in equation 12. Additionally, to ensure that the translation vector is in the hyperplane, each $\mathbf{r}$ and $\mathbf{w}_r$ pair must be orthogonal to each other. This is ensured by restricting the dot product between the hyperplane normal and the predicate to be roughly equal to the length of the translation vector, i.e.,: $\frac{\mathbf{w}_r \bullet \mathbf{r}}{||\mathbf{r}||_2} \leq \epsilon$.

$$\phi(s, p, o) = ||(\mathbf{s} - (\mathbf{w}_p \bullet \mathbf{s})\mathbf{w}_p) + \mathbf{p} - (\mathbf{o} - (\mathbf{w}_p \bullet \mathbf{o})\mathbf{w}_p)||_{1,2} \tag{12}$$

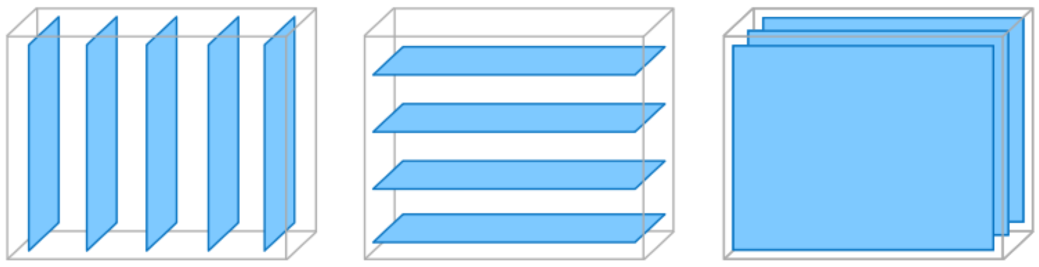


Figure 3: Examples of three dimensional tensors. From left to right, we can take lateral, horizontal or frontal slices of the tensor. Rabanser et al. (2017).

3.2. Tensor factorization

We will start this section with a short overview of tensor definitions required for the this thesis. For a more complete introduction to tensors, we refer to Rabanser et al. (2017).

A tensor is an extension of a matrix to an arbitrary number of dimensions. In the same way that one can stack multiple (row) vectors on top of each other to produce a two-dimensional object (a matrix), one could stack multiple matrices on top of each other in order to create a three-dimensional object; a tensor. Tensors can be thought of as multidimensional arrays. The number of dimensions is also called the order, way, or mode of the tensor: a first-order tensor is a vector, a second-order tensor is a matrix.

The outer product between two vectors a, b is defined as $ab^T = a \circledcirc b$ and produces a matrix. This definition can be extended to the tensor outer product between n vectors, producing an n-dimensional tensor. In the case of three dimensions, i.e. $a \circledcirc b \circledcirc c$, one could imagine each slice of the tensor as being obtained through scaling the $a \circledcirc b$ matrix through the corresponding index in c.

An n-dimensional tensor is defined to be a rank-1 tensor if it can be decomposed into the outer product of n vectors. Building from this, the rank of tensor $\mathcal{X}$ is defined to be the smallest number of rank-1 tensors that sum up to $\mathcal{X}$. Unlike the matrix rank which is well understood and can be efficiently calculated, tensor rank is poorly understood. Calculating the rank of a tensor is known to be NP-hard even for a three-dimensional tensor (Hillar and Lim, 2013).

A KG can be modelled as third-order tensor containing every possible triple, i.e. $\mathcal{D} = \mathcal{E} \times \mathcal{P} \times \mathcal{E}$. Any possible triple can be modelled as a binary random value: $y_{spo} \in (1, -1)$ where 1 indicates a true triple and -1 indicates a false triple. However, in actuality the KG is only partially observed, and we only store positive facts. As a result not every triple has a known truth value. Therefore, we let $\mathbf{Y} \in \mathbb{R}^{n \times m \times n}$ denote the partially observed knowledge graph, also called the adjacency tensor.

We can use this new formulation to rephrase our embedding task. Rather than applying an intuition to the embeddings and enforcing them as constraints like in translation based models, we instead view the problem as attempting to learn a conditional probability distribution from $\mathbf{Y}$ that generalizes to the unseen triples. That is, we want to produce some scoring tensor $\mathbf{X} \in \mathbb{R}^{n \times n \times m}$, whose scores can be converted into the probability of a triple being true (e.g. through the logistic function $\frac{1}{1+e^{-x}}.$), i.e. $P(\mathbf{Y}_{spo} = 1) = \sigma(X_{spo})$. Or, phrased in the context of a scoring function: $P(\mathbf{Y}_{spo} = 1) = \sigma(\phi(s, p, o))$.

The idea is that $\mathbf{X}$ can be modelled as the result of the interactions between the latent features of the entities and predicates: given a compositional function $\phi(s, p, o)$ and embedding vectors, we should be able to produce any value in $\mathbf{Y}$. Learning $\mathbf{X}$ is done through tensor factorization: the partially observed tensor is decomposed into a product of *factor matrices* with smaller, fixed dimensions, resulting in more compact representation of the original tensor. Tensor factorization models differ the number of components used and how they are composed.

In our case, factor matrices represent the entity and predicate embeddings. Since both determining the rank of a 3-way tensor and determining the best rank-1 approximation of a tensor are proven NP-hard, the factorization is generally an approximation of the true tensor. That is, a dimensionality is chosen as the number of latent features to model, and the tensor is approximated with those elements.

3.2.1. Polyadic Decomposition

A well known tensor factorization method is polyadic decomposition (PD) introduced by Hitchcock (1927). PD can be considered an extension of matrix rank decomposition to tensors, as such, PD decomposes a tensor as the finite sum of a series of rank-1 tensors. I.e. it decomposes an n-dimensional tensor as the finite sum of a series of outer products between n different vectors. If the number of rank-1 tensors is equal to the rank of the tensor, the decomposition is called *canonical*.

The formalization of the objective function for a PD of a three-way tensor $\mathcal{X}$ can be seen in equation 13. Here, R indicates the number of of rank-1 matrices used in the approximation, which is equivalent to the length of the embedding vector k. This can be thought of as the number of latent features used to model the entities and predicates. a_r, b_r, c_r denote the vectors making up the rank-1 tensor with index r.

$$min||\mathcal{X} - \sum_{r \in R} a_r \circledcirc b_r \circledcirc c_r|| \tag{13}$$

Applying polyadic decomposition to knowledge graphs has a large downside. Seeing as it decomposes an n-way tensor into a combination of outer products between N vectors, our 3-way KG tensor is decomposed into 3 factor matrices. One factor matrix contains the predicate embeddings and the other two contain the entity embeddings. That means that each entity has two embeddings: one for its occurrences as a subject and one for its occurrences as an object. As a result, an entity's subject representation is completely independent of its object representation.

To understand why this is problematic, consider the following two facts: (*The Dark Knight (Movie), leading actor, Christian Bale*), (*John, likes, The Dark Knight (Movie)*). In the first fact *The Dark Knight (Movie)* occurs as a subject and in the latter it occurs as an object. That means that observing the first fact will only update the subject representation of the movie and by extension thus will not affect the score and plausibility of the second fact. Yet, we would expect that this would happen, because if we were to observe that John likes Christian Bale, we would find it more plausible that John will like the movie.

3.2.2. RESCAL & DistMult

Among the first to apply tensor factorization to the problem of KG embeddings, Nickel et al. (2011) introduce RESCAL. RESCAL is based on the tucker decomposition which decomposes a tensor into a matrix for every dimensional, along with a single core-tensor of the same dimension Tucker (1966). In the naive case, applying the Tucker decomposition would result in a core tensor along with three component matrices. In addition to having the same problem with independent representation that PD has, this naive approach would also require more parameters due to the inclusion of a core tensor.

RESCAL solves these issues by applying two constraints. Firstly, one factor matrix is restricted to be the identity matrix, meaning it does not affect the computation. Secondly, the two matrices representing the entity embeddings are restricted to be equal to each other. Thus, RESCAL actually decomposes a tensor into two components. The first is an entity matrix $E \in \mathbb{R}^{n \times k}$ that contains the latent representations of all entities. The second is a predicate tensor $P \in \mathbb{R}^{m \times k \times k}$ where each slice is a matrix modelling the interactions between entities for that predicate. Each of these slices is factorized as $\mathbf{X}_p \approx EP_rE^T \; \forall r \in \mathcal{P}$. RESCAL thus has $\theta(nk + mk^2)$ parameters.

$$\phi(s,p,o) = \mathbf{s}^T R_p \mathbf{o} = \sum_{a=1}^{K} \sum_{b=1}^{K} R_{pab} * s_a * o_b \tag{14}$$

The scoring function for RESCAL is displayed in equation 14. Note that in order to compute the score of a single triple under RESCAL requires that each predicate slice of the predicate tensor is multiplied with two vectors of length k from the entity matrix, which puts the time complexity at $\mathcal{O}(k^2)$. Consequently, computing the score of all triples would imply a time complexity of $\mathcal{O}(nmk^2)$. However, the authors provide an efficient updating procedure based on alternating least squares, that allows for updating P_r independently from the number of entities and lets m only occur as a linear factor. The result is that the algorithm scales linearly with the size of the embedding, the time complexity of RESCAL is thus $\mathcal{O}(k^2)$.

Due to its large number of parameters (i.e., quadratic to the dimensionality of the embedding and the number of predicates), RESCAL is prone to overfitting to the available data. To alleviate this problem, Yang et al. (2015) introduce DistMult. DistMult is

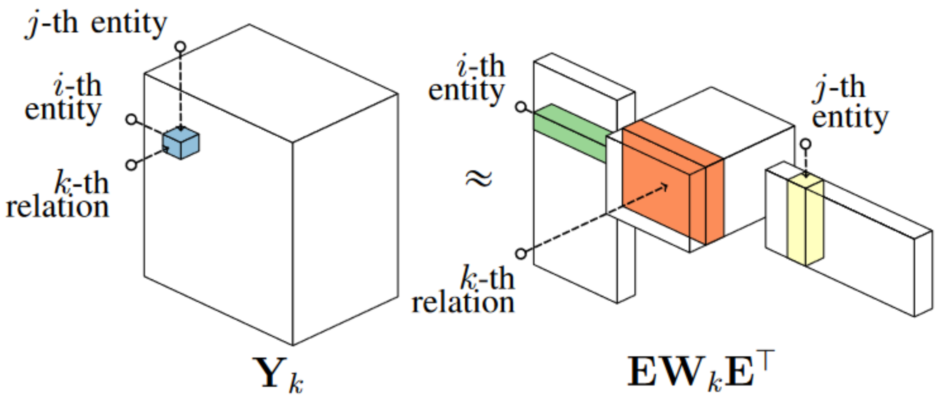


Figure 4: Illustration of the tensor decomposition applied by RESCAL. Nickel et al. (2015).

a special case of RESCAL where every predicate matrix is restricted to be a diagonal matrix, meaning each predicate can be modelled as just a single vector. As a result, DistMult requires just $\theta(nk + mk)$ parameters, which is equal to other models such as TransE.

Specifically, under DistMult each slice $\mathbf{X}_k$ of the tensor is decomposed as $\mathbf{X}_k = EP_rE^T$, where P_r is the diagonal matrix representing the embedding of predicate r, and E is the matrix containing the entity embeddings. The corresponding scoring function is displayed in equation 15. A factorization of this form is also called the *eigendecomposition* or *spectral decomposition*. If this decomposition exists, the matrix is said to be *diagonalizable*. However, it is known that a matrix is diagonalizable if and only if it is symmetric. As a consequence, the decomposition only exists when $\mathbf{X}_k$ is symmetric, meaning that DistMult can only properly model symmetric relations. Nevertheless, DistMult has proven to be a quite effective embedding model.

$$\phi(s,p,o) = \mathbf{s}^T P_p \mathbf{o} = \sum_{i=1}^{k} \mathbf{s}_i * \mathbf{p}_i * \mathbf{o}_i \tag{15}$$

3.2.3. ComplEx

A quick review of complex numbers: a complex number consist of a real and imaginary part, i.e., it can be expressed in the form $a + bi$ where a and b are real and i is imaginary. The *complex conjugate* of a complex number is the number with the same real part, but with the sign of the imaginary component inverted, e.g., the complex conjugate of $2+5i$ is $2-5i$. The complex conjugate of $z \in \mathbb{C}$ is denoted $\bar{z}$.

The dot product or scalar product between two real vectors of equal length $\mathbf{a}, \mathbf{b} \in \mathbb{R}^k$ is defined as $\sum_{i=1}^{k} \mathbf{a}_i * \mathbf{b}_i$ and produces a scalar. Under this definition, the square root of the dot product between a vector by itself produces the length of the vector. However, if the same definition was applied to complex vectors this property would not hold. Instead, the dot product between a complex vector with itself would be an arbitrary complex number and could even be the zero vector. To salvage this property, the dot product between two complex numbers $\mathbf{c}, \mathbf{d} \in \mathbb{C}^k$ is defined as $\sum_{i=1}^{k} \mathbf{a}_i * \overline{\mathbf{b}_i}$. As a result of this definition, the dot product of two complex vectors is not commutative: $a \bullet b \neq b \bullet a$.

Trouillon et al. (2016) note that while the representation of an entity should be equal regardless of whether it is used as an object or subject, its behavior should not be. Therefore, they suggest combining complex vectors with a dot product based scoring function. Now the representation is the same, but the behavior depends on whether the entity is used as object or subject. This way, the dot product is a very effective composition function.

Their model, called ComplEx, factorizes $\mathbf{X}$ into two components just like RESCAL and DistMult. The first component is an entity matrix $E \in \mathbb{C}^{n \times k}$ which contains the entity embeddings. The second component is a predicate tensor $R \in \mathbb{C}^{m \times k \times k}$, where each slice r is a *diagonal* matrix $R_r \in \mathbb{C}^{k \times k}$ which models the behavior of a specific predicate. Just like in DistMult, these are restricted to be diagonal matrices. Therefore, ComplEx only has $\theta(mk + nk)$ parameters, i.e., its number of parameters grows linearly with the number of entities and predicates.

The scoring function for ComplEx is displayed in equation 16. Here Re returns only the real part of the complex number.

$$\phi(s, p, o) = Re(\sum_{i=1}^{k} \mathbf{p}_i * \mathbf{s}_i * \overline{\mathbf{o}_i}) \tag{16}$$

3.2.4. SimplE

Recall that the major downside to polyadic decomposition is that it learns two independent embedding vectors for each entity even though they should be shared. Noting this, Kazemi and Poole (2018) introduce an enhancement to PD which allows dependent learning of the two embeddings. They name their model SimplE after the simplicity of the change.

SimplE considers two embeddings for each entity $e \in \mathcal{E}$: one for its occurrences as subject $\mathbf{e}_s \in \mathbb{R}^k$ and one for its occurrences as object $\mathbf{e}_o \in \mathbb{R}^k$. This is the same as under polyadic decomposition. However, they also consider two embeddings for each predicate, i.e. $\{(\mathbf{r}, \mathbf{r}_i) \mid r \in \mathcal{P}\}$. The latter can be thought of as the inverse predicate. The score of a triple (s, p, o) is then calculated by taking the average of its score and its inverse score, i.e. the score obtained by taking the alternative embedding and predicate vectors. The resulting scoring function can be seen in equation 17. Here, x^{-1} refers to the inverse embedding of entity or predicate x.

$$\phi(s,p,o) = \frac{1}{2}\sum_{i=1}^{k}(\mathbf{s}_i * \mathbf{p}_i * \mathbf{o}_i) + (\mathbf{o}_i^{-1} * \mathbf{p}_i^{-1} * \mathbf{s}_i^{-1}) \tag{17}$$

3.3. Temporal KG embeddings

The embeddings that we discussed previously only work on static knowledge graphs and cannot utilize temporal information of facts. Yet, it is plausible that embeddings of a TKG that do take the temporal scope into account produce better embeddings than those that do not. In this section, we will discuss a selection temporal knowledge graph embedding models. Some of these are *model agnostic*, meaning that they operate in conjunction with any static KG embedding model. Additionally, some are *hybrid* embedding models, meaning that they can embed both temporal and atemporal (static) graphs. This is an important feature, as most TKGs contain many atemporal facts.

3.3.1. t-TransE and variants

Noting that traditional KB embedding models such as TransE often confuse predicates such as *BornIn* and *DiedIn*, Jiang et al. (2016) introduce a time-aware embedding approach which takes advantage of the temporal ordering of facts: *BornIn* must occur before *DiedIn*. The underlying assumption is that through a temporal transition, a predicate vector at time $t_1 \in \mathcal{T}$ evolves into a successive predicate vector at time $t_2 \in \mathcal{T}$ (where $t_1 < t_2$). E.g., given embeddings of two temporal facts sharing the same subject: (s, p_i, o_m, t_1), (s, p_j, o_n, t_2), and a temporal order transformation matrix $M \in \mathbb{R}^{m \times m}$, the two predicates should be similar after applying the transformation matrix: $\mathbf{p}_i M \approx \mathbf{p}_j$. As a result, the transformation matrix can be scored as

$$g(\mathbf{p}_i, \mathbf{p}_j) = ||\mathbf{p}_i M - \mathbf{p}_j||_{1,2} \tag{18}$$

This approach is model agnostic and can thus be added to all existing KG embedding approaches by minimizing the joint score of existing KG embedding approach and the temporal order score function. The relative importance of the static and temporal scoring functions can be controlled with the hyperparameter λ.

In order to generate the required samples for training M, a set of positive and negative predicate pairs is required for each entity. Given a pair $(r_i, r_j) \mid r_i, r_j \in \mathcal{P}$ for entity e, it is positive if (e, r_i) occurs before (e, r_j) does, and negative otherwise. More formally, let Ω_e^+ be the set of positive predicate pairs for entity e. Then it can be constructed as $\Omega_e^+ = \{(r_i, r_j) | (s, r_i, o, h_1) \in T^{s=e}, (s, r_j, o, h_2) \in T^{s=e}, h_1 < h_2\}$. The corresponding set of negative predicate pairs Ω_e^- can be obtained by reversing the order of each pair in Ω_e^+.

3.3.2. Temporal-Aware Sequence Encoders

García-Durán et al. (2018) introduce a recurrent neural network architecture to learn time aware representations of entities and predicates. Like t-TransE, their method is model agnostic, meaning that it can be used in conjunction with traditional KG embedding methods. Their method is referred to as TA-*model*, depending on which scoring function is used (e.g. TA-TransE). Additionally, their method represent facts with their original granularity.

Specifically, temporal facts are modelled by extending a triple with either the temporal predicate *'occursSince'* or *'occursUntil'* and a timestamp. This means that for closed time intervals (intervals that have a defined start and end date), two facts are modelled. One fact signifying the start time and one fact signifying the end time. Therefore, knowledge graphs using the more commonly used (s, p, o, b, e) format will have to be pre-processed before this model can be used.

The timestamp is constructed as a sequence of characters combined with a suffix $\in \{y, m, d\}$ indicating whether the character refers to year, month or day information respectively. An example conversion of several triples can be seen in Table 2. The combination of temporal predicate and timestamp combination is called a *temporal token*.

Fact	**Temporal token**
(Obama, born, US, 1961)	[born, 1y, 9y, 6y,1y]
(Obama, president, USA, since, 2009)	[president, since, 2y, 0y, 0y, 9y, 01m]

Table 2: Facts and their corresponding predicate sequence. Adapted from García-Durán et al. (2018).

Each token is then mapped to its corresponding embedding and used as input for the recurrent neural network. The resulting predicate embedding is obtained from the final layer of the network. The output of that final layer can then be used as input for any scoring function. The parameters for the scoring function (i.e. subject and object embedding) are learned jointly with the parameters of the network generating the predicate embeddings using stochastic gradient descent.

3.3.3. HyTE

The next TKG embedding approach that we will discuss is *Hyperplane-based Temporally aware knowledge graph Embedding* (HyTE) (Dasgupta et al., 2018). HyTE takes inspiration from the TransH model discussed earlier. Instead of projecting embeddings on a predicate-specific hyperplane during scoring, it projects them on a timestamp-specific hyperplane. This idea is grounded in the observation that the main source of different one-to-many, many-to-one or many-to-many predicate pairs are based in different points in time, e.g., someone who moves to New York after living in Berlin.

HyTE considers a TKG as series of static KGs, where each static KG represents a singular point in time containing only the set of facts that were valid at that time. I.e. $T = \bigcup_{t \in \mathcal{T}} \mathcal{D}_t$. To this end, any fact that is true over an interval of time is deconstructed into a fact for every year it is true and included in the relevant subgraph. Therefore, one must take care that the time granularity is not set too high which could result in very sparse subgraphs or imbalances between the sizes of the subgraphs. As a countermeasure, the authors use an adaptive granularity so that a minimum number of triples is included in every subgraph.

Unlike t-TransE, in HyTE the temporal scope is directly included in the embedding rather than being enforced through constraints. This results in a distributed representation where the embeddings differs depending on the time. Therefore, HyTE requires a different distance function that takes into account the timestamp at which the fact occurred. Like TransH, it computes the dot ($\bullet$) product between the embedding and the hyperplane normal, noted **h**. The distance function can be seen in equation 19.

$$\phi(s, p, o, h) = ||(\mathbf{s} - (\mathbf{h} \bullet \mathbf{s})\mathbf{h}) + (\mathbf{p} - (\mathbf{h} \bullet \mathbf{p})\mathbf{h}) + (\mathbf{o} - (\mathbf{h} \bullet \mathbf{o})\mathbf{h})||_{1,2} \tag{19}$$

Calculating the loss for a specific embedding is done using by minimizing the margin based ranking loss. It is therefore required to iterate over all time steps. The result can be seen in equation 20. Here, negative samples are obtained through TDNS or TANS, as explained in section 2.6.

$$\mathcal{L} = \sum_{t \in \mathcal{T}} \sum_{x^+ \in \mathcal{D}_t^+} \sum_{x^- \in \mathcal{D}_t^-} \gamma + \phi(x^+) - \phi(x^-) \tag{20}$$

3.3.4. Diachronic Embeddings

The concept of Diachronic Entity Embeddings (DEE), introduced in Goel et al. (2019) combines the distributed representation present in HyTE and combinatory capabilities of t-TransE. That is, the temporal aspects are included inside the embedding vectors and the method can be combined with any existing KG-embedding. This is achieved by applying a time-dependent function to each embedding. The underlying intuition is that some features of an entity may differ over time, whereas other features remain static.

A diachronic entity embedding (DEE) function is a function which maps every (entity, timestamp) pair to its representation. Noting that the choice of DEE function can differ based on the properties of the TKG which is being embedded, an example provided by Goel et al. (2019) can be seen in equation 21. Here **z** represents the DEE of length k of a specific entity at a specific time. Given an entity-specific embedding $a \in \mathbb{R}^d$, its first γk indices are transformed using a sigmoid function, where $0 \leq \gamma \leq 1$ is a hyperparameter, and two additional weight vectors $\mathbf{w}, \mathbf{b}_e \in \mathbb{R}^{\gamma k}$.

Furthermore, the authors advocate for using the sine function in favour of a classic sigmoid function such as the logistic function, as the periodicity of the sine function allows for multiple on and off states.

$$z_e^t[n] = \begin{cases} \mathbf{a}_e[n]\sigma(\mathbf{w}_e[n]t + \mathbf{b}_e) & \text{if } 1 \leq n \leq \gamma k \\ \mathbf{a}_e[n] & \text{if } \gamma d < n \leq k \end{cases} \tag{21}$$

3.3.5. ATiSE

In order to take the temporal uncertainty into account that occurs during the evolution of entity and predicate representations over time, Xu et al. (2019) propose mapping their representations into the space of multi-dimensional Gaussian distributions. Here, an embedding at a specific timestamp is represented as the mean of the distribution, and its uncertainty is represented as the covariance of the distribution. Evolution of entities and predicates is modelled by using additive time series. For this reason the model is named AtiSE, short for additive time series decomposition embedding. ATiSE operates on timestamped facts (i.e. quadruples), but can model valid time facts by constructing a quadruplet for every timestamp at which the fact holds.

Additive time series decomposition decomposes a time series into three components. A trend component, a seasonal component and an irregular component. In ATiSE, the first is modelled as a linear combination of the embedding vector and an evolution component The seasonal component is modelled as a sine function and the irregular component is modelled as gaussian noise.

Scoring a triple is done through Kullback-Leibler (KL) divergence, which measures how much one probability distribution differs from another distribution (Kullback and Leibler, 1951). The score of a triple is the KL divergence between the entity distribution and the predicate distribution, i.e. the subject and object distribution are first combined. Therefore, ATiSE can be considered a translational model. The resulting equation can be seen in equation 22, here $\delta_{\mathcal{KL}}$ refers to the KL divergence of the two distributions. For a full expansion of the scoring function we refer to Xu et al. (2019).

$$\phi(s,p,o,h) = \delta_{\mathcal{KL}}(P_{p,h},(P_{s,t} - P_{o,h})) \tag{22}$$

4. Method

Noting that the results obtained by many TKG embedding models are only marginally better than those obtained using static KG embedding models, we have created a new method called SpliMe. The name is short for *SPL*it and *ME*rge, its two main approaches. SpliMe leverages the valid time of facts to embed time inside entities or predicates themselves. It is important to note that the output of SpliMe is not a static KG. Instead, it is a temporal knowledge graph denoted in triples. SpliMe is model agnostic, meaning it can be used with any of the existing static embedding models. Additionally, it is capable of embedding both static and temporal facts.

At the core of SpliMe is a transformation function $f : T \mapsto T'$ which transforms a given TKG into another TKG that incorporates time at the level of entities or predicates. This function is applied repeatedly until a stopping criteria c_s is met. The simplest implementation of f would be to create a new predicate for every unique (*predicate, valid time*) combination in the TKG. However, due to the large domain of possible intervals, this is generally not feasible as the resulting KG would be too sparse.

To counter this, we have created four implementations of f, namely *timestamping*, *splitting*, *merging* and *proximity*. We will now present an explanation of each method applied to predicates, the versions for entities can be defined analogously. Additionally, while the implementations assume a valid time KG, we note that any event KG can be trivially expanded into a valid time KG by setting $b = e = h$ for every fact.

4.1. Timestamping

Conceptually the simplest approach, timestamping converts each temporal fact in the KG into a set of facts, one for each timestamp for which the fact was true. A variant of this approach was previously defined in Leblay and Chekol (2018) under the name *Naive-TTransE*. Formally, given an injective function $u(\mathcal{P}, \mathcal{T}) \mapsto \mathcal{P}'$ which creates a new predicate for every *(predicate, timestamp)* combination, timestamping creates a set of facts for a given quintuple, i.e. $f(s, p, o, b, e) = \{(s, u(p, t), o) \mid \forall t \in \mathcal{T} \; s.t. \; b \leq t \leq e\}$. To obtain a fully timestamped KG, this process must be applied to every fact in the KG. An example of timestamping can be seen in Figure 5.

(Trump, president, USA, 2017, 2020)

$\downarrow$

(Trump, president[2017], USA)
(Trump, president[2018], USA)
(Trump, president[2019], USA)
(Trump, president[2020], USA)

Figure 5: Illustration of how a fact is converted into a set of facts when timestamping.

Pseudocode for the timestamping method can be seen in Algorithm 1. We start off by initializing an empty predicate set and temporal knowledge graph in lines 4 and 5.

Algorithm 1 : SpliMe: timestamping strategy

```
function TIMESTAMP($T$, $\mathcal{P}$ $u$)
input: TKG, predicate lookup function
output: TKG, new predicate set
    $T' \leftarrow \{\}$
    $\mathcal{P}' \leftarrow \{\}$
    for all $(s, p, o, b, e) \in T$ do
        for $t = b$; $t \leq e$ ; $t$++ do
            r $\leftarrow u(p, t)$          ▷ Lookup/create new a predicate for this (predicate, timestamp) combination
            $\mathcal{P}' \leftarrow \mathcal{P}' \cup r$
            $T' \leftarrow T' \cup (s, r, o, t)$
        end for
    end for
return $T', \mathcal{P}'$
end function
```

Then, for every quintuple in the knowledge graph, we iterate over all all timestamps in its valid time (line 6). For each such timestamp we create and add a new predicate (to the predicate set), and then add the modified fact to the TKG (lines 9, 10).

Note that the timestamp of the fact is also included in the resultant TKG. I.e., a set of quadruples is returned instead of the expected set of triples. This is done because timestamping is also used as a pre-processing step for the *merging* approach discussed in section 4.3. However, because the temporal information is already included in the predicates, the time element can simple be removed when one wishes to train a timestamped dataset.

4.2. Splitting

Given a predicate $r \in \mathcal{P}$ and a timestamp $t \in \mathcal{T}$, splitting adds two new predicates to the predicate set: $\mathcal{P}' = \{r_1, r_2\} \cup \mathcal{P}$. All $(s, p, o, b, e) \in T^{p=r}$ are then updated to contain either r_1 if $e \leq t$, or r_2 if $b \geq t$. Otherwise, the fact is split into two: $\{(s, r_1, o, b, t), (s, r_2, o, t, e)\}$. An example of splitting a TKG containing two facts can be seen in Figure 6. This figure also shows that splitting does not necessarily increase the size of the data set. Instead, splitting reduces the number of facts associated with each predicate.

More formally, the splitting function can be defined as $f : (T, c_p, c_t) \mapsto T'$, which given a TKG T, splitting criteria c_p which selects a predicate, and criteria c_t which selects a timestamp, maps to a new TKG T'. The challenge here lies in defining an effective c_p and c_t. For c_p, a general approach that produced a good performance in our experimental evaluation is selecting the predicate that occurs the most, i.e. $c_p = \max_{r \in \mathcal{P}} |T^{p=r}|$. Intuitively, this is the predicate that benefits the most by making it more specific.

(Obama, president, USA, 2009, 2017)
(Trump, president, USA, 2017, 2020)
$\downarrow$
(Obama, president[2009-2017], USA)
(Trump, president[2017-2020], USA)

Figure 6: Illustration of the splitting process on a small TKG containing two facts using the *count* criteria. Splitting on the year 2017 results in an equal number of facts on both sides of the splits. If the *time* criteria had been applied, the split year would have been 2014.

For c_t, we propose two methods: *count* and *split*. Let t_f^r and t_l^r denote the first and last timestamps associated with predicate r in the TKG. The *time method* splits at $t = (t_f^r + t_l^r)/2$. The *count method* selects for predicate r the point in time that results in the most balanced number of triples on both sides of the split. This is done by counting how many facts end before, or end after a specific timestamp. Formally, the count function can be written as $c_t(r) = \arg\min_{t \in \mathcal{T}} abs(|T^{p=r,e\leq t}| - |T^{p=r,b\geq t}|)$ where $T^{p=r,e\leq t} \neq \emptyset$ and $T^{p=r,b\geq t} \neq \emptyset$ in order to prevent the consideration of non-existing (*predicate, timestamp*) combinations. We have performed experiments with both methods and found that both achieve similar results. Which one performs best depends on the dataset used.

For the stopping condition c_s, we allow a user selected grow factor $g \geq 1$ and stop when the number of predicates has become more than g times the original number of predicates. Specifically, let m and m' denote the number of predicates original KG and the modified KG respectively, then we continue until $c_s(g, m, m') = g * m \geq m'$. In our experimental evaluation we used grow factors between 5 and 30. If lower values are used not enough new predicates are added, for higher values there may not be enough room to split on.

Pseudocode for the merging approach is given in Algorithm 2. Continuing until the stop condition is met, each iteration of the algorithm starts by finding the most common predicate in the TKG (line 5) and selecting a timestamp based on the either the time or count method (line 6). On line 7, two new predicates are created: the first represent the predicate until the split timestamp t, the second represents it from that point forward. Then for every fact in the TKG which contains the original predicate, we remove that fact in favour of the new 'split' facts. Which predicate (r_1, r_2) is used depends on the valid time of the quintuple. Finally, we remove the original predicate from the predicate set (line 19).

4.3. Merging

Merging can be considered as the opposite of splitting. The method starts with a timestamped TKG (consisting of (s, p, o, h) quadruples) and selectively merges back predicates which belong to the same original predicate and are subsequent in time. This reduces the number of unique predicates in the TKG. That is, given a predicate pair (r_1, r_2) to merge, a new predicate r_c is generated, and every $(s, p, o, h) \in \{T^{p=r_1} \cup T^{p=r_2}\}$ is

Algorithm 2 : SpliMe: splitting strategy

```
function Split($T$, $\mathcal{P}$, $c_t$)
input: TKG, Predicate set, time selection condition
output: TKG, Predicate set
    while not *stop condition met* do
        r ← $\arg\max_{r \in \mathcal{P}|T^{p=r}|}$
        t ← $c_t(|T^{p=r}|)$                          ▷ Using either *Time* or *Count* method
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{r_1, r_2\}$
        for all $(s, p, o, b, e) \in T^{p=r}$ do
            $T \leftarrow T \setminus \{(s, p, o, b, e)\}$          ▷ Remove the original fact from the TKG
            if $b \leq t \leq e$ then                    ▷ If fact spans the split time
                $T \leftarrow T \cup (s, r_1, o, b, t)$
                $T \leftarrow T \cup (s, r_2, o, t, e)$
            else if $e \leq t$ then                      ▷ Fact ends before split
                $T \leftarrow T \cup (s, r_1, o, b, e)$
            else                                         ▷ Fact begins after split
                $T \leftarrow T \cup (s, r_2, o, b, e)$
            end if
        end for
        $\mathcal{P} \leftarrow \mathcal{P} \setminus \{r\}$                    ▷ Remove the original predicate
    end while
return $T, \mathcal{P}$
end function
```

(Macron, president[2017], France)
(Macron, president[2018], France)
(Trump, president[2017], USA)
(Trump, president[2018], USA)
$\downarrow$
(Macron, president[2017-2018], France)
(Trump, president[2017-2018], USA)

Figure 7: Illustration of how a small timestamped TKG containing four facts is merged.

then updated to contain r_c, lowering the number of unique predicates. Like splitting, the difficulty lies in defining a good predicate selection criterion c_p, and a good time selection criterion c_t.

To this end, we define function $l_p(\mathcal{P}') \mapsto \mathcal{P}$ which given a predicate of the timestamped TKG, returns the source predicate in the original TKG and a function $l_t(\mathcal{P}') \mapsto \mathcal{T}$ which given a predicate of the timestamped TKG, returns the time associated with that predicate. Recall from section 4.1 that $u(\mathcal{P}, \mathcal{T}) \mapsto \mathcal{P}'$ returns a unique (new) predicate for every (*predicate, timestamp*) pair. l_p and l_t are defined s.t. for predicate $r \in \mathcal{P}'$, $u(l_p(r), l_t(r)) = r$.

A predicate pair (r_1, r_2) is only valid as a merge candidate if I) the source predicates are the same: $l_p(r_1) = l_p(r_2)$, and II) there exists no predicate whose timestamp is in between those associated with r_1 and r_2. More formally, $\nexists r_c \in \mathcal{P}'\ s.t.\ l_t(r_1) \leq l_t(r_c) \leq l_t(r_2)$.

Based on the intuition that we want to increase the number of examples for every predicate, we select the (r_1, r_2) pair that occurs the least in the TKG, i.e. $(r_1, r_2) = \arg\min_{r_1, r_2 \in \mathcal{P}'} |T^{p=r_1} \cup T^{p=r_2}|$. Naturally, the conditions described above must also hold.

The stopping criterion c_s is defined using a shrink factor $s \geq 1$. Recall that timestamping introduces a new predicate for every (*predicate, timestamp*) combination. Let m, m', m'' denote the number of predicates in the original KG, the *extra* predicates in the timestamped KG and the *current* number of predicates in the merged KG respectively. Then, merging continues until $c_s(s, m, m', m'') = m'' \leq p + m' - (m' - \frac{m'}{s})$ returns true. The higher the value for s, the smaller the resulting number of predicates. In our experiments, we have generally investigated values of s between 1 and 10. At higher values, there are often not enough predicates left to merge.

Pseudocode for the merging strategy can be seen in Algorithm 3. The first step is to apply the SpliMe timestamping method (line 4). The following procedure is performed until the stop condition is met. On line 6 we generate all possible merge options according to the constraints laid out above. From this, we select the pair that occurs the least (line 8). Next, we find all facts in the TKG containing either predicate selected to merge, and iterate over them on line 9. Each of these we remove from the TKG (line 10) and then re-add it (line 11) with the predicate replaced for the merged predicate we created on line 7. Once we have iterated over all predicates we exit the loop. Then we update the predicate set by adding the new, and removing the old predicate (lines 13, 14)

Algorithm 3 : SpliMe: merging strategy

```
1: function MERGE($T$, $\mathcal{P}$)
2: input: TKG, Predicate set
3: output: TKG, Predicate set
4:     $T', \mathcal{P}' \leftarrow$ TIMESTAMP($T$)    ▷ Apply timestamping procedure in Algorithm 1
5:     while not stop condition met do
6:         $\mathcal{O} \leftarrow$ all possible merge options
7:         $(r_1, r_2) \leftarrow \arg\min_{(r_1,r_2)\in\mathcal{O}} |T'^{\,p=r_1} \cup T'^{\,p=r_2}|$    ▷ Select the least occuring pair
8:         $r_n \leftarrow$ generate new predicate
9:         for all $(s,p,o,h) \in T'^{\,p=r_1} \cup T'^{\,p=r_2}$ do
10:            $T' \leftarrow T' \setminus \{(s,p,o,h)\}$
11:            $T' \leftarrow T' \cup \{(s,r_n,o,h)\}$    ▷ Replace all occurrences of the predicates
12:        end for
13:        $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{r_n\}$    ▷ Add the new predicate
14:        $\mathcal{P}' \leftarrow \mathcal{P}' \setminus \{r_1, r_2\}$    ▷ Remove the old predicates
15:    end while
16: return $T', \mathcal{P}'$
17: end function
```

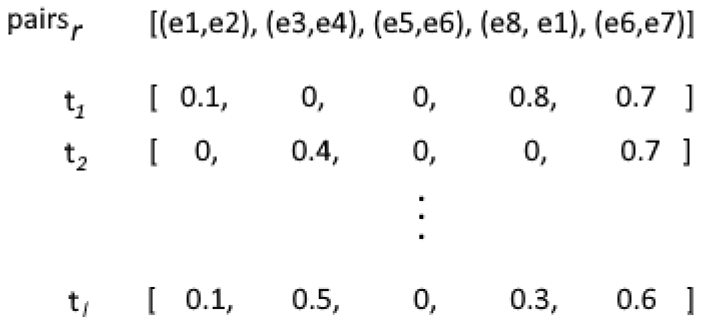
4.4. Proximity

Our final approach utilizes CPD (section 2.7.3) to determine the predicate and timesteps to split at. However, CPD cannot be immediately applied to knowledge graphs as it operates exclusively on numeric values. Therefore, we combine this approach with network proximity measures (section 2.8): the proximity measures output a ”signature” vector for every predicate, at every timestamp, to which we then apply change point detection.

We will firstly discuss the creation of the signature vectors. Given the constraints of the CPD, these must contain numeric values, and for a single predicate the signatures at every timestamp must be of equal length. Inspired by the format used in evolution summaries (section 2.7.2), each entry in the vector represents the proximity score of a specific entity pair, where the order of that entity pair does not matter. Consequently, the KG is considered an undirected graph in this scenario.

Specifically, signatures are created first slicing the KG into subgraphs representing the different (predicate, timestep) combination, such that each subgraph contains only the facts containing the given predicate, and which are true at that timestep. Then, for every predicate we create a list of entity pairs which exist in any of that predicates subgraphs, and calculate their proximity scores on those subgraphs. If the entity pair does not exist in a given subgraph, its score is 0. This proximity score is added to each timesteps signature vector.

More formally, we create a subgraph $s_{r,t} = T^{p=r, b\leq t\leq e}$ for every (*predicate, timestamp*) combination. Let $f((e_1, e_2), T) \mapsto \mathbb{R}$ denote a proximity function (e.g. pref-



$pairs_r$	[(e1,e2),	(e3,e4),	(e5,e6),	(e8, e1),	(e6,e7)]
t_1	[0.1,	0,	0,	0.8,	0.7]
t_2	[0,	0.4,	0,	0,	0.7]
			⋮		
t_l	[0.1,	0.5,	0,	0.3,	0.6]

Figure 8: Illustration of an entity pair vector and the associated proximity score vectors for a single predicate. A proximity vector is created for each timestamp.

erential attachment) that calculates the proximity score for a given entity pair on the given subgraph. Let $pairs_r$ denote the set of entity pairs for predicate r, i.e. $pairs_r = \{(e_1, e_2) \mid e_1, e_2 \in \mathcal{E} \wedge ((e_1, p, e_2) \vee (e_2, p, e_1) \in T^{p=r})\}$. Now, for every predicate we calculate the proximity score of each pair in its pair set for every timestep and add the result to the signature vector. That is, we calculate $f(a, s_{r_t})\ \forall a \in pair_r,\ \forall t \in \mathcal{T}$ for every predicate. An illustration of what these vectors look like is given in Figure 8.

On these signature vectors we can then apply CPD to find out at which timesteps the signature, and thus the predicate changes significantly. This would be a split point. Applying this to all entities or predicates would result in a list of split points for each, which can then be applied to the original data set.

The number of change points for a given predicate is unknown. Meaning that a complexity penalty or residual value has to be used to determine the optimal number of split points. In our experimental evaluation, we utilize a bottom-up segmentation algorithm in combination with a residual ϵ, where ϵ is in the range [1, 100] depending on the data being transformed. Additionally, to ensure that the found change points do not depend on the magnitude of the signatures, we normalize all signatures before feeding them into the CPD algorithm.

4.5. Granularity and Temporal Distortion

Many TKG embedding methods that deal with valid time knowledge graphs aggregate multiple timesteps into a time interval or *bin* due to the sparsity of time information (e.g. (Dasgupta et al., 2018; Xu et al., 2019)). By requiring a minimum number of facts for each bin, the number of distinct timesteps in the KG is decreased. The effect is that the temporal information of facts is distributed more uniformly compared to the raw data. The minimum number of facts in each bin is called the *granularity* level. Unfortunately, the bin creation algorithms used in other papers are not clearly defined. Instead, only the granularity level or the number of bins is reported.

Algorithm 4 : SpliMe: binning

```
1: function Bin($T$, $g$)
2: input: TKG, granularity level
3: output: Set of bins and their first and last timesteps
4:     $t_{min} \leftarrow \min t \in T$                                ▷ First timestep in the TKG.
5:     $t_{max} \leftarrow \max t \in T$                                ▷ Last timestep in the TKG.
6:     $C \leftarrow \{\}$                                              ▷ Initialize bin set.
7:     $c \leftarrow \{min : t_{min}, max : t_{min}\}$                  ▷ Initialize first bin.
8:     while $c_{max} \leq t_{max}$ do
9:         while $(|T^{(c_{min} \leq b \leq c_{max}) \vee (c_{min} \leq e \leq c_{max})}| < g) \vee (|T^{e \geq c_{max}}| < g)$ do
10:            $c_{max} \leftarrow c_{max} + 1$                         ▷ Increase # timesteps in current bin.
11:        end while
12:        $C \leftarrow C \cup c$                                      ▷ Update bin set.
13:        $c \leftarrow \{min : c_{max}, max : c_{max}\}$              ▷ Create the next bin to be filled.
14:    end while
15: return $C$
16: end function
```

(Macron, president[2017], France)
(Macron, president[2018], France)
(Trump, president[2016], USA)
(Trump, president[2017], USA)
(Trump, president[2018], USA)
↓
(Macron, president[2016-2017], France)
(Trump, president[2016-2017], USA)
(Trump, president[2018-2018], USA)

Figure 9: Illustration of how a set of facts for the original predicate 'president' between 2016 and 2020 are merged, and how this can result in temporal distortion.

Therefore, we propose the bin creation procedure given in Algorithm 4. The algorithm starts by initializing t_{min} and t_{max} as the first and last timesteps in the TKG (lines 4, 5). C is initialized as the set of all bins, and c as the first bin (lines 6, 7). For bin c, its start and end timesteps are referred to as c_{min} and c_{max} respectively.

We continue creating bins until we have reached the final timestep in the TKG (line 8). Facts are added to a bin if their valid time overlaps with with the interval of the current bin. The interval of the bin is extended until the required number of facts have been added, or if it is impossible to create a new bin because there will not enough facts to fill it to the required granularity level. These conditions are represented by the first and last term on line 9 respectively. The bin set is then updated to contain the newly created bin (line 12), and a new bin is initialized to the timestep at which the current bin ended (line 13).

For splitting and merging, SpliMe does not require a granularity level to be applied explicitly because both perform it implicitly. In the case of splitting, the most common predicate is split into two and the number of bins is increased. In the case of merging, the least common predicates are combined and the granularity is decreased. For this reason, rather than requiring a minimum number of facts per timestep globally, SpliMe can be said to operate at the level of individual predicates.

However, both the approach used in SpliMe and the traditional approach seen in the literature result in what we call *temporal distortion*. This entails association of a fact with a timestamp at which it is not true. An example can be seen in Figure 9. In this example, Macron is noted as being the president of France in 2016. However, the timestamped KG shows that he only became president in 2017. The temporal scope of the fact *(Macron, president, France)* has been

Notably, the model proposed García-Durán et al. (2018) does not suffer from temporal distortion because it embeds facts with their original granularity. For other models, the only way to reduce the amount of distortion is by reducing the granularity level. That is, requiring less facts per bin. However, this in turn increases the sparseness of the data which may worsen the quality of the embeddings. An investigation of the effect of the granularity level on the performance of SpliMe is available in section 5.7.1.

4.6. Complexity Analysis

Because SpliMe is a pre-processing method, it has a dual effect on the running time and space requirements of knowledge graph embedding. Firstly, there is a time associated with running SpliMe which depends on the method utilized. Secondly, because SpliMe increases the number of entities or predicates by introducing more facts and reducing the number of facts associated with each element, this in turn can cause an increase in space and time used during the embedding process. As in the description of the models, we will discuss the complexity of splitting on predicates. However, that of entities is again defined analogously.

4.6.1. SpliMe

Lets first consider the time complexity of the proposed methods. Let k denote the number of facts in the original KG, and n, m, t denote the amount of entities, predicates and timesteps. The worst case for SpliMe is when every fact in the KG spans the entire time range. In this scenario the most iterations can be performed.

Timestamping In the case of timestamping, a new fact is created for every (*predicate, timestamp*) combination, which results in a time complexity of $\mathcal{O}(kt)$. For the other methods, the runtime depends on the number of merges and splits performed, which is influenced by the stopping criterion c_s, or the ϵ residual in the case of the proximity approach.

Splitting The pathological case for the splitting method also assumes that the KG originally contains only one predicate. Under this scenario the *time* and *count* methods have the same result. Every time a fact is split, it produces two facts. Thus, at first there is only one predicate and splitting it doubles the size of the KG. The second and third split then take these two predicates and split them again, doubling the size again. After x splits, the size of the TKG is thus $\mathcal{O}(k * \log_2(x+1))$.

The number of occurrences for each predicate can be calculated before the first split is applied with a single pass, and then updated at every split in constant time. The difficulty therefore lies in efficiently updating all facts. Assuming one stores all facts in an index in which inserts, deletes and lookups can be performed in (amortized) constant time (e.g. a hashmap), the time is linear in the number of facts being split. In the given scenario, this is $\mathcal{O}(k)$. Combining this, the time complexity for applying x splits is $\mathcal{O}(xk)$.

Merging Recall that merging starts of with a timestamped KG and merges back pairs of predicates. I.e., we begin with $\mathcal{O}(kt)$ facts, to which we can apply t merges, one for each (*predicate, timestamp*) combination. Let us again assume that inserts, deletes and lookups of facts can be done in constant time. Obviously, calculating the number of occurrences for each predicate pair can be done in linear time. These counts can then be stored as a sorted list ($\mathcal{O}(m \log_2(m)$ using Quicksort for example) and updated when required. The time complexity of the merging method is thus $\mathcal{O}(kt^2 + m \log_2(m))$.

Proximity Finally, we consider the proximity approach. Here the complexity depends on the complexity of the creation of the signatures, and on that of the CPD algorithm utilized. Regarding the latter, we used bottom-up segmentation in our experiments, which has a runtime complexity of $\mathcal{O}(s \log(s))$, where s is the number of signatures. This is then applied to every predicate.

Signature creation requires calculation of the node neighborhoods of every entity on every subgraph. Assuming that the data has already been processed into the form of an adjacency list (s.t. creation of each signature is equal to the number of entity pairs, $\binom{n}{2}$ in the worst case[2]). Calculating a signature for every timestep leads to a worst-case complexity of $\mathcal{O}(tn^2)$. In real-world use cases however, the number of entities associated with each predicate is generally much smaller than the complete entity set.

4.6.2. Embedding

Lastly, lets consider the effect of the increased number of entities and predicates on the static embedding methods. The maximum number of new predicates that can be created is also the number that is created in the worst case of the timestamping approach, which is when every facts spans the entire time period. This is equal to applying splits until no longer possible, or performing no merging. This would result in $\mathcal{O}(mt)$ predicates.

Assuming a model whose parameters grow linearly with respect to the number of entities and predicates, its space complexity would be $\mathcal{O}(mt{+}n)$. However, we note that generally $m, t \ll n$, especially when a minimum granularity is applied. As a result, in all normal cases applying SpliMe does not have a large effect on the number of parameters learned by embedding models, and thus by extension their runtime.

[2]Equal to $\frac{n(n-1)}{2}$, or $\mathcal{O}(n^2)$ asymptotic complexity.

5. Experiments

In order to evaluate the effectiveness of SpliMe we have implemented the aforementioned methods and executed them on several knowledge graphs commonly used for the evaluation of TKG embedding methods. In this section we will explain our methodology, list our results and draw conclusions.

All SpliMe methods were implemented in the Python programming language[3] using the NumPy[4] and Pandas[5] libraries. Additionally, we use the Ruptures[6] library for change point detection. Lastly, the source code and the data used in this thesis are available on Github[7] for reproducibility.

Unless noted otherwise, all models were trained and evaluated using the Ampligraph framework, a suite of machine learning tools used for supervised learning created by Costabello et al. (2019). This framework contains implementations for many popular embeddings such as TransE and ComplEx. All models were trained on the UU gemini computing cluster. This cluster is a shared computing environment consisting of several Intel Xeon E5-2683 CPUs running at 2.10GHz. Each CPU has 16 cores, and each machine has 256GB ram. The cluster does not have GPUs, so models were trained on the CPU instead.

5.1. Data

We will now give an overview of the knowledge graphs used in our evaluation. Additionally, making the KGs viable for our purposes required some pre-processing and cleanup. These will be noted and motivated as well. Lastly, an overview of characteristics of each KG can be seen in Table 3. For our experiment, we maintained the original train/test/validation splits.

Valid time KGs The first KG that we will use is *Wikidata*. Wikidata is the large, open knowledge graph which acts as central storage for the structured data of other Wikimedia projects such as Wikipedia (Erxleben et al., 2014). Secondly, we use *YAGO*, an open source knowledge graph created by extracting information from Wikipedia (Suchanek et al., 2007). Specifically, we use the Wikidata12k and YAGO11k subsets extracted by Dasgupta et al. (2018). Both have time granularity set to the year level.

For many facts in these KGs, either the begin or end time is missing. This is denoted with four hashtags: ####. If the begin time is missing, this is because it is unknown. If the end year is missing, this can be because the end date is unknown, or because the fact is still true. However, as in Dasgupta et al. (2018), we do not discern these cases.

[3] https://www.python.org Python documentation
[4] https://numpy.org NumPy documentation
[5] https://pandas.pydata.org Pandas documentation
[6] https://centre-borelli.github.io/ruptures-docs Ruptures documentation
[7] https://github.com/wradstok/thesis_radstok Source code and data repository

Dataset	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert train\rvert$	$\lvert valid\rvert$	$\lvert test\rvert$
YAGO11k	10,526	10	-	16,408	2,050	2,051
Wikidata12k	12,554	24	-	32,497	4,062	4,062
ICEWS14	7,128	230	365	72,826	8,941	8,963

Table 3: Overview of the characteristics of the used datasets. Number of timesteps is not noted for the valid time KGs because this depends on the applied granularity level.

In either situation we replace a missing begin time or end time with the first and last timestep in the TKG respectively. Additionally, there are facts whose temporal scope is invalid either because they cannot be parsed correctly (e.g. 19#5), or because the end time is before the start time. These facts are removed before any SpliMe methods are applied. Unless noted otherwise, valid time KGs are converted as described in section 4.5 with a granularity level of 300.

Event based KGs Additionally, we run our experiments on the Integrated Crisis Early Warning System (ICEWS) data set. ICEWS is a system designed to monitor and forecast national, sub-national and internal crises. Its data consists of timestamped political events at day granularity (Boschee et al., 2015). In our experiments, we use ICEWS14 subset created by García-Durán et al. (2018). *ICEWS14* contains all events in 2014. All timesteps are of day level granularity, meaning it has 365 timesteps. Alternatively, there exists a subset called *ICEWS05-15* which contains all events from 2005 until 2015. However, we opted not to use this one due to computing power constraints.

5.2. Hyperparameters

SpliMe is model agnostic and can thus be combined with any KG embedding method. However, in our experiments we have opted to use TransE. This is for two reasons. Firstly, TransE is the simplest model and therefore provides a good baseline to show the viability of our approach. Secondly, many TKG embedding models are built on top of the TransE family (e.g. (Dasgupta et al., 2018; Jiang et al., 2016)), and thus using TransE provides a fair comparison.

We have adapted our hyperparameters from García-Durán et al. (2018) and use the same hyperparameters for all models. Specifically, we train for 200 epochs with embedding size = 100 and learning rate = 10^{-3}. We set the batch size to 500 and generate 500 negative samples per batch. Optimization was done using the ADAM optimizer in combination with a self-adversarial loss function (Kingma and Ba, 2014). For the proximity method, we use a kernelized mean change cost function with bottom-up search for the CPD algorithm. Jump size and minimal section length are set to 1.

		Hits@10 performance		
Paper	**Model**	**Wikidata12k**	**YAGO11k**	**ICEWS14**
Dasgupta et al. (2018)	HyTE	8.5%	2.8%	-
Tang et al. (2020)	TDG2E	8.5%	2.8%	-
Xu et al. (2019)	ATiSE	33.9%	24.4%	-
García-Durán et al. (2018)	TA-TransE	-	-	63.7%
Goel et al. (2019)	DE-SimplE	-	-	63.7%
Jin et al. (2019)	RE-Net	-	-	47.1%
Ours (HyTe)	-	10.1%	2.6%	54.1%
Ours (RotatE)	-	50.4%	28.3%	70.3%
Ours (Ampligraph)	-	52.7%	35.6%	56.1%

Table 4: Comparison between the results obtained by the TransE model on the same dataset, but in different implementations. Dashes denote that the result is not available. Below the horizontal line are our experiments using three different frameworks. Note that *model* refers to the model introduced in the paper to make it easier to reference the papers contributions. It does not refer to the model tested.

5.3. Entity Prediction

For evaluation, we follow the ranking procedure laid out in Bordes et al. (2013) which is explained in detail in section 2.4. Specifically, we use the *filtered* setting. To re-iterate, for a triple in the test set (s, p, o) both the subject or object is replaced with all $e \in \mathcal{E}$ in turn. That is, subject and object evaluation is combined. In line with the *filtered* setting, we remove any resulting triples which occur in the train, test or validation sets. All triples are then scored by the model and the result is sorted. The rank of the original triple is recorded.

We report the MRR, Hits@1, Hits@3 and Hits@10. Following recent TKG embedding literature, we report the combined subject and object prediction results. In addition to our three transformation methods, we also report results on a reference version. Here, all temporal information is removed from the TKG turning it into a static KG. We refer to this as a *vanilla* model. A further explanation is given in section 5.4.

5.4. On Evaluating TKGs with Static Methods

In Table 4 we have collected the TransE results for several different data sets from a number of papers. Many other papers choose to operate on their own subset of Wikidata or YAGO and can thus not be compared. Nevertheless, we note a big discrepancy between the valid time KG results of HyTE and TDG2E and those in ATiSE. Furthermore, there exists also a large discrepancy for ICEWS14 between the RE-Net and TA-TransE papers. In order to properly compare our results to the current state of the art, we need to understand what causes these differences.

We have evaluated the performance of TransE under three different implementations:

Parameter	**HyTE**	**RotatE**	**Ampligraph**
Dimension	100	100	100
Margin	1	24	1
Neg samples	5	500	500
Norm	l_2	l_1	l_1
Batch size	50,000	512	500
Learning rate	10^{-4}	10^{-3}	10^{-3}
Loss	Pairwise	Self-adversarial	Self-adversarial
Sampling temperature	-	1.0	0.5
Initializer	Xavier	Uniform	Xavier
Steps/Epochs	500	150k	200

Table 5: Overview of the applied hyperparameters used for the three different models. RotatE uses steps instead of epochs as a stopping condition.

Data set	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
Wikidata12k	0.038	1.9%	3.6%	6.9%
YAGO11k	0.034	1.1%	3.0%	7.4%
ICEWS14	0.146	0.3%	21.3%	41.8%

Table 6: Results for the reference datasets in the Ampligraph framework. The same parameters as described in section 5.2 are applied, but with pairwise loss instead of self-adversarial loss.

HyTE, RotatE and Ampligraph. All experiments were run on the same dataset, applying only transformations to put it in the required format for the framework. Applied hyperparameters are displayed in Table 5.

From the results, we see that the HyTE framework results match closely with the ATiSE results on Wikidata12k and YAGO11k. The RotatE and Ampligraph framework experiments however perform much higher. Yet, on the ICEWS14 data set we see similar results across all frameworks, with the performance being roughly in between those obtained in García-Durán et al. (2018) and Jin et al. (2019).

We believe that this result is due to the use of self-adversarial negative sampling implemented in RotatE and Ampligraph. To evaluate this hypothesis we used Ampligraph to train TransE on all three data sets using the same hyperparameters, but with pairwise loss instead of self-adversarial loss. The results are available in Table 6 and show that both valid time KGs suffer from this change, with their results being approximately equal those seen in the original HyTE paper. However, the ICEWS14 result suffers much less. Thus, this indeed explains the observed difference.

	Wikidata12k	**YAGO11k**	**ICEWS14**
# duplicates in train (%)	4720 (14.53%)	0 (0%)	30136 (41.38%)
# duplicates in test (%)	214 (5.27%)	0 (0%)	1544 (17.23%)
# duplicates in valid (%)	193 (4.75%)	0 (0%)	1610 (18.01%)
# test triples in train (%)	1042 (27.10%)	0 (0%)	3499 (47.16%)
# valid triples in train (%)	1027 (26.54%)	0 (0%)	3527 (48.11%)

Table 7: Overview of the number of inter and intra set duplicates after stripping temporal information for each of the used datasets. Note how YAGO is unaffected.

5.4.1. Filtering

Generally, when temporal KGs are embedded using static methods in order to generate a baseline result, the temporal information is simply discarded. However, we note that doing so leads to duplicate information which might distort the evaluation result. To give an example, consider the following two facts: (*Obama, visited, the Netherlands, 2014*) and (*Obama, visited, the Netherlands, 2014*). When temporal information is stripped these both result in (*Obama, visited, the Netherlands*).

This process has two issues. Firstly, triples can be duplicated inside any given train/test/test split: a set might contain multiple examples of the exact same triple if they previously denoted different occurrences in time. If this happens in the train set, the model will get multiple instances of the triple to fit to. If it happens in the test set, the model's performance will be abnormally determined by such triples. Secondly, and of greater importance, examples may be duplicated between sets, causing test-leakage.

A full overview of the number of duplicate triples in each dataset is given in Table 7. Investigations show that YAGO11k is unaffected by both types. However, Wikidata12k and ICEWS are not, with ICEWS14 scoring the worst, having almost half of its test triples appear in the training set. This is probably due to its large number of timesteps.

To counteract this issue and show how it affects performance, we introduce two new filtering methods. *Intra-set filtering* which filters out any duplicate triples inside a given split, and *inter-set filtering* which removes any triples from the train/validation split if they occur in the test set. Next, we re-run the experiments from Table 4 but, apply both filtering approaches. These results are displayed in Table 8 and confirm our hypothesis: the model performance on the filtered data sets is much lower than on the original.

Comparing these results to those found in the papers in Table 4, it is unclear what filtering approaches are used for some papers. As we showed in our comparison with pairwise loss functions, hyperparameter tunings also have a large effect on the result. For instance, after consulting the authors, we learned that both Goel et al. (2019) and Jin et al. (2019) did not apply any filtering. Yet, their results on ICEWS14 are very different. We hope that now we have made this issue explicit, applied filtering methods will be noted along the results.

		Hits@10 performance			
Dataset	**Implementation**	**No Filter**	**Inter**	**Intra**	**Both**
Wikidata12k	HyTE	10.1%	3.8%	5.3%	3.5%
	RotatE	53.8%	36.4%	52.4%	33.9%
	Ampligraph	52.7%	37.7%	52.8%	37.9%
YAGO11k	HyTE	2.6%	-	-	2.6%
	RotatE	28.9%	-	-	28.8%
	Ampligraph	35.6%	-	-	35.6%
ICEWS14	HyTE	54.1%	25.7%	40.6%	23.7%
	RotatE	70.3%	38.4%	65.1%	37.9%
	Ampligraph	56.1%	42.2%	56.1%	42.2%

Table 8: Results on Wikidata12k and ICEWS14 using three different TransE implementations, combined with the different filter settings. YAGO11k results for intra/inter settings were not generated, because these are the same as the both/unfiltered.

5.5. Results

In order to evaluate the effectiveness of our proposed methods, we also create a random baseline, which works by applying splits randomly across predicates and timesteps in a uniform manner. Specifically, at every split step, a predicate is chosen from the currently existing predicates. Then, from the time span of that predicate (i.e., its first and last occurrence in the data set) a timestamp is chosen. This is repeated until the desired number of predicates have been created. The method is denoted as *Random* in our results. For the valid time KGs we take the average of 7 runs. For ICEWS14 we take the average between 5 runs. The per-run results are available in section D.7 in the appendix.

Because SpliMe applies transformations to the knowledge graphs, we also have hyperparameters for the data. For both the *time* and *count* splitting methods we experimented with $x \in \{5, 10, 15, 20, 25, 30\}$ for all data sets. Regarding merging, we tested a shrink factor $\in \{1.5, 2, 4, 6, 8, 10\}$. Lastly, for the *proximity* method, we tested the three proximity metrics described in section 2.8 with ϵ (residuals) $\in \{1.25, 2.5, 5, 10, 15, 20\}$ for Wikidata12k and YAGO11k, and $\epsilon \in \{5, 12.5, 25, 50, 100, 150, 200\}$ for ICEWS14. The best results are displayed in Table 9. Results of all runs are available in the appendix.

Method	Setting	MRR	Hits@1	Hits@3	Hits@10	# Preds
Vanilla	-	0.209	12.4%	22.7%	37.9%	24
Random	-	0.289	17.9%	33.3%	50.7%	423
Timestamp	-	0.340	21.1%	40.8%	58.1%	1622
Split (time)	*Grow* $= 10$	0.320	20.1%	37.2%	54.9%	240
Split (count)	*Grow* $= 25$	0.300	18.3%	34.5%	53.8%	600
Merge	*Shrink* $= 4$	**0.358**	**22.2%**	**43.3%**	**61.0%**	423
Proximity	*Pref*, $\epsilon = 2.5$	0.328	20.9%	38.1%	56.0%	726

(a) Wikidata12k results

Method	Setting	MRR	Hits@1	Hits@3	Hits@10	# Preds
Vanilla	-	0.188	8.2%	23.8%	35.6%	10
Random	-	0.197	7.8%	25.2%	39.9%	200
Timestamp	-	0.197	6.9%	26.0%	41.2%	570
Split (time)	*Grow* $= 20$	0.213	**9.0%**	27.0%	43.2%	200
Split (count)	*Grow* $= 25$	0.196	8.1%	24.1%	40.3%	250
Merge	*Shrink* $= 2$	0.195	6.2%	26.3%	42.0%	290
Proximity	*Pref*, $\epsilon = 5$	**0.214**	6.5%	**29.9%**	**45.8%**	177

(b) YAGO11k results

Method	Setting	MRR	Hits@1	Hits@3	Hits@10	# Preds
Vanilla	-	0.141	0.1%	18.9%	42.2%	230
Random	-	0.172	2.0%	23.4%	48.1%	3500
Timestamp	-	**0.213**	**4.7%**	**29.4%**	**54.4%**	17061
Split (time)	*Grow* $= 20$	0.190	3.0%	26.3%	51.6%	4600
Split (count)	*Grow* $= 25$	0.191	3.0%	26.2%	52.2%	5750
Merge	*Shrink* $= 1.5$	0.207	4.1%	28.7%	53.9%	11449
Proximity	*Adar*, $\epsilon = 25$	0.196	2.9%	27.3%	53.0%	5866

(c) ICEWS14 results

Table 9: Overview of the best results obtained for each method. All results were obtained for the *inter*-filtering setting. Best results among for each dataset all are highlighted in **bold**. For a complete overview of all runs, we refer to the appendix.

The best results displayed in Table 9 show that all methods, even random baseline, provide an increase in performance on all metrics with regards to the vanilla model. Therefore, we can say that incorporating time at the level of predicates improves the link prediction capabilities of static KG embedding models. Furthermore, all SpliMe methods have improved performance compared to the random baseline at a comparable number of predicates, suggesting that they indeed capture temporal information in a more efficient manner.

On Wikidata12k the best result is achieved using the merge method creating a version of the dataset with 423 predicates (17.6 times increase). Here, merging outperforms all other approaches on every recorded metric. It outperforms vanilla TransE and the random baseline by just over 23 and 10 percentage points at the hits@10 level respectively. This represents a 60% and 20% increase in performance respectively.

On YAGO11k the best result is achieved using change point detection in combination with preferential attachment as a proximity function. Notably, this method produces 177 predicates (17.7 times increase), which are 113 and 23 fewer than the best merge and split approaches respectively. *Proximity* outperforms the vanilla baseline by just over 10 percentage points at the hits@10 level, which is a 28% increase in performance.

Lastly, on ICEWS14 the best result is achieved with the timestamping approach, which achieves the highest scores on all metrics. Timestamping outperforms the random baseline with 6 percentage points (13% increase) at the hits@10 level. On the hits@10 metric, the difference is 2.7 percentage points, representing a 135% increase. Compared to our vanilla TransE baseline the results are even better, especially on hits@1 as the vanilla baseline scores just 0.1% here.

5.5.1. Comparison with other models

In Table 10 we have compared the best results obtained by SPLIME with the current literature. The results show that SPLIME outperforms the current state-of-the-art on the valid time datasets. Indeed, SPLIME performs 15 and 12 percentage points better than the second-best model on the hits@10 metric for Wikidata12k and YAGO11k respectively. Only on the hits@1 metric on the YAGO11k dataset it is outperformed by ATiSE. Yet, it still outperforms the other models.

The ICEWS14 results are not as good. Here, SPLIME is significantly outperformed by the other models. However, we still observe a noteworthy increase in performance compared to our vanilla baseline, which as explained in section 5.4.1 differs from those seen in most literature. Furthermore, we note that by not applying any filtering, we have previously shown vanilla (i.e. without temporal information) results comparable to the state-of-the-art in link prediction performance on this data set. Therefore, we believe SPLIME is best evaluated against itself and not its competitors in this case.

Method	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
SpliMe	**0.358**	**22.2%**	**43.3%**	**61.0%**
ATiSE*	0.252	14.8%	28.8%	46.2%
HyTE*	0.180	9.8%	19.7%	33.3%
TTransE*	0.172	9.6%	18.4%	32.9%
SEDE	-	-	-	45.8%
TDG2E	-		-	40.2%

(a) Wikidata12k results. Best SpliMe result was achieved with the *merge* method.

Method	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
SpliMe	**0.214**	6.5%	**29.9%**	**45.8%**
ATiSE*	0.185	**12.6%**	18.9%	30.1%
HyTE*	0.105	1.5%	14.3%	27.2%
TTransE*	0.108	2.0%	15.0%	25.1%
SEDE	-	-	-	30.1%
TDG2E	-	-	-	31.1%

(b) YAGO11k results. Best SpliMe result was achieved using *change point detection.*

Method	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
SpliMe	0.213	4.7%	29.4%	54.4%
ATiSE*	**0.545**	**42.3%**	**63.2%**	**75.7%**
HyTE*	0.297	10.8%	41.6%	60.1%
TA-DistMult	0.477	36.3%	-	68.6%
RE-Net	0.457	38.2%	49.1%	59.1%

(c) ICEWS14 results. Best SpliMe result was achieved with the *timestamp* method.

Table 10: Comparison of the best SpliMe approach for each data set and the current state of the art. Best results among all were highlighted in **bold**. *: results were obtained from Xu et al. (2019), other results were taken from their respective papers. SpliMe results are from using the *inter* filter setting.

5.6. Baseline

We note that through increasing the number of predicates or entities, SpliMe also increases the number of parameters in the model. In this section, we will show that the increased performance is not just due to this, but because SpliMe helps embeddings models capture more information by including temporal information at the level of entities or predicates.

To this end, we learn a baseline model with 1000 parameters per predicate which we call *vanilla-1k*. Next, we compare this to the *split (time)* and *merge* approaches which allow for 10 predicates for each original predicate. We then learn these with 100 parameters per predicate such that the total number of parameters is equal for each model. The results from this experiment are displayed in Table 11.

Dataset	Version	MRR	Hits@1	Hits@3	Hits@10	Runtime
Wikidata12k	Vanilla-1k	0.115	5.7%	12.4%	22.3%	12:26:28
	Split (time)	0.272	15.3%	32.3%	50.1%	04:22:09
	Merge	0.346	21.2%	41.9%	59.2%	06:58:48
YAGO11k	Vanilla-1k	0.188	8.2%	23.8%	35.6%	08:01:18
	Split (time)	0.199	7.8%	26.2%	40.1%	05:14:11
	Merge	0.172	6.8%	19.5%	37.2%	05:24:50
ICEWS14	Vanilla-1k	0.107	0.0%	14.5%	31.4%	16:33:45
	Split (time)	0.187	2.8%	25.7%	51.1%	03:02:13
	Merge	0.192	2.6%	26.9%	52.3%	03:03:00

Table 11: Comparison between a vanilla dataset and two SpliMe methods with the same total number of parameters for each. Both *inter* and *intra* set filtering were applied to the data sets. Runtime is given in *hh:mm:ss*.

Results show for both Wikidata12k and ICEWS14 that the *vanilla-1k* variant has overfitted to the data: the performance is worse than that observed of the standard (100 parameters/predicate) model. Yet, the SpliMe methods still perform very will. From this, we conclude that SpliMe indeed helps extract more information from the data. Additionally, we also note the much lower fitting time compared to *vanilla1k*. For instance, in the case of ICEWS14 the vanilla dataset takes over five times as long to fit.

5.7. Ablation Studies

In this section, we will perform investigations regarding several components and hyperparameters of SpliMe, and of (temporal) KG embedding models in general. This allows us to measure their effect on the effectiveness of the learned embeddings.

5.7.1. Granularity

Many TKG embedding models which operate on valid time KGs aggregate time periods such that a minimum number of facts is achieved for every timestep (e.g. Dasgupta et al. (2018); Xu et al. (2019); Chen et al. (2019); Zhou et al. (2019)). However, to the best of our knowledge the effect of different granularity levels has not been extensively studied. To this end, we perform an ablation study to investigate effect of enforcing different granularity levels when applying SpliMe. Specifically, using the merge and split (time) approaches. The results are available in tables 12a and 12b.

Note that since merging uses timestamping for pre-processing, the number of predicates in the data set depends on the applied granularity level. As a result, the lower the granularity level, the more parameters are available. Therefore, we would expect better results at lower granularity. This indeed proves to be the case. However, seeing as the split method also performs best at a low granularity level (≈ 10) even though the number

Wikidata12k	*Split (time), grow=10, inter set filtering*						
Granularity	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{P}\rvert$
0	0.334	**89**	21.5%	38.8%	56.7%	620	240
5	0.309	101	18.2%	35.1%	56.2%	334	240
10	0.333	89	**20.4%**	**39.3%**	**58.6%**	266	240
25	**0.345**	350	23.5%	38.8%	57.7%	191	240
50	0.285	120	16.1%	33.7%	54.1%	150	240
150	0.303	128	19.8%	33.3%	53.3%	98	240
300	0.272	155	15.3%	32.3%	50.1%	70	240
500	0.294	116	16.8%	35.1%	54.1%	56	240
1000	0.277	118	16.4%	32.2%	50.0%	39	240

(a) Results on the Wikidata12k dataset after applying the SpliMe *split* method using different levels of granularity. Best results are highlighted in **bold**.

Wikidata12k	*Merge, shrink=4, inter set filtering*						
Granularity	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{P}\rvert$
0	-	-	-	-	-	2002	9194
5	**0.401**	96	**27.0%**	46.7%	**64.7%**	334	1684
10	0.384	83	24.8%	45.6%	62.9%	266	1378
25	0.396	100	25.7%	**47.6%**	64.6%	191	1030
50	0.383	**82**	24.8%	45.2%	63.8%	150	838
150	0.380	114	24.1%	46.1%	62.7%	98	576
300	0.358	100	22.2%	43.3%	61.0%	70	423
500	0.345	107	21.3%	41.6%	59.1%	56	344
1000	0.322	117	20.3%	37.5%	56.1%	39	246

(b) Results on the Wikidata12k dataset after applying the SpliMe *merge* method using different levels of granularity. Results for granularity 0 have been excluded due to processing time constraints. Best results are highlighted in **bold**.

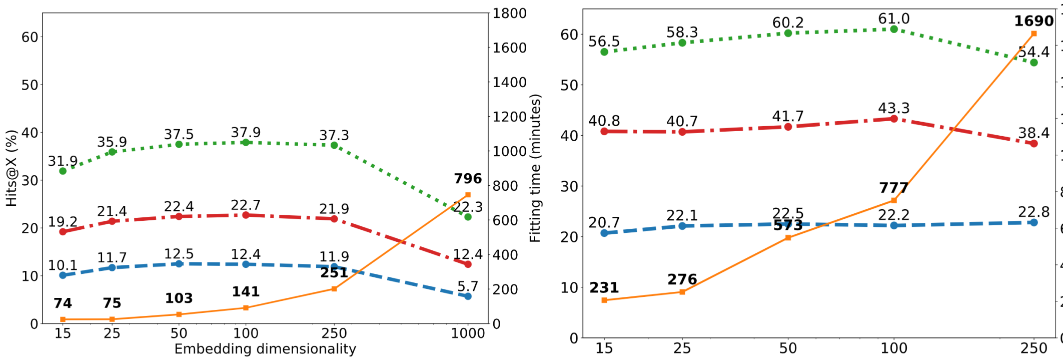
of predicates remains the same, the increased performance does not solely come from an increased parameter count.

5.7.2. Embedding size

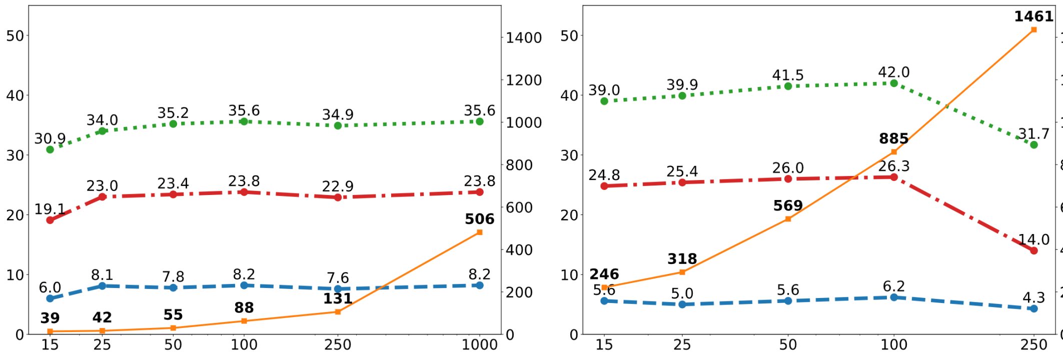
One of the most important hyperparameter is the dimensionality of the embedding (k). The higher the number of parameters, the more flexible the model is. However, in addition to increasing the time and memory consumption of the learning process, setting the parameter count too high might also lead to overfitting. To this end, we perform an ablation study to investigate the effect of k on the hits@1, hits@3 and hits@10 metrics. We also measure the total fitting time.

All evaluations were performed using the vanilla model (i.e. regular TransE) and applying both filtering approaches. Both valid time KGs had granularity 300 applied. The results of this investigation are displayed in Figure 10. While generally embedding sizes of 100 are applied in literature, our results show that similar performance can be achieved at much lower levels. This poses an interesting tradeoff with regards to training time versus maximum performance. Furthermore, as expected, performance degrades when k becomes very large, with the 1000 parameter model performing worse than the 15 parameter model on Wikidata12k and ICEWS14.

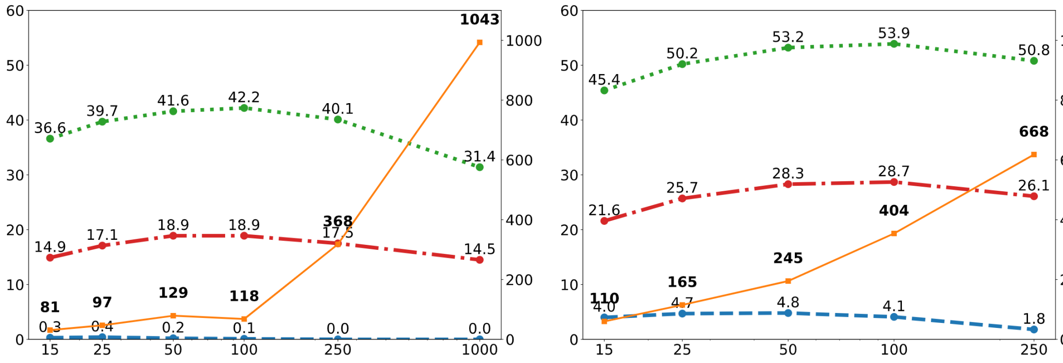
Note that the given fitting times can only be seen as an indication of the expected fitting time. There are many external factors that influence this time such as which server the model was fitted on, and the load of the that server. For instance, on ICEWS14 the 100 parameter model was fitted more quickly than the 50 parameter model. Futhermore, when generating the baseline results we found that fitting times on the same dataset with the same number of predicates could differ by a factor of 4.



(a) Wikidata12k results. Left: TransE vanilla. Right: SPLIME merge with *shrink* = 4



(b) YAGO11k results. Left: TransE vanilla. Right: SPLIME merge with *shrink* = 2



(c) ICEWS14 results. Left: TransE vanilla. Right: SPLIME merge with *shrink* = 1.5

● Hits@1 ● Hits@3 ● Hits@10 ● Fitting time

Figure 10: Ablation study regarding the effect of different embedding dimensionality. Left y-axis plots the hits@x percentage. Right y-axis plots the fitting time in minutes. For vanilla models, embedding sizes $\in \{15, 25, 50, 100, 250, 1000\}$ were tested. For merge models, embeddings sizes $\in \{15, 25, 50, 100, 250\}$ were tested.

5.8. Qualitative Analysis

To strengthen the claim that SpliMe improves embeddings we also perform a qualitative analysis of the result. We investigate this through two avenues. The first is link prediction. Specifically we perform predicate prediction and compare the results from a vanilla TransE model and a SpliMe TransE model. The second avenue is through a t-SNE plot, which allows us visualize the embeddings learned by a model. Each method will be explained further in the relevant section.

5.8.1. Predicate Prediction

In this section, we will perform qualitative analysis with regards to the predicate prediction task. (Temporal) predicate prediction is defined analogously to entity prediction. Instead of replacing the subject and object with every entity, the predicate is replaced with every other predicate. That is, given an $(s,?,o,b,e)$ quintuple we are tasked with predicting the most likely predicate. Since SpliMe operates on triples, we cannot directly pass (s,p,o,b,e) quintuples or (s,p,o,h) quadruples to the model. However, we can include the temporal aspect by filtering any answers which are not of the correct temporal scope. Specifically, we feed the model with a head and tail entity and ask for the 25 most likely predicates. From this list of predicates we then remove any predicate whose time span does not at least partially overlap with the time span of the original quintuple.

A selection of such queries is shown in Table 13. Here, the top two results are shown for every query. The correct answer is highlighted in bold. These queries were performed on a vanilla dataset and on a dataset transformed with the split (time) method with growth set to 20. The first four questions are the same as in the original HyTE paper. From these results, it appears that the vanilla TransE model and SpliMe seems to achieve equal results. We do note however that our TransE vanilla model also performs better than the one used in the original HyTE paper.

Examples where SpliMe works best are those where similar predicates are queried, but at different timesteps. For example, when a person has both the ‘wasBornIn‘ and ‘diedIn‘ predicates, but at (significantly) different timestamps. Unfortunately, the YAGO11k test set contains just one such example. Instead, the few people for which both the ‘wasBornIn‘ and ‘diedIn‘ relation are present in the test set have them occur in (almost) the same internal timestamp. Therefore, we believe that this qualitative analysis does not paint the full picture, and SpliMe will outperform TransE in a more exhaustive test.

Original quintuple (s,?,o,b,e)	**TransE**	**SpliMe**
G. Carroll, *wasBornIn*, Baltimore, 1928, 1928	**wasBornIn**, DiedIn	**wasBornIn**, diedIn
S.A. Laubenthal, *diedIn*, Washington., 2002, 2002	**diedIn**, BornIn	**diedIn**, isMarriedTo
E. G. Sander, *graduatedFrom*, Cornell Univ., 1959, 1965	worksAt, **graduatedFrom**	worksAt, **graduatedFrom**
E. Maceda, *isAffiliatedTo*, Nacionalista Party, 1971, 1987	**isAffiliatedTo**, isMarriedTo	**isAffiliatedTo**, isMarriedTo
A.Rothschild, *isMarriedTo*, E.J. Rothschild, 1877, 2020	**isMarriedTo**, wasBornIn	**isMarriedTo**, owns
A.Rothschild, *diedIn*, Paris, 1935, 1935	**diedIn**, wasBornIn	**diedIn**, wasBornIn
E.J. Rothschild, *wasBornIn*, Boulogne-Bi., 1845 , 1845	**wasBornIn**, diedIn	**wasBornIn**, diedIn
E.J. Rothschild, *diedIn*, Boulogne-Bi., 1934, 1934	wasBornIn, **diedIn**	**diedIn**, wasBornIn
Vilhelm Aubert, *worksAt*, University of Oslo, 1954, 1988	**worksAt**, graduatedFrom	**worksAt**, graduatedFrom
Marie Curie, *isMarriedTo*, Pierre Curie, 1859, 1906	**isMarriedTo**, created	**isMarriedTo**, created
Marie Curie, *hasWonPrize*, W.G. Award, 1921, 1921	**hasWonPrize**, wasBornIn	**hasWonPrize**, isMarriedTo
Marie Curie, *wasBornIn*, Warsaw, 1867, 1867	**wasBornIn**, diedIn	**wasBornIn**, diedIn
Pierre Curie, *wasBornIn*, Paris, 1859, 1859	diedIn, **wasBornIn**	**wasBornIn**, diedIn
Pierre Curie, *diedIn*, Paris, 1906, 1906	**diedIn**, wasBornIn	wasBornIn, **diedIn**
King-Sun Fu, *worksAt*, MIT, 1961, 2020	**worksAt**, graduatedFrom	**worksAt**, graduatedFrom
King-Sun Fu, *graduatedFrom*, Univ. of Toronto, 1955, 2020	worksAt, **graduatedFrom**	worksAt, **graduatedFrom**

Table 13: A comparison between link prediction results generated by a vanilla TransE model, and the best SpliMe model on the YAGO11k dataset. Some entities have had their names shortened for readability. Queries above the horizontal line are the same as in Dasgupta et al. (2018). Below the horizontal line are original. All examples were taken from the test set to ensure that the model has not seen them before.

5.8.2. t-SNE plots

t-distributed stochastic neighborhood embedding (t-SNE) is a non-parametric high-dimensional data visualization technique. Each high-dimensional data point is mapped to a location on a two or three dimensional map (van der Maaten and Hinton, 2008). We will use t-SNE to provide a legible visualization of the high-dimensional predicate embeddings that are learned by our model.

Specifically, we use the embeddings of a model learned on the Wikidata12k dataset transformed with the SpliMe merge method, with a granularity of 300 and a shrink factor of 4. The accompanying t-SNE plot is displayed in Figure 11. We observe that predicates of the same type are mostly clustered together in the embedding space. While we create the t-SNE plot for all predicates, only the ten most common predicates are plotted for legibility. Otherwise, there would be some predicates scattered throughout. This is most true for the predicates to which few splits have been applied, for example *winner of an event (P1346)*, which was only split once.

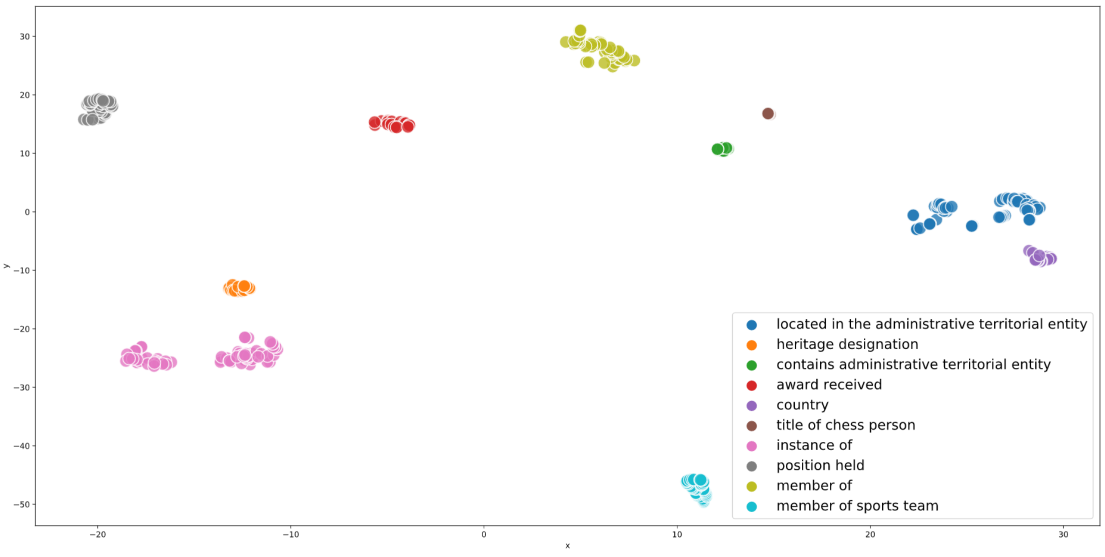
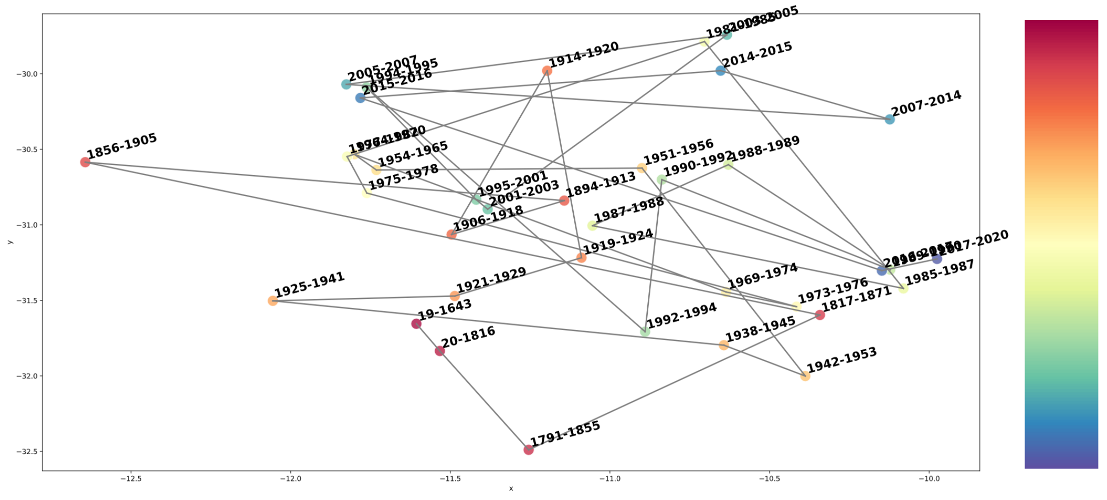


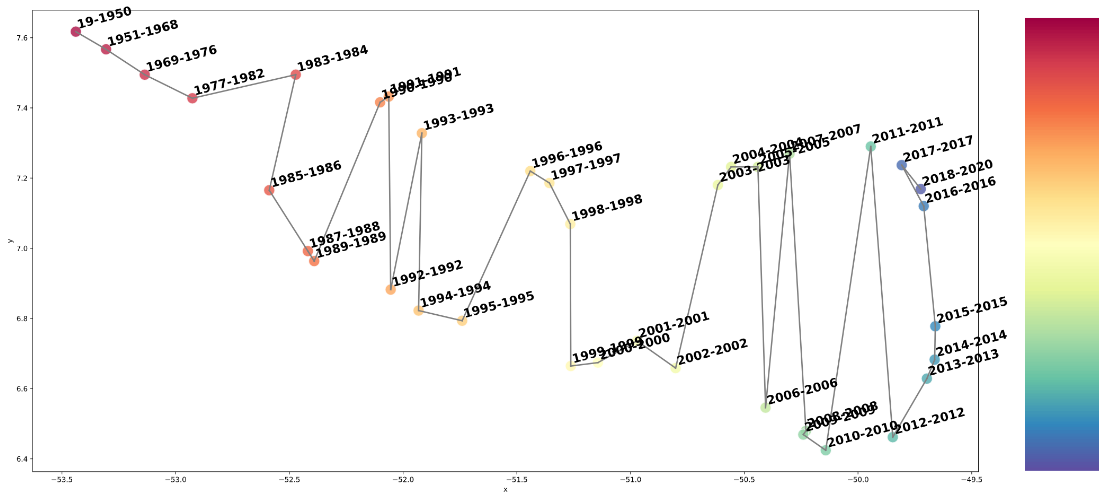
Figure 11: Two-dimensional t-SNE projection of the embedding learned by the model. For legibility, only the ten most common predicates are plotted.

Additionally, we note that predicates with many points seem to form lines or circles in the reduced dimensionality plot. To investigate this, we took the same t-SNE embeddings and created a plot containing just the predicate *member of sports team (P54)*. This plot is displayed in Figure 12b. It shows that the model has successfully learned a somewhat smooth temporal evolution of the embedding: each step in time moves the embedding in a similar direction, and the different points in time are well separated.

To evaluate how much of this is due to SpliMe, we also created a plot on a baseline dataset (i.e. one obtained through random splits) for a predicate that had an approximately equal number of splits. The original could not be used as none of the random baseline models had the required number of splits. Figure 12a shows that here the temporal evolution of the embedding is completely erratic, jumping all over the place. This implies that SpliMe includes temporal information in an intelligent manner.



(a) t-SNE plot of a dataset created by applying splits randomly. Only the *residence (P551)* predicate is drawn.



(b) t-SNE plot of a dataset created using the SpliMe merge method. Only the *member of sports team (P54)* predicate is drawn.

Figure 12: Two t-SNE projections. The colors are based on the end-time of the predicate, ranging from red (long ago) to blue (recent). The lines between the points illustrate how the embedding moves through the vector space, and the labels denote the time period during which the predicate is valid.

5.9. Temporal Evaluation

Lastly, we provide an alternative evaluation metric for temporal prediction. As a reminder, in temporal prediction a model is given an (*s,p,o*) triple and is asked to predict the most likely time point. In designing a new evaluation metric, we note that the following aspects influence the result: I) whether the fact re-occurs, II) the valid time of the original quintuple, III) the number of temporal classes in the dataset.

To show why these requirements are important and non-trivial, consider Grover Cleveland, who was president of the United States from 1885 to 1889 and 1893 to 1897 (i.e. he was president for two non-consecutive terms). Given that we are evaluating quadruples with year level granularity, the query (*s,p,o, ?*) has 8 correct answers. If we apply the normal ranking approach, given a data set consisting of 20 temporal classes (e.g. 1880-1900), there is a large chance that an alternative answer will be ranked above our answer. However, if we apply filtering to counteract this, we are left with only 13 classes, which also greatly distorts the result.

To counteract this, we propose two different evaluation procedures. These procedures are similar to the one used in entity and predicate prediction, but with a few changes to the associated metric. That is, we apply an additional penalty term depending on the length of the valid time interval. This term normalizes the difference in magnitude between small and long intervals. To be specific, given an (*s,p,o,b,e*), we first convert it to a set of (*s,p,o,h*) quadruples. Next, all possible (*s,p,o,?*) quadruples are scored by the model the scores are sorted.

Approach 1 records the sum of the ranks of all valid quadruples rather than that of the best quadruple, and divides this by the *sum of best possible ranks* to normalize the result. Suppose that we are attempting to predict the first presidential term of Grover Cleveland as described earlier. In this case, the sum of best possible ranks is $(1+2+3+4=10)$. Assuming that the model output the correct years at ranks $1, 4, 5$ and 6 respectively, the sum of recorded ranks is 16. The true rank using this approach can then be calculated as $\frac{16}{10} = 1.6$. More generally, given a set of ranks of valid quadruples R_v for a single (*s,p,o,b,e*) quintuple, the true rank can be calculated as[8]

$$\frac{\sum_{r \in R_v} r}{\sum_{i=0}^{e-b+1} i} \tag{23}$$

Approach 2 goes one step further and operates by calculating scores of time intervals rather than time points. Specifically, given a quintuple (*s,p,o,b,e*), the scores of all intervals of length $(e-b+1)$ are calculated by summing the score of the relevant (*s,p,o,h*) quadruples. These intervals are then sorted according to score, and the rank of the original interval is recorded. However, the number of classes is depends on the

[8]The calculation in the denominator, i.e. the calculation of the sum of all elements in the interval [b,e] can be efficiently calculated as $(e-b+1)*((2+e-b)/2)$.

length of the interval: a large interval will have fewer classes, and thus a lower expected rank. For a quintuple (s,p,o,b,e) whose interval was ranked at position r, the true rank can be calculated as

$$1 + (r - 1) * \frac{|\mathcal{T}|}{|\mathcal{T}| - (e - b)} \tag{24}$$

Using our Grover Cleveland example, this approach would calculate the score of every interval of length four between 1880 and 1900 by summing the scores of the quadruples in it. Suppose that the correct interval was ranked at position 4. Since the data contains 20 temporal classes, this means we apply a penalty term of $\frac{20}{20-4} = 1.25$. The rank obtained with this approach would then be $1 + (4 - 1) * 1.25 = 4.75$.

5.9.1. Evaluation

The aforementioned approaches cannot be utilized for SpliMe , as SpliMe is not made to perform temporal prediction in general. This is due to the granularity level being applied implicitly at the level of individual predicates. To illustrate, assume that the predicate *presidentOf* receives a single timestep for the period 2000-2016. The query (*?, presidentOf, USA, 2008, 2012*), will now return both Bush and Obama s answers. Yet, only the latter is actually correct.

We have implemented[9] the aforementioned approaches in the code created by Dasgupta et al. (2018). We compare these novel approaches with the original approach, which simply returns the lowest correct rank. Additionally, we compare the results for both the TANS and TDNS approaches (see section 2.6 for more information).

The results for YAGO11K and ICEWS14 are displayed in Table 14. We do not perform our experiments on ICEWS14, as here all approaches will have equal results. This is because ICEWS14 is an event dataset where the scope of every fact is exactly one day long. As a result, the penalty terms in both our approaches is just 1. All models were trained using the same hyperparameters: l_2 distance norm, margin 10 and 5 negative samples per batch. No filtering was applied. The models were run for 500 epochs.

To begin with, the results confirm the findings from Dasgupta et al. (2018) in that TDNS improves the result of temporal prediction. The difference is smaller, but is present everywhere but the YAGO11k Hits@10. It appears regardless of which evaluation metric is used. However, the choice of evaluation metric has a large influence on how the result should be interpreted.

In general, it seems that the original approach is too optimistic. By taking the best rank of any quadruple in the interval of the quintuple, it overestimates the hits@1 performance of the models. This is reflected in the fact that approach 1 has much lower hits@1 performance, but similar hits@10 performance to it. Furthermore, we observe that that approach 2 is especially strict, having lower scores across the board.

[9] https://github.com/wradstok/HyTE temporal evaluation implementation in HyTE codebase.

Sampler	**Approach**	**MR**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
TANS	original	27	0.055	5.5%	12.2%	28.8%
	1	26	0.109	0.9%	8.0%	22.3%
	2	34	0.016	1.6%	4.1%	13.5%
TDNS	original	27	0.060	6.0%	12.7%	30.2%
	1	26	0.114	1.2%	8.5%	24.1%
	2	33	0.021	2.1%	5.2%	14.8%

(a) Wikidata12k results. This dataset contains 78 temporal classes.

Sampler	**Approach**	**MR**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
TANS	original	15	0.171	17.1%	32.1%	54.9%
	1	14	0.238	1.4%	27.5%	55.0%
	2	27	0.077	7.7%	13.7%	25.2%
TDNS	original	15	0.171	17.1%	32.2%	53.0%
	1	14	0.238	1.5%	28.5%	54.9%
	2	27	0.067	6.7%	11.4%	23.5%

(b) YAGO11k results. This data set contains 61 temporal classes.

Table 14: Results for the different temporal evaluation metrics.

6. Conclusion

The first contribution of this thesis is the design and implementation of SpliMe, a model-agnostic method that uses static KGE models to embed temporal knowledge graphs. SpliMe operates through selectively splitting and merging predicates or entities, and we have created five different methods for doing so. We have shown that even incorporating temporal information by randomly applying splits improves the link prediction performance of TransE, indicating the viability of the inclusion of temporal information and the power of our approach.

Additionally, all our methods achieve results above both the vanilla TransE baseline and the random baseline. Our results show that SpliMe achieves state-of-the-art results in link prediction performance on two datasets commonly used for temporal knowledge graph embedding model evaluation (*Wikidata12k, Yago11k*), and has significantly increased performance compared to our baseline on another dataset (*ICEWS14*). We have further shown the strength of our results through our qualitative analysis. All things combined, we have shown that our approach is sound.

Secondly, we have uncovered issues with the evaluation procedures used for static KG embedding models on temporal knowledge graphs. Through this problem, static KGE models achieved higher scores than they should have, which causes literature to underestimate the effect of their temporal KGE methods. To combat this, we have introduced two explicit filtering methods which remove this effect.

Thirdly, we have reformulated the link prediction task on temporal knowledge graphs to better suit temporal scope prediction. Specifically, we introduced two new approaches which apply a penalty to make comparison between scores achieved on a different number of temporal classes possible.

As a final contribution, we have made available a fully runnable implementation of the system[10]. In addition to providing third parties with the ability to reproduce the research in this thesis, this repository contains an extensible framework which can be used for continued research in this topic.

6.1. Future Work

Lastly, we consider some possible avenues for future research. In this section, we discuss several future works related to both SpliMe and the general area of temporal KG embedding.

Temporal scope prediction: Due to both many models not having an implementation available and time constraints, we were only able to apply our new method temporal scope prediction evaluation metrics to HyTE. In future work, it will be interesting to evaluate other TKG embedding models as well. These models could be obtained by

[10]https://github.com/wradstok/thesis_radstok

acquiring their source code, or through an own implementation. This will provide yet another avenue to investigate the strengths and weaknesses of different approaches.

More static models: To keep evaluation simple and to provide a fair baseline to compare SpliMe with to other models, we have continually used TransE to learn the embeddings in this thesis. Generally, more advanced models such as ComplEx, RotatE or SimplE outperform TransE. Noting that SpliMe is model-agnostic, it will be interesting to investigate the performance of our approach in conjunction with more advanced embedding models.

Background knowledge: Several KG embedding models (e.g. (Kazemi and Poole, 2018)) have support to utilize background information in the learning process. A similar approach can be imagined for SpliMe. For instance, we could imagine implementing a hybrid approach which applies different methods depending on the class of the predicate being split. It is likely that such an approach could lead to better results.

Entities: after preliminary research showed that SpliMe was better applied to predicates than to entities, we have not invested any time in the latter. However, since this research we have developed several different split approaches which were not tested on entities. It is plausible that some of these operate just as well, or even better on entities than on predicates.

Better data: Both ICEWS14 and Wikidata12k require filtering before results can be properly evaluated. It would be nice if better subsets of the original data could be extracted so that this is no longer required. Furthermore, with only 20.000 and 40.000 triples for YAGO11k and Wikidata12k respectively, the data sets can be considered small. Larger data sets are required to more closely resemble a real-world scenario. For Wikidata this step is already being undertaken (e.g. Lacroix et al. (2020)).

References

Lada A Adamic and Eytan Adar. 2003. Friends and neighbors on the web. *Social Networks*, 25(3):211 – 230.

Farahnaz Akrami, Lingbing Guo, Wei Hu, and Chengkai Li. 2018. Re-evaluating embedding-based knowledge graph completion methods. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 1779–1782.

Samaneh Aminikhanghahi and Diane J Cook. 2017. A survey of methods for time series change point detection. *Knowledge and information systems*, 51(2):339–367.

François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. 2008. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795.

Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. 2011. Learning structured embeddings of knowledge bases. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Elizabeth Boschee, Jennifer Lautenschlager, Sean O'Brien, Steve Shellman, James Starz, and Michael Ward. 2015. Icews coded event data. *Harvard Dataverse*, 12.

Shuo Chen, Lin Qiao, Biqi Liu, Jue Bo, Yuanning Cui, and Jing Li. 2019. Knowledge graph embedding based on hyperplane and quantitative credibility. In *Machine Learning and Intelligent Communications*, pages 583–594, Cham. Springer International Publishing.

Luca Costabello, Sumit Pai, Nicholas McCarthy, and Adrianna Janik. 2020. Knowledge graph embeddings tutorial: From theory to practice. In order to view the presentation, a free account has to be created. Slides are avaiable without registration.

Luca Costabello, Sumit Pai, Chan Le Van, Rory McGrath, Nicholas McCarthy, and Pedro Tabacof. 2019. AmpliGraph: a Library for Representation Learning on Knowledge Graphs.

Shib Sankar Dasgupta, Swayambhu Nath Ray, and Partha Talukdar. 2018. Hyte: Hyperplane-based temporally aware knowledge graph embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2001–2011.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2017. Convolutional 2d knowledge graph embeddings. *arXiv preprint arXiv:1707.01476*.

Xin Luna Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 601–610. Evgeniy Gabrilovich Wilko Horn Ni Lao Kevin Murphy Thomas Strohmann Shaohua Sun Wei Zhang Geremy Heitz.

Anton Dries and Ulrich Rückert. 2009. Adaptive concept drift detection. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 2(5-6):311–327.

Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. 2014. Introducing wikidata to the linked data web. In *The Semantic Web – ISWC 2014*, pages 50–65, Cham. Springer International Publishing.

João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4).

Alberto García-Durán, Sebastijan Dumančić, and Mathias Niepert. 2018. Learning sequence encoders for temporal knowledge graph completion. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4816–4821, Brussels, Belgium. Association for Computational Linguistics.

Damien Garreau. 2017. *Change-point detection and kernel methods*. Theses, Université Paris sciences et lettres.

Rishab Goel, Seyed Mehran Kazemi, Marcus Brubaker, and Pascal Poupart. 2019. Diachronic embedding for temporal knowledge graph completion.

Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. 2005. Temporal rdf. In *European Semantic Web Conference*, pages 93–107. Springer.

Christopher J. Hillar and Lek-Heng Lim. 2013. Most tensor problems are np-hard. *J. ACM*, 60(6).

Frank L Hitchcock. 1927. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189.

Tingsong Jiang, Tianyu Liu, Tao Ge, Lei Sha, Sujian Li, Baobao Chang, and Zhifang Sui. 2016. Encoding temporal information for time-aware link prediction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2350–2354.

Woojeong Jin, Changlin Zhang, Pedro A. Szekely, and Xiang Ren. 2019. Recurrent event network for reasoning over temporal knowledge graphs. *CoRR*, abs/1904.05530.

Seyed Mehran Kazemi and David Poole. 2018. Simple embedding for link prediction in knowledge graphs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 4284–4295. Curran Associates, Inc.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

Timothée Lacroix, Guillaume Obozinski, and Nicolas Usunier. 2020. Tensor decompositions for temporal knowledge base completion.

Julien Leblay and Melisachew Wudage Chekol. 2018. Deriving validity time in knowledge graph. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 1771–1776, Republic and Canton of Geneva, CHE. International World Wide Web Conferences Steering Committee.

Kalev Leetaru and Philip A Schrodt. 2013. Gdelt: Global data on events, location, and tone, 1979–2012. In *ISA annual convention*, volume 2, pages 1–49. Citeseer.

Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195.

Pei Li, Xin Luna Dong, Andrea Maurino, and Divesh Srivastava. 2011. Linking temporal records. *Proceedings of the VLDB Endowment*, 4(11):956–967.

David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605.

Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. 2013. YAGO3: A Knowledge Base from Multilingual Wikipedias. In *CIDR*, Asilomar, United States.

George A Miller. 1995. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.

Sameh K Mohamed, Vít Nováček, Pierre-Yves Vandenbussche, and Emir Muñoz. 2019. Loss functions in knowledge graph embedding models. In *DL4KG@ ESWC*, volume 2377, pages 1–10. CEUR-WS. org.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. 2011. A three-way model for collective learning on multi-relational data. In *International Conference on Machine Learning*, volume 11, pages 809–816.

Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. 2020. Yago 4: A reason-able knowledge base. In *The Semantic Web*, pages 583–596, Cham. Springer International Publishing.

Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. 2017. Introduction to tensor decompositions and their applications in machine learning. *arXiv preprint arXiv:1711.10781*.

Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 697–706, New York, NY, USA. Association for Computing Machinery.

Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. Rotate: Knowledge graph embedding by relational rotation in complex space. In *International Conference on Learning Representations*.

Xiaoli Tang, R. Yuan, Q. Li, T. Wang, H. Yang, Yundong Cai, and H. Song. 2020. Timespan-aware dynamic knowledge graph embedding by incorporating temporal evolution. *IEEE Access*, 8:6849–6860.

Mayesha Tasnim, Diego Collarana, Damien Graux, Fabrizio Orlandi, and Maria-Esther Vidal. 2019. Summarizing entity temporal evolution in knowledge graphs. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 961–965, New York, NY, USA. Association for Computing Machinery.

Kristina Toutanova and Danqi Chen. 2015. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 57–66.

Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. 2017. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3462–3471. JMLR. org.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. International Conference on Machine Learning (ICML).

Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing*, 167:107299.

Ledyard R Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311.

Piek Vossen. 1998. Introduction to eurowordnet. In *EuroWordNet: A multilingual database with lexical semantic networks*, pages 1–17. Springer.

PJTM Vossen, Laura Bloksma, and Paul Boersma. 1999. The dutch wordnet.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *Twenty-Eighth AAAI conference on artificial intelligence.*

Robert West, Evgeniy Gabrilovich, Kevin Murphy, Shaohua Sun, Rahul Gupta, and Dekang Lin. 2014. Knowledge base completion via search-based question answering. In *Proceedings of the 23rd international conference on World wide web*, pages 515–526.

Chengjin Xu, Mojtaba Nayyeri, Fouad Alkhoury, Jens Lehmann, and Hamed Shariat Yazdi. 2019. Temporal knowledge graph embedding model based on additive time series decomposition. *arXiv preprint arXiv:1911.07893.*

Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding entities and relations for learning and inference in knowledge bases. In *Proceedings of the International Conference on Learning Representations (ICLR) 2015.*

Yujing Zhou, Jia Peng, Lei Wang, Daren Zha, and Nan Mu. 2019. Sede: Semantic evolution-based dynamic knowledge graph embedding. *Aust. J. Intell. Inf. Process. Syst.*, 16(4):64–73.

A. Alternative SpliMe Methods

A.1. Snapshots

We briefly investigated embedding a TKG by deconstructing it into a series of static knowledge graphs, and learning and evaluating each instance separately rather than applying some form of joint learning. I.e. TKG T can be written as $T = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_{|\mathcal{T}|}$ where each KG represents the facts which are true at or during that particular instant in time. The total score for the TKG could be calculated by taking the average of the scores according to the number of triples in each instance.

Unfortunately, preliminary investigations showed that this method is not feasible while maintaining the original train/test/validation splits of the original knowledge graphs. Since each time instance contains only a subset of all entities, the search space for entity link prediction is significantly reduced. Furthermore, it is very common for an entity to appear in the test set for a particular time instant, while not appearing in that instants training set. In these cases prediction is not possible at all. For these reasons, we decided not to continue our investigations in this direction.

A.2. Embedding Refinement

One could also imagine *refining* embeddings through change point detection (CPD). Specifically, applying SpliMe timestamping, and then learning the dataset would result in an embedding vector for every predicate or entity at every timestamp. Under the assumption that similar entities are close together in the vector space (i.e. distance models), we could analyse the evolution of an entity by calculating how much its embedding shifts between two timesteps using the l_1 or l_2 norm. Doing this would give a new list of the 'most important' split points, which can then be applied to the original data set.

We implemented the above description. We have included the refinement results in Table 15. Here, *Reference* refers to the timestamped model. Unfortunately, the result turned out to be generally worse than those achieved by just timestamping, which is a required pre-processing step. Only on the YAGO11k dataset was there a slight increase in performance. We believe that this is due to the fact that the learned embeddings are too *smooth*; they do not differ enough from timestep to timestep for CPD to find meaningful changes. This is emphasized by the t-SNE plots seen in section 5.8.2.

Epsilon	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# Preds**
Reference	0.337	**21.1%**	**39.9%**	**57.4%**	1485
5	0.208	10.8%	22.5%	41.6%	337
10	0.216	11.3%	23.9%	43.6%	455
20	0.237	12.8%	27.2%	46.8%	642
50	0.264	15.6%	29.3%	49.1%	673
100	0.297	18.0%	34.0%	52.7%	703
250	0.323	20.2%	38.1%	55.6%	717

(a) Wikidata12k results. Any increase of ϵ past 250 did not result in a larger number of predicates. For CPD we used bottom-up search in conjunction with a jump of 1 and a minimum section size of 2.

Epsilon	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# Preds**
Reference	0.197	6.9%	26.0%	41.2%	570
5	0.139	5.1%	15.4%	30.1%	82
10	0.153	4.8%	18.6%	33.3%	116
20	0.194	6.3%	25.2%	41.2%	166
50	**0.196**	6.3%	26.0%	**41.7%**	228
100	0.195	**6.4%**	**26.1%**	41.0%	257

(b) YAGO11k results. Any further increase in ϵ did not increase the number of predicates. For CPD we used bottom-up search in conjunction with a jump of 1 and a minimum section size of 2.

Epsilon	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# Preds**
Reference	**0.213**	**4.7%**	**29.4%**	**54.4%**	17061
5	0.170	1.5%	23.6%	48.1%	4364
10	0.187	2.3%	26.2%	51.9%	6312
20	0.198	3.1%	27.9%	53.1%	7156
50	0.198	3.2%	27.6%	53.3%	7480
100	0.199	3.3%	27.7%	53.4%	7545
250	0.199	3.2%	27.8%	53.2%	7559

(c) ICEWS14 results. For CPD we used the pelt with a jump of 1 and a minimum section size of 2.

Table 15: Refinement results. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

B. Hybrid Graphs: YAGO15k

Because SpliMe utilizes static embedding models to embed temporal graphs, it also has the ability to embed hybrid knowledge graphs. Specifically, we can apply our transformation to the temporal part of the knowledge graph while leaving the atemporal part untouched. The result can then be merged back together and fed into a static KGE model.

To investigate this claim we used the YAGO15k dataset released in García-Durán et al. (2018). This dataset was also recently used in Lacroix et al. (2020). As per the requirements of the embedding model which was introduced alongside it, temporal facts are modelled by extending a triple with a temporal modifier (*occursSince*, *occursUntil*) and a timestamp. Therefore, the dataset first needs to be converted to the (s,p,o,b,e) format used in this thesis. Characteristics of the transformed data set are given in Table 16.

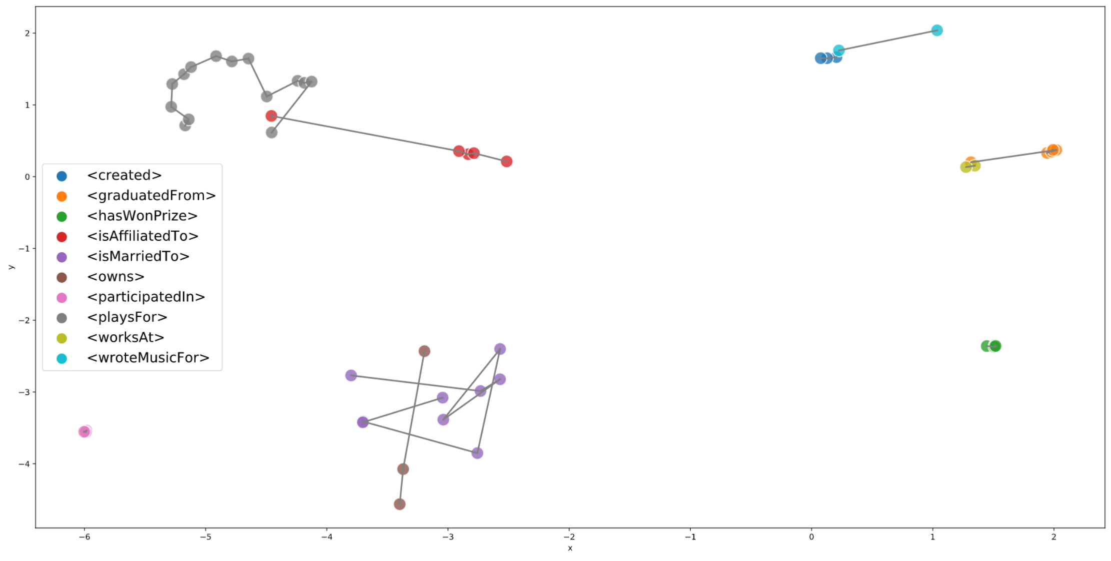
Dataset	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{E}_{temp}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}_{temp}\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert train\rvert$	$\lvert valid\rvert$	$\lvert test\rvert$
YAGO15k	15403	4193	32	10	48	98,156	12,290	12,268

Table 16: Characteristics of the YAGO15k dataset. $\mathcal{E}_{temp}$ and $\mathcal{P}_{temp}$ denote the sets of entities and predicates which are used at least once in a fact with a temporal scope respectively. A granularity level of 300 was applied to the data.

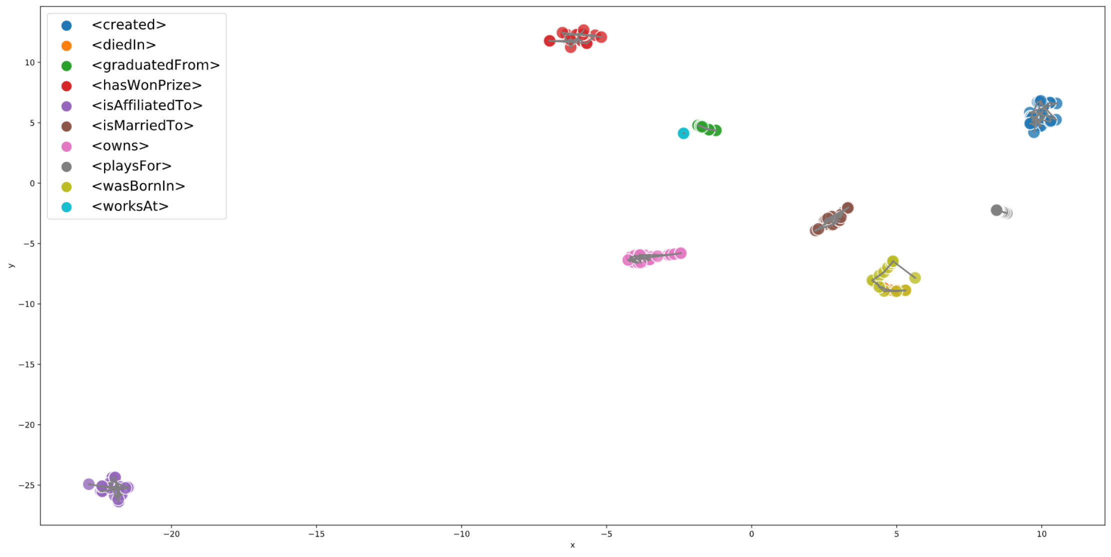
Unfortunately, preliminary investigations showed a decrease in performance with regards to the vanilla version rather than the expected increase. This effect is not unique to our evaluation. The original paper also notes a decrease in performance for several temporal measures. In fact, only one out of their four approaches improves over their baseline, and only by 0.1% and 0.2% at the hits@1 and hits@10 metrics respectively. For this reason we have not performed a full hyperparameter search to find the parameters that give the best result on the dataset. Instead, this section will be dedicated to investigating why the results are so poor. Firstly, the initial experimental result are shown in Table 17.

Method	**Setting**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# Preds**
Vanilla	*N/A*	0.170	6.1%	21.1%	36.3%	32
Random	*N/A*	0.144	7.0%	15.7%	28.7%	152
Timestamp	*N/A*	0.112	5.1%	11.4%	23.2%	464
Split (time)	$Grow = 4$	0.122	5.7%	12.7%	24.9%	72
Merge	$Shrink = 2$	0.107	4.6%	10.9%	22.4%	253
Proximity	$Pref, \epsilon = 10$	0.147	7.1%	15.9%	29.7%	74

Table 17: Overview of the experiments performed on the YAGO15k dataset. All results were obtained using both inter and intra set filtering. Random is the average of 5 runs.



(a) t-SNE plot of the ten most common predicates in the YAGO15k dataset after applying the SpliMe proximity method.



(b) t-SNE plot of the ten most common predicates in the YAGO11k dataset after applying the SpliMe split (time) method.

Figure 13: Two t-SNE plots. Each point represent a temporal predicate, the color of the points groups them based on their original predicate. The lines between the points denote the same predicate at different points in time.

To further investigate the quality of the YAGO15k embeddings, we have created two more t-SNE plots. One for YAGO15k and one for YAGO11k to compare it to. These are pictured in figures 13a and 13b respectively. This comparison is useful because the datasets share a large number of facts, and by extension entities and predicates.

Looking at the first plot we observe that different predicates receive similar embeddings, and this happens in pairs. For example, we see the pairs *WorksAt* and *graduatedFrom*, *created* and *wroteMusicFor*, and *isMarriedTo* and *owns*. Furthermore, the embeddings for a single predicate seem much more erratic: the interval between different timesteps and the direction of that interval seem to change much more than in the comparable YAGO11k dataset. This confirms that the embeddings learned on the YAGO15k dataset are of lower quality than those learned on other datasets.

A partial explanation is that only 27% of entities and 21% of the predicates present in YAGO15k are actually temporal. Additionally, temporal facts only make up just over 17% of the data in the training set. Consequently, the average number of facts associated with each temporal predicate is lower than that of an average atemporal predicate.

However, we believe the largest problem is the poor distribution of predicates in the temporal part of the dataset. In Table 18 we have displayed the ten most common predicates in each half of the dataset. The temporal part of the data is dominated by the predicate *playsFor*, occurring in almost 98% percent of all facts. As a result of this, any approach that attempts to learn a separate representation for temporal and atemporal will have issues with the lack of temporal data available for most predicates. Additionally, we hypothesize that the *playsFor* relation is exceptionally hard to learn due to its semantics. To illustrate, consider that any football club in Europe might be interesting in picking up a young talent. Predicting which one exactly requires information that is just not present in the data.

Atemporal			**Temporal**		
Predicate	**Count**	**Share**	**Predicate**	**Count**	**Share**
isAffiliatedTo	32,176	31.7%	playsFor	20,854	97.6%
isCitizenOf	18,497	18.2%	isMarriedTo	229	1.1%
isLocatedIn	12,270	12.1%	hasWonPrize	140	0.7%
playsFor	9,169	9.0%	graduatedFrom	46	0.2%
actedIn	6,487	6.4%	participatedIn	33	0.2%
wasBornIn	5,411	5.3%	created	26	0.1%
hasWonPrize	3,355	3.3%	isAffiliatedTo	15	0.1%
influences	1,537	1.5%	owns	11	0.1%
dealsWith	1,458	1.4%	worksAt	5	0.0%
happenedIn	1,243	1.2%	wroteMusicFor	1	0.0%
Total	*101,354*	*90.4%*	*Total*	*21,360*	*100%*

Table 18: The 10 most common predicates in the atemporal and temporal part of the YAGO15k dataset.

As a further investigation we also learned models on the atemporal and temporal parts of the data set separately. These results are available in Table 19. Naturally, SpliMe can only be applied to the temporal subset of the data. Our results show that both halves individually perform worse than when combined, with the temporal part performing the worst. Applying any further transformations using SpliMe also worsens the result, though by a smaller margin that previously.

All together, we conclude that YAGO15k is not a representative dataset to measure the performance of (current) temporal knowledge graph embedding methods.

Method	**Setting**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# Preds**
Vanilla	*N/A*	0.134	7.1%	14.6%	25.4%	32
Vanilla	*N/A*	0.055	1.7%	4.7%	11.4%	10
Timestamped	*N/A*	0.046	1.1%	3.8%	10.4%	432
Split (time)	$grow = 4$	0.036	0.7%	2.1%	8.6%	40
Merge	$shrink = 2$	0.044	0.7%	3.4%	10.5%	221

Table 19: YAGO15k results split out over the atemporal and temporal parts of the dataset. Above the double horizontal line are atemporal results, below are the temporal result.

C. SpliMe on Entities

Preliminary research showed that SpliMe operates better on predicates than on entities. For this reason most of our research has focussed on the former. However, the *timestamp*, *split* (both time and count) and *merge* approaches have also been implemented for entities. In this section we will list the results obtained using these methods.

After observing that the split (count) method performed worse than the split(time) method at equivalent number of entities, we did not perform further investigations into this method. As a result, only two different grow values were tested on each data set. Furthermore, on the Wikidata12k and ICEWS14 datasets, the best results from the *merge* strategy are competitive with their equivalent predicate version. However, on YAGO11k the result is significantly worse.

Dataset	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
Wikidata12k	12554	0.209	12.4%	22.7%	37.9%
Yago11k	10526	0.188	8.2%	23.8%	35.6%
ICEWS14	7128	0.141	0.1%	18.9%	42.2%

Table 20: Results for the SpliMe timestamping method on entities. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	18831	**0.218**	**13.6%**	**23.9%**	**38.1%**
2	25108	0.192	12.0%	21.2%	33.8%
5	38976	0.106	2.9%	13.3%	23.2%

(a) Wikidata12k results. Increasing the grow factor past 5 did not result in more splits being applied.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	15789	0.118	4.0%	14.6%	24.9%
2	21052	0.117	3.5%	14.8%	25.0%
5	22794	**0.165**	**9.9%**	**18.1%**	**29.5%**

(b) YAGO11k results. Increasing the grow factor past 5 did not result in more splits being applied.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	10692	0.135	0.0%	18.9%	39.8%
2	14256	0.149	0.0%	21.6%	44.5%
5	35640	**0.182**	0.0%	**31.1%**	**50.8%**

(c) ICEWS14 results.

Table 21: Results for the SpliMe split (time) method on entities. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	18831	**0.203**	**12.8%**	**21.9%**	**35.6%**
2	25108	0.178	11.1%	19.2%	31.2%

(a) Wikidata12k results.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	15789	0.100	2.8%	12.4%	21.6%
2	21052	**0.101**	2.6%	**13.1%**	21.6%

(b) YAGO11k results.

Grow factor	**# Entities**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**
1.5	10692	0.133	0.0%	18.4%	39.4%
2	14256	**0.146**	0.0%	**20.7%**	**43.5%**

(c) ICEWS14 results.

Table 22: Results for the SpliMe split (count) method on entities. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

Shrink Factor	**# Entities**	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**
1.0125	15788	0.342	**115**	23.5%	36.6%	**61.1%**
1.025	18944	**0.344**	200	**24.7%**	**37.3%**	54.8%
1.05	25030	0.307	404	20.6%	34.1%	52.8%
1.1	36372	0.272	593	18.0%	29.8%	46.5%

(a) Wikidata12k results.

Shrink Factor	**# Entities**	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**
1.025	16436	0.088	**358**	2.8%	9.1%	19.5%
1.05	22065	0.090	451	3.2%	9.2%	19.6%
1.1	32556	0.096	636	3.3%	10.0%	21.4%
1.3	66449	**0.123**	1248	**4.6%**	**13.8%**	**26.9%**

(b) YAGO11k results.

Shrink Factor	**# Entities**	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**
1.025	8978	0.125	**190**	0.0%	16.7%	36.6%
1.05	10740	0.132	232	0.0%	18.3%	38.8%
1.1	14025	0.146	314	0.0%	20.8%	43.6%
1.3	24636	0.173	773	0.0%	27.9%	50.0%
2	45063	**0.193**	5410	0.1%	**35.1%**	**51.2%**

(c) ICEWS14 results.

Table 23: Results for the SpliMe split (count) method on entities. Both inter and intra set filtering was applied for all runs. For Wikidata12k and YAGO11k granularity 300 was used, for ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D. Full results

D.1. Vanilla TransE Results

Granularity	Filter	MRR	MR	Hits@1	Hits@3	Hits@10
300	None	0.357	113	26.9%	38.8%	52.8%
300	Inter	0.209	153	12.4%	22.7%	37.9%
300	Intra	0.357	113	26.9%	38.8%	52.8%
300	Both	0.209	153	12.4%	22.7%	37.9%

(a) Wikidata12k results.

Granularity	Filter	MRR	MR	Hits@1	Hits@3	Hits@10
300	None	0.188	443	8.2%	23.8%	35.6%
300	Inter	0.188	443	8.2%	23.8%	35.6%
300	Intra	0.188	443	8.2%	23.8%	35.6%
300	Both	0.188	443	8.2%	23.8%	35.6%

(b) YAGO11k results.

Granularity	Filter	MRR	MR	Hits@1	Hits@3	Hits@10
0	None	0.201	80	0.7%	30.8%	56.1%
0	Inter	0.141	140	0.1%	18.9%	42.2%
0	Intra	0.201	80	0.7%	30.8%	56.1%
0	Both	0.141	140	0.1%	18.9%	42.2%

(c) ICEWS14 results.

Table 24: Results for vanilla TransE for the different filtering methods. Embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D.2. Timestamp Results

Dataset	**# Preds**	**MRR**	**MR**	**Hits@1**	**Hits@3**	**Hits@10**
Wikidata12k	1622	0.340	115	21.1%	40.8%	58.1%
Yago11k	570	0.197	175	6.9%	26.0%	41.2%
ICEWS14	17061	0.213	106	4.7%	29.4%	54.4%

Table 25: Results for the SpliMe timestamping method. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D.3. Split (time) Results

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	120	0.279	**94**	15.2%	32.9%	52.9%
10	240	**0.320**	111	**20.1%**	**37.2%**	**54.9%**
15	360	0.295	105	17.9%	34.3%	53.2%
20	480	0.299	165	18.8%	35.1%	51.2%
25	525	0.306	112	19.0%	35.3%	53.8%

(a) Wikidata12k results.

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	50	0.154	194	6.0%	17.5%	33.6%
10	100	0.194	200	8.0%	24.4%	39.4%
15	150	0.210	185	9.0%	26.6%	42.4%
20	200	**0.213**	**183**	**9.0%**	**27.0%**	**43.2%**
25	250	0.201	211	8.9%	24.9%	39.6%
30	273	0.202	201	8.2%	25.7%	40.5%

(b) YAGO11k results.

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	1150	0.185	100	2.6%	25.6%	50.7%
10	2300	0.187	99	2.8%	25.7%	51.3%
15	3450	0.188	**98**	2.7%	25.8%	51.5%
20	4600	0.190	101	3.0%	**26.3%**	51.6%
25	5750	0.190	103	3.2%	25.6%	**52.1%**
30	6900	**0.192**	107	**3.3%**	25.8%	51.8%

(c) ICEWS14 results.

Table 26: Results for the SpliMe split (time) method. Best results among all were highlighted in **bold**. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D.4. Split (count) Results

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	120	0.292	**92**	17.7%	33.4%	52.0%
10	240	0.281	117	16.6%	31.6%	51.3%
15	360	0.290	119	17.4%	32.7%	53.3%
20	480	0.290	119	17.4%	32.7%	53.3%
25	600	**0.300**	119	**18.3%**	**34.5%**	**53.8%**
30	604	0.290	105	16.8%	34.1%	53.6%

(a) Wikidata12k results.

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	50	0.126	251	5.8%	12.9%	26.3%
10	100	0.182	206	7.2%	22.2%	37.9%
15	150	0.184	309	7.5%	23.2%	36.8%
20	200	0.189	314	**8.6%**	22.5%	37.6%
25	250	**0.196**	**174**	8.1%	**24.1%**	**40.3%**
30	300	0.191	201	8.2%	23.4%	37.7%

(b) YAGO11k results.

Grow factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
5	1150	0.182	100	2.5%	24.9%	49.9%
10	2300	0.190	**98**	2.9%	26.1%	51.6%
15	3450	0.190	100	3.0%	25.8%	51.9%
20	4600	0.189	99	2.7%	25.9%	51.9%
25	5750	0.191	101	3.0%	**26.2%**	**52.2%**
30	6900	**0.192**	104	**3.2%**	26.0%	52.2%

(c) ICEWS14 results.

Table 27: Results for the SpliMe split (count) method. Best results among all were highlighted in **bold**. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D.5. Merge Results

Shrink factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
1.5	1088	0.348	119	21.9%	41.0%	58.9%
2	823	0.348	121	**22.0%**	**41.0%**	**59.4%**
4	423	**0.358**	100	22.2%	43.3%	61.0%
6	290	0.345	96	20.9%	41.6%	59.8%
8	223	0.342	**90**	20.3%	42.3%	59.1%
10	183	0.330	94	19.8%	39.6%	57.2%

(a) Wikidata12k results.

Shrink factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
1.5	383	0.192	191	6.0%	25.5%	41.4%
2	290	**0.195**	**183**	6.2%	**26.3%**	**42.0%**
4	150	0.180	204	**6.5%**	22.4%	38.1%
6	103	0.169	213	6.1%	20.6%	37.5%
8	80	0.173	207	6.2%	21.3%	37.9%
10	66	0.180	208	6.3%	22.6%	38.6%

(b) YAGO11k results.

Shrink factor	# preds	MRR	MR	Hits@1	Hits@3	Hits@10
1.5	11449	**0.207**	99	**4.1%**	**28.7%**	**53.9%**
2	8645	0.203	95	3.7%	28.2%	53.6%
4	4437	0.198	94	3.1%	27.7%	53.5%
6	3034	0.193	**94**	2.6%	27.3%	52.7%
8	2332	0.192	96	2.6%	27.1%	52.3%
10	1912	0.191	97	2.6%	26.8%	52.0%

(c) ICEWS14 results.

Table 28: Results for the SpliMe merge method. Best results among all were highlighted in **bold**. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

D.6. Distance Results

Measure	Epsilon	MRR	Hits@1	Hits@3	Hits@10	# preds
Adar	20	0.246	14.6%	27.0%	45.4%	182
	15	0.245	14.4%	27.4%	45.7%	258
	10	0.254	15.1%	28.4%	46.4%	356
	5	0.287	17.8%	32.7%	50.5%	545
	2.5	**0.318**	**19.6%**	**37.6%**	**55.2%**	707
	1.25	0.318	19.6%	37.6%	55.2%	742
Jaccard	20	0.248	15.1%	27.2%	45.0%	188
	15	0.248	14.6%	27.5%	45.7%	267
	10	0.253	15.3%	27.7%	46.6%	361
	5	0.287	17.6%	32.9%	50.6%	547
	2.5	**0.316**	**19.3%**	**37.6%**	**55.4%**	706
	1.25	0.316	19.3%	37.6%	55.4%	742
Pref	20	0.227	13.3%	25.0%	42.6%	154
	15	0.236	13.7%	26.0%	44.5%	217
	10	0.237	13.8%	25.6%	44.7%	315
	5	0.284	18.0%	31.5%	49.8%	504
	2.5	**0.322**	**20.1%**	**37.7%**	**55.4%**	726
	1.25	0.322	20.1%	37.7%	55.4%	742

(a) Wikidata12k results. For each measure, there was no difference between $\epsilon = 2.5$ and $\epsilon = 1.25$ at any number of significant digits.

Measure	**Epsilon**	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**# preds**
Adar	20	**0.186**	**7.1%**	24.3%	36.5%	60
	15	0.180	5.9%	24.4%	37.1%	84
	10	0.183	5.9%	24.7%	38.1%	120
	5	0.180	4.8%	24.7%	39.7%	226
	2.5	0.182	5.0%	**25.3%**	**40.0%**	257
	1.25	0.182	5.0%	25.3%	40.0%	257
Jaccard	20	**0.185**	**7.1%**	24.2%	36.4%	60
	15	0.179	5.8%	24.4%	36.8%	89
	10	0.174	5.6%	23.1%	36.8%	126
	5	0.181	5.4%	24.2%	**40.1%**	236
	2.5	0.182	5.0%	**25.3%**	40.0%	257
	1.25	0.182	5.0%	25.3%	40.0%	257
Pref	20	0.191	**7.4%**	24.9%	37.0%	52
	15	0.190	6.4%	25.9%	39.1%	72
	10	0.183	5.9%	24.7%	38.1%	98
	5	**0.202**	5.4%	**28.8%**	**43.7%**	177
	2.5	0.182	5.0%	25.3%	40.0%	257
	1.25	0.182	5.0%	25.3%	40.0%	257

(b) YAGO11k results. Note that for each split the maximum number of splits is already achieved at $\epsilon = 2.5$

Measure	Epsilon	MRR	Hits@1	Hits@3	Hits@10	# preds
Adar	200	0.156	1.0%	21.1%	45.6%	495
	150	0.182	3.1%	24.1%	50.1%	1699
	100	0.188	3.7%	24.6%	50.4%	3418
	50	0.189	3.7%	**25.3%**	50.6%	4918
	25	0.190	3.7%	25.2%	**50.7%**	5866
	12.5	0.190	3.9%	24.9%	50.6%	6390
	5	**0.190**	**3.8%**	25.2%	50.1%	6699
Jaccard	200	0.168	1.9%	22.9%	47.7%	671
	150	0.186	3.4%	24.8%	49.9%	2183
	100	0.188	3.6%	24.8%	50.0%	3629
	50	**0.189**	3.6%	25.1%	**50.5%**	4900
	25	0.188	3.5%	**25.2%**	50.3%	5958
	12.5	0.189	**3.8%**	25.1%	50.2%	6431
	5	0.189	3.8%	24.8%	49.9%	6717
Pref	200	0.167	1.7%	22.6%	47.4%	604
	150	0.184	3.2%	24.8%	49.8%	2184
	100	0.189	3.7%	24.9%	50.2%	3689
	50	0.190	3.8%	**25.4%**	50.2%	5009
	25	0.188	3.6%	25.1%	50.2%	6026
	12.5	**0.191**	**4.1%**	25.1%	50.3%	6451
	5	0.190	4.0%	25.0%	**50.3%**	6730

(c) ICEWS14 results.

Table 29: Similarity measures with CPD results. Best results for each measure are highlighted in **bold**. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used, for ICEWS14 granularity 0 was used. Used hyperparameters are: embedding size = 25, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.

Notably, the effect of the proximity measure utilized does not seem to have a large effect on the resulting performance. We observe changes of around a percentage point at most, which is well within the realm of random difference obtained in the fitting process. Instead, the result seems to depend only on the ϵ value used. The best results are obtained with $\epsilon = 2.5$ on Wikidata12k and $\epsilon = 5.0$ on Yago11k.

On ICEWS14, ϵ values seems to have little effect on the link prediction performance of the embeddings. While maximum performance is achieved at lower ϵ values, it may be more advantageous to use higher values instead. Seeing that difference in performance is minimal, the increase in training time due to the larger number of predicates might not be worth it.

D.7. Random Baseline Results

	Seed	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**Fit. Time**
Run 1	2312384781	0.271	16.1%	31.1%	49.2%	03:41:05
Run 2	463751933	0.308	19.8%	35.5%	52.0%	07:38:30
Run 3	682021680	0.289	17.7%	33.5%	50.7%	01:55:26
Run 4	93266380	0.292	18.5%	32.6%	51.2%	02:33:29
Run 5	56546590	0.329	20.5%	39.1%	55.6%	02:02:50
Run 6	88033396	0.275	16.6%	31.6%	49.6%	03:50:40
Run 7	96240958	0.262	16.1%	29.5%	46.7%	04:14:13
Average		*0.289*	*17.9%*	*33.3%*	*50.7%*	*03:42:19*

(a) Wikidata12k results, splits were applied at random until the data set contained 423 predicates.

	Seed	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**Fit. Time**
Run 1	2312384781	0.219	9.9%	27.8%	41.9%	03:45:51
Run 2	463751933	0.216	9.4%	26.6%	43.4%	02:15:28
Run 3	682021680	0.213	8.6%	27.9%	41.7%	02:18:29
Run 4	93266380	0.194	7.2%	25.0%	39.6%	02:54:38
Run 5	56546590	0.166	6.2%	20.6%	34.8%	03:19:57
Run 6	88033396	0.166	6.1%	20.5%	35.0%	03:56:16
Run 7	96240958	0.208	7.3%	27.9%	43.2%	03:33:50
Average		*0.197*	*7.8%*	*25.2%*	*39.9%*	*03:09:13*

(b) YAGO11k results, splits were applied at random until the data set contained 200 predicates.

	Seed	**MRR**	**Hits@1**	**Hits@3**	**Hits@10**	**Fit. Time**
Run 1	2312384781	0.171	2.0%	23.1%	47.9%	01:29:43
Run 2	463751933	0.173	1.8%	23.7%	48.5%	01:35:12
Run 3	682021680	0.175	2.4%	23.8%	48.2%	05:22:31
Run 4	93266380	0.172	1.7%	23.8%	48.5%	02:47:32
Run 5	56546590	0.171	2.1%	22.8%	47.2%	02:46:42
Average		*0.172*	*2.0%*	*23.4%*	*48.1%*	*02:39:14*

(c) ICEWS14 results, splits were applied at random until the data set contained 3500 predicates.

Table 30: Random baseline results. Each run represents a new creation of a baseline data set. Inter-set filtering was used for all runs. For Wikidata12k and YAGO11k granularity 300 was used. For ICEWS14 granularity 0 was used. Used hyper-parameters are: embedding size = 100, learning rate = 0.001, self-adversarial sampling loss, 500 negative examples and trained for 200 epochs.