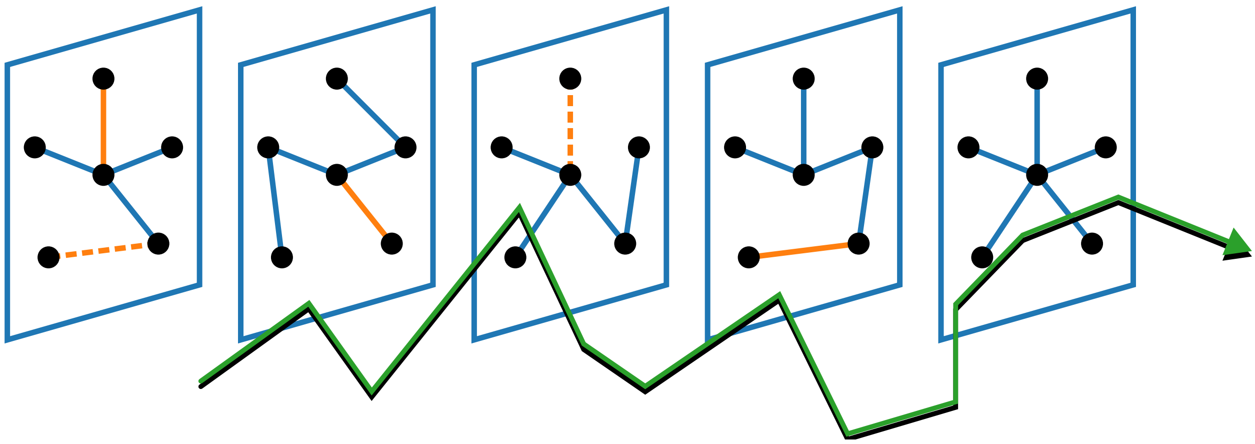

Financial News Event Prediction Using Temporal Knowledge Graphs

Master's Thesis in Data Science & AI

AXEL BLOM
ANDREAS HELGESSON

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Financial News Event Prediction Using Temporal Knowledge Graphs

AXEL BLOM
ANDREAS HELGESSON

Department of Mathematical Sciences
Division of Analysis and Probability Theory

Chalmers University of Technology
Gothenburg, Sweden 2025

Financial News Event Prediction Using Temporal Knowledge Graphs
AXEL BLOM
ANDREAS HELGESSON

Supervisor: Georg Schuppe, SEBx (part of Skandinaviska Enskilda Banken AB)
Examiner: Peter Helgesson, Department of Mathematical Sciences

Master's Thesis 2025
Department of Mathematical Sciences
Division of Analysis and Probability Theory
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of entities and relationships evolving in a temporal knowledge graph, as well as a line graph to symbolise evolving trends.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Financial News Event Prediction Using Temporal Knowledge Graphs
AXEL BLOM
ANDREAS HELGESSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Predicting future financial events and trends is crucial for effective decision-making in financial markets. Traditional approaches utilizing Temporal Knowledge Graphs (TKGs) primarily rely on specific relationship forecasting within knowledge graphs, often falling short in capturing the broader, more dynamic shifts that characterize industry trends. This thesis explores the feasibility of using TKGs in concert with advanced machine learning to predict such trends, shifting the focus from specific relationship predictions to identifying broader patterns in entity categories and relationship types. We investigated multiple techniques, including adapting existing graph convolutional network models and implementing custom model architectures leveraging Transformers and Multi-Head Latent Attention. Performance was evaluated across several publicly available TKG datasets. Our findings suggest that while defining a clear and actionable trend within a TKG presents a significant challenge, a trend-based approach using carefully curated datasets and appropriate machine-learning architectures has the potential to improve prediction accuracy, particularly when combined with methods addressing model generalization and data scarcity. Future research directions include exploring more sophisticated trend definitions, incorporating entity-specific category information, and further optimizing model architectures for enhanced performance and reduced computational costs.

Keywords: Financial Event Prediction, Temporal Knowledge Graphs (TKGs), Dynamic Knowledge Graphs (DKGs), Graph Convolutional Networks (GCNs), Transformers, Multi-Head Latent Attention (MLA), Time Series Analysis.

Acknowledgements

We are deeply grateful to Georg Schuppe, Researcher at SEBx, for his invaluable expertise and unwavering support throughout this study. His insights were crucial; and his guidance consistently pushed us to refine our approach. We also extend our sincere thanks to Anastasia Varava and the entire SEBx team for their warm welcome, collaborative spirit, and provision of essential resources that significantly contributed to the success of this project.

We thank Balazs Toth, Principal Machine Learning Engineer at AMD Silo AI, for his insightful guidance and keen interest in our work. His advice on existing machine learning frameworks was particularly helpful.

Finally, we extend our profound appreciation to our academic supervisor, Peter Helgesson. His insightful feedback, unwavering encouragement, and dedication to our progress were instrumental in shaping this thesis. We are immensely grateful for his commitment to our intellectual growth and academic success.

Axel Blom & Andreas Helgesson, Gothenburg, June 2025

List of Acronyms

Below is a list of acronyms used throughout this thesis:

AI	Artificial Intelligence
CDF	Cumulative Distribution Function
CENET	Contrastive Event Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture (NVIDIA)
GCN	Graph Convolutional Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
KS	Kolmogorov-Smirnov
LLM	Large Language Model
MHA	Multi-Head Attention
MLA	Multi-Head Latent Attention
MQE	Mean Quartic Error
MSE	Mean Squared Error
RE-GCN	Recurrent Evolution Graph Convolutional Network
RE-Net	Recurrent Event Network
RMSE	Root-Mean Squared Error
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
TKG	Temporal Knowledge Graph

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables used throughout this thesis:

Indices

i,j	Indices for design and target matrices
l	Layer index for graph convolutional network
t	Timestep index for temporal knowledge graph

Sets

E	Set of distinct entities and objects in the knowledge graph
F	Set of possible facts in the knowledge graph
F	Set of timesteps in the temporal knowledge graph
G	Set of entities, objects, relations, and facts making up the knowledge graph
R	Set of distinct relations in the knowledge graph
U	Set of time gates in the gate recurrent unit

Parameters

H	Number of past timesteps to use as history
K	Key matrix
N	Number of nodes in the TKG
Q	Query matrix
T	Number of training timesteps in the TKG
V	Value matrix

X	Design matrix
Y	Target matrix
τ	Timestep specifier in the temporal knowledge graph quadruple
d_{model}	Latent dimension
d_k	Key dimension (attention)
f	Fact in the set F
h	Number of timsteps into the future to predict (prediction horizon)
o	Object entity
r	Relation
s	Subject entity

Variables

V	Time gate recurrent unit coefficients (weights)
W	Graph Convolutional Network coefficients (weights)
β (beta)	Matrix of regression coefficients
ϵ (epsilon)	Matrix of regression error terms
μ (mu)	Feature mean
σ (sigma)	Feature standard deviation
b	Bias (intercept)
d	Time gate recurrent unit coefficient (intercept)
$\mathbf{e}$	Evolution unit entity embedding
w	Vector of coefficients (weights)
x	Design variable
y	Target variable

Contents

List of Figures

List of Tables

1

Introduction

Understanding and anticipating future events is a cornerstone of numerous applications in machine learning, from recommendation systems to predictive analytics. It is also very important for applications that aim to assist financial trading decisions. Knowledge Graph (KG) reasoning is central to this endeavour, as it enables machines to infer relationships and derive insights from structured data. Traditional knowledge graphs, however, often represent static snapshots of information, neglecting the temporal dynamics inherent in real-world events.

Temporal Knowledge Graphs (TKGs) have emerged as a potent solution to this limitation by associating each fact within the graph with a specific timestep. This temporal dimension enriches the graph's representational capacity, allowing for a more nuanced understanding of how entities and their relationships evolve over time. Despite their promise, most existing reasoning methods on TKGs are predominantly retrospective, focusing on interpreting past events without the capability to forecast future occurrences.

SEBx[1], an independent innovation studio within the SEB Group[2], is exploring the possibility to use TKGs to predict future news events or trends that might affect global financial markets, such as currency trading markets.

1.1 Background

In recent years, research on extrapolation using TKGs has advanced considerably. A notable development is the Recurrent Event Network (RE-Net) [2]. RE-Net is an autoregressive architecture designed to predict future interactions within TKGs. It models the occurrence of an event as a probability distribution conditioned on sequences of past KGs. The network employs a recurrent event encoder to process historical facts and a neighbourhood aggregator to capture connections of facts at specific timestep. This dual-module approach enables RE-Net to sequentially infer future facts, achieving high performance in multi-step inference tasks across several public TKG datasets. This advancement has significantly contributed to the foundation of this research area.

[1] `https://sebx.io/`
[2] `https://sebgroup.com/`

Building upon the foundation laid by RE-Net, the Recurrent Evolution Graph Convolutional Network (RE-GCN) further introduces significant enhancements in TKG reasoning [1]. While RE-Net employs a recurrent architecture to model temporal sequences, RE-GCN integrates a relation-aware Graph Convolutional Network (GCN) to capture both structural dependencies within each timestep and sequential patterns across time. This combination allows RE-GCN to learn evolutional representations of entities and relations more effectively, and by incorporating static properties such as entity types, it further refines these representations. Empirical evaluations on benchmark datasets have demonstrated that RE-GCN achieves substantial improvements in performance and efficiency over RE-Net in temporal reasoning tasks.

However, many opportunities remain for further advancement in TKG reasoning. One notable development is the Contrastive Event Network (CENET) model, which builds upon the successes of RE-GCN [3]. In CENET, both non-historic events (those corresponding to the immediate prediction target) and historic events (those providing long-range temporal context) are leveraged to enhance prediction accuracy. The model employs two complementary branches: one encodes the current or non-historic event, while the other aggregates information from relevant historical events. A contrastive loss is then computed to align the embeddings of semantically similar historical and current events, while simultaneously pushing apart dissimilar ones. This dual-objective loss, combining the traditional prediction loss with the contrastive loss, enables CENET to learn more discriminative temporal representations that capture both the evolution of events and the underlying relational structure, thereby achieving superior performance in future event prediction tasks.

Building on the advances made by RE-Net and RE-GCN, FinDKG [4] extends TKG reasoning into potential use cases in the context of financial markets. While RE-Net and RE-GCN focus on modelling static temporal sequences and capturing structural dependencies within fixed datasets, FinDKG addresses the challenges of rapidly changing financial environments by leveraging large language models to continuously curate a dynamic financial news dataset. This dynamic curation process extracts and integrates unstructured textual data, such as breaking news and financial reports, into a knowledge graph, ensuring that the graph can stay current with developing news events. FinDKG builds upon the established methodologies of RE-Net and RE-GCN in its deployment on TKGs, although it does leave some room for further improvements, such as those proposed by CENET, to be explored. FinDKG does, however, pioneer the dynamic integration of continuously updated data, setting a new standard for TKG applications in finance and providing an excellent foundation for our study.

1.2 Problem & Aims

While the majority of existing research concentrates on extrapolating specific future relationships or entities within TKGs, there is a notable gap in the exploration of utilizing TKGs to derive more general insights from future predictions. This study addresses this gap by investigating the feasibility of leveraging TKGs to forecast broader trends.

This research deviates from the traditional approach of predicting specific relationships between entities. Instead, it seeks to predict the *type* of relationships that are likely to form and the *type* of entities they are likely to connect. This shift in focus is motivated by the inherent difficulty in predicting specific relationships, particularly when considering timesteps beyond a single unit into the future, as highlighted by Jin et al. [2]. The central hypothesis is that this generalized approach can achieve superior performance in predicting broader industry trends compared to the conventional method of predicting specific relationships.

Consider the example of predicting an increase in the AI trend. Rather than attempting to predict the specific relationship triple (Microsoft, invests in, AI), the focus is shifted to predicting whether the future will contain more relationships of the form (?, invests in, AI) than currently exist. This would indicate a general increase in investment within the AI domain, signalling the emergence of an AI trend. However, it is important to note that defining and deriving an emerging 'trend' in the context of TKGs requires careful investigation and formalization.

Therefore, the primary objective of this project is to investigate, implement, and evaluate advanced models for predicting future industry trends by leveraging the structural and dynamic properties of TKGs. The model will be trained and evaluated on open-source TKG datasets, providing a realistic context where accurate future event prediction is crucial.

To establish a non-neural-network-based performance baseline, a simple linear regression model will be implemented. Furthermore, an existing machine learning model will be adapted, and a novel model will be developed to ascertain whether modern machine learning techniques can yield better performance in practice.

1.3 Research Questions

As detailed in section 1.2, this study is designed to address the following specific research questions:

1. How do we define a 'trend' in the context of TKGs?

2. Can the proposed trend-based prediction method yield better prediction accuracy than previous machine learning approaches, albeit with lower specificity?

3. How does the proposed trend-based method compare to simpler mathematical and statistical approaches, such as linear regression?

1.4 Structure

This report is structured into five main parts. Initially, a theoretical background (chapter 2) provides the essential concepts underpinning this study. Following this, chapter 3 specifically defines and contextualizes the definition of 'trend' within the context of TKGs. Chapters 4 through 5 then detail the process of initial dataset exploration and analysis, culminating in the curation of a custom dataset designed to address identified limitations of currently available datasets. The subsequent model implementation and results are presented in chapters 6 through 9. Finally, the report concludes with a comprehensive presentation of results, discussion, and conclusion, found in chapters 10, 11, and 12, respectively.

2

Theory

This chapter introduces the theoretic concepts, frameworks, and models used throughout this study.

2.1 Classification vs. Regression

In supervised learning, the choice between classification and regression depends on the nature of the output variable. Classification involves mapping inputs to one of a finite set of discrete labels, while regression predicts a continuous quantity [5].

2.1.1 Classification

In classification, we assign each input example $\mathbf{x} \in \mathcal{X}$ to one of C discrete classes by learning a decision function

$$f : \mathcal{X} \to \{1, 2, \ldots, C\},$$

parameterized by a model's outputs $\mathbf{z} = (z_1, \ldots, z_C)$. Typically a softmax activation is applied to $\mathbf{z}$ to obtain class probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)}, \quad i = 1, \ldots, C,$$

so that $p_i \in (0, 1)$ and $\sum_i p_i = 1$ [6]. The predicted class is $\arg\max_i p_i$, and the magnitude of p_i serves as a confidence score.

Correctly set up training minimizes the *cross-entropy* loss between the one-hot true label $\mathbf{y}$ and the predicted distribution $\mathbf{p}$:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{C} y_i \log p_i = -\log p_{y^*},$$

where y^* is the true class index. Cross-entropy penalizes incorrect predictions heavily, guiding the network to assign high probability mass to the correct class.

- **Example tasks**: Image recognition (e.g., dog vs. cat), text sentiment analysis (positive vs. negative).

- **Original RE-GCN**: Casts relation prediction as multi-class classification over relation types for each entity pair, using softmax and cross-entropy.

2.1.2 Regression

In regression, we model a continuous target variable $y \in \mathbb{R}$ by learning a function $f\colon \mathcal{X} \to \mathbb{R}$ that predicts real-valued outputs [5]. Typical regression tasks include house price estimation, time series forecasting, and, in our adaptation, predicting node degrees in TKGs as real-valued counts.

This learning task requires minimizing error-based loss functions to optimize predictions. Here, we outline several loss functions to explore theoretically, focusing on their penalty mechanisms for prediction errors.

The Mean Squared Error (MSE), defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true},i} - y_{\text{predicted},i})^2$$

where $y_{\text{predicted},i}$ is the predicted value, $y_{\text{true},i}$ is the true value, and n is the number of samples, penalizes larger errors more due to squaring [6].

A related metric, the Root Mean Squared Error (RMSE):

$$\text{RMSE} = \sqrt{\text{MSE}}$$

retains this emphasis on larger errors but aligns with the unit of the output variable [5].

Alternatively, the Mean Absolute Error (MAE):

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_{\text{true},i} - y_{\text{predicted},i}|$$

penalizes errors linearly, reducing sensitivity to outliers [6].

For greater emphasis on large errors, the Quartic Loss uses the same structure as MSE but raises the error to the fourth power:

$$\text{MQE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true},i} - y_{\text{predicted},i})^4$$

This amplifies penalties for significant deviations, potentially suiting critical predictions in TKGs.

These loss functions offer varied approaches to error penalization, each with distinct theoretical properties for regression tasks.

2.2 Linear Regression & Stochastic Gradient Descent (SGD)

This section outlines the fundamental principles of Linear Regression and Stochastic Gradient Descent (SGD), elaborating on the computational advantages of employing SGD for estimating linear regression models, as is relevant for this study (linear regression is used as a baseline model). For a more comprehensive explanation of these concepts, we refer to Bishop [7].

Linear Regression seeks to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. The objective is to minimize the MSE loss function, which quantifies the difference between the predicted and actual values. The linear model is expressed as:

$$y = X \cdot w + b$$

where y represents the dependent variable, X is the matrix of feature values, w denotes the vector of coefficients (weights) associated with each feature, and b signifies the bias (intercept) term.

SGD is an iterative optimization algorithm employed to locate the minimum of a function. In the context of linear regression, this function is the MSE loss function. Unlike batch gradient descent, which computes the gradient using the entire dataset, SGD approximates the gradient based on a single, randomly selected data point (or a small subset referred to as a mini-batch) in each iteration. The iterative process is generally described as follows (simplified):

Algorithm:

1. **Initialization**: Start with some initial guess for the coefficients w and the bias b. These can be random or zero.

2. **Iteration**: Repeat until convergence (or a maximum number of iterations is reached):

   (a) **Sample**: Randomly select a data point (x_i, y_i) from the dataset.

   (b) **Prediction**: Calculate the predicted value using the current w and b:

   $$y_{\text{predicted}} = X[i] \cdot w + b$$

   (c) **Calculate Error**: Calculate the error (loss) for this single data point using the MSE:

   $$\text{loss} = (y_{\text{true}} - y_{\text{predicted}})^2$$

(d) **Calculate Gradients**: Calculate the gradients of the loss function with respect to w and b. These gradients tell us how much we need to adjust w and b to reduce the error:

$$dw = -2 \cdot X[i] \cdot (y_{\text{true}} - y_{\text{predicted}})$$

$$b = -2 \cdot (y_{\text{true}} - y_{\text{predicted}})$$

(e) **Update Parameters**: Update the coefficients w and the bias b in the direction opposite to the gradient, scaled by a learning rate (α):

$$w = w - \alpha \cdot dw$$

$$b = b - \alpha \cdot db$$

3. **Convergence**: The algorithm stops when the changes in w and b become very small or when a maximum number of iterations is reached.

It is crucial to recognize that SGD optimizes the same loss function as standard linear regression and, consequently, should approximate similar values for the weight vector, w, and the bias term, b. Unlike standard linear regression which finds the exact minimum analytically (often requiring significant computational resources, especially for large datasets), SGD iteratively approximates the minimum through sequential parameter updates.

This iterative approach allows SGD to be very efficient, particularly when dealing with large datasets. With a sufficiently small learning rate and enough iterations, SGD should converge to a solution closely resembling the optimal one, but with a fraction of the computational cost [7]. While SGD updates parameters based on individual data points (or mini-batches), these updates average out over iterations, approximating the full-dataset gradient. The oscillations inherent in SGD's stochastic nature also decrease as it converges.

2.3 Knowledge Graphs (KGs)

A knowledge graph is a powerful data structure for representing structured knowledge. They organize information into entities and the relationships that connect them, creating a framework that facilitates storage, retrieval, and reasoning over complex and heterogeneous data. This structure has long been foundational in fields like logic and Artificial Intelligence (AI), where the ability to interpret and reason over relationships is critical.

A knowledge graph is typically represented by a set of triples (entity, relation, object) [8]. Both the 'entity' and 'object' are entities, with the latter being the target node when the relation is a directed edge the the KG. For example, consider the news story: 'The collapse of Silicon Valley Bank caused turmoil in the global financial markets'. This can be expressed through the triplets in Table 2.1.

Table 2.1: Table listing triples representing a global news story impacting a market

Entity	Relation	Object
'SiliconValleyBank'	'collapse'	'GlobalFinancialCrisis'
'SiliconValleyBank'	'operatesIn'	'BankingSector'
'GlobalFinancialCrisis'	'impacts'	'GlobalMarkets'

The knowledge graph as a whole is formally expressed as $G = \{E, R, F\}$, where:

- E is the set of all distinct entities and objects (e.g., 'SiliconValleyBank', 'GlobalMarkets').
- R is the set of all distinct relations (e.g., 'collapse', 'impacts').
- $F \subseteq E \times R \times E$ represents the set of all possible facts. Each fact $f \in F$ is represented as (entity, relation, object).

Through this representation, knowledge graphs enable structured reasoning and provide a rich basis for downstream tasks like question answering, information retrieval, and, in our case, predictive modeling.

2.3.1 Temporal Knowledge Graphs (TKGs)

A TKG is a directed, multi-relational graph that extends a static KG by incorporating a temporal dimension, allowing for the representation of time-sensitive facts [9]. Formally, a TKG is expressed as $G = (E, R, T, F)$, where T denote the timesteps expanding on the KG definition. $F \subseteq E \times R \times E \times T$ represents the set of all possible timestep facts. Each fact $f \in F$ is represented as a quadruple (entity, relation, object, τ), where τ is the timestep.

A TKG G_t consists of time-specific facts, capturing both structural and temporal dynamics. The graph $G_{0:t}$ aggregates all subgraphs G_τ over time, formally represented as $G_{0:t} = \bigcup_{\tau=0}^{t} G_\tau$.

For example, consider the scenario of geopolitical events. The entity set E could include {Barack Obama, Iran, Iraq}, the relation set R might include {make statement, provide aid, receive aid}, and the timestep set T could include dates such as {2014-06-18, 2014-06-22}. The fact set F could then represent events such as in the Table 2.2.

Table 2.2: Table showing example TKG facts

Entity	Relation	Object	Timestep
'Iran'	'Provide Protection'	'Iraq'	2014-06-18
'Barack Obama'	'Make Statement'	'Iran'	2014-06-18
'Barack Obama'	'Provide Aid'	'Iraq'	2014-06-22

2.3.2 Event Prediction Using Temporal Knowledge Graphs

TKGs can be leveraged for event prediction, primarily through two approaches:

- **Interpolation**: Predicting missing aspects of past or present events within a known timeframe $[0, T]$ [10].
- **Extrapolation**: Predicting future events beyond the observed time horizon T [2].

Both cases can be further divided into two types of predictions, interpolation is used in the example below:

- **Time Prediction**: Given a query with a missing timestep, (entity, relation, object, ?), the goal is to determine when in the historical timeline the event occurred.
- **Time-Dependent Query Answering**: Given a query with a missing entity or relation, the objective is to infer the missing element based on prior knowledge.

Extrapolation follows a similar structure but aims to predict missing facts for events that have not yet occurred. Most prior research has focused on extrapolation of the two types, where the objective is to forecast entities or relations in future timesteps. For example, given a query (entity, ?, object, τ) for a future time $\tau > T$, the goal is to predict the most probable relation. Building on the example in Table 2.2, consider the quadruple (Barack Obama, ?, Iran, 2014-06-24) - what is a likely relation (?) based on the data from the previous timesteps?

2.4 Data Scaling

Scaling is a crucial preprocessing step in machine learning, particularly when dealing with algorithms sensitive to feature magnitudes. Common scaling methods aim to transform features to a similar range, preventing features with larger values from dominating the learning process. This section explains two scaling methods, as used in the baseline Linear Regression Model (refer to chapter 6). For a more comprehensive explanation of these concepts, please refer to Igual et al. [11].

Two popular techniques are Min-Max Scaling and Standard Score Normalization, each offering distinct advantages and disadvantages based on the dataset characteristics. Min-Max Scaling transforms features by scaling them to a specific range, typically between zero and one. Mathematically, for a feature x, the scaled value x' is calculated as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Where $\min(x)$ and $\max(x)$ are the minimum and maximum values of the feature in the dataset, respectively. A primary advantage of Min-Max Scaling is its simplicity and the guaranteed boundedness of the resulting values. However, it is highly sensitive to outliers, as these extreme values can significantly compress the range of the remaining data points.

Conversely, Standard Score Normalization standardizes features by removing the mean and scaling to unit variance. The scaled value x' is computed as:

$$x' = \frac{x - \mu}{\sigma}$$

Where μ is the mean of the feature and σ is its standard deviation. Standard Score Normalization is less sensitive to outliers compared to Min-Max Scaling, as it relies on the mean and standard deviation, which are more robust to extreme values than minimum and maximum values. Its main advantage is making the data more Gaussian-like. On the other hand, it does not guarantee a specific range for the feature values, potentially leading to values outside the [0, 1] interval, and it may not be suitable for datasets with non-Gaussian distributions.

2.5 Data Analysis

The Kolmogorov-Smirnov (KS) test is a non-parametric test comparing the empirical cumulative distribution function (CDF) of a sample to a theoretical CDF (one-sample) or the ECDFs of two samples (two-sample). The KS statistic quantifies the maximum absolute difference between the CDFs, with a larger statistic value indicating greater dissimilarity. The p-value, generated alongside the statistic value, represents the probability of observing a statistic value as extreme or more extreme than the calculated one, assuming the null hypothesis (data comes from the specified distribution or both samples from the same distribution) is true. A low p-value (typically < 0.05) suggests rejecting the null hypothesis, indicating evidence against the assumption that the samples originate from the hypothesized distribution. For a more thorough treatment of the KS test, refer to Hollander et al. [12].

2.6 RE-GCN

All models implemented in this study, with the exception of the baseline regression model, are based on RE-GCN [1], a machine learning model designed to predict future relations in a TKG by learning evolutional representations of entities and relations within the TKG.

This section outlines the fundamental design of this model as is relevant for this study. For a more in-depth explanation, please refer to Li [1].

RE-GCN handles entities and relations dynamically when addressing temporal aspects in TKGs. Entities are represented by dynamic embeddings that evolve over time based on interactions and the KG structure, maintaining consistency with inherent properties through an optional static graph constraint. Relations are also represented by dynamic embeddings, reflecting evolving relationships between entities, with a Gated Recurrent Unit (GRU) component explicitly modeling temporal dependencies. To address the temporal nature of TKGs, RE-GCN processes the TKG as a sequence of KG snapshots, captures temporal patterns using recurrent components (GRU), and integrates previous information into updated entity embeddings with a Time Gate. This Time Gate weighs new information, preventing entities from converging to the same values.

2.6.1 Model Structure

RE-GCN processes the TKG as a sequence of graph snapshots, $G = [G_1, G_2, ..., G_t, ...]$, where each G_t represents the KG at timestep t. The model's architecture is built around two key components: the evolution unit and the score functions.

The evolution unit is responsible for learning the dynamic representations of entities and relations at each timestep. It iteratively updates these representations by considering: structural dependencies within the KG at the current timestep, sequential patterns across different timesteps in the TKG, and static properties of the entities, as encoded in a static graph. The evolution unit is described in further detail in section 2.6.1.

The score functions use the evolved entity and relation embeddings to predict missing facts (entities or relations) at future timesteps. The RE-GCN employs a custom scoring function (`ConvTransE`) but this can be replaced by other score functions.

Figure 2.1 illustrates the architecture of the RE-GCN model and its components.

Evolution Unit

The evolution unit comprises these key components:

- **Relation-Aware GCN**: A GCN is applied to each KG snapshot G_t to capture structural dependencies. The GCN propagates information between entities based on their relationships. By being relation-aware, the GCN considers the specific relation type when aggregating information from neighbouring entities. This means the influence of a neighbouring entity depends on the type of relation connecting them. Formally, the embedding of an entity o at layer $l + 1$ and time t, denoted $\mathbf{e}_{o,t}^{(l+1)}$, is computed by aggregating messages

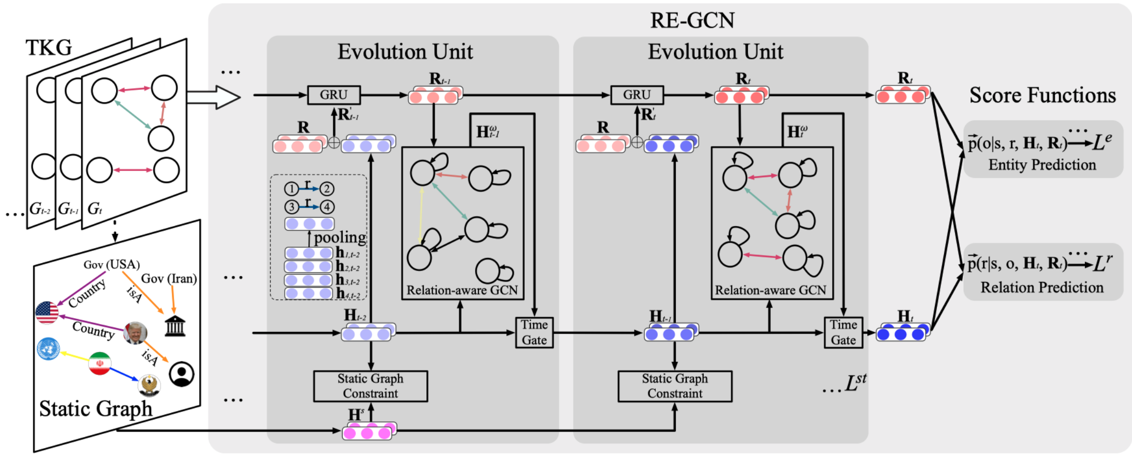

Figure 2.1: Figure depicting the overarching architecture of the RE-GCN model [1]

from its neighbours s connected via relation r:

$$\mathbf{e}_{o,t}^{(l+1)} = f\left(\sum_{(s,r):(s,r,o)\in E_t} \left(W_1^{(l)}\mathbf{e}_{s,t}^{(l)} + W_2^{(l)}\mathbf{e}_{r,t}^{(l)}\right) + W_3^{(l)}\mathbf{e}_{o,t}^{(l)}\right)$$

where $\mathbf{e}_{s,t}^{(l)}$, $\mathbf{e}_{r,t}^{(l)}$, and $\mathbf{e}_{o,t}^{(l)}$ are the embeddings of subject entity s, relation r, and object entity o at layer l and time t, respectively; $W_1^{(l)}$, $W_2^{(l)}$, and $W_3^{(l)}$ are learnable weight matrices; and $f(\cdot)$ is a non-linear activation function. The final term accounts for self-evolution of the node.

- **GRU for Relations**: The GRU explicitly models the temporal dependencies of the relations in the TKG. It uses the current representation of a relation, combined with the representation of its related entities from the previous timestep, to produce an updated representation. The GRU takes as input a combined representation r' of a relation r and the representations of the entities involved in that relation. This is used in conjunction with the previous embedding of the relation R_{t-1} as input to the GRU:

$$R_t = \text{GRU}(R_{t-1}, r')$$

- **Time Gate Recurrent Component for Entities**: This is crucial in preventing over-smoothing and vanishing gradients in deeper, stacked GCNs. It effectively weighs the information from the current GCN layer output against the entity embeddings from the previous time step. The Time Gate controls how much of the new information to incorporate and how much to retain from the past. Formally, a time gate U_t is calculated:

$$U_t = \text{sigmoid}(V\mathbf{e}_{t-1} + d)$$

where V and d are parameters to be learned and $\mathbf{e}_{t-1}$ are embeddings of the entities at the previous timestep. The updated entity embeddings $\mathbf{e}_t$ are then calculated using the time gate as:

$$\mathbf{e}_t = U_t \odot \mathbf{e}'_t + (1 - U_t) \odot \mathbf{e}_{t-1}$$

where $\mathbf{e}'_t$ is the output of the relation-aware GCN at timestep t and $\odot$ is the element-wise product.

- **Static Graph Constraint Component**: This component incorporates static properties of entities (e.g., entity types). A static graph is built representing these properties, and the model encourages the learned dynamic entity embeddings to remain consistent with the static graph structure.

2.7 Transformers

The Transformer architecture, introduced by Vaswani et al. [13], has become the cornerstone of machine learning areas such as modern LLMs. A central component of Transformer architecture is the Multi-Head Attention (MHA) mechanism, which allows the model to weigh the importance of different tokens in the input sequence when generating representations, potentially outperforming RNNs in capturing long-range dependencies [13]. Section 2.7.1 briefly outlines the key operating concepts of a Transformer, as necessary to understand in the context of this study.

In this study, we utilize the PyTorch[1] implementation `TransformerDecoder`[2] , whose architecture is described in further detail in section 2.7.2.

While highly effective, the standard MHA mechanism presents a significant bottleneck, particularly during the inference phase. A recent innovation in combating these issues is Multi-Head Latent Attention (MLA) [14]. This Transformer architecture will also be tested in this study, and its architecture is explained in further detail in section 2.7.3.

2.7.1 Key Operating Concepts

The core of the Transformer's attention mechanism is the scaled dot-product attention, defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where:

- Q is the query matrix (represents the current token).
- K is the key matrix (represents all tokens).

[1] `https://pytorch.org/`
[2] `https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html`

- V is the value matrix (holds token content).
- d_k is the key dimension.
- $Token$ is, in the case of a time series, a specific time step.

This mechanism computes attention scores by:

1. Calculating the dot product QK^T to measure similarity between queries and keys.
2. Scaling the result by $\sqrt{d_k}$ to prevent large values.
3. Applying the softmax function to normalize the scores into attention weights.
4. Weighting the value matrix V with these scores to focus on relevant tokens.

Multi-Head Attention extends this by performing multiple parallel attention operations, enhancing sequence modelling [13].

Unlike RNNs, which process sequences sequentially, Transformers lack inherent sequence order. Why positional embeddings are essential. One method is using sinusoidal positional embeddings:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where pos is the sequence position of the token, i is the dimension index, and d_{model} is the model's latent dimension. These embeddings encode positional context, enabling order-aware processing [13].

2.7.2 PyTorch TransformerDecoder

The PyTorch `TransformerDecoder` is a flexible architecture composed of a stack of layers, each integrating multiple attention heads and a latent dimension d_{model}, aligned with the dimensionality of entity embeddings in our TKG task [15]. The number of layers and attention heads are determined through hyperparameter tuning to optimize the trade-off between computational efficiency and the model's ability to model intricate temporal and relational patterns in TKGs.

Each layer comprises three primary sub-components:

- **Multi-head Self-Attention**: This mechanism enables the model to focus on different parts of the input sequence when predicting each output token. By leveraging multiple attention heads, it captures diverse contextual relationships across the sequence, with each head attending to a distinct subspace of the data. A causal mask ensures that only preceding positions influence the current prediction, maintaining the autoregressive nature of the task [13].

- **Multi-head Attention with Dummy Memory**: Typically, this layer incorporates context from an encoder's output. In our TKG setup, lacking an encoder, we employ a dummy memory tensor, allowing the decoder to operate independently using the input sequence alone. The attention heads here enhance the model's ability to extract varied relational insights from this sequence.

- **Feed-forward Network**: A position-wise feed-forward network follows the attention layers, applying a linear transformation and ReLU activation to each position independently. This sub-layer enriches the model's expressive power, enabling it to discern complex patterns within the TKG data.

The number of layers governs the depth of the model, with additional layers allowing for progressively abstract representations of temporal and relational dependencies. Meanwhile, the attention heads facilitate parallel processing of different aspects of the sequence, enhancing the model's capacity to learn multifaceted relationships. Layer normalization and residual connections stabilize training across this stack, supporting effective learning regardless of the tuned values for the number of layers and attention heads [13].

2.7.3 Multi-Head Latent Attention (MLA)

A central component of Transformer architecture is the MHA mechanism, which allows the model to weigh the importance of different tokens in the input sequence when generating representations. While highly effective, the standard MHA mechanism presents a significant bottleneck, particularly during the inference phase. This bottleneck arises from the Key-Value (KV) cache, where the computed key and value vectors for each token in the context must be stored to facilitate efficient generation of subsequent tokens. As the context length l increases, the size of this KV cache grows linearly ($O(l \cdot n_h \cdot d_h)$, where n_h is the number of heads and d_{model} is the dimension per head), consuming substantial memory resources and limiting inference throughput, maximum batch size, and the practical feasibility of handling very long sequences [16].

Several approaches have been proposed to mitigate this KV cache burden, such as Multi-Query Attention [17] and Grouped-Query Attention [18]. These methods reduce the cache size by sharing key and value heads across multiple query heads. However, as noted in the DeepSeek study, this reduction often comes at the expense of model performance, failing to fully match the capabilities of standard MHA [14].

To address the dual challenge of inference efficiency and model performance, the DeepSeek study introduces an innovative attention mechanism termed MLA. The core theoretical innovation of MLA lies in its low-rank joint compression of key and value representations.

Instead of computing and caching the full key and value matrices for each token, MLA employs a projection mechanism to compress the key and value information derived from the input hidden state into a shared, low-dimensional latent vector. This low-rank factorization effectively decouples the cache size from the number of attention heads, dramatically reducing memory requirements.

The introduction of MLA represents a significant theoretical advancement in designing efficient Transformer architectures. By fundamentally altering how key and value information is represented and cached, MLA provides a mechanism to drastically reduce the memory and computational overhead associated with long-context inference. The DeepSeek study empirically demonstrates that this theoretical innovation translates into substantial practical benefits, including a reported 93.3% reduction in KV cache size and a 5.76-fold increase in maximum generation throughput compared to a dense model of comparable capability, crucially, while maintaining strong model performance [14]. MLA thus offers a promising pathway towards building more scalable, economical, and performant large language models capable of processing extensive contextual information efficiently.

3

Defining Trends in a TKG Context

For effective prediction, a robust definition of an industry 'trend' within the TKGs is necessary. Defining a 'trend' in this context necessitates the consideration of multiple criteria. An ideal metric for describing a trend should be comparative, understandable, and actionable (inspired by Croll et al. [19]).

The comparative aspect requires that the measure of a 'trend' is easily comparable across entities, relations, and timesteps. This allows for analysis of how a 'trend' for a particular topic changes over time, with the ultimate goal of predicting future changes. Consequently, the metric must be quantitative rather than qualitative.

Understandability implies that the metric should be intuitively understood by individuals and exhibit a clear and logical connection to what an individual would consider a 'trend' in global markets or world events. Simplicity in terms of what the metric measures is therefore crucial.

Actionability means that the ability to predict changes in the metric provides value. For instance, the metric could indicate which topics will become more trending and thus likely more valuable in the future, thereby supporting investment and trading decisions. Alternatively, it could highlight markets likely to face turmoil, which would aid in credit underwriting and risk management decisions.

An exploratory approach is employed to identify possible metrics, keeping in mind that such a metric could utilize entities, relations, or a combination of information regarding both to create a measure of 'trendiness'. To this end, a dataset is initially investigated to determine what combination of its aspects can be used to discern patterns that align with the aforementioned criteria for a 'trend'.

We begin by investigating a weighted count that assigns weight to positive and negative relation types, hypothesizing that this constitutes a highly actionable metric given its capacity to indicate positive or negative movement. However, the interpretability and comparability of this metric are limited. Furthermore, it is important to note that it is only applicable to the FinDKG dataset, as it is the sole dataset that incorporates positive and negative relations. Data scarcity also emerges as a challenge when relation types are taken into account, resulting in a highly sparse knowledge graph upon filtering specific relation types. While these limitations are not ideal, we acknowledge their presence and proceed within their bounds for the purposes of this study.

To enhance understandability and comparability, we simplify the approach by omitting relation types to combat data scarcity and ensure applicability across all datasets. This simplification leads to a focus on examining the change in degree (number of relations for an entity) for each graph node between timesteps. This approach allows for a more generalized and robust analysis across different knowledge graph datasets. See Figure 3.1 illustrating this metric for the entity 'United Kingdom' in the ICEWS18 dataset.

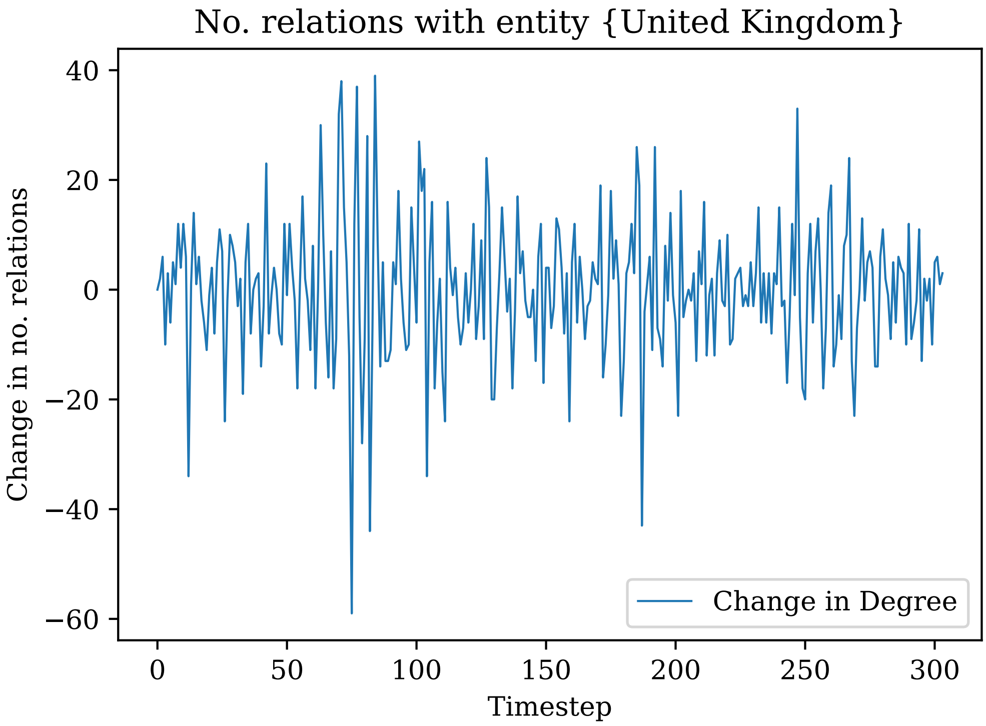

Figure 3.1: Figure illustrating the change in degree for a node in the ICEWS18 dataset

This metric also offers actionability and enhanced understandability, but it exhibits limitations in terms of comparability. For instance, a entity transitioning from a degree of 10 in the previous timestep to 15 in the current timestep demonstrates a change of 5. However, if another entity experiences a similar change from a degree of 90 to 95, the change is equivalent, despite the latter entity exhibiting a more prominent trend (higher degree overall).

To address these limitations, we further simplify the metric. Acknowledging that change, or the derivative, directly correlates with the degree of the entity, we can focus directly on the degree count itself. This simplification allows for a more direct and easily interpretable measure of entity prominence within the knowledge graph.

Figure 3.2 illustrates this metric for the entity 'United Kingdom' within the ICEWS18 dataset. By simply plotting the degree of the entity at each timestep, we can clearly identify periods of higher degree, indicating an abundance of news stories pertaining to the entity during those times, and periods of lower degree, signifying a scarcity of related news. This visualization enhances the metric's understandability and facilitates comparison, as the degree can be compared between entities, and the change or derivative can be calculated and compared as well. This comparative capability also makes it actionable. A model predicting directly on this metric is essentially forecasting the actual trend, whereas the change or derivative is a derived or causal metric of this trend.

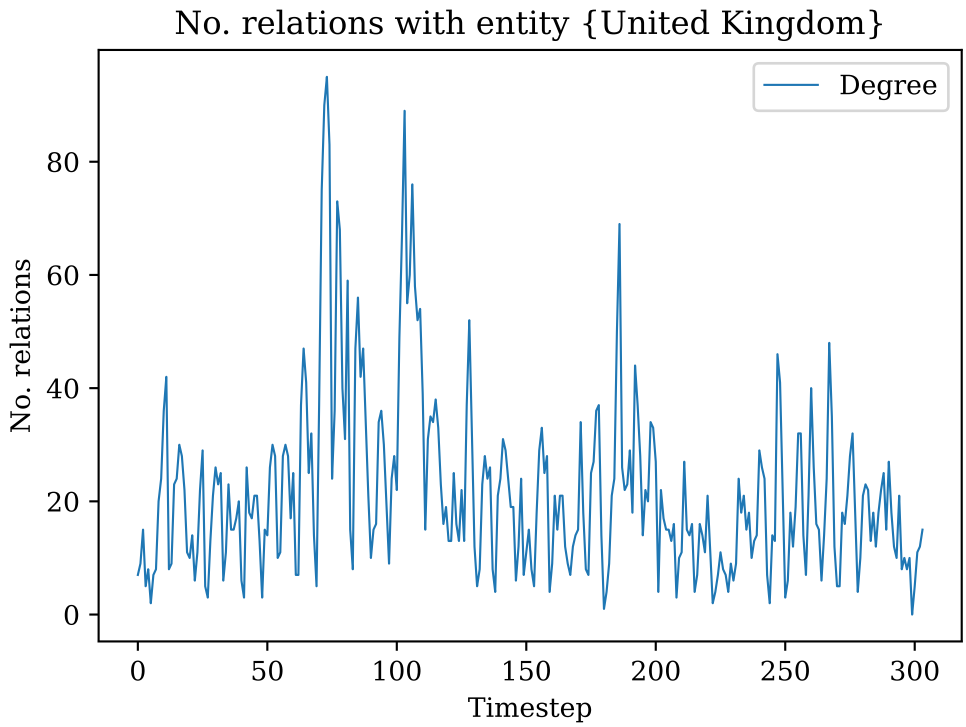

Figure 3.2: Figure illustrating the degree for a entity in the ICEWS18 dataset

The correspondence of this metric to a real-world perception of the trendiness of a particular topic is also logical and reasonable. Most individuals are likely to associate trendiness with topics receiving extensive news coverage, suggesting that this metric aligns with common understanding.

Therefore, we adopt this definition — entity degree — as our operational definition of 'trend' for the purposes of this study. This provides a clear and measurable basis for analysing and predicting trends within TKGs.

However, we remain cognizant that this definition is not without flaws. More complex metrics were briefly explored, but designing such a metric was subsequently deemed out of scope for this study. These, along with the the implications of the generalizations and simplifications made, as described in this chapter, are further discussed in section 11.5.

4

Exploring The Datasets

This section details the datasets employed in this study and analyses their fundamental characteristics. This analysis reveals limitations in current TKG datasets, suggesting improvements for future datasets to enhance TKG prediction performance.

There are six publicly available TKG datasets commonly used in previous works [1]: ICEWS18, ICEWS14, ICEWS05-15, GDELT, WIKI, and YAGO. This study utilizes the first four of these datasets, and it omits the latter two because they are not grounded in real-world events. Instead, they focus on updates of facts and links between Wikipedia articles.

Additionally, the FinDKG dataset [4] is used. This dataset incorporates financial concepts as entities, such as 'interest rate' and 'energy prices', and concentrates exclusively on financial news. A distinguishing feature of this dataset is the categorization of all entities, a feature absent in the datasets mentioned earlier. For example, 'Donald Trump' is classified as a 'person', while 'Amazon' is categorized as a 'company'. However, this categorization is not explicitly incorporated into the model.

The first four datasets are used as provided by the RE-GCN repository [1], and the FinDKG dataset is used as provided by the FinDKG repository [4]. The FinDKG dataset is available in two sizes, both of which are used in this study (FinDKG and FinDKG-full). Table 4.1 summarizes key information about the datasets. It is worth noting that ICEWS05-15 and GDELT have significantly more timesteps than the other datasets. However, ICEWS05-15 and ICEWS14 also exhibit fewer edges per timestep on average, which results in a sparser knowledge graph at each timestep.

The datasets are split into training, validation, and testing sets. This splitting adheres to a chronological sequence, with the validation set succeeding the training set, and the testing set following the validation set.

Table 4.1: Table listing key characteristics for the datasets used in the study

	ICEWS18	**ICEWS14s**	**ICEWS05-15**	**GDELT**	**FinDKG**	**FinDKG-full**
No. Entities	23033	7128	10488	7691	13645	13645
No. Relations	256	230	251	240	15	15
No. Edges (Training)	373018	74845	368868	1734399	119549	222732
No. Edges (Validation)	45995	8514	46302	238765	11444	9404
No. Edges (Testing)	49545	7371	46159	305241	13069	10013
No. Timesteps (Training)	240	304	3243	2138	100	234
No. Timesteps (Validation)	30	30	404	265	13	13
No. Timesteps (Testing)	34	31	370	348	13	14
Ave. No. Edges/Timestep	1541	249	115	828	1143	928
Timestep	24 hours	24 hours	24 hours	15 min	1 week	1 week

4.1 Data Analysis

We conduct preliminary data analysis focusing on the degree values within the datasets to assess the feasibility of making predictions on the degrees, based on our established definition of a trend.

Table 4.2 presents the mean degree value and variance for each entity in each dataset. The mean and variance remain generally stable across the training, validation, and testing data splits, with the exceptions of FinDKG and FinDKG-full. The variance in FinDKG-full is especially notable. This discrepancy suggests dissimilarities between the training, validation, and testing data. Consequently, predicting on the validation and testing data based on insights gained from training on the training data may prove challenging.

Table 4.2: Table listing degree mean and variance for the datasets used in the study

Dataset	Split	Mean	Variance
ICEWS18	Training	0.13496	0.61044
	Validation	0.13313	0.49957
	Testing	0.12652	0.43878
GDELT	Training	0.21095	1.61792
	Validation	0.23430	1.94864
	Testing	0.22809	1.74127
ICEWS14s	Training	0.06908	0.19175
	Validation	0.07963	0.22051
	Testing	0.06672	0.17002
ICEWS05-15	Training	0.02169	0.06474
	Validation	0.02186	0.05559
	Testing	0.02379	0.06605
FinDKG	Training	0.17523	0.84071
	Validation	0.12903	0.66104
	Testing	0.14735	0.51677
FinDKG-full	Training	0.13951	0.97596
	Validation	0.10603	0.30690
	Testing	0.10483	0.41158

To quantify the differences between the training, validation, and testing data, a two-sample KS test is employed for each entity's degree time series across these splits.

Table 4.3 shows the mean statistic and p-value for each split pair within each dataset. The KS statistic represents the maximum distance between the CDFs of the two samples. A larger statistic indicates a greater difference between the distributions. The p-value is the probability of observing a KS statistic as large as or larger than the one calculated, assuming that the two samples originate from the same distribution. The mean p-values for all datasets are high, with the lowest value of 93.5% observed in the GDELT dataset. These high p-values suggest a considerable similarity between the training, validation, and testing data splits for each dataset.

However, it is important to note that neither the mean, variance, nor KS test considers the temporal aspect of the data.

Table 4.3: Table listing the KS statistics and p-value for each dataset's training, validation, and testing data splits

Dataset	Split	Mean Statistic	Mean p-value
ICEWS18	Train vs. Valid	0.02799	0.97252
	Train vs. Test	0.02792	0.96877
	Valid vs. Test	0.03074	0.97429
GDELT	Train vs. Valid	0.01280	0.94014
	Train vs. Test	0.01222	0.93482
	Valid vs. Test	0.01350	0.95234
ICEWS14s	Train vs. Valid	0.02507	0.97380
	Train vs. Test	0.02353	0.97831
	Valid vs. Test	0.02643	0.98056
ICEWS05-15	Train vs. Valid	0.00709	0.96646
	Train vs. Test	0.00802	0.96228
	Valid vs. Test	0.00550	0.98319
FinDKG	Train vs. Valid	0.05177	0.96102
	Train vs. Test	0.06242	0.94192
	Valid vs. Test	0.04888	0.97147
FinDKG-full	Train vs. Valid	0.04780	0.96536
	Train vs. Test	0.04760	0.96164
	Valid vs. Test	0.03512	0.97817

Lastly, Figure 4.1 illustrates the proportion of non-zero degree values in each timestep. For all datasets, the mean number of non-zero degrees per timestep remains below 7%, with ICEWS05-15 exhibiting the lowest proportion at just 1%. This indicates that, in the training data for ICEWS05-15, an average of 99% of all entities in each timestep has a degree of 0. FinDKG emerges as the 'densest' graph, with over 6% of all entity degrees being non-zero (if we use degree as a measure of graph density).

4.2 Selecting Evaluation metric

The presence of a significant proportion of zero-value degrees can skew or noise into the evaluation metrics employed to assess the models' performance. To mitigate this issue, the models are configured to compute the RMSE error for zero-valued and non-zero targets independently. Figure 4.2 confirms the initial hypothesis, show-

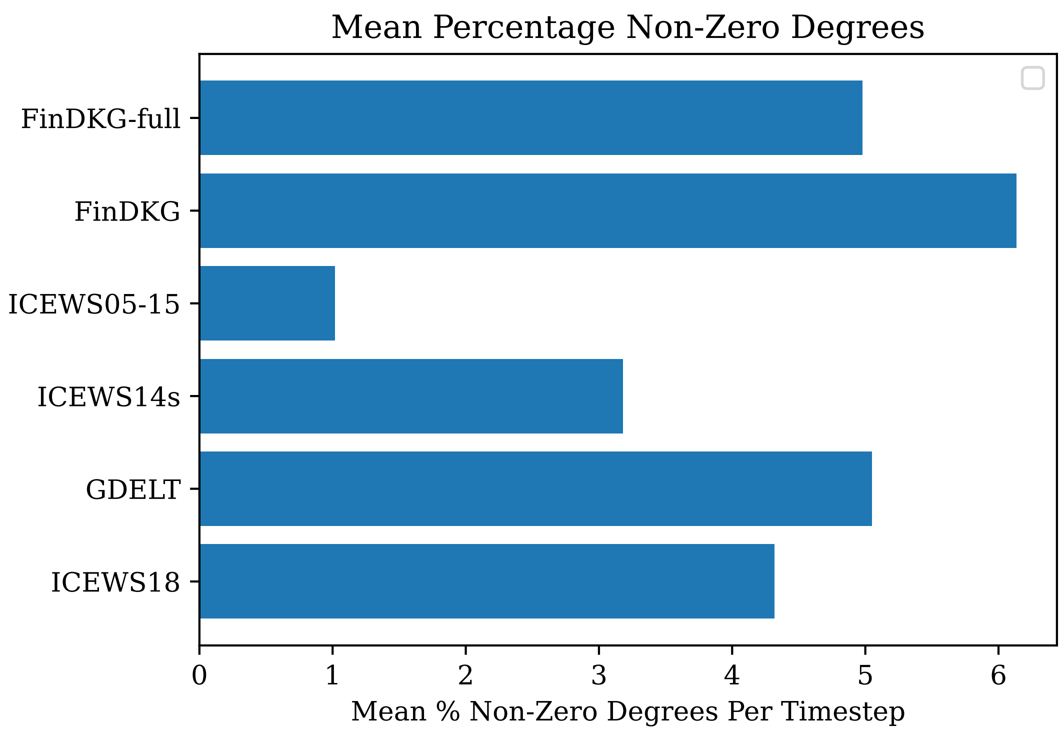

Figure 4.1: Figure showing the mean share of non-zero degree values for each timestep, for the training data of each dataset

ing that the combined RMSE is noisy and unstable, caused by the corresponding characteristics for zero-valued targets.

Consequently, the RMSE error for non-zero-valued targets is selected as the primary evaluation metric for identifying the optimal epoch. This metric demonstrates greater stability and focuses on evaluating the predictions for entities with higher degree values, which is the main interest.

4.3 Exploring Dataset Category Aggregation

A common issue with many knowledge graph datasets is their sparsity (as illustrated in Figure 4.1), where a significant number of entities have few or no connections. This sparsity poses a challenge for machine learning models, as isolated or weakly connected entities provide limited contextual information, making it difficult to learn meaningful representations and generate accurate predictions. To address this, we explored an aggregation-based approach to enhance the connectivity of the graph and improve model performance.

For this purpose, we selected the FinDKG dataset, as it already included a set of predefined categories and covered a diverse range of topics. This made it a suitable candidate for evaluating the impact of entity aggregation on graph density and model effectiveness. By grouping entities into broader categories, we aimed to

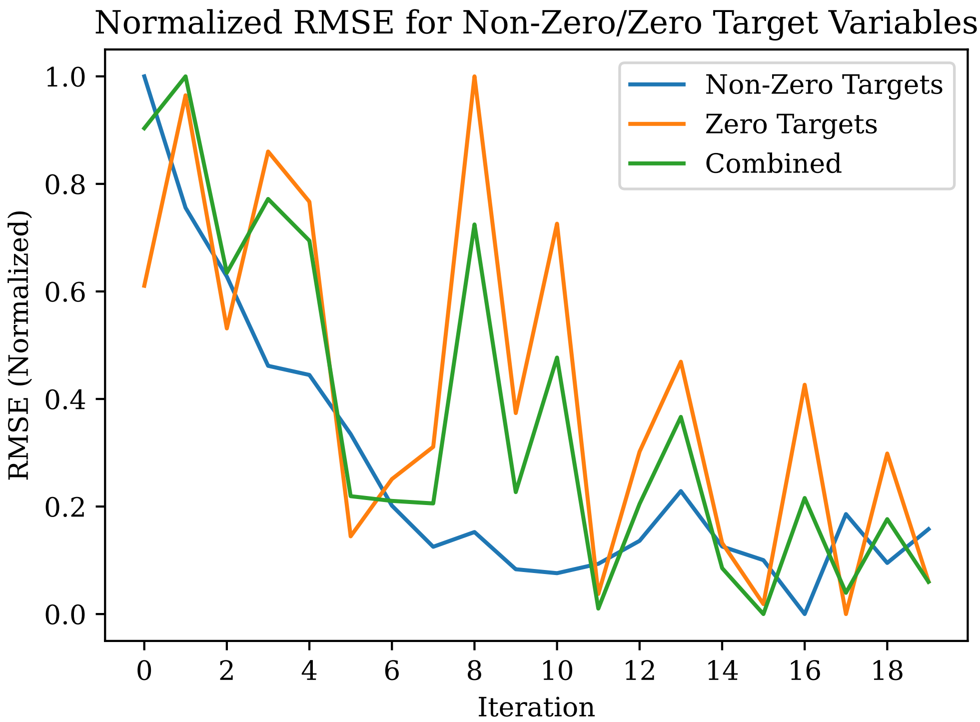

Figure 4.2: Figure showing how RMSE error for non-zero targets, zero targets, and their total differ

create a knowledge graph with fewer but more connected entities, thereby improving the overall structure and informativeness of the dataset.

To generate these aggregated categories, we utilized Gemini, an advanced language model capable of semantic clustering. By applying prompt engineering techniques, we ensured that the generated categories preserved meaningful distinctions between different types of entities. For example, we explicitly separated technology companies from financial institutions to maintain clear conceptual boundaries. This structured aggregation enabled entities sharing similar functional or industrial characteristics to be grouped under the same category.

However, a challenge with this approach is the difficulty of categorizing ambiguous or unconventional entities. Some entities do not fit neatly into predefined categories and may be incorrectly assigned. For instance, the term 'Thanksgiving' was misclassified as a financial concept entity, illustrating the limitations of automated category generation and the need for manual refinement in certain cases.

Once the aggregated categories were established, each entity in the dataset was assigned a corresponding label. This transformation significantly reduced the number of distinct entities while increasing the number of relationships per entity. Specifically, the aggregated version of FinDKG reduced the number of entities from 13625 to just 35 categories, leading to a more compact but well-connected knowledge graph. By minimizing the number of zero-degree entities and increasing relational density, this restructuring facilitates more efficient learning and enhances the model's ability to capture meaningful patterns.

By increasing the number of connections per entity, the aggregated dataset improves the model's ability to capture meaningful patterns and dependencies, as can be seen in Table 4.4.

Table 4.4: Table showing performance comparison of the aggregated and standard models on the FinDKG dataset

	Aggregated Model	**Standard Model**
Validation data:		
Normalized RMSE	0.6936	9.9598
Normalized RMSE (non-zero)	0.6601	1.2475
Training data:		
Normalized RMSE	0.9009	10.9705
Normalized RMSE (non-zero)	0.8489	1.7717

However, the effectiveness of this approach depends on the quality and relevance of the aggregated categories. If the chosen categories do not align well with the predictive task, the model's performance may suffer due to the loss of crucial distinctions between entities. This highlights the need for carefully constructed aggregation strategies tailored to the specific problem domain.

Despite this challenge, our findings reinforce the importance of knowledge graph density. A graph with an excessive number of sparsely connected entities can hinder model performance, as many entities contribute little meaningful information. By reducing the number of low-degree entities and increasing relational density, the aggregated dataset provides a more structured and informative representation, ultimately improving learning efficiency and predictive accuracy.

We build on these learnings by curating our own custom dataset, GDELT-big. See chapter 5.

5

Curating A Real-Time Dataset

Experimentation indicate that dataset aggregation (refer to section 4.3) represents a viable strategy for generating a denser graph, thereby enhancing model prediction performance with respect to evaluation error.

Furthermore, for a prediction model to possess practical utility from an SEBx viewpoint, it is crucial that the model operates on a live data stream. The GDELT dataset, as employed in this study, draws upon such a source[1]. We curate a custom dataset derived from the live version of the GDELT 1.0 Global Knowledge Graph (GKG).

This custom curated dataset spans the period from 2013-04-01 to 2025-04-14, incorporating training, validation, and testing splits analogous to those of the other datasets utilized. This dataset undergoes testing exclusively on our Custom Model, owing to its substantial size and consequent increased training runtime (refer to chapter 9).

5.1 Processing The GDELT 1.0 GKG

A data pipeline, written in Python, downloads snapshots from the GDELT 1.0 GKG[2], decompresses the data, merges it with a defined set of entities and relations, aggregates the relations per week, filters out unwanted data fields, and writes the dataset to text files matching the format of the other datasets used in this study. The use of a defined set of entities (consisting of all nations and Fortune Global 500 companies[3]) ensures that this pipeline is completely deterministic unlike FinDKG, which used an LLM-based pipeline for entity extraction [4]. Parallel computing and vectorized functions in `Pandas` and `Numpy` accelerate data processing, as necessitated by the massive size of the GDELT 1.0 GKG (the uncompressed size of the raw datafiles, for the entire time period, amount to 507.072 GB).

We denote this dataset 'GDELT-big' (available on Hugging Face[4]). Table 5.1

[1] `https://www.gdeltproject.org/`
[2] `https://www.gdeltproject.org/data.html#rawdatafiles`
[3] `https://fortune.com/ranking/global500/`
[4] `https://huggingface.co/datasets/andreas-helgesson/gdelt-big`

presents the characteristics of GDELT-big, mirroring the information provided for the other datasets in Table 4.1. This dataset exhibits a mean percentage of non-zero degrees of 22%, exceeding the other datasets used by a factor of three (refer to Figure 4.1).

Table 5.1: Table listing characteristics for the custom GDELT-big dataset

	GDELT-big
No. Entities	1302
No. Relations	5
No. Edges (Training)	388876937
No. Edges (Validation)	48609616
No. Edges (Testing)	48609617
No. Timesteps (Training)	483
No. Timesteps (Validation)	78
No. Timesteps (Testing)	68
Ave. No. Edges/Timestep	772808
Timestep	Week

In addition to this dataset, a pipeline is implemented that creates a real-time sub-dataset for evaluation and prediction on the live GDELT 1.0 GKG feed.

6

Implementing Linear Regression Model

To establish a baseline prediction model not using deep neural-networks, we implement a linear regression model. Expected to be trivial, this in practice poses a significant challenge due to three important aspects of the regression problem: the size of the datasets; the fact that the output variables are multidimensional, having one element for each entity's degree prediction; and the fact that these elements are not entirely independent.

How this challenge is overcome is discussed in more detail in sections 6.1 and 6.1.2. To start, we define the regression model in its theoretical form as referenced throughout this chapter:

- Let N be the number of nodes in the TKG
- Let T be the number of timesteps available in the training data for the TKG
- Let H be the history length, a parameter denoting how many past timesteps the model shall base its prediction on
- Let h be the horizon length, a parameter denoting how many timesteps into the future the model shall predict
- Let X be the design matrix, having dimensions $(T-h-H) \times (N \times H)$. Note that the last $h+H$ training samples must be omitted, as training targets are not available for this data. This is further explained in section 6.2.1
- Let Y be the target matrix, having dimensions $(T-h-H) \times N$. Note that the first $h+H$ training samples must be omitted, as training inputs are not available for this data
- Let β (beta) be the matrix of regression coefficients, having dimensions $(N \times H) \times N$
- Let ϵ (epsilon) be the matrix of regression error terms, having dimensions $(T-h) \times N$. ϵ will be random noise following some distribution

With this notation established, the linear relationships to be solved using a linear regression model can be described by equation 6.1, where β and ϵ are unknowns that training the model will find.

$$Y = X \cdot \beta + \epsilon \tag{6.1}$$

Note that this definition does not disallow prediction of negative degree values, which would be non-nonsensical. The implications of this is discussed further in section 11.2.

Lastly, it is important to note that neither X nor Y contain any information about the structure of the underlying TKG, other than the degree of each entity for each timestep. Generating embeddings of this data, compatible with linear regression, was deemed out of scope for this study. Instead, to incorporate the temporal aspect, the rows of X are created by concatenating the degrees of the past H timesteps. As such, the elements of X and Y can be denoted as follows:

- Let x_{ij}, the element of X at $[i, j]$, be the degree of entity $j \mod N$ at timestep i. We take $j \mod N$ as each row of X is a concatenation of N degree vectors, see section 6.2.1 for details on the construction of matrix X
- Let y_{ij}, the element of Y at $[i, j]$, be the degree of entity j at timestep i

6.1 Overcoming Implementation Challenges

Using off-the-shelf implementations of linear regression, such as scikit-learn's[1] `LinearRegression`[2], was not possible due to memory restraints. Similarly, off-the-shelf implementations of SGD regression proved unusable due to the interdependencies of entity degrees. How these challenges are overcome is described in further detail in section 6.1.1 and 6.1.2. In the end, we opted to create our own implementation of multi-output SGD regression.

6.1.1 Limiting Memory Usage

`LinearRegression`, in scikit-learn, requires the entire dataset to be in memory at the same time. This poses a significant challenge memory-wise, as we in this study therefore need the following items needed to be kept in memory simultaneously:

- Design matrix X
- Target matrix Y
- Coefficient matrix, having size $N \times H$

[1] `https://scikit-learn.org/stable/`

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression`

- Intermediate coefficients matrix, used during training, having a maximum size of $(N \times H) \times (N \times H)$

As such, the maximum total memory required, assuming the data is stored in NumPy[3] arrays as 64 bit floating point values, is approximately:

$$\text{size}(X) + \text{size}(Y) + \text{size}(V) + \text{size}(W)$$

where V and W are the coefficient and intermediate coefficient matrices, respectively, and *size* refers to the memory required to store each matrix. This is equivalent to:

$$((T-h) \times H \times N + (T-h) \times N + H \times N + (H \times N)^2) \times 8 \text{ bytes} \tag{6.2}$$

For the dataset ICEWS14s, with history length 10 and a horizon of 1, this equals 40.8 GB. For the dataset ICEWS18, with history length 10 and a horizon of 1, this equals 424.9 GB. While the former might be accommodable, the latter is not in the scope of this study as it requires access to computer systems highly specialized for memory-heavy computations.

Considering equation 6.2, it is clear that the term $(H \times N)^2$ dominates how memory usage will increase with increasing history length or number of entities. Plotting memory usage as a function of history length and the number of entities in the dataset, see Figure 6.1, makes it clear that even with a modest history length, memory requirements are unfeasible for larger datasets. Recall that ICEWS18, the dataset with the largest number of entities, has $N = 23033$ (see Table 4.1).

Therefore, typical off-the-shelf linear regression models, such as scikit-learn's `LinearRegressor`, are not usable for this study. However, SGD regression can be used to mimic linear regression if mean squared error is used as loss function.

Scikit-learn's implementation, `SGDRegressor`[4], does not require the entire dataset to be in memory at once and can train in batches. However, this implementation does not support multi-output regression making it also unusable in this study - see section 6.1.2.

6.1.2 Enabling Multi-Output Regression

The linear regression model has to output one predicted degree for each entity in the graph, meaning the target matrix Y has dimensions $(T-h) \times N$. Scikit-learn's `SGDRegressor`, however, expects the target vector Y to be one-dimensional and

[3] `https://numpy.org/`

[4] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor`

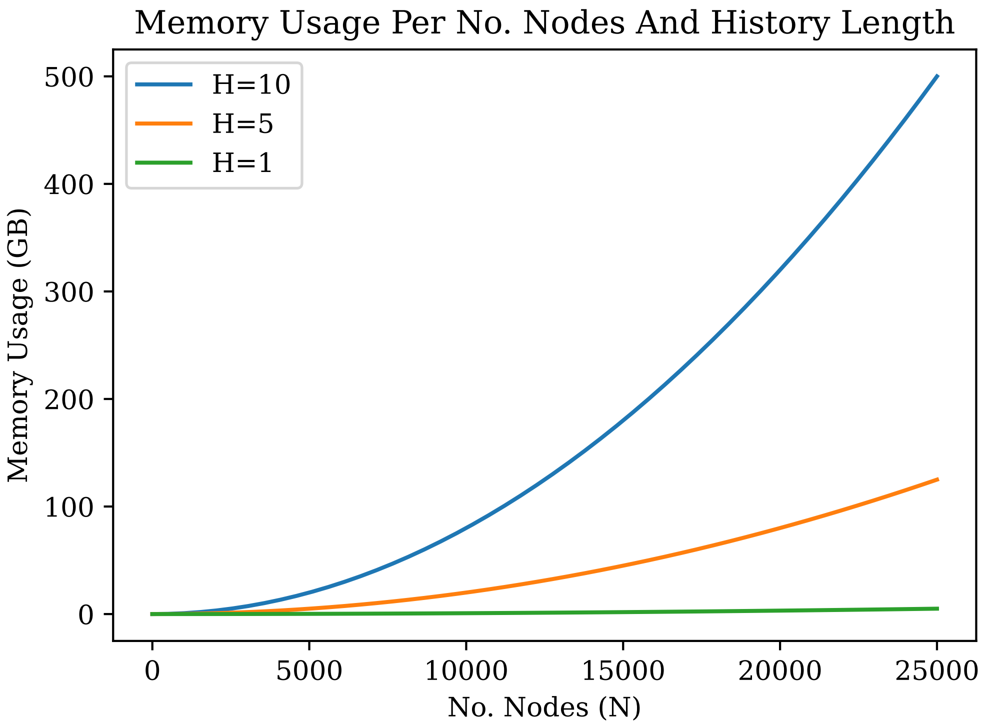

Figure 6.1: Figure illustrating how memory usage for linear regression increases with the term $(H \times N)^2$

cannot natively handle multi-output regression. Scikit-learn instead provides two possible workarounds: `MultiOutputRegressor`[5] and `RegressorChain`[6].

`MultiOutputRegressor` is an aggregate model that works by training one regression model of choice per output variable, in our case one model for each entity in the target Y. This approach comes with some downsides, the first being computation. For the dataset with the fewest number of entities, ICEWS14s, `MultiOutputRegressor` trains 7128 instances of `SGDRegressor` - one for each entity. Even when utilizing parallelization, this would require significant computational power. Likewise, the trained `MultiOutputRegressor` model would be extremely large as each trained `SGDRegressor` has its set own weights and biases that needs to be stored.

The second challenge is that of interdependence in the target variables in Y. The entire premise of this study is that the future degree for entities are dependent of each other based on the underlying TKG structure, and as such we can not assume that the output target variables were independent. Because `MultiOutputRegressor` trains each model separately, target variable interdependence is taken into account - making this alternative unsuitable. An alternative approach, which does take interdependency into account, is to use `RegressorChain`. However, `Regressor-`

[5]`https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor`

[6]`https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.RegressorChain`

`Chain` requires the interdependencies to be sequential, such that that $y_{i,n}$ depends on $y_{i,n-1}$, and $y_{i,n-1}$ depends on $y_{i,n-2}$ and so on. This is an assumption we can not make for our datasets, as the entities are interconnected without any such inherent ordering. Thus this alternative is also not suitable.

Therefore, we implement our own version of SGD regression that does support a multidimensional target matrix Y.

6.2 Implementation

Based on the considerations described in section 6.1, we implement a custom multi-output SGD regression model. This section outlines how the implementation was done, and how the datasets were prepared for the finished model.

6.2.1 Loading The Data

Before implementing and training the model, the matrices X and Y are assembled to correctly capture entity degrees, history length, and horizon. We accomplish this by first loading in the datasets, calculating the entity degrees for each training timestep, and storing these in a matrix `degrees` having dimensions $T \times N$. X is then assembled as:

$$X = \begin{bmatrix} X_{0,*} \\ X_{1,*} \\ \vdots \\ X_{T-h,*} \end{bmatrix} \in R^2$$

where each row $X_{i,*}$ is the concatenation of the degree vectors of the past H timesteps. Letting $||$ denote vector concatenation; such that for $a = [1, 2]$ and $b = [3, 4]$, $a || b = [1, 2, 3, 4]$; $X_{i,*}$ is then:

$$X_{i,*} = \mathop{\Big|\Big|}_{j=i-H}^{i-1} \texttt{degrees[}j, *\texttt{]}, \quad i = \{H, \ldots, T-h\}$$

Similarly, we assemble Y as:

$$Y = \begin{bmatrix} Y_{0,*} \\ Y_{1,*} \\ \vdots \\ Y_{T-h,*} \end{bmatrix} \in R^2$$

where each row $Y_{i,*}$ is:

$$Y_{i,*} = \texttt{degrees}[i-1+h,*], \quad i = \{H, \ldots, T-h\}$$

Figure 6.2 shows a visual explanation of this approach, ensuring that for each index i, $X_{i,*}$ is a concatenated list of training entity degrees for the H timesteps leading up to timestep i, and that $Y_{i,*}$ is a list of target entity degrees h timesteps ahead of i. Thus, the length of $X_{i,*} = H$ and $Y_{i,*}$ corresponds to the timestep h steps ahead of $X_{i,*}$. Listing 6.1 shows how this is computed in Python.

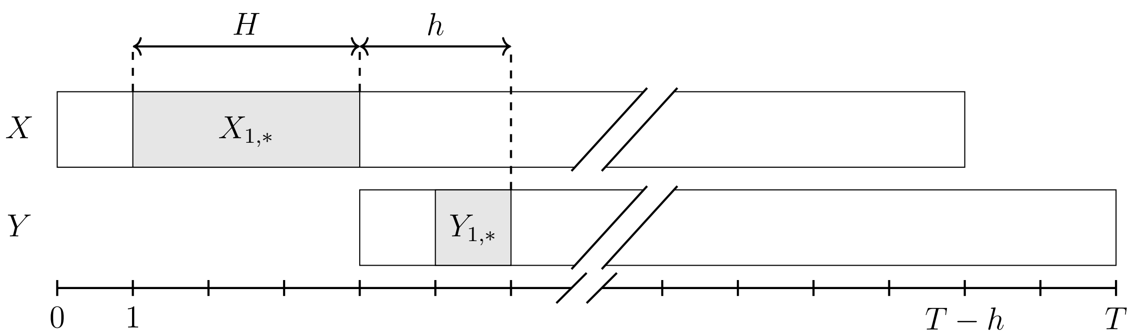

Figure 6.2: Figure illustrating how X and Y relate to each other, timestep-wise, in terms of horizon h, history-length H, and number of training timesteps T

Listing 6.1: Python code used to prepare design and target matrices X and Y.

```python
X = []
Y = []
for i in range(history, len(degrees)):
    if i-1+horizon >= len(degrees):
        continue
    X.append(concatenate(degrees[i-history:i]))
    Y.append(degrees[i-1+horizon])
```

6.2.2 Scaling The Data

A crucial aspect to consider when employing SGD regression is that of normalizing the data. Given that our data contains a majority share zero values (refer to Figure 4.1), we use scikit-learn's `MinMaxScaler` (which implements Min-Max Scaling), rather than `StandardScaler` (which implements Standard Score Normalization), due to our desire to:

- avoid numerical instability: `StandardScaler` centres the data around zero. When a dataset already has a large percentage of zero values, like ours, `StandardScaler` can introduce negative values and drastically scale the non-zero data points. This scaling can lead to very large or very small values, potentially causing numerical instability during the SGD optimization process. `MinMaxScaler`, on the other hand, maps the data to a specific range

([0, 1]), preserving the existing sparsity pattern and avoiding the creation of artificially extreme values.

- improve convergence speed: the compressed range of data achieved by `MinMaxScaler` can lead to faster convergence in SGD. The more data points that deviate far from the mean in unscaled data can cause slow convergence because the SGD method may be trying to account for outliers at each step.

- improve modeling results: because SGD is sensitive to initial conditions, `StandardScaler` introduces negative values and can cause issues that would not be present in the 0 to 1 range.

Although using `MinMaxScaler`, we remain cognizant that this method also comes with some downsides, such as:

- sensitivity to non-uniform distributions: if the data distribution is not consistent throughout all timesteps, `MinMaxScaler` might squeeze and distort the original relationships.

- scaling variability: if the validation or testing data follows distributions dissimilar to that of the training data, their data points might fall outside the original min-max range of the training data - leading to unsatisfactory model performance.

6.2.3 Implementing Custom Multi-Output SGD Regression Model

We create a custom implementation of SGD regression that mimicks linear regression and supports a multidimensional target matrix Y. Although SGD does not require all values to be in memory at any one time, it still needs to allocate large arrays to store the following data:

- Design matrix X

- Target matrix Y

- Weights matrix, having size $(N \times H) \times N$

- Biases matrix, having size $N \times 1$

As such, the maximum total memory required, assuming the data is stored in NumPy arrays as 64 bit floating point values, this is approximately:

$$((T - h) * H * N + (T - h) * N + N * H * N + N) * 8 \text{ bytes} \tag{6.3}$$

For the dataset ICEWS14s, with history length 10 and a horizon of 1, this equalls 4.3 GB. For the dataset ICEWS18, with history 10 length 10 and a horizon of 1, this equalls 42.5 GB. Although this is much less than the memory required for linear regression, as outlined in section 6.1.1, it is still too high to be easily accessible for a personal computer with typical hardware.

We therefore utilize Dask[7], a Python package for handling parallel computations on large scale data, to directly compute the RMSE on training and validation data after each training iteration - without ever storing the weights or biases matrices in their entireties. Dask achieves this by dynamically loading only the values required for computations into memory; any further explanation on how Dask works is outside the scope of this study. Because Dask can operate under an arbitrarily set memory limit, our implementation can easily scale to larger datasets without requiring additional memory - see Figure 6.3.

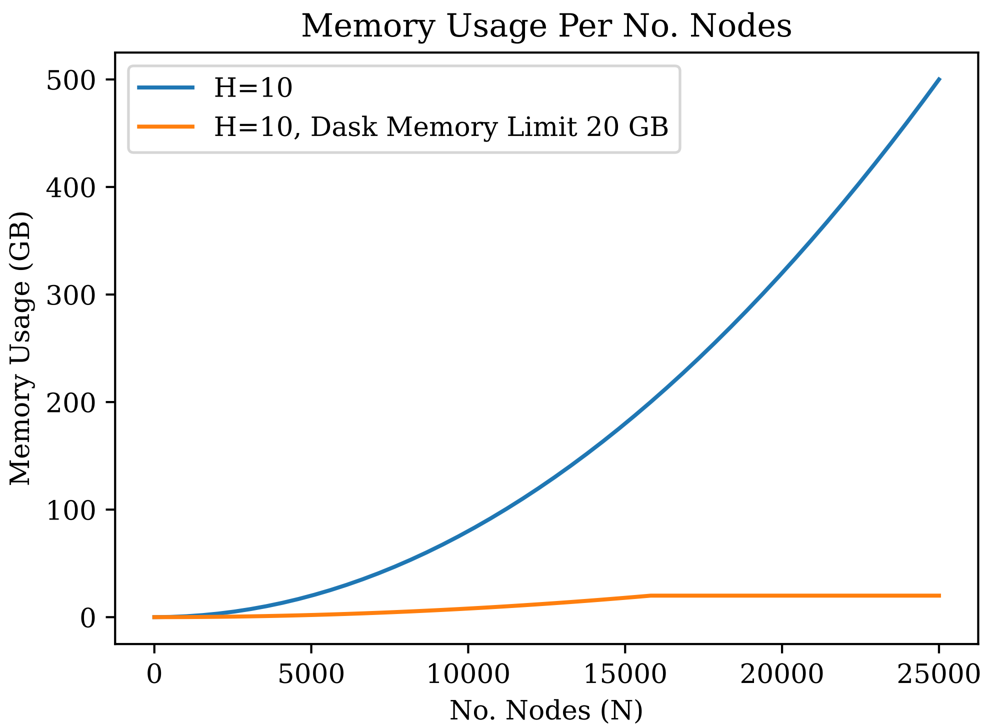

Figure 6.3: Figure illustrating how memory usage will scale for SGD regression using Dask with a set memory limit, compared to linear regression. Recall Figure 6.1

We implement the model in its own class, `MultiOutputSGD`, which performs SGD regression using mean squared error as loss function and without using L1 or L2 penalties. Therefore, this model produces same results as `LinearRegression`.

We implement `MultiOutputSGD` in such a way that training the model generates an output of RMSE values for the training and validation data after each training iteration. No actual computations are performed until `dask.persist()` or

[7]`https://www.dask.org/`

`dask.compute()` is called, after which Dask automatically parallelizes and performs the computations needed to generate these specific output values.

6.3 Results

The model is fitted using a virtual machine running in Google Cloud with 32 vCPU cores. See Figure 6.4 showing the RMSE non-zero errors for the validation and training data after each training iteration. Note that the RMSE non-zero values are based on the scaled data.

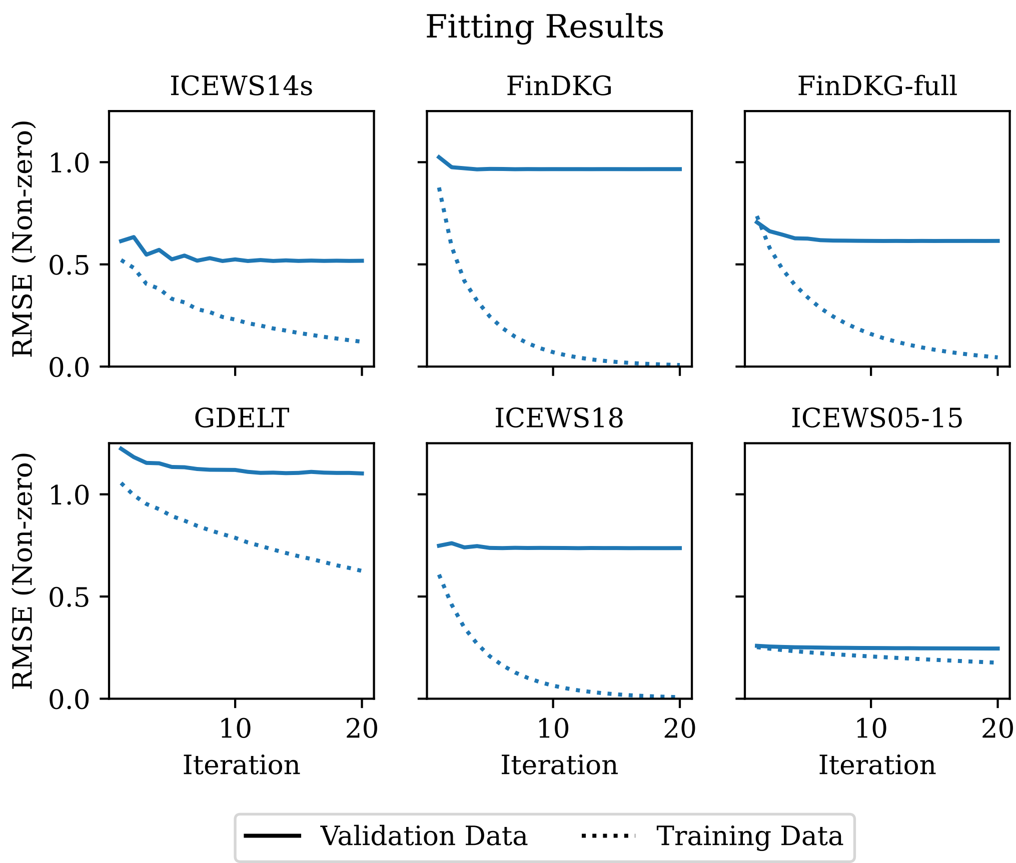

Figure 6.4: Figure showing the RMSE non-zero error after each fitting iteration of the baseline SGD regression model, when prediction horizon is 1

As illustrated by the figure, validation data evaluation error stabilizes in under 10 iterations for all models, with training data loss continuing to decrease. The dataset ICEWS05-15 stands out with no significant reduction in evaluation error being seen for either validation data nor training data after the first iteration, having stabilized from the outset.

Scaling the RMSE inversely, to match the original data as used in the subsequent

RE-GCN-based models (see chapter 7 and 8), results in Table 6.1 for the best performance after 20 iterations.

Table 6.1: Table listing the RMSE non-zero errors for validation data and training data on the linear regression mode

Dataset	Training (Non-zero)	Validation (Non-zero)
ICEWS18	0.681	66.315
GDELT	25.636	45.189
ICEWS14s	5.717	24.328
ICEWS05-15	9.871	13.755
FinDKG	1.357	171.952
FinDKG-full	9.505	129.724

7

Adapting Existing Machine Learning Model

In order to establish a performance baseline for neural-network-based machine learning models, we start by adapting an existing machine learning model used for predictions on TKG. We create two such baselines, the first being an unchanged model where we let it predict the edges in the next timestep and simply count the degrees from this prediction. This approach is presented in section 7.2.

Additionally, as the second baseline, we change the goal function of the model by simply changing the last layer in the decoder to a linear layer corresponding to the new goal function, as well as other necessary changes needed to facilitate this. This approach is outlined in section 7.4.1, with results presented in section 7.5.

7.1 Acquiring Existing Model

Initially we attempted to use FinDKG [4] code but found it too complex to easily implement our changes. Given that the FinDKG model is highly specialized for the particular task of of predicting edges and entities, we opted to try using RE-GCN [1] instead which we found has a much smaller and less complex codebase (16 files compared to 29, and 2318 lines of code compared to 4393).

The RE-GCN[1] repository was cloned to a Google Cloud Vertex AI Workbench running with a CUDA enabled GPU. Some packages needed to be updated and DGL[2] installed. With these minor changes, the RE-GCN model could be run without errors.

In addition to speed of deployment and development, RE-GCN has the added benefit of already running all the datasets ICEWS18, ICEWS14, ICEWS05-15, GDELT in addition to the FinDKG dataset. With FinDKG code, additional data processing would have been required to use these other datasets.

[1] `https://github.com/Lee-zix/RE-GCN`
[2] `https://www.dgl.ai/`

7.2 Unchanged RE-GCN Baseline

As a naive baseline, we implemented a process that leverages the original RE-GCN model's predictions. The initial step involves obtaining the RE-GCN's prediction at a given timestep, which forecasts the most likely entities and relationships associated with each node. Following this prediction, we construct a snapshot of the entire graph at that timestep. For each individual node in this predicted future graph snapshot, we then count the total number of edges connected to it. These degree values in the predicted snapshot serve as the final output. We used the default hyperparameter configuration of the RE-GCN as provided, without any tuning.

7.2.1 Results

This section presents the results obtained from the fitted Unchanged RE-GCN Model.

Table 7.1 presents the validation evaluation errors observed at the optimal epoch, which is selected from 15 training iterations conducted for each dataset. To conserve time, the model is not evaluated on either the training or testing data.

Table 7.1: Table listing the RMSE non-zero errors for validation data on the Unchanged RE-GCN Model

Dataset	Validation (Non-zero)
ICEWS18	125.039
GDELT	88.782
ICEWS14s	56.068
ICEWS05-15	41.032
FinDKG	97.946
FinDKG-full	68.254

7.3 Changing Goal Function

The original RE-GCN code employs two decoders: `ConvTransE` for entities and `ConvTransR` for relations. In an initial attempt to adapt the model, these decoders are replaced with a `DegreePredictor` class. This class features a single linear layer with an output dimension corresponding to the number of degree values to be predicted. Although this naive approach functions, its performance proves unsatisfactory.

Consequently, the `DegreePredictor` is superseded by a `ConvTransDegreePredictor` class. This new class mirrors the functionality of `ConvTransE` and `ConvTransR`, but incorporates flattening layers and an output linear layer similar to that in `DegreePredictor`.

Finally, the model code undergoes adaptation to utilize this new decoder. This entails implementing functions such as `_calculate_degrees` and `split_degrees-`

`_by_time`, that properly calculates the degree value vectors from the TKG so they can be fed to the model during training and evaluation.

7.3.1 Changing Loss Function

Given the modification of the goal function, a corresponding adjustment to the loss function is also necessary, reflecting the shift from a classification to a regression prediction task.

The selection of an appropriate loss function warrants careful consideration. MAE proves suboptimal in effectively penalizing substantial deviations, particularly for nodes exhibiting large degrees. Instead, a loss measurement is required that adequately scales to accommodate significant errors associated with large degree values, thereby ensuring sensitivity across the entire spectrum of degree magnitudes. It is crucial to penalize errors commensurate with their magnitude, especially when dealing with high degree values.

The prevalence of nodes with very low degrees introduces a challenge, as the primary interest in this study lies on nodes characterized by high degree values . Therefore, the loss function should prioritize accurate prediction for these nodes.

Considering the limitations of MAE, MSE or RMSE offer superior alternatives, as they impose a greater penalty on larger errors without introducing necessary complexity. Consequently, `MSELoss`, as implemented in PyTorch, is adopted as the loss function during model fitting.

7.4 Hyperparameter Tuning

See Table 7.2, describing each identified hyperparameter for the Adapted Model.

Hyperparameter tuning is performed on the ICEWS14s dataset. Given the extensive hyperparameter space, a complete grid search is computationally prohibitive. Therefore, the hyperparameters are partitioned into manageable subsets and tuned iteratively. For all datasets containing static graph information, this information is incorporated. The default dropout values from RE-GCN are employed. Furthermore, the default learning rate is used, unless there is evidence of non-convergence.

The optimal hyperparameters consist of a history size of 28; 2 layers with a dimension of 400; self-loop connections and layer normalization disabled; a static graph weight of 0.5, discount of 1, and angle of 5; and a convolutional decoder with 50 channels and a kernel size of 3.

7.4.1 RE-GCN Hyperparameters

The history length is initially tuned. History lengths of 1, 7, 28, and 56 are evaluated. These lengths correspond to 1 day, 1 week, 4 weeks and 8 weeks, respectively, reflecting the temporal granularity of the data. It is important to note that history

Table 7.2: Table detailing hyperparameters for the Adapted Model

Hyperparameter	Description
History	The number of historic timesteps, in addition to the current timestep, that the model should base its predictions on.
Layers	The number of hidden neural layers in the neural network model.
Dimension	The dimension (number of nodes) in each hidden layer in the neural network model.
Self-Loop	Boolean determining whether a self-referencing edge should be added to each node.
Layers Normalization	Boolean determining whether layer normalization should be applied during each forward pass.
Weight	Controls the contribution of the static graph's loss to overall loss.
Discount	Binary value controlling whether the angle should increase of time (1), or not (0).
Angle	The angle (in degrees) used in the cosine similarity constraint between the dynamic and static embeddings. It determines the threshold for how similar the embeddings need to be.
Channels	The number of output channels for the convolutional layers in the convolutional degree predictor.
Kernel Size	The size of the convolutional filter along the spatial dimension in the convolutional degree predictor.

length significantly impacts epoch runtime, with mean epoch times of 33.6, 86.7, 248.1, and 493.4 seconds, respectively. The history length of 28 continues to yield the lowest evaluation error, achieving a RMSE non-zero of 2.516 on the validation data. This value is adopted for subsequent experiments.

A grid search is conducted to determine the optimal number of hidden layers and their respective dimensions. The number of layers is varied among 2, 4, and 8, while the dimensions are tested with values of 200, 400, and 600. Among the nine possible combinations, the configuration consisting of 2 layers with a dimension of 400 produces the lowest evaluation error, resulting in an RMSE non-zero of 2.484 on the validation data. This architecture is therefore selected for continued use.

A grid search is performed to assess the impact of self-loop connections and layer normalization, with each being tested as either true or false. The configuration yielding the lowest evaluation error is both being false, resulting in an RMSE non-zero of 2.473 on the validation data. This configuration is therefore selected for continued use.

7.4.2 RE-GCN Static Graph Hyperparameters

A grid search is performed to optimize the weight and discount parameters. Weights of 0.25, 0.5, 1, and 2 are evaluated, while discounts of 0 and 1 are tested. Among the eight possible combinations, two resulted in unstable training and produced null results. It was identified that disabling self-loop connections and layer normalization led to these instability issues/non-convergence. The configuration yielding the lowest evaluation error was a weight of 0.5 and a discount of 1, resulting in an RMSE non-zero of 2.473 on the validation data. This configuration is therefore selected for continued use, and it is important to note that these were the default values in RE-GCN.

The angle parameter is then tuned, with values of 2, 5, and 10 being evaluated. An angle of 5 resulted in the lowest evaluation error, achieving an RMSE non-zero of 2.434 on the validation data. This value is adopted for subsequent experiments.

7.4.3 Convolutional Decoder Hyperparameters

A grid search is performed to optimize the number of channels and kernel size. Channels of 50, 100, and 200 are evaluated, while kernel sizes of 3 and 7 are tested. The combination yielding the lowest evaluation error is 50 channels and a kernel size of 3, resulting in an RMSE non-zero of 2.434 on the validation data. This configuration is therefore selected for continued use, and we note that these were the default values in RE-GCN.

7.5 Results

This section presents the results obtained from the fitted Adapted Model.

Table 7.3 presents the findings for the optimal epoch, selected from 15 training iterations for each dataset. A notable observation is that the FinDKG-full dataset exhibits a lower evaluation error on the validation set compared to the training set. This is not the case for any other dataset.

Turning now to computational efficiency, Table 7.4 lists the runtime associated with fitting and evaluating each model. The times reported for each epoch represent the average duration across all epochs for the respective dataset. This provides insight into the computational cost of training and deploying the Adapted Model.

Table 7.3: Table listing the RMSE non-zero errors for training, validation, and testing data on the Adapted Model

	Training	**Validation**	**Testing**
ICEWS18	3.218	3.510	3.506
GDELT	3.890	4.517	4.394
ICEWS14s	2.176	2.434	1.817
ICEWS05-15	2.060	2.087	1.842
FinDKG	3.464	4.252	5.424
FinDKG-full	3.282	2.963	5.442

Table 7.4: Table listing the mean runtime in seconds for fitting operations on the Adapted Model (epoch, and evaluation on training, validation, and testing data, respectively)

	Epoch	**Training**	**Validation**	**Testing**
ICEWS18	494.8	176.1	23.7	26.9
GDELT	3204.8	1282.8	165.5	217.5
ICEWS14s	273.5	100.9	10.9	11.3
ICEWS05-15	2598.4	1094.8	137.6	126.0
FinDKG	77.5	32.8	5.3	5.3
FinDKG-full	194.6	92.4	5.3	5.7

8

Implementing Custom Machine Learning Model

In this chapter, we present further refinements to our previously Adapted Model (see chapter 7) with the aim of enhancing its performance, specifically in terms of error minimization, and to explore the insights derivable from its output.

Initially, we investigate the impact of replacing the GRU component with a `TransformerDecoder`. Thereafter, we substitute the `TransformerDecoder` with an MLA Transformer. This alteration yields a negligible increase in the model's evaluation error, but a decrease in training time by 27.4% compared to `TransformerDecoder`.

Furthermore, we address the issue of data sparsity, characterized by a high frequency of zero values, by modifying the training loss function. Specifically, we replace the conventional MSE loss function with a Mean Quartic Error (MQE) function.

Following these architectural and functional modifications, we conduct hyperparameter tuning and train the model.

These modifications results in a mean evaluation error increase of 11.8% on the validation data and a mean evaluation error increase of 17.7% on the training data. Additionally, we observe a limitation in the model's ability to perform accurate regression, with the model tending to converge towards average degree values that minimizes the RMQE validation error across all timesteps. In chapter 9, to assess the model's potential utility in a different capacity, we reformulate it as a classification model it an attempt to improve its performance

8.1 Exploring Model Improvements

This section describes the changes implemented to improve the model's performance, specifically in terms of evaluation error minimization.

8.1.1 Replacing Gated Recurrent Unit (GRU) With Transformer

To augment our model with Transformer capabilities, we first implement positional encoding, a fundamental prerequisite for enabling the Transformer architecture. Two distinct strategies for Transformer integration are then explored. The first strategy leverages the readily available `TransformerDecoder` module from scikit-learn. The second strategy entails the implementation of a custom MLA Transformer, a technique gaining prominence and notably employed in the DeepSeek LLM [14].

Traditional temporal models for knowledge graphs typically employ a GRU to capture sequential dependencies. However, GRUs are inherently sequential and may struggle with long-range dependencies. To improve the model performance, the GRU is replaced with the specific Transformer implementation to leverage its self-attention mechanisms.

In this approach, the input to the Transformer is not the raw sequence of entity embeddings but the sequence extracted using the previously described model logic, which leverages convolutional layers from the RE-GCN. Unlike GRUs, which process sequences sequentially, Transformers lack built-in mechanisms to capture the sequential nature of the input. To address this, a positional encoding module is introduced to preserve order information, as described in section 2.7.1.

The precomputed positional encodings are incorporated into the model by summing them element-wise with the sequence of learned embeddings of the input tokens. This addition injects explicit information about the relative positions of elements within the sequence into the Transformer architecture. The implementation of this process in Python is detailed in listing 8.1.

Listing 8.1: Python class used for the positional encoding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-np.log(10000) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(0), :]
        return x
```

8.1.1.1 Using Off-The-Shelf Transformer

We replace the GRU with the `TransformerDecoder` module in PyTorch. The `TransformerDecoder` is a core element of the Transformer architecture, initially presented by Vaswani et al. [13]The PyTorch `TransformerDecoder` implementation adheres to the original Transformer design.

We apply the Transformer architecture exclusively to dynamic data, as applying the Transformer to the static graph components of the model does not offer a clear advantage and would be memory-intensive. The static graph portions of the codebase, therefore, remain unchanged. `TransformerDecoder` introduces two hyperparameters that need tuning: the number of layers and the number of heads.

8.1.1.2 Using MLA Transformer

Inspired by recent progress in latent attention mechanisms, specifically the work of DeepSeek [14], we explore the substitution of the `TransformerDecoder` layers with MLA layers, hypothesizing that MLA can reduce the computational cost associated with training, aligning with findings reported by DeepSeek.

The implementation comprises three primary classes: `DeepSeekDecoder`, which encapsulates several `DeepSeekDecoderLayer` instances, where each `DeepSeekDecoderLayer` contains a single `MultiHeadLatentAttention` layer. Within the overall model architecture, the conventional `TransformerDecoder` is directly replaced by the newly designed `DeepSeekDecoder`.

This modification introduces an additional hyperparameter: the latent dimension. While the `TransformerDecoder` is parameterized by the number of layers and attention heads, the introduction of necessitates specifying the latent dimension. This parameter is constrained to be a positive integer divisible by the number of attention heads, but is otherwise unconstrained.

8.1.2 Using different Loss Function

Recall Figure 4.1, illustrating that, on average, 93+% of all entities exhibit a degree of 0 per timestep. However, our focus lies primarily on outlier entities with significantly larger or rapidly growing degree values, as these are more relevant for trend prediction. Consequently, entities with a constant degree of 0 are of lesser interest. The abundance of zero values leads to a substantial cumulative loss, even if the loss per entity remains small, when training. We hypothesize that the model may be optimizing to minimize these numerous small losses associated with zero values, rather than focusing on reducing the larger losses from outlier values.

To address this issue, two distinct approaches are investigated. The first involves rounding all model predictions to the nearest integer, while the second explores altering the loss function employed during training.

Initially, rounding to the nearest integer was considered. The rationale is that any prediction within an absolute values less than 0.5 would be rounded to 0, effectively excluding it from the loss function calculation. For large degree values, the effect is negligible, as integer predictions are desired. This is implemented using `torch.round()` in PyTorch. However, this approach led to the model failing to learn during training, as the loss remained constant.

We realized that rounding values prevent the model from training on the corresponding entities, losing crucial information about the relationships between degree counts. Essentially, the model would only train on positive values, representing less than 7% of the total dataset. Consequently, the poor results obtained are not surprising.

As an alternative, the loss function is modified from MSE to MQE, which involves raising the loss to the power of four before calculating the mean. The cubic loss is avoided because it allows negative values to pass through, which would skew the mean calculation by lowering the mean loss value. This change is hypothesized to improve the loss metric in two ways: by emphasizing errors for larger degree values, which are more relevant for trend prediction; and by minimizing values less than 1, mitigating the issue of zero values dominating the total loss, similar to rounding only values close to zero towards zero.

8.2 Other Improvements

The substantial size of GDELT-big, relative to the other datasets employed in this study (compare Table 5.1 and 4.1) necessitates several code optimizations to mitigate GPU memory consumption and curtail model training time. To address the former, we implement a more efficient method for representing duplicate edges in memory and proactively clear the GPU cache. We reduce model training time by optimizing data read functions and pre-computing the TKGs. These optimizations result in a significant reduction in total runtime, achieving an 87.5% decrease for a 15-epoch training run using a history length of 52 on GDELT-big. While these enhancements are primarily motivated by the demands of training GDELT-big, the resultant speed improvements also benefit the training process for the other datasets.

8.2.1 Reducing Data Read In Times

The process of reading in data, calculating degree vectors, and splitting these vectors by timestep poses a significant bottleneck in achieving fast iterative development and hyperparameter tuning when working with GDELT-big. To address this, the data read-in function is optimized by reading the entire data file at once, instead of processing it line by line. Additionally, unnecessary type conversions are eliminated. This results in a substantial reduction of the total read-in time, decreasing it from 5 minutes and 35 seconds to 33.1 seconds.

Similarly, the calculation of degree vectors and their subsequent splitting by timestep is improved. The original for-loop logic is replaced with vectorized `NumPy` functions.

This modification drastically reduces the execution time of these methods from a total of 9 minutes and 40 seconds to 1 minute and 26 seconds.

8.2.2 Reducing GPU Memory Usage

GDELT-big poses significant GPU memory demands when operating with extended history lengths. Specifically, using a history of 52 weeks on an NVIDIA T4 GPU equipped with 16 GB of memory results in a CUDA out-of-memory error. Debugging reveals that memory fragmentation is the primary cause. This issue is mitigated by explicitly deleting unneeded torch variables, invoking the garbage collector, and calling `torch.cuda.empty_cache()` after each epoch step during training. While this approach resolves the memory fragmentation, it introduces a slight increase in training runtime. This adjustment is sufficient for training the Recurrent model on GDELT-big, but memory limitations persist when employing the `TransformerDecoder` or MLA architectures.

To facilitate training with `TransformerDecoder` or MLA, the representation of duplicate edges within the graph is modified. GDELT-big, unlike other datasets in this study, exhibits higher density and contains numerous duplicate edges, because each timestep represents the aggregation of a week's worth of news. For instance, if a timestep contains 100 identical relations, instead of adding 100 edges to the graph, a single edge is added with a `count` property set to 100. This approach yields a substantial reduction in graph memory sizes. Empirical analysis reveals that, for the arbitrarily selected timestep 1000, the 52 history graphs exhibit a reduction in the average number of edges per timestep from 28833 to 9171, a reduction of 68.2%.

Consequently, the average number of edges in the graphs is reduced by 68.2%, significantly decreasing the memory overhead associated with storing and processing the tensor representation of these graphs. To accurately account for the new `count` property, the message tensor is multiplied by the corresponding counts in the DGL graph, effectively encoding the duplicate vectors and ensuring equivalent matrix multiplication results.

While GDELT-big can now be trained on the orignal model more efficiently, training `TransformerDecoder` and MLA architectures necessitates additional optimizations. Initially, `DataParallel` from PyTorch was explored to distribute memory usage across multiple GPUs. However, the extensive modifications required to facilitate data transfer between devices rendered this approach infeasible within the given time constraints.

Subsequently, offloading the most memory-intensive computation, specifically `torch.mm` within the `msg_func` layers, to the CPU was investigated. Although the CPU offers significantly more memory than the GPU, this approach ultimately leads to out-of-memory errors when attempting to transfer data back to the GPU. However, this strategy is viable, albeit slower, with a 26-week history length.

Finally, the propagation operation is performed on each graph in the history separately, instead of processing all graphs concurrently. This solution proves effective.

In summary, GPU memory usage is reduced by modifying the representation of duplicate edges in the graphs, performing propagation on each history graph independently, and manually deleting unused torch tensors while clearing the GPU cache.

8.2.3 Reducing TKG Build Time

In RE-GCN, the function `build_sub_graph`, responsible for constructing the DGL TKG for historical timesteps, rebuilds historical graphs for each step within an epoch. Specifically, with a history length of H, each graph is constructed H times per epoch.

Empirical analysis confirms that this graph construction process constitutes a significant portion of the runtime when employing longer history lengths. In a test using timestep 1000 from GDELT-big, one epoch step with $H = 52$ consumes 5.8 seconds for building historical graphs, compared to 5.6 seconds for training the model weights. Therefore, historical graph construction accounts for over 50% of the total execution time for this step.

Attempts to optimize the `build_sub_graph` function using vectorized `NumPy` functions or `MultiThreading` did not yield meaningful reductions in runtime. The latter is likely due to the overhead associated with multithreading. Therefore, directly reducing the computation time for a single graph proved challenging.

To address this bottleneck, a strategy of pre-computing all graphs before training is implemented. This ensures that each graph is built only once, reducing the total time spent on graph construction by a factor of $1/H$. The pre-computed graphs are stored in a list, introducing minimal latency when indexing them during training.

The same pre-computation strategy is applied to the test function, accelerating evaluation by passing pre-computed graphs and preventing redundant computations.

Experimental results on GDELT-big with a history length of 52 indicate that without pre-computation, each epoch takes 9 hours and 46 minutes, 34 minutes for validation evaluation, and 91 minutes for training evaluation, for a total of 11 hours and 51 minutes. With pre-computation, the pre-computing phase takes 11 minutes and 12 seconds, followed by 50 minutes and 44 seconds per epoch, 2 minutes and 37 seconds for validation evaluation, and 35 minutes and 1 second for training evaluation, for a total of 1 hour 28 minutes and 22 seconds per epoch.

This represents a 87.6% reduction in runtime per epoch, plus the initial pre-computing overhead.

In a hypothetical scenario involving 15 epochs of training with a history length of 52 on GDELT-big, the pre-computation strategy achieves a 87.5% reduction in total runtime. This reduces the total training time from 7 days, 9 hours, and 45 minutes to 22 hours, 16 minutes, and 42 seconds.

8.3 Hyperparameter Tuning

In this section, we tune the Custom Model's hyperparameters. In addition the the hyperparameters used in the Adapted Model, recall Table 7.2, the Custom Model's Transformer layers introduce additional hyperparameters to tune. These are described in Table 8.1.

Table 8.1: Table detailing hyperparameters for the Custom Model, in addition to those described in Table 7.2

Hyperparameter	Description
Heads	The number of attention heads in the Transformer. Must be tuned for both `TransformerDecoder` and our MLA Transformer Decoder separately.
Layers	The number of attention layers in the Transformer. Must be tuned for both `TransformerDecoder` and our MLA Transformer Decoder separately.
Latent Dimension	In the MLA Transformer Decoder, the latent dimension is the dimensionality of the hidden representation that the model learns to encode the input data.

Hyperparameter tuning is performed on the ICEWS14s dataset. Given the extensive hyperparameter space, a complete grid search is computationally prohibitive. Therefore, the hyperparameters are partitioned into manageable subsets and tuned iteratively. For all datasets containing static graph information, this information is incorporated. The default dropout values from RE-GCN are employed. Furthermore, the default learning rate is used, unless there is evidence of non-convergence.

The static graph parameters are not tuned in this phase; instead, the parameters tuned in the Adapted Model are utilized to focus on tuning the Transformer components. Self-loop connections and layer normalization are kept activated due to observed instability issues when deactivated (recall section 7.4.2).

The identified optimal hyperparameters consist of a history size of 28, 2 layers with a dimension of 600, 50 channels and a kernel size of 7 for the convolutional decoder, and 8 heads and 2 layers in the `TransformerDecoder`.

The model architecture can use either `TransformerDecoder` or MLA. For MLA, the optimal configuration involves a latent dimension of 1200, 4 layers, and 8 heads. However, this configuration yields a evaluation error approximately 1% higher than that achieved with `TransformerDecoder`, and is therefore not used. Nevertheless, it is important to note that the epoch training time with this MLA setup is 27.5% lower than that of the best `TransformerDecoder`.

8.3.1 RE-GCN Hyperparameters

The history length is initially tuned. Although a history length of 28 proves optimal in the Adapted Model (see section 7.4.1), the potential for Transformers to leverage longer histories is hypothesized. Consequently, history lengths of 28 and 56 are evaluated. It is important to note that history length significantly impacts epoch runtime, with mean epoch times of 177.9 and 302.1 seconds, respectively. These times are substantially shorter compared to those observed when recurrent networks are used (refer to section 7.5). The history length of 28 continues to yield the lowest evaluation error, achieving a RMSE non-zero of 2.464 on the validation data. This value is adopted for subsequent experiments.

A grid search is conducted to determine the optimal number of hidden layers and their respective dimensions. The number of layers is varied among 2, 4, and 8, while the dimensions are tested with values of 200, 400, and 600. Among the nine possible combinations, the configuration consisting of 2 layers with a dimension of 600 produces the lowest evaluation error, resulting in an RMSE non-zero of 2.418 on the validation data. This architecture is therefore selected for continued use.

8.3.2 Convolutional Decoder Hyperparameters

A grid search is performed to optimize the number of channels and kernel size. We try 50, 100, and 150 for channels; and 3 and 7 for kernel size. A preliminary experiment with 200 channels resulted in an out-of-memory error, thus the maximum number of channels is capped at 150. The combination yielding the lowest evaluation error is 50 and 7, giving a RMSE non-zero of 2.362 on the validation data, so this is what we use hereafter.

8.3.3 Transformer Decoder Hyperparameters

A grid search is performed to optimize the number of heads and layers. We try 8 and 20 heads; and 2 and 4 layers. The combination yielding the lowest evaluation error is 8 and 2, respectively, giving a RMSE non-zero of 2.362 on the validation data, so this is what we use hereafter.

8.3.4 MLA Transformer Decoder Hyperparameters

We start by tuning the latent dimension. It is the hidden layers dimension, so 600, multiplied by a latent dimension factor. We try 2 and 4 for this factor, of which the former resulted in the lowest RMSE non-zero of 2.403 on the validation data.

A grid search is thereafter performed to optimize the number of heads and layers. We try 8 and 20 heads; and 2 and 4 layers. The combination yielding the lowest evaluation error is 8 and 4, respectively, giving a RMSE non-zero of 2.385 on the validation data.

8.4 Results

This section presents the results obtained from the fitted Custom Model.

The `TransformerDecoder` model achieved a lower RMSE non-zero errors during hyperparameter tuning than the MLA Transformer. Consequently, it is employed to evaluate the model's performance across all datasets. However, it is important to note that while the evaluation error for the `TransformerDecoder` is lower, it is only marginally so (1.0%), whilst its runtime is significantly higher, averaging 37.8% more per epoch compared to the MLA Transformer.

Table 8.2 presents the findings for the optimal epoch, selected from 15 training iterations for each dataset. A notable observation is that, in contrast to the Adapted Model (as discussed in chapter 7), the FinDKG-full dataset does not exhibit a lower evaluation error on the validation set compared to the training set. This suggests a potential difference in the Custom Model's generalization behaviour on this particular dataset when.

Table 8.2: Table listing the RMSE non-zero errors for training, validation, and testing data on the Custom Model

	Training	**Validation**	**Testing**
ICEWS18	3.727	3.599	3.691
GDELT	5.032	5.567	5.282
ICEWS14s	2.091	2.362	2.184
ICEWS05-15	2.045	2.032	2.214
FinDKG	3.191	4.350	6.520
FinDKG-full	3.283	4.398	6.543

Turning now to computational efficiency, Table 8.3 lists the runtime associated with fitting and evaluating each model. The times reported for each epoch represent the average duration across all epochs for the respective dataset. This provides insight into the computational cost of training and deploying the Custom Model.

8.4.1 Tendency of Learning Average

The model exhibits a tendency to learn an average value, which results in the lowest RMSE, without effectively predicting the specific degree values. Figure 8.1 illustrates this behaviour, demonstrating the model's propensity to predict nearly identical values at each time step, irrespective of the input history.

Several factors could contribute to this outcome. The model may be underfitting the data, indicating insufficient model complexity to capture the underlying patterns. Alternatively, the model might be unable to effectively learn from the dataset, suggesting potential inaccuracies in the assumptions made when defining trend.

Table 8.3: Table listing the mean runtime in seconds for fitting operations on the Custom Model epoch, and evaluation on training, validation, and testing data, respectively)

	Epoch	**Training**	**Validation**	**Testing**
ICEWS18	314.7	177.5	24.2	27.4
GDELT	2758.8	1339.7	170.2	223.5
ICEWS14s	477.7	177.3	19.1	19.7
ICEWS05-15	4885.3	2029.3	254.5	233.1
FinDKG	161.3	51.5	8.7	8.7
FinDKG-full	162.7	51.3	8.7	9.4

These findings suggest that the model is not particularly adept at regression tasks. Consequently, chapter 9 reformulates the problem as a classification task to assess whether the model demonstrates improved performance under this alternative framing.

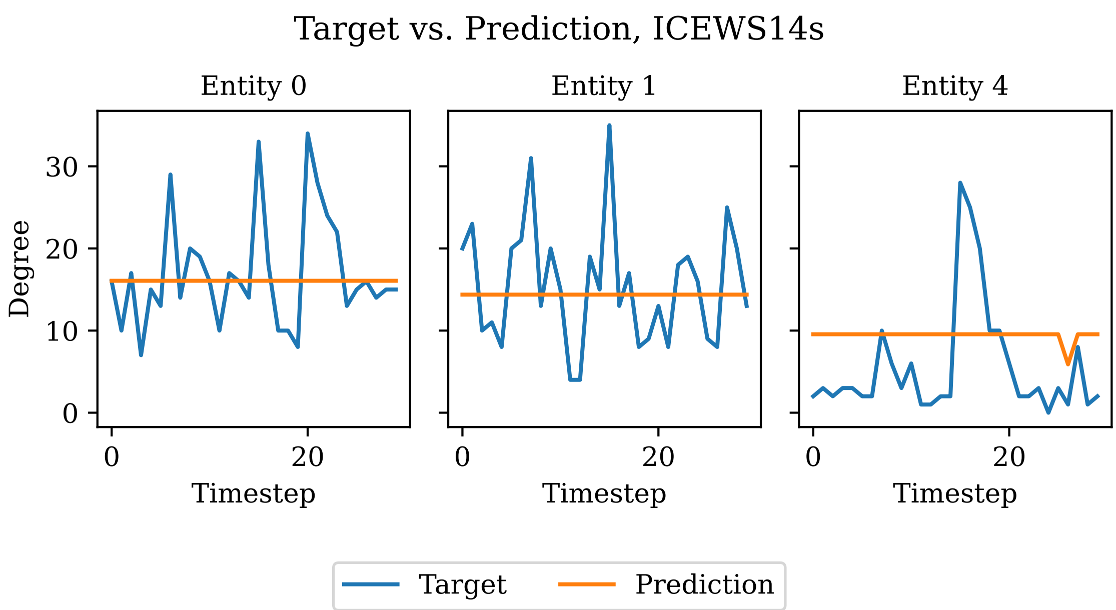

Figure 8.1: Figure showing the target degree and the corresponding model prediction

9

Changing Custom Machine Learning Model To Classification

Following the training of the Custom Model, as detailed in chapter 8, a limitation is observed in its ability to perform accurate regression. The model tends to converge towards average degree values that minimize the RMSE validation loss across all timesteps.

To assess the model's potential utility in an alternative capacity, it is in this chapter reformulated as a classification model to investigate whether this reformulation enhances performance. Instead of predicting the precise degree value, the model is tasked with classifying whether the degree value will increase, decrease, or remain constant in the subsequent timestep. While this represents an additional generalization of the original task, the model exhibits a more nuanced behaviour, avoiding the prediction of a single average value. The classification model is evaluated exclusively on the GDELT-big dataset and achieves an overall accuracy of 52%, which improves to a maximum of 92% when evaluated only on predictions with a confidence level exceeding 99%.

Model training and evaluation is limited to the GDELT-big dataset in the interest of time. Furthermore, this dataset is of particular interest for inclusion in the final model, as it provides the clearest use case value for SEBx given its live data feed. Because classification employs accuracy, instead of RMSE, to evaluate performance, this model's performance is not comparable to the regression models presented in previous chapters regardless.

9.1 Model Changes

The data loading methodology is refined to represent degree information as changes between consecutive timesteps rather than absolute degrees at each timestep. To facilitate this, a function, `calculate_trend_arrays`, is implemented. This function computes vectors containing one of three values that indicate whether the degree has increased ('UP'), remained the same ('NO CHANGE'), or decreased ('DOWN').

Subsequently, the model's training paradigm is changed to match the classification task. The return value for `ConvTransDegreePredictor` in RE-GCN is modified to be a PyTorch `log_softmax` layer. The training loss function is also changed to cross entropy loss. Furthermore, the evaluation function is adapted to use accuracy, precision, recall, and F1-score, metrics which are suitable for evaluating classification performance, instead of root RMSE.

To enable efficient training on the large GDELT-big dataset, a dynamic learning rate is employed. This learning rate automatically decreases when the training loss stop decreasing. The Python implementation of this dynamic learning rate adjustment is detailed in listing 9.1.

Listing 9.1: Python code for dynamically lowering the optimizer learning rate.

```
if len(epoch_mean_losses) >= 2:
    reg = LinearRegression().fit([[0],[1]],
       epoch_mean_losses[-2:])
    if reg.coef_[0] > 0: # If loss increasing
        for g in optimizer.param_groups:
            g['lr'] = g['lr'] * 0.1
            print(f"New learning rate: {g['lr']}")
```

9.2 Results

This section presents the results obtained from the fitted custom classification model. This model is fitted exclusively on the GDELT-big dataset.

Table 9.1 details the results of the fitted custom classification model. Additionally, the training time for each epoch is 540.5 seconds. We note that the accuracy, precision, and recall are all above 0.5 - indicating that the model performs better than random chance, which is expected to achieve results around 0.5.

Table 9.1: Table listing the results of the fitted custom classification model

Training Data Evaluation:	
Runtime (seconds)	443.9
Accuracy	0.546
Precision	0.554
Recall	0.597
F1 score	0.517
Validation Data Evaluation:	
Runtime (seconds)	68.9
Accuracy	0.519
Precision	0.525
Recall	0.578
F1 score	0.495

Figure 9.1 illustrates the relationship between the model's accuracy and the number of entities, as they vary with the confidence level. This means that by only trusting the model when its prediction reaches at least a specific confidence threshold, we can observe the effect on both the accuracy and the number of entities for which the model attains that confidence level.

These values are calculated empirically using testing data, and the resulting relationships are depicted in the figure. For instance, when we choose to act on the model's prediction only when it exhibits at least 75% confidence, the median case yields 130 nodes for which the prediction reaches that confidence level. Among these predictions, 81.1% are correct.

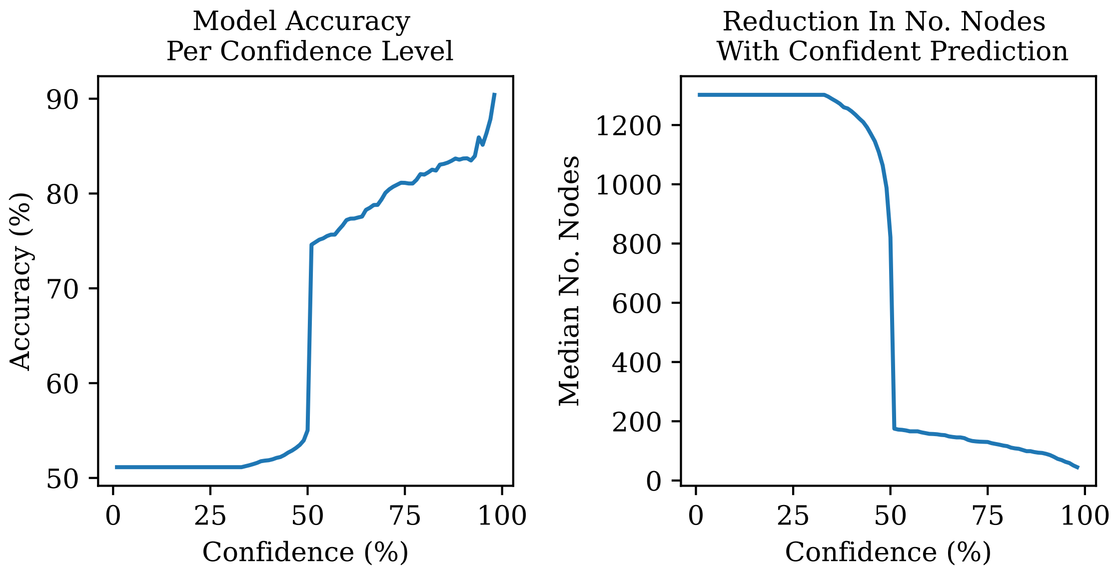

Figure 9.1: Figure showing model accuracy, as a function of confidence, on the GDELT-big dataset

10

Results

This chapter synthesizes the results obtained from each model, as detailed in the preceding sections on the baseline Unchanged RE-GCN Model (refer to section 7.2.1), the baseline Adapted Model (refer to section 7.5), the baseline Linear Regression Model (refer to section 6.3), and the Custom Model (refer to section 8.4). It then proceeds to compare their respective performances. The GDELT-big classification model, as presented in Chapter 9, is excluded from this comparative analysis due to its inherent nature as a classification model, which renders direct comparison with the regression models problematic.

Table 10.1 presents the RMSE-non zero error (evaluation error) for each fitted model. It is important to note that the Unchanged RE-GCN Model is not evaluated on the training data and that neither the Unchanged RE-GCN Model nor the Linear Regression Model are evaluated on the testing data, thus accounting for their absence in the table (this omission is due to time restrictions). A significant observation is the substantially lower evaluation error exhibited by the Adapted Model and the Custom Model on the validation data, when compared to the Unchanged RE-GCN Model and the Linear Regression Model. The reduction in evaluation error is in excess of 80% across all datasets. Additionally, the Linear Regression Model demonstrates a considerably lower error on the training data in contrast to the validation data, a phenomenon that potentially indicates overfitting.

Table 10.2 presents the mean runtime, measured in seconds, for fitting operations performed on each model across all datasets. The timing focuses solely on the Adapted Model and the Custom Model (using `TransformerDecoder`), as timing functionality remains unimplemented in the original RE-GCN and the Linear Regression Model. The Custom Model exhibits a reduced epoch time for the ICEWS18, GDELT, and FinDKG-full datasets, while demonstrating an increased epoch time for the ICEWS14s, ICEWS05-15, and FinDKG datasets. Furthermore, the evaluation runtimes for the Custom Model exceed those of the Adapted Model for all datasets, with the exception of training data evaluation on FinDKG-full.

The mean percentage change in runtime between the Custom Model and the Adapted Model for fitting operations reveals a decrease of -9.6% for ICEWS18s and -1.3% for GDELT. Conversely, increases of $+75.0\%$ for ICEWS14s, $+85.8\%$ for ICEWS05-15, $+72.3\%$ for FinDKG, and $+5.9\%$ for FinDKG-full are observed. Thus, ICEWS18s experiences the most substantial reduction in runtime, while ICEWS05-15 incurs the most significant increase when employing the Custom Model compared to the baseline Adapted Model.

Figure 10.1 further illustrates the difference in runtime by presenting the total time for training one epoch and evaluating the training and validation data, which occurs after each epoch. This figure facilitates a comparison between the Adapted Model and the Custom Model, and allows for an estimation of the runtime for the Custom Model using MLA, given it is 37.8% lower than for the Custom Model (as discussed in section 8.4).

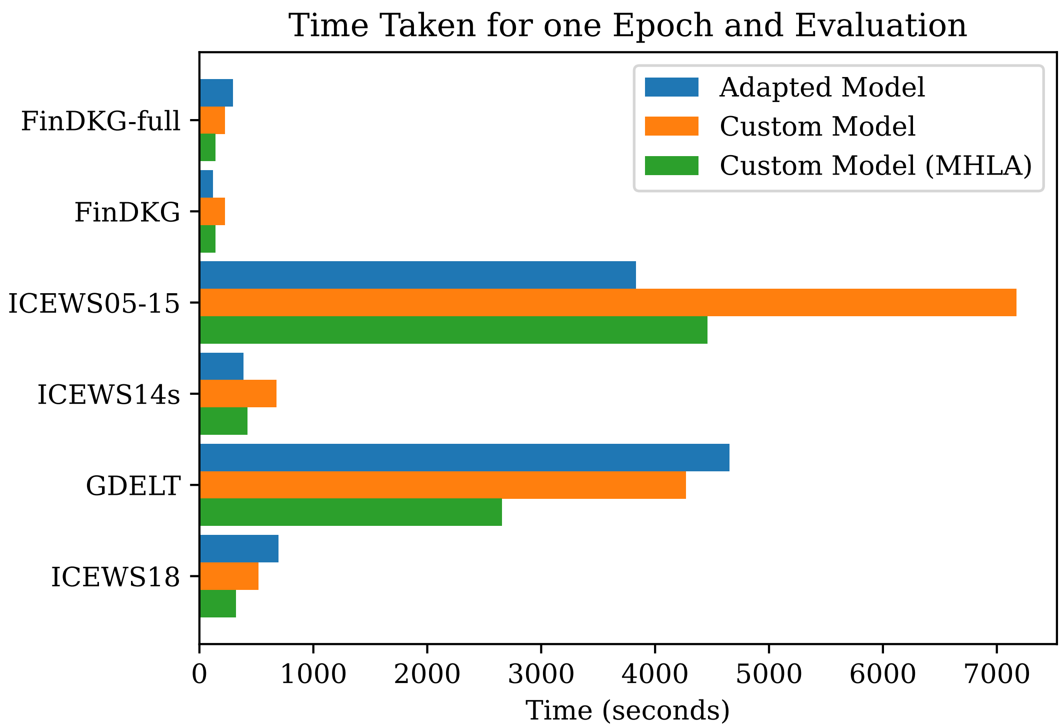

Figure 10.1: Figure showing the execution time for one training epoch and evaluation on training and validation data, for each model

A trend emerges from the data. For FinDKG-full, GDELT, and ICEWS18, the introduction of the Transformer in the Custom Model results in a reduction in runtime, a trend that continues with the MLA Transformer. However, this pattern does not hold true for FinDKG, ICEWS05-15, and ICEWS14s, where the Custom Model exhibits longer execution times than the Adapted Model. It is worth noting that for these latter datasets, MLA also demonstrates longer execution times compared to the baseline Adapted Model, suggesting a dataset-specific influence on the performance of the different model architectures.

Table 10.1: Table listing RMSE non-zero errors for each fitted model, for each dataset

	Training	**Validation**	**Testing**
Unchanged RE-GCN Model:			
ICEWS18	-	125.039	-
GDELT	-	88.782	-
ICEWS14s	-	56.068	-
ICEWS05-15	-	41.032	-
FinDKG	-	97.946	-
FinDKG-full	-	68.254	-
Adapted Model:			
ICEWS18	3.218	3.510	3.506
GDELT	3.890	4.517	4.394
ICEWS14s	2.176	2.434	1.817
ICEWS05-15	2.060	2.087	1.842
FinDKG	3.464	4.252	5.424
FinDKG-full	3.282	2.963	5.442
Linear Regression Model:			
ICEWS18	0.681	66.315	-
GDELT	25.636	45.189	-
ICEWS14s	5.717	24.328	-
ICEWS05-15	9.871	13.755	-
FinDKG	1.357	171.952	-
FinDKG-full	9.505	129.724	-
Custom Model:			
ICEWS18	3.727	3.599	3.691
GDELT	5.032	5.567	5.282
ICEWS14s	2.091	2.362	2.184
ICEWS05-15	2.045	2.032	2.214
FinDKG	3.191	4.350	6.520
FinDKG-full	3.283	4.398	6.543

Table 10.2: Table listing mean runtime in seconds for fitting operations on each model, for each dataset (epoch, and evaluation on training, validation, and testing data, respectively)

	Epoch	**Training**	**Validation**	**Testing**
Adapted Model:				
ICEWS18	494.8	176.1	23.7	26.9
GDELT	3204.8	1282.8	165.5	217.5
ICEWS14s	273.5	100.9	10.9	11.3
ICEWS05-15	2598.4	1094.8	137.6	126.0
FinDKG	77.5	32.8	5.3	5.3
FinDKG-full	194.6	92.4	5.3	5.7
Custom Model:				
ICEWS18	314.7	177.5	24.2	27.4
GDELT	2758.8	1339.7	170.2	223.5
ICEWS14s	477.7	177.3	19.1	19.7
ICEWS05-15	4885.3	2029.3	254.5	233.1
FinDKG	161.3	51.5	8.7	8.7
FinDKG-full	162.7	51.3	8.7	9.4

11

Discussion

This chapter delves into a comprehensive discussion of the results presented in the previous chapter. It aims to interpret these findings in the context of existing literature, highlighting their significance and potential implications for the broader field. We critically analyse the observed results, address any limitations of the study, and propose potential avenues for future research.

11.1 Datasets

The characteristics of a dataset significantly influence the efficacy of TKG trend prediction models. Our experiments with dataset category aggregation, as detailed in section 4.3, revealed that datasets with fewer entities, while maintaining a comparable number of edges and therefore exhibiting a higher average number of edges per node in each timestep, have a lower model evaluation error.

However, it is crucial to recognize that the selection of aggregation categories should align with the intended application of the model. For instance, in the context of financial news event prediction, as explored in our study, or in other domains, the choice of categories should be carefully considered. We did not delve deeply into this aspect, and thus, we propose that future research should focus on developing a comprehensive framework for selecting appropriate aggregation categories.

Regarding other dataset characteristics, we observed that datasets with a greater number of timesteps (spanning a longer time period), such as GDELT and ICEWS05-15, incurred significantly longer runtimes for fitting operations (epochs and evaluation). Importantly, the increased number of timesteps did not consistently translate into a substantial reduction in model evaluation error compared to datasets with fewer timesteps.

Furthermore, the performance of the Transformer model exhibited variability across datasets. While it demonstrated faster processing times on ICEWS18, GDELT, and FinDKG-full, it was comparatively slower on FinDKG, ICEWS05-15, and ICEWS14s. Surprisingly, we can not identify any apparent common factors among these two groups of datasets concerning the number of entities, relations, timesteps, or the average number of edges per timestep. The underlying cause of this runtime discrepancy therefore remains unclear and warrants further investigation.

In addition to the aforementioned characteristics, we acknowledge the existence of quality issues within the datasets, which, while not explicitly addressed in this report, deserve attention. For example, while FinDKG includes "Thanksgiving" as an entity, we find its significance in regards to global financial news events highly debatable. This observation underscores the inherent challenges in creating and sourcing high-quality TKG datasets. We recognize that there as a limited selection of TKG datasets available in the field.

Also, we find FinDKG particularly interesting due to its entity-specific category information. While we did not incorporate this information during model implementation in the current study, we believe it presents a promising avenue for future research. Integrating category information into the TKG embeddings could potentially mimic the effects of category aggregation without actually reducing the number of entities.

Finally, in an attempt to address the shortcomings of existing datasets, we explored an alternative dataset curation approach. We sought to mitigate issues related to non-sensical entities and aggregation categories by employing a pre-defined list of entities based in part on Fortune 500 companies. This resulted in the creation of the GDELT-big dataset, which exhibited an average number of edges per timestep more than three times higher than that of any other datasets. It was hypothesized that these modifications would enable models to achieve superior performance on GDELT-big compared to existing datasets, given the previously discussed observations. However, due to time constraints, we were unable to validate this hypothesis within the scope of this study. Notably, GDELT-big demands greater memory and computing resources due to its increased size.

11.2 Baseline Linear Regression Model

The convergence and runtime characteristics of SGD were carefully considered for the Linear Regression Model. Although SGD is expected to converge to a solution comparable to that of standard linear regression, we acknowledge the absence of a definitive guarantee. To ensure the consistency and reliability of our results, the algorithm could have been executed multiple times with varying initial conditions and learning rates, but this was not undertaken in the interest of time. Such an approach could have mitigated the risk of being trapped in local minima or being overly sensitive to initial parameters.

It is also crucial to emphasize that the design matrix X and the target matrix Y where intentionally constructed to exclude explicit structural information from the underlying TKG. Beyond the degree of each entity at each time step, the Linear Regression Model remains agnostic to the intricate relationships within the TKG. To incorporate this structural information, a TKG embedding compatible with linear regression would be required, a task deemed beyond the scope of the current study.

Nevertheless, we recognize this as a potentially fruitful direction for future research, promising to enrich the model's ability to capture complex temporal dependencies.

One limitation of the Linear Regression Model is its ability to generate nonsensical negative degree predictions. This arises from the inherent assumption of a linear relationship and the unbounded nature of the model. When extrapolating the linear trend beyond the observed data range, particularly into negative regions, the model may produce unrealistic predictions, especially when dealing with data characterized by natural boundaries. We acknowledge this limitation and recognize the need to explore alternative modelling approaches or data transformations that can effectively address this issue, although such measures were not implemented within the scope of this study.

Lastly, the choice of `MinMaxScaler` for data scaling was driven by the prevalence of zero values in our dataset, and our desire to prioritize stability, speed, and robustness. Unlike `StandardScaler`, `MinMaxScaler` preserves sparsity and prevents the creation of artificially extreme values by scaling the data to the interval $[0, 1]$. While this approach offers significant advantages, it is important to recognize its sensitivity to distribution shifts. However, our initial dataset analysis indicated that the training, validation, and testing sets exhibited similar data distributions, for all datasets, suggesting that the potential impact of this sensitivity on our results should be minimal.

11.3 Baseline Adapted Model

The baseline model, employing the original RE-GCN architecture to predict edges and subsequently derive degree values from these predictions, exhibited significantly inferior performance compared to its adapted counterpart. The Adapted Model, which retained the unchanged RE-GCN architecture but incorporated a modified goal function tailored to our regression task, demonstrated substantially improved results. Specifically, the Adapted Model achieved evaluation losses that were over 90% lower across all datasets, highlighting the effectiveness of aligning the objective function with the regression task.

Interestingly, the Adapted Model displayed a unique behaviour on the FinDKG-full dataset, where the validation loss was lower than the training loss. This phenomenon was not observed for any other dataset, and we have yet to determine the underlying cause for this discrepancy. Further investigation is warranted to understand the specific characteristics of the FinDKG-full dataset that may contribute to this behaviour.

Concerning the baseline Linear Regression Model, as detailed in section 11.2, we did not implement any safeguards against predicting negative values. While a well-fitted model should ideally avoid predicting negative values, this is not guaranteed. The inclusion of a mechanism to prevent negative predictions would have been beneficial to ensure the physical plausibility of the model's outputs.

11.4 Custom Model

The initial experiments revealed that the `TransformerDecoder` architecture facilitated a faster training time compared to the recurrently Adapted Model, maintaining identical settings. However, subsequent hyperparameter tuning involved increasing both the number of layers and hidden dimensions in the `TransformerDecoder`, which consequently led to a decrease in training speed. It is important to emphasize that when configured with the same number of layers and dimensions, the `TransformerDecoder` consistently demonstrated superior speed.

The MLA Transformer Decoder model, albeit only mentioned briefly, demonstrated a evaluation error only marginally higher than the standard `TransformerDecoder` model while offering significantly faster execution. This promising result suggests that further optimization could lead to an MLA Transformer Decoder achieving both a lower evaluation error than `TransformerDecoder` and a substantially improved runtime. Considering the considerable memory and runtime requirements of the GDELT-big dataset, the MLA Transformer Decoder is particularly appealing. Its enhanced efficiency has the potential to significantly mitigate the computational challenges associated with analysing such large-scale datasets.

Importantly, a crucial observation regarding the Custom Model's performance was its tendency to learn only an average degree value for each entity, rather than generating more insightful predictions. This issue could stem from several factors, including the possibility of model underfitting, necessitating a more complex architecture. Moreover, the validity of the trend definition itself may need to be reevaluated (as discussed in section 11.5). Notably, the Adapted Model, employing a GRU, achieved a lower evaluation error on most datasets compared to the Custom Model using a Transformer. Whether this outcome was due to random chance or an inherent advantage of recurrent layers for TKG trend prediction remains unclear, but it certainly merits further investigation.

Finally, the original research plan included estimating confidence intervals for the Custom Model to analyse the distribution of predictions and, more importantly, to assess how the confidence interval changes with varying model horizons. Considering the limitations of previous research, such as RE-GCN, in predicting more than one timestep into the future with satisfactory results, this analysis aimed to determine the prediction horizon beyond which the confidence interval becomes excessively large for reliable predictions. However, the observed behaviour of the Custom Model, which primarily learned an average value that exhibited minimal temporal variation, rendered this analysis less meaningful and led to its omission. Nevertheless, we believe that exploring the evolution of confidence intervals in TKG forecasting remains a valuable direction for future research.

11.5 Trend Definition

Defining a 'trend' within TKGs presents a multifaceted challenge, necessitating a careful balance between comprehensiveness and practicality. In our study, we adopted a singular trend definition, which, in retrospect, may have limited our exploration of the TKG's dynamic nature. A more robust approach could involve formulating a suite of plausible trend definitions, implementing each, and rigorously comparing the resultant model performances. This comparative analysis would offer valuable insights into which definition best captures the underlying temporal patterns relevant to our research questions. Part of the motivation for our choice of trend definition is that we wanted to begin with a simple definition. Given that the models have a hard time picking up this simplified trend and given the problems discussed about the low connectivity of the dataset, it is probable that other trend definitions would also see a similar model learning struggle.

The trend definition does, however, neglect several potentially informative aspects of the TKG structure. It does not account for the specific types of relationships, the temporal evolution of these relationships, or the diversity of an entity's neighbours. A more comprehensive trend definition would ideally incorporate a broader spectrum of these features, providing a richer representation of the TKG's dynamics. For instance, dividing the relation types into positive, negative, and neutral, is an alternative to the chosen trend definition that could be promising. The relationships did differ between the different datasets that we used, and some relationships had some ambiguity, why this was not pursued.

Furthermore, we posit that trends may not always manifest directly within the entities primarily associated with them. Instead, their influence might be indirectly reflected in the trendiness of related entities. For example, the increasing prevalence of AI might be more apparent through the rising prominence of company entities like Microsoft and Nvidia, rather than in the AI entity itself. This observation highlights the importance of considering indirect effects and network-wide patterns when interpreting model results and designing future investigations. Understanding these potential indirect manifestations is crucial for extracting meaningful insights from our models and for developing more nuanced trend definitions in the future.

The pursuit of maximal information capture must be tempered by the practical constraints of interpretability, actionability, and alignment with the model's intended use case. A trend definition, regardless of its sophistication, should facilitate meaningful comparisons, be understandable, and inform actionable insights. We acknowledge that our current definition, while serving as a starting point, likely falls short of these ideals. The very nature of trends in TKGs allows for numerous interpretations, and our chosen approach may not be optimal across all relevant criteria.

11.6 Loss Function

The unsatisfactory performance of the Custom Model suggests that employing the MQE loss function was inadequate to address the complexities arising from a substantial proportion of zero-valued degrees at each time step.

Although alternative strategies were considered, their thorough investigation fell outside the scope of this study due to time constraints. One such approach involved defining the loss function based on a selected subset of entities. This would entail using the entire TKG as model input, but restricting degree prediction to a specific group of entities deemed most relevant. Ideally, this subset would comprise entities exhibiting high connectivity and significant utility in the intended prediction task. By forcing the model to optimize the loss specifically for these entities, it may be possible to improve predictive performance by mitigating noise. This method shares conceptual similarities with the creation of aggregated categories tailored for specific prediction tasks, as discussed in section 11.1, as both approaches have the potential to enhance connectivity but necessitate task-specific customization to achieve optimal performance.

Another alternative, which was briefly explored, involved designing a weighted loss function that considers the types of relations connected to a entity, rather than simply the total number of relations. A relation indicating a positive impact would be assigned a positive weight, while a relation likely associated with a negative impact on the entity would receive a negative weight. Preliminary experimentation was conducted; however, the results did not demonstrate s substantial insight upside. We believe that a more refined design and further experimentation are necessary to fully realize the potential benefits of this approach.

11.7 GDELT-big & Benefits of Classification

The classification model fitted using the extensive GDELT-big dataset achieves accuracy exceeding 80% at certain confidence levels, strongly indicating the solvability of the classification problem. This suggests the model's ability to learn meaningful patterns, despite the limitations encountered previously in this study. Consequently, our definition of a trend, despite its shortcomings as discussed in section 11.5, proves predictable to a certain extent. Had the data been inherently unpredictable, the achievable accuracy would have likely been capped at approximately 50% (random guesses are either right or wrong).

These findings also suggest that a classification-based approach may be more effective than a regression-based one. The initial shift to regression aimed to enable sorting based on the magnitude of percentage change, facilitating the identification of the most actionable results. However, we now recognize that classification "buckets", such as "5%", "10%", and "20+%" increase and likewise for decrease, could have served a similar purpose in lieu of the simple "Up", "NO CHANGE", and "DOWN" categories we used. Exploring this alternative classification approach, preferably in conjunction with a revised trend definition, remains a promising avenue for future research.

Furthermore, we acknowledge that large datasets, such as GDELT-big, presented significant implementation challenges related to GPU memory constraints and data processing times. To facilitate faster iterative development in future work, adapting the codebase to support multi-GPU processing and distributed computing may be necessary. We also recognize that data ingestion can consume a significant amount of time and become a bottleneck for iterative development and improvement. Employing additional parallelized computing, vectorized functions, and pre-computing techniques could substantially reduce processing times further, thereby accelerating the iterative development process.

12

Conclusion

This thesis set out to investigate the feasibility of leveraging TKGs to predict future industry trends, shifting the focus from predicting specific relationships to identifying broader patterns. This approach was motivated by the inherent difficulties in predicting exact relationships, particularly over longer time horizons. We hypothesized that generalizing predictions to trends could yield better performance in capturing overarching industry shifts. The research was guided by three key questions: how can a trend be defined within the context of TKGs; can a trend-based prediction method achieve better accuracy than previous machine learning approaches, even with lower specificity; and how does the proposed trend-based method compare to traditional mathematical models?

Our exploration revealed several key insights and challenges. Defining a trend within a TKG proved to be a complex undertaking. The initial trend definition, while simple, may have been too limited to fully capture the dynamic nature of the graphs. As discussed in section 11.5, future research should explore a suite of plausible trend definitions, comparing their impact on model performance. Such definitions could incorporate relationship types, temporal evolution, and the diversity of an entity's neighbours, offering a richer representation of TKG dynamics.

The results highlighted the significant influence of dataset characteristics on model efficacy, as elaborated in section 11.1. Datasets with varying numbers of entities, relations, and timesteps exhibited different performance profiles. While datasets with more timesteps didn't consistently translate into reduced model loss, the variability in Transformer model performance across datasets suggests the existence of underlying factors yet to be fully understood. Furthermore, dataset quality issues, such as the presence of nonsensical entities and entities that in most timesteps have no connections, underscore the challenges of creating and sourcing high-quality TKG datasets. The overall connectivity density of the dataset have a significant impact on the performance of models trained on the data, as can be seen from the performance improvements achieved by the category aggregation test in section 4.3.

Comparing the RE-GCN baseline Adapted Model with the baseline Unchanged RE-GCN Model, we found that the Adapted Model, with a modified goal function, performed considerably better. As discussed in section 11.3, this highlights the importance of aligning the objective function with the specific regression task. The baseline Linear Regression Model, while simple, suffered from limitations, including the generation of nonsensical negative degree predictions, as noted in section 11.2.

The custom `TransformerDecoder` model, while initially demonstrating faster training times, ultimately underperformed compared to the Adapted Model on most datasets. As noted in section 11.4, the model tended to learn an average degree value for each entity, rather than generating more insightful predictions. This suggests potential underfitting or the need for a more complex architecture. The MLA Transformer Decoder model showed promise in balancing accuracy and runtime, suggesting a potential avenue for future optimization.

Ultimately, the classification model fitted using the GDELT-big dataset achieved accuracies exceeding 80% at certain confidence levels, indicating the solvability of the classification problem and suggesting the predictability of the trend definition, albeit with limitations (only a subset of entities having such a confidence level, refer to section 11.7). This finding also suggests that a classification-based approach might be more effective than a regression-based one for TKG trend prediction.

Moreover, the Custom Model's performance, primarily learning an average value, diminished the meaningfulness of analysing confidence intervals across varying model horizons. As discussed in section 11.4, this remains a valuable direction for future research, particularly concerning the evolution of confidence intervals in TKG forecasting.

In conclusion, this thesis provides valuable insights into the challenges and opportunities of using TKGs to predict future industry trends. While the initial trend definition and Custom Model architecture require further refinement, the results suggest that a trend-based approach can be viable. Future research should focus on developing more sophisticated trend definitions, exploring alternative model architectures (including investigating why recurrent layers outperformed Transformers) and loss functions, addressing dataset quality issues, and leveraging larger datasets through multi-GPU and distributed computing. Furthermore, the potential of classification-based approaches warrants further investigation. By addressing these challenges, future studies can unlock the full potential of TKGs for predicting and understanding complex industry dynamics.

Bibliography

[1] Zixuan Li, Xiaolong Jin, Wei Li, Saiping Guan, Jiafeng Guo, Huawei Shen, Yuanzhuo Wang, and Xueqi Cheng. Temporal knowledge graph reasoning based on evolutional representation learning. 2021.

[2] Woojeong Jin, Meng Qu, Xisen Jin, and Xiang Ren. Recurrent event network: Autoregressive structure inference over temporal knowledge graphs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6669–6683. Association for Computational Linguistics, 2020.

[3] Xiaoming Zhang, Fang Li, and Qiang Liu. Temporal knowledge graph reasoning with historical contrastive learning. In *Proceedings of the 2023 Conference on Artificial Intelligence and Knowledge Graphs*, pages 102–110. ACM, 2023.

[4] Xiaohui Li. Findkg: Dynamic knowledge graph with large language models for global finance. October 20, 2023, 2023.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, New York, NY, 2nd edition, 2009.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, Cambridge, MA, 2016.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, New York, 1st edition, 2006.

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

[9] Li Cai, Xin Mao, Yuhao Zhou, Zhaoguang Long, Changxu Wu, and Man Lan. A survey on temporal knowledge graph: Representation learning and applications. *arXiv preprint arXiv:2403.04782*, 2024.

[10] Julien Leblay and Melisachew Wudage Chekol. Deriving validity time in knowl-

edge graph. In *International Conference on Extending Database Technology (EDBT)*, Vienna, Austria, 2018. OpenProceedings.

[11] Laura Igual and Santi Seguí. *Introduction to Data Science*. Springer, 2017.

[12] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. *Nonparametric Statistical Methods*. John Wiley & Sons Inc., 2014.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[14] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[15] PyTorch Contributors. TransformerDecoder - PyTorch 2.7 Documentation, 2023. [Online; accessed 2023-10-01].

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient mem-

ory management for large language model serving with pagedattention, 2023.

[17] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[18] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[19] A. Croll and B. Yoskovitz. *Lean Analytics: Use Data to Build a Better Startup Faster*. Lean (O'Reilly). O'Reilly Media, Incorporated, 2013.

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY