UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Dice

Master's Thesis

Submitted to the Dice Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science

A Recommender System for Basket Items

by
Nilanjan Das
Matriculation No: 6830549

Thesis Supervisor:
Prof. Dr. Axel-Cyrille Ngonga Ngomo
Prof. Dr. Gregor Engels
Advisor:
Michael Röder, M. Sc.

Paderborn, November 12, 2020

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Abstract. A recommender system provides personalized recommendations to the user in a large space of possible options. Most of the existing recommender systems utilize user's profile data for recommending an item. However, these conventional recommender systems are under scrutiny due to strict personal data protection laws around the world. Additionally, we investigated the performance of a graph-based approach for the recommendation systems. In this thesis, we present a graph-based scalable and novel approach for the recommendation which doesn't depend on the profile data for predictions. We evaluate our method on an extensive transaction dataset from the retail domain (700k transactions with 150k different items) and compare it to a baseline. The proposed approach relies on knowledge graph embeddings. During our evaluation, we have used two knowledge graph embedding algorithms. The suggested method first applies Pyke, a knowledge graph embedding algorithm with a close to linear runtime complexity. Later Pyke was replaced by a convolutional complex knowledge graph embedding algorithm, Conex. The evaluation results suggest that Conex fits better than Pyke to the approach. Our implementation is open-source, and it is available at https://github.com/nil9/Master-Thesis.

Keywords: Knowledge graph embedding, Recommender system, Machine learning, LSTM, Graph based recommendation

Contents

List of Figures

List of Tables

Listings

1 Introduction

Since the invention of the web, we are flooded with data. Several companies have successfully invested in decision support systems which helps to make the decision easier within this large chunk of available information. These decision system influences to build modern recommender systems. A recommender system has several advantages as it can improve customer satisfaction which in turn provides a competitive edge. Therefore, the recommender systems have received a lot of attention for the past two decades from industry and researchers because of its money-making potential. Thus, we can see many impressive methods/algorithms in this field of work [BOHG13].

A recommender system provides an advantage for both the customers and sellers. For the customer, it is much easier to find the item of choice because otherwise, they need to search in a much larger space of products, while still exploring the new products. For sellers, it increases the chance of a sale by converting a mere visitor to a customer [BOHG13]. A recommender system increases sale by automatically suggesting items to potential customers with the help of their past purchase data. It also increases the loyal customer base by adding more "value-added relationship" between the customer and the sellers' website [SKR99]. Moreover, the recommender can predict future sale by analysing past buying patterns of customers.

1.1 Recommender System

Within this thesis, a recommender system is defined as follows:

Definition: "Any system that produces individualized recommendations as output or has the effect of guiding the user in a personalized way to interesting or useful objects in a large space of possible options." [Bur02]

This thesis is written in collaboration with Diebold Nixdorf [1]. Diebold Nixdorf is an American firm specialized in Banking and Technology consultation which has around 23,000 employees with presence in more than fifty countries around the world. As a consultancy firm, which provides services to medium and large size business partners, Diebold Nixdorf often requires technical know-how which provides itself with a strategic benefit. The recommender system is one such scenario where even large firms may not have the resources to design, develop and implement it because of its mathematical and computational complexity as well as practical challenges to process a large amount of data and make a recommendation in a real-time scenario. Many solutions in the domain of the recommender system are already existent. A simple

[1] `www.dieboldnixdorf.com`

working solution in this sector may not give a competitive edge to Diebold Nixdorf. For this, an innovative idea would be beneficial which motivates for this collaboration.

Many existing recommender systems use traditional methods like matrix factorization. Matrix factorization uses previous user-item interaction data to predict an item for the user. But this approach has its own drawback [KBV09, SLH14]. It can not predict correctly when user-item interaction data are small in size [SLH14].

With the advent of Artificial Intelligence (AI) and big data, the relevance of recommender systems has increased multi-fold. However, the increase in the popularity in natural language processing and the deep neural network has influenced the creation of the state-of-the-art models based on these technologies [QKHC17, HK18, HKBT16, RDS+15, KGB14, HDY+12b]. But most of these recommender systems use customer profile data to train the models. The introduction of data security law may create hindrance to use profile data. This thesis intends to address the mentioned problem with a graph-based approach.

In this thesis, we prepare a recommender system without using customer profile data.

1.2 Problem Definition

Though a handful numbers of recommender systems exist which can recommend well, their performance drops in some restricted settings. One such scenario is recommending items without using user profile data. The problem of excessive access to customer personal data is a years-old topic for discussion [SFR+06], but this has gained momentum after the Facebook – Cambridge Analytica Scandal [2]. The General Data Protection Regulation [3], issued by the European Union (EU) in 2016, which came to effect on 25 May 2018 emphasizes the personal data privacy. This new rule empowered EU citizens with their data (e.g. provide consent to use or erase personal data). This rule also mentioned non-EU companies to be in-sync with their EU counterparts while processing personal data of EU citizens. Moreover, in a globalized market setting a global player competes with the local sellers. In some cases, local consumers are not a frequent visitor to the global sellers' website, which makes it harder for the sellers to understand/predict customers choice.

We represent the verbal problem formally, let S be the set of all items, S_b be the non-empty set of basket item/s (the items customer already choose in the current session) and s_p is the predicted item. Therefore, a recommender system approximates a function f:

$$f(S_b) \rightarrow s_p \quad \text{with} \quad \forall s_p \in S, S_b \subset S \tag{1.1}$$

It should be noted that $s_p \in S$ means the recommended item can also be a basket item.

1.3 Objectives and Challenges

This thesis intends to attain the following objectives:

1. The primary objective of this thesis is to develop a feature-based recommender system.
2. The approach should be based on knowledge graphs and machine learning techniques.
3. The dataset is given as XML and has to be transformed into a better representation.
4. The final solution should be scalable and efficient (e.g. memory and processing time).

[2] www.wikipedia.org/wiki/Facebook-CambridgeAnalyticadatascandal
[3] www.eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679

Challenges

This thesis deals with a large dataset, containing around 700000 transactions and 150000 items. Therefore, the primary challenge of this thesis is to find a scalable solution which has a reasonable runtime. Secondly, the transaction data contains additional information like store location, cashier name, their details, item returns, discounts, etc. Thus, it is challenging to identify the details of our interest. Finally, we need to find a knowledge graph embedding algorithm which can generate different vectors for distinguished items.

1.4 Related Work

The need for a recommendation system was increased with the invention of the internet in the early '90s. The first generation of recommender systems were based on different variations on collaborative filtering. Tapestry [GNOT92] was developed in 1992 to filter the incoming mails. Group Lens [RIS+94] was designed in 1994 to recommend online news items. One prominent example of early recommender system is Amazon's item-item collaborative filtering recommendation [LSY03], which was developed in 2003. In 2006, Netflix had released a dataset with 100M movie rating and announced prize money of 1M USD for the team which can generate the most accurate prediction. It influenced to build many collaborative filter-based recommendation algorithms [BLN09]. The matrix factorization (MF) based collaborative filtering was introduced in 2004 [Mar04, Hof04], which further increases the accuracy of prediction. Matrix factorization approach outperformed the classical methods of recommendation in terms of accuracy, constant time prediction, and compact model representation. In 2007, generative artificial neural network-based collaborative filtering was introduced to manage large dataset [SMH07] efficiently. The graph-based recommendation is another approach. This thesis is also a graph-based recommendation. In graph-based recommendation, the items are represented by nodes. The graph-based method can produce recommendations with low latency, without the need for specific data about the user [JKH07]. Pixie by Pinterest from 2018 is a notable example of this approach which can generate image recommendations in a real-time scenario [EJL+18]. The recurrent neural network-based recommendation is a popular approach for time series data. LSTM, a version of the recurrent neural network can produce state-of-the-art recommendations in a variety of fields such as text generation [Kar15], music generation [LR+14], language translation [FCB16], etc. There are a plethora of recommendation approaches based on deep learning that exists today. But very few of them can outperform the simpler algorithms. Therefore, in 2019 a question was raised [DCJ19] over the performance of these deep learning-based approaches.This thesis recommends items based on anonymous customer's transaction data. Therefore, the graph-based approach is better suited for our purpose.

In this thesis, we use embeddings (vector representations) to distinguish between items. Word2Vec [MCCD13] and GLOVE [PSM14] can represent the words in vector space, capturing their semantic meaning. MetaProd2Vec can generate item embeddings was introduced in [VSC16]. Doc2Vec can classify documents in categories such as positive, negative was introduced in [LM14].

Identifying a suitable metric is essential to understand the accuracy of predictions. In [CVW11] authors suggest Novelty and Diversity as important metrics wherein [GDBJ10], authors suggest Catalog Coverage and Serendipity are the most critical metrics. The authors conclude saying that there is no perfect metric for recommendation system. Therefore, the developers/authors need to choose a metric based on how they expect the recommendation system to behave. In different research [XY09], authors have stated that there is a trade-off between diversity and accuracy metrics. All these studies lead to the conclusion that there is no primary metric to identify the accuracy of the recommendation. The metric should be chosen based on

the context of recommendation.

1.5 Structure of the Thesis

This thesis conveys all relating researches and findings in six chapters.

- Chapter 2 introduces many topics that are essential to understand before going deep into further discussion. In this chapter, we provide brief discussions on the Knowledge graph, Neural networks and XML parsers.
- Chapter 3 presents the methodology to fulfil this thesis. It exhibits proposed approach along with the baseline method. Each phase of the approach is elaborated. This chapter forms the theoretical ideas of this thesis.
- Chapter 4 shows the implementation of the whole workflow in detail. It first introduces the technologies and packages used. Then the chapter elaborates the execution of each component along with the necessary diagrams in a stepwise manner.
- Chapter 5 evaluates the outcomes from different experiments that are performed in this thesis. First, we introduce the objectives of the evaluation. Then it discusses the list of experiments that are performed. Later, it describes the experimental design along with the execution plan for each experiment. Finally, this chapter interprets and analyses the experimental results.
- Chapter 6 summaries the outline and the results of this thesis. Furthermore, it shows the limitations of this thesis as well as the prospect for further research in this domain.

2 Background

This chapter discusses essential concepts which are relevant for this thesis.

2.1 Artificial Neural Network

Artificial neural network is a function approximator where an approximation function y=f(x) maps an input vector x to $y, y \in Y$, a set of possible outputs with |Y|>= 2. A set of parameters θ are tuned during the learning process for the best approximation. y' is the predicted output from the artificial neural network.

$$y' = f * (x; \theta) \tag{2.1}$$

2.1.1 Feed Forward Neural Network

This is the simplest form of artificial neural network where the input layer receives x which traverse through possible multiple hidden layers (layers between the input and output) to a final layer to produce the output y'. In this architecture, each layer is only connected to the preceding layer [Hea18].

Neurons are the building block of the artificial neural network layers. The neurons transfer a vector input to a scaler output space. In 2.2 a_j represents the scaler output from j-th neuron.

$$a_j = \phi(\sum_i w_{ij} x_i + w_{j0}) \tag{2.2}$$

Where $\sum_i w_{ij} x_i$ is the weighted sum on the input space and w_{j0} is the bias term. The ϕ represents nonlinearity which is called an activation function. ϕ increases the capacity of the model [Bis06]. The basic structure of the neuron is shown below (Fig.2.1).

In the following notations, neurons are denoted as $h_j^{(l)}$ where the subscription shows the j-th neuron in a layer and the superscription denotes the number of the layer. So the input layer is represented as $h^{(0)}$.

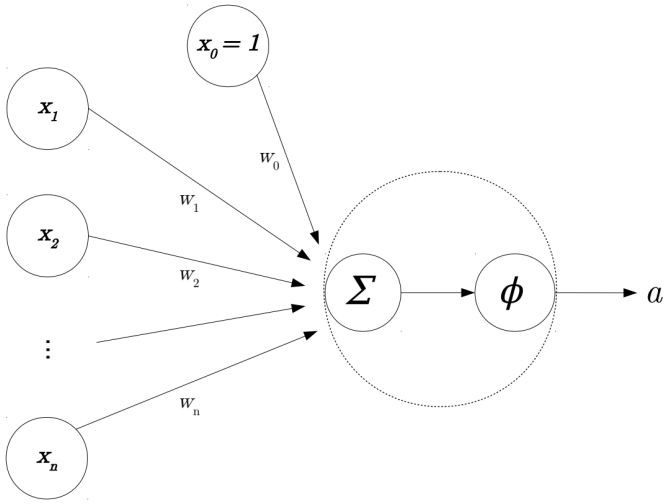

Figure 2.1: Structure of an artificial Neuron[Fal18]

The Forward Pass

In the forward pass, the input x is received through the layer $h^{(0)}$, which is passed through the hidden layers following the equation 2.2 and produce:

$$h_j^{(l)} = \phi(\sum_i w_{ij}^{(l)} h_j^{(l-1)} + w_{j0}^{(l)}) \tag{2.3}$$

The vectorised form of equation 2.3 is :

$$h^{(l)} = \phi(w^{(l)} h^{(l-1)}) \tag{2.4}$$

Finally, a loss function checks the difference between the prediction y' and the real data y:

$$Loss = \mathcal{L}(y', y) = (h^{(L)}, y) \tag{2.5}$$

Two assumptions of loss function are made here. They are: the loss can be calculated by averaging over the loss of the individual output. Second, the loss function is at least once differentiable w.r.t its input [Bis06].

The Back-propagation Algorithm

After the calculation of loss function, we want to adapt the weights for a better approximation of the unknown function f. The back-propagation algorithm does this [RHW86].

The idea is to take partial derivative w.r.t weights and biases of the network. So, a small amount of change in the weights is made to check where the loss function has the highest increase. Formally the changes in the weight are $w = w + \delta w$ and the changes in loss function are $\delta\mathcal{L} \simeq \delta w^T \nabla\mathcal{L}$, where δw and $\delta\mathcal{L}$ is the changes in the weight and loss function respectively. $\nabla\mathcal{L}$, the gradient of the loss function, points out where the loss function has the highest rate of increase [Bis06]. So, back-propagation uses the chain rule of calculus to calculate the partial derivatives on individual weights [Hea18].

$$\frac{\partial\mathcal{L}}{\partial w_{ij}} = \frac{\partial\mathcal{L}}{\partial h_j}\frac{\partial h_j}{\partial w_{ij}} \tag{2.6}$$

The vector notation of the equation 2.6 is :

$$\nabla_w \mathcal{L} = (\frac{\partial h}{\partial w})^T \nabla_h \mathcal{L} \tag{2.7}$$

In equation 2.7 ∇_w and ∇_h represent gradient w.r.t weight and height respectively.

Activation Function

This function provides a non-linear transformation of the inputs. The very first simple Perceptron model was built by Rosenblatt [Ros58].

$$\phi(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geqslant 0 \end{cases} \tag{2.8}$$

This model fails in complex scenarios like backpropagation cannot manage discontinuous character. Another popular activation function is sigmoid [Hea18].

$$\phi(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

But sigmoid is not used in practice because it has a neuron saturation problem. It means that the output becomes close to zero for very small and very large input values because the slope is flat near the edges [GB10]. To overcome this drawback hyperbolic tangent is used:

$$\phi(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.10}$$

The activation function which is widely used for feedforward network is Rectified Linear Unit (ReLU) [NH10]:

$$\phi(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geqslant 0 \end{cases} \tag{2.11}$$

A slightly altered version of ReLU is Leaky Rectified Linear Unit [Fal18], which uses slope parameter λ as hyperparameter:

$$\phi(x) = \begin{cases} \lambda_x & \text{for } x < 0 \\ x & \text{for } x \geqslant 0 \end{cases} \tag{2.12}$$

Output Functions

Compared to the hidden layer, activation functions have different criteria to match in the output layer. This layer needs to transform final activation into a label space of y. For classical regression problem and Gaussian distribution, the label space is standard $\mathbb{R}$ or $\mathbb{R}^n$. However, this is not true for the binary or multi-class problem where ϕ must be a discriminant function that maps input to a discrete space C_k[Bis06]. So a sigmoid function of form 2.10 is essential for assigning a class probability to the binary or multi-class case. For multi-class case, the approximation function is called softmax that creates a mapping $\mathbb{R}^k \to [0, 1]$, a map from a k- dimensional vector space to a real-world value closed by the interval $[0, 1]$. Therefore, the probability vector over k classes is:

$$\phi(x) = softmax(x) = \frac{e^x}{\sum_i e^{x_i}} \tag{2.13}$$

We get familiar with the artificial neural network. In the following sections we discuss a complex version of artificial neural network, Recurrent Neural Network (RNN).

2.1.2 Recurrent Neural Network (RNN)

A recurrent neural network is a type of neural network [HDY+12a, DYDA11] which has proved its utility in various domains (e.g. speech recognition, time series modelling, etc.) [PDS+16, PWD16, Elm90, Rob94, MKB10, Gra12, BBLP12, MHDB13]. An RNN is applicable for cases where the context is important. The RNN uses feedback loops to remember the sequential inputs. Because of this feedback loop, an RNN remembers information of the past. The output from a neuron propagates to the next neuron of an RNN (Fig. 2.2).

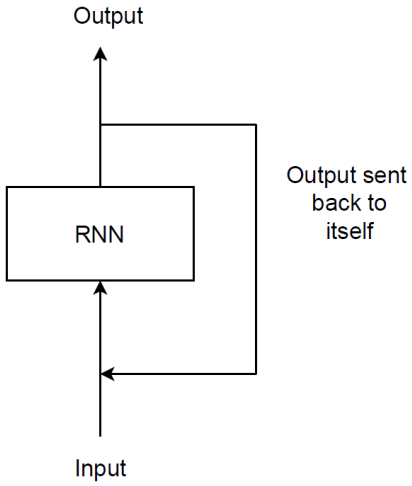

Figure 2.2: Single Neuron of an RNN[ASAAA18]

Suppose in the sentence *This is a dog* the last word *dog* is missing. An RNN wants to predict the last word of this sentence. For this example an RNN needs to be fed with first three words namely *This*, *is* and *a* to predict the forth word *dog*. So, one can think of RNN is a neural network with memory. Theoretically, RNN can capture information of any length, but in practice, it can only remember the information of short distance. LSTM and GRU are types of RNN.

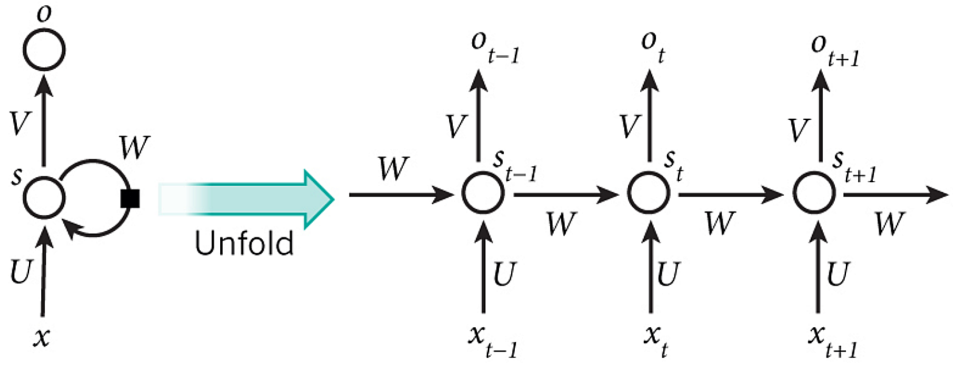

Figure 2.3: Architecture of the RNN[BYR17]

Fig.2.3 shows an RNN. The following points describe the notations of Fig.2.3 [BYR17].

- x_t is the input at timestamp t. For example, x_t represents the one-hot encoding for the second word.
- s_t is the hidden state at timestamp t. s_t is the memory of the network which depends on previously hidden step output s_{t-1} and the input to that state x_t. so, $s_t = f(Ux_t + Ws_{t-1})$ where f_t is the non-linearity like tanh or ReLU.
- o_t is the output at timestamp t.
- V is the output from each step.

Additional information on Fig.2.3

- s_t is the memory which can remember all the information from previous steps and o_t is dependent on s_t. But in practice s_t cannot capture information from long previous steps.
- The regular neural network uses different parameters for each step. But an RNN uses the same parameters (U, V, W in Fig.2.3) in all the steps which not only reduce the numbers of parameters but it easily explain that the RNN is doing the same operation in all the steps.
- The hidden memory (s in Fig.2.3) is the most striking feature of an RNN, which captures information of some past states.

2.1.3 Long Short Term Memory (LSTM)

In a neural network, one common technique is back-propagation which adjusts the weights of the neurons with the gradient descent method. If the length of the hidden layer is long enough, a neural network suffers from vanishing gradient problem [PWD16]. In vanishing gradient problem, the gradients in the initial layers become so nominal that a neural network cannot learn anything by making changes in the weights of these neurons. An LSTM resolves this problem [HS97]. The LSTM is a type of recurrent neural network (RNN) which first came into the picture in 1997. An LSTM has long term memory which solves significant problems in image recognition, speech recognition and language modelling. In LSTM, each neuron has four interacting layers: input gate, a forget gate, an output gate and a self-recurrent neuron (Fig. 2.4).

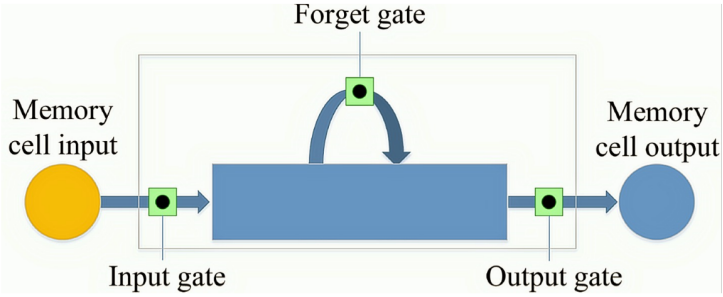

Figure 2.4: An LSTM memory cell architecture[BYR17]

Inside a neuron, there is a cell state (i.e. C_t) which works like a conveyer belt means information passes through it. An LSTM can hold or ignore the data. An LSTM performs this through gates. The first gate (first blue colour in Fig.2.5) in the LSTM checks which data to ignore. A sigmoid function inside this gate outputs a number between 0 to 1 where 0 means

completely forget the number in the cell state and 1 means remember the number. Next step decides which information to store in cell state. This step works in two parts (second blue colour part in Fig.2.5). The first is a sigmoid function which determines the data to store. Then a *tanh* layer generates the vector of the selected number. The last step decides the output. It also works in two stages. A sigmoid function determines which part of the cell to be output. Then the sigmoid value is multiplied by a *tanh* function which compresses the value between -1 and +1. Fig.2.5 shows an LSTM.

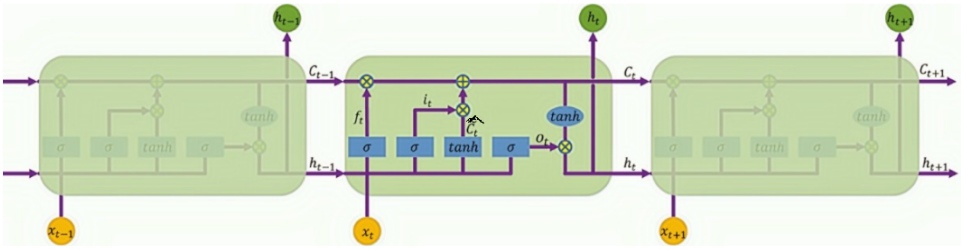

Figure 2.5: An LSTM module [BYR17]

The following points elaborate on the mathematical symbols which are present in Fig. 2.5.

- x_t is the input vector at time t.
- The weight matrices are W_i, W_f, W_c,W_o, U_i, U_f, U_c, U_o and V_o.
- The bias vectors are b_i, b_f,b_c and b_o.
- The value which propagates to next state, memory cell value is h_t at time t.
- The input gate value i_t and candidate state value $\hat{C}$ can be expressed :

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2.14}$$

$$\hat{C} = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.15}$$

- Values at the forget gate f_t and state memory cell can be formulated as:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.16}$$

$$C_t = i_t * \hat{C}_t + f_t * C_{t-1} \tag{2.17}$$

- Output gate o_t and memory cell value h_t at time t are:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o) \tag{2.18}$$

$$h_t = o_t * tanh(C_t) \tag{2.19}$$

2.1.4 Different optimizers used in LSTM

In the following equations $J(\theta)$ is considered as a constant objective function which is minimized by iterative algorithms. $\nabla J(\theta)$ is the gradient of the function evaluated at θ.

Stochastic Gradient Descent (SGD)

SGD optimizer is one of the oldest and most used optimizers [Bot12] for deep learning problems which works on iterations to find the minimum error.
Drawback: If the target function is not convex or pseudo-convex, then the SGD get stuck into a local minimum.

Momentum

This method gives direction and reduces the high variance of SGD [Qia99]. Momentum uses an extra parameter γ to update vector of past time.

$$V(t) = \gamma V(t-1) + \alpha * \nabla J(\theta) \tag{2.20}$$

In practice, γ is set to 0.9. In the equation 2.20, α represents the learning rate and t is time step.
Advantage: Reduce the high variance of SGD.
Drawback: Hyper-parameter γ is set manually.

Nesterov Accelerated Gradient (NAG)

Momentum can be thought of a ball which blindly falls from a hill following a slope without knowing the direction where it is going. So, a smarter ball is needed, which has the notion of its direction. NAG provides this precession over the momentum term γv_{t-1}. In NAG the momentum term γv_{t-1} is used to move the parameter θ. So, $\theta - \gamma v_{t-1}$ is computed to get the approximation of the next position of parameters. Thus, NAG calculates gradient w.r.t the approximate future position of parameters [Rud17]. In practice, γ is set to 0.9. In equation 2.21, η denotes the learning rate.

$$V_t = \gamma V_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{2.21}$$

$$\theta = \theta - v_t \tag{2.22}$$

Advantage: NAG does not get stuck at the local minima.
Drawback: Hyper-parameter γ is set manually.

Adagrad

Adagrad "adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data."[Rud17]. All the previous optimizers that are already discussed using the same learning rate η for every parameter θ_i. Adagrade changes the learning rate η for every parameter θ_i at each timestep t. $g_{t,i}$ is the gradient of the objective function w.r.t the parameter θ_i at timestep t. Thus, SGD update on every parameter θ_i at each timestep t takes the below form [2.24]. The updated learning rate is represented by equation 2.25

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \tag{2.23}$$

$$\theta_{t+1,i} = \theta_{t,i} - \eta * g_{t,i} \tag{2.24}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} * g_{t,i} \tag{2.25}$$

"$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t[11], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse." [Rud17]

G_t contains the sum of squares of the past gradients w.r.t. all the parameters θ. Therefore equation 2.25 can be vectorized by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{2.26}$$

Advantage: This method does not require to tune the learning rate manually.
Drawback: This approach is Computationally expensive.

Adam

Adam optimizer [KB14] calculates the exponentially decaying average of the gradients m_t, along with the exponentially decaying average of the past squared gradients v_t. So the equations of m_t and v_t are [2.27 and 2.28]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.27}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.28}$$

As m_t and v_t are vectors if initialized with 0, creates a bias towards zero. Thus, after the bias correction the equations [2.27 and 2.28] convert into [2.29 and 2.30]:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.29}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.30}$$

So, the update rule of Adam takes the following form[2.31]:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} * \hat{m}_t \tag{2.31}$$

The default values of the above equations [2.27,2.28,2.29,2.30 and 2.31] are 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.
Advantage: This approach is fast and converges quickly.
Drawback: Adam is Computationally expensive.

2.2 RDF Graph

2.2.1 Graph-Based Data Model (GBDM)

Graph-based data model RDF first came into existence in 1991 to make a semantic relation between the data coming from different sources of the internet. This was the basis of the graph-based data model. In the beginning, only XML can serialize GBDM. XML had syntactical

drawbacks in representing collections, reification and quoting [CS04]. Therefore, several other serializers ware introduced namely N3 [BLC11], N-Triples [SR14], and TURTLE [BBL08].

The sole principal of the RDF is to represent resources (i.e. subjects) in relation (i.e. predicate) to other resources or literals (objects). In short, an RDF represents the relations among resources in subject, predicate and object format (Fig.2.6). Thus, each combination of subject, predicate and object is a molecular representation of the RDF. For example, *Bob is a student*. Here *Bob* is subject, *student* is an object and the relation between them *is* is a predicate. In RDF, subjects and objects depict the nodes of a graph and predicates are the edges. The edge of an RDF is only IRI, subject and object can either be IRI or a blank node. The literals can also represent objects because sometimes objects need to show values such as strings, numbers, and dates [Dav20].

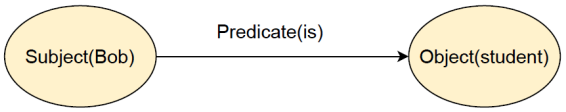

Figure 2.6: Sample triple in RDF graph

2.2.2 URI and IRI

URIs (Unique Resource Identifier) are strings which help to represent data model, resources and properties in an unambiguous fashion. URIs cannot be shown in Unicode format. For this reason W3C [BG14] recommends to use IRIs (Internationalized Resource Identifier). In a nutshell, IRIs are URIs in Unicode format. For example, in Fig.2.7 IRIs represent each node. The length of the IRI is long, which made it difficult to read. IRI can be shortened by using a prefix. In Fig.2.8 *rdf:type* is the abbreviation form of the node *http://www.w3.org/2000/01/rdf-schema#type* in Fig.2.7. Thus, Fig.2.8 is easily readable.

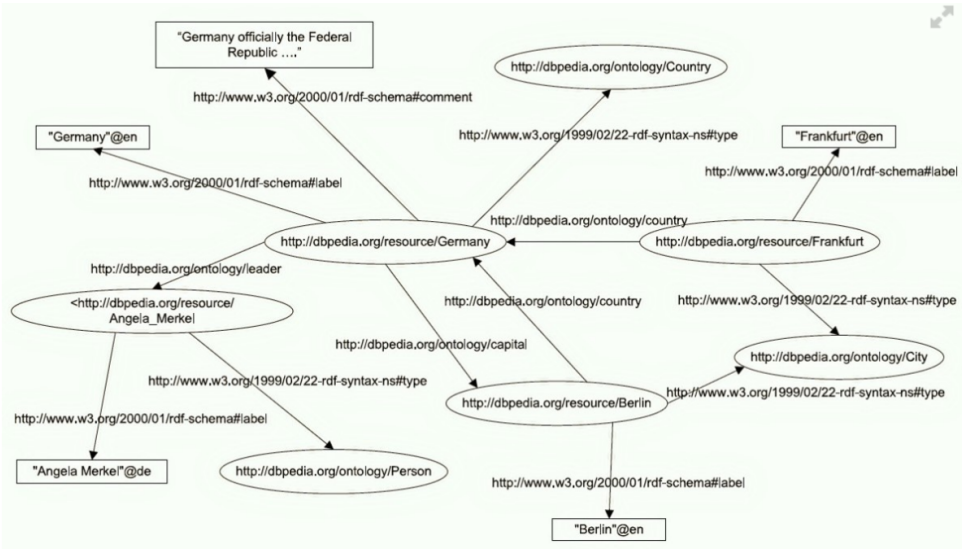

Figure 2.7: RDF graph example [Ngu18]

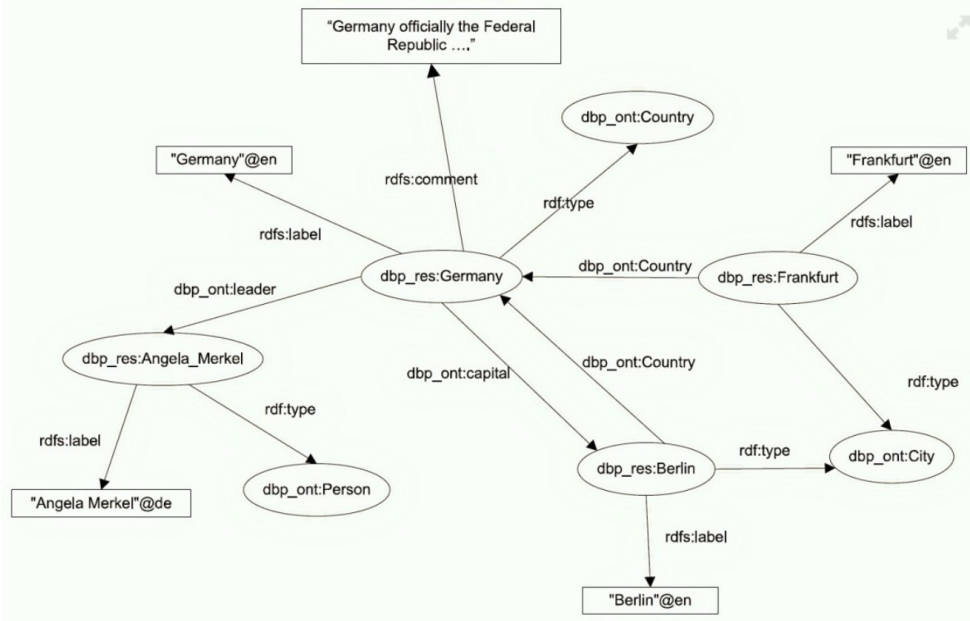

Figure 2.8: RDF graph example with shorted IRIs [Ngu18]

2.2.3 Literal

Literals can only represent the objects in an RDF. Literals are of two types, namely plain literals and typed literals. Both types represent strings, numbers, date-times in a lexical form. The plain literal has an optional language tag. The typed literal's type is an IRI. For example in Fig.2.8 *dbp_res:Angela_Markel* has an *rdfs:label* called *"Angela Markel"@de*, this denotes the literal in German.

2.2.4 Blank node

If the subject and object of an RDF are neither IRI nor Literal (for objects only), then it can be represented with a blank node. The name "Blank" denotes unknown resources where no IRI can be given. RDF Data model assigns an id of form_*: id*, which is a local identifier to represent a blank node.

2.2.5 RDF Graph

RDF graph consists of triples. Triple (subject, predicate and object), is a directed graph where subjects and objects form nodes and the relation between those is a predicate, which forms the directed edge between the nodes.

2.2.6 Knowledge Graph Embedding

Knowledge Graph Embedding represents entities and their relations in a lower-dimensional vector space (embedding space). In embedding space, the similar vectors are nearer compared to the dis-similar vectors. Knowledge Graph Embedding is useful to extract the overall relation of Knowledge Graph. Knowledge Graph Embedding helps to infer relationships among entities coming from different sources. Knowledge Graph Embedding applications can solve many problems in the field of collective machine learning, type prediction, link prediction, entity resolution, knowledge graph completion and question answering. In this thesis, we have used two knowledge

graph embedding algorithms, namely, Pyke [DN19] and Conex [DN20]. The necessary details of these algorithms are discussed below.

Pyke

To understand Pyke, knowledge on Hooke's Law is necessary because "PYKE, combines a physical model (based on Hooke's law) with an optimization technique inspired by simulated annealing" [DN19]

Fig.2.9 shows the list of notations that are used by Pyke.

Notation	Description
$\mathcal{G}$	An RDF knowledge graph
$\mathcal{R}, \mathcal{P}, \mathcal{B}, \mathcal{L}$	Set of all RDF resources, predicates, blank nodes and literals respectively
$\mathcal{S}$	Set of all RDF subjects with type information
$\mathcal{V}$	Vocabulary of $\mathcal{G}$
σ	Similarity function on $\mathcal{V}$
$\overrightarrow{x}_t$	Embedding of x at time t
F_a , F_r	Attractive and repulsive forces, respectively
K	Threshold for positive and negative examples
P	Function mapping each $x \in \mathcal{V}$ to a set of attracting elements of $\mathcal{V}$
N	Function mapping each $x \in \mathcal{V}$ to a set of repulsive elements of $\mathcal{V}$
$\mathbb{P}$	Probability
ω	Repulsive constant
$\mathcal{E}$	System energy
ϵ	Upper bound on alteration of locations of $x \in \mathcal{V}$ across two iterations
Δe	Energy release

Figure 2.9: List of notations used in Pyke [DN19]

Hook's Law

Hook's Law defines the relationship between the deformation force on the spring and the magnitude of deformation on the realm of the spring. If the deformation force increases, then the volume of deformation also linearly increases [DN19]. Hence, the formal representation of Hook's Law is 2.32:

$$F = -k\Delta \tag{2.32}$$

In 2.32 F is the deformation force, k is the spring constant while, Δ is the magnitude of deformation. The inverse of Hooks law is 2.33:

$$F_{inverse} = \frac{-k}{\Delta} \tag{2.33}$$

The force ($F_{inverse}$) between the two mass points becomes weaker with the increase of the distance between the two mass points it connects.

Positive Pointwise Mutual information (PPMI)

"The Positive Pointwise Mutual Information (PPMI) is a means to capture the strength of the association between two events (e.g., appearing in a triple of a KG). Let a and b be two events. Let P(a; b) stand for the joint probability of a and b, P(a) for the probability of a and P(b) for the probability of b. Then, PPMI(a; b) is defined as (2.34):" [DN19]

$$PPMI(a, b) = max(0, \log \frac{P(a, b)}{P(a)P(b)}) \tag{2.34}$$

Pyke Methodology

Pyke is an iterative process where each element of a vocabulary $\mathcal{V}$ of a knowledge graph $\mathcal{G}$ can be represented as a unit mass in n-dimensional vector space $\mathbb{R}^n$ by an n-dimensional vector randomly allocated at iteration $t = 0$. The goal is to improve the representation of embeddings $\vec{x_t}$ over the iteration where $x \in \mathcal{V}$ at iteration t. So, the final objective of Pyke is to improve $\vec{x_t}$ iteratively such that after a predefined number of iteration (denoted as T) it ensures:

1. The embedding of similar elements of $\mathcal{V}$ are closer to each other in the embedding space while
2. The embedding of dis-similar elements of $\mathcal{V}$ is distant from one another.

Pyke assumes a PPMI based similarity matrix is given where similarity function σ: $\mathcal{V} \times \mathcal{V} \longrightarrow [0, \infty)$

Let d be the distance between two embeddings in $\mathbb{R}^n$ where $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+$. Therefore the goals of Pyke are:

1. C1: "If $\sigma(x, y) > 0$, then $d(\vec{x_t}, \vec{y_t}) \leqslant d(\vec{x_{t-1}}, \vec{y_{t-1}})$" [DN19]. It means the similar vectors should be near in embedding space while
2. C2: "If $\sigma(x, y) = 0$, then $d(\vec{x_t}, \vec{y_t}) \geqslant d(\vec{x_{t-1}}, \vec{y_{t-1}})$" [DN19], the dis-similar vectors are at a distance in embedding space.

Therefore, as per Hook's law, we can formally represent C1, the force($F_a(\vec{x_t}, \vec{y_t})$) between two similar masses x and y at time t as (2.35):

$$||F_a(\vec{x_t}, \vec{y_t})|| = \sigma(x, y) \times d(\vec{x_t}, \vec{y_t}) \tag{2.35}$$

Where $d(\vec{x_t}, \vec{y_t})$ is the distance between the masses and $\sigma(x, y)$ is the spring constant. That means the force between the masses increases with distance.

The inverse of Hooks law represents C2, the force between two dis-similar masses x and y at time t as (2.36)

$$||F_a(\vec{x_t}, \vec{y_t})|| = -\frac{\omega}{d(\vec{x_t}, \vec{y_t})} \tag{2.36}$$

Where $\omega > 0$ is the repulsive constant.

In a nutshell, Pyke tries to minimize the total distance between similar elements and maximizes the distance between dissimilar elements. Therefore, the formal goal of Pyke is: "Let $P : \mathcal{V} \to 2^{\mathcal{V}}$ be a function which maps each element of $\mathcal{V}$ to the subset of $\mathcal{V}$ it is similar to. Analogously, let $N : \mathcal{V} \to 2^{\mathcal{V}}$ map each element of $\mathcal{V}$ to the subset of $\mathcal{V}$ it is dissimilar to. PYKE aims to optimize the following objective function : (2.37) " [DN19]

$$J(\mathcal{V}) = \left(\sum_{x \in \mathcal{V}} \sum_{y \in P(x)} d(\vec{x}, \vec{y}) \right) - \left(\sum_{x \in \mathcal{V}} \sum_{y \in N(x)} d(\vec{x}, \vec{y}) \right) \tag{2.37}$$

As per the ideas described above Pyke first constructs symmetric similarity matrix $\mathcal{A}$, of dimension $|\mathcal{V}| \times |\mathcal{V}|$. The similarity co-efficient between $x \in \mathcal{V}$ and $y \in \mathcal{V}$ stored in $\mathcal{A}$ is represented by $a_{x,y}$. Pyke builds the similarity matrix in the following way:"For any two elements $x, y \in \mathcal{V}$ we set $a_{x,y} = \sigma(x, y) = PPMI(x, y)$ in our current implementation. We compute the probabilities P(x), P(y) and P(x,y) as follows:" [DN19]

$$P(x) = \frac{|\{(s, p, o) \in \mathcal{G} : x \in \{s, p, o\}\}|}{|\{(s, p, o) \in \mathcal{G}\}|} \tag{2.38}$$

Likewise,

$$P(y) = \frac{|\{(s,p,o) \in \mathcal{G} : y \in \{s,p,o\}\}|}{|\{(s,p,o) \in \mathcal{G}\}|} \tag{2.39}$$

so,

$$P(x,y) = \frac{|\{(s,p,o) \in \mathcal{G} : \{x,y\} \subseteq \{s,p,o\}\}|}{|\{(s,p,o) \in \mathcal{G}\}|} \tag{2.40}$$

Pyke calculates two sets namely P and N for accumulating similar and dissimilar elements. The similar elements attract each other while the dis-similar elements repulse each other.

Embedding Initialization and iterations

As mentioned earlier Pyke maps each $x \in \mathcal{V}$ to a single point $\vec{x_t}$ in embedding space $\mathbb{R}^n$. The very first iteration at $t = 0$ this vector is randomly allocated. In each iteration t elements of similar elements set attracts x with a force of :

$$F_a(\vec{x_t}) = \sum_{y \in P(x)} \sigma(x,y) \times (\vec{y_t} - \vec{x_t}) \tag{2.41}$$

And the elements of N(x) repulse x with force:

$$F_r(\vec{x_t}) = -\sum_{y \in N(x)} \frac{\omega}{(\vec{y_t} - \vec{x_t})} \tag{2.42}$$

The embedding of x at an iteration $t+1$ is proportional to $(F_a(\vec{x_t}) + F_r(\vec{x_t}))$. However, using this approach leads to non-converging behaviour in the model. To avoid this situation Pyke uses *simulated annealing* (i.e. reduces the total energy of the system by a constant factor Δe after each iteration).

Fig.2.10 depicts the Pyke algorithm.

Conex

Conex is a knowledge graph embedding algorithm which "infers missing links by leveraging the composition of a 2D convolution with a Hermitian inner product of complex-valued embedding vectors." [DN19]. Conex achieves superior performance on link prediction task on most of the state-of-the-art approaches such as RotatE, QuatE and TuckER with at least eight times fewer parameters. The following discussions present the necessary notations and terminologies that are used by Conex.

Knowledge Graph: A knowledge graph $\mathcal{G}$ is a set of triples $(s,p,o) \in \xi \times \mathcal{R} \times \xi$ where ξ is the set of all entities and P is the set of all relations [DN20].

Link Prediction: Predicting the typed directed edges in $\mathcal{G}$ is known as link prediction which is a subtask of Knowledge Graph Completion [JPC+20]. Therefore, the formalized scoring function for the link prediction is $\phi : \xi \times \mathcal{R} \times \xi \mapsto \mathbb{R}$ where $\phi((s,p,o)) >> \phi((x,y,z))$ if $(s,p,o) \in \mathcal{G} \wedge (x,y,z) \notin \mathcal{G}$ [NMTG15].

The Convolution Operation: "A convolution is an integral expressing the amount of overlap of one function f as it is shifted over another function g. Formally, the convolution operation over a finite range $[0,\tau]$ is given by" [DN20].

$$(f * g)(t) = \int_0^\tau f(\tau) g(t-\tau)\, d\tau \tag{2.43}$$

Algorithm 1 PYKE

Require: $T, \mathcal{V}, K, \epsilon, \Delta e, \omega, n$

```
//initialize embeddings
for each x in V do
    x_0 = random vector in R^n;
end for
//initialize similarity matrix
A = new Matrix[|V|][|V|];
for each x in V do
    for each y in V do
        A_xy = PPMI(x, y);
    end for
end for
// perform positive and negative sampling
for each x in V do
    P(x) = getPositives(A, x, K) ;
    N(x) = getNegatives(A, x, K) ;
end for
// iteration
t = 1;
E = 1;
while t < T do
    for each x in V do
        F_a = sum_{y in P(x)} sigma(x, y) x (y_{t-1} - x_{t-1});
        F_r = - sum_{y in N(x)} w / (y_{t-1} - x_{t-1});
        x_t = x_{t-1} + E x (F_a + F_r);
    end for
    E = E - De;
    if sum_{x in V} ||x_t - x_{t-1}|| < e then
        break
    end if
    t = t + 1;
end while
return Embeddings x_t
```

Figure 2.10: Pyke algorithm [DN19]

In equation 2.43 $*$ denotes the convolution operation, f is the input and g is kernel. The output of $f * g$ denotes the feature map [Hea18]. "Suppose that f represents a 2-dimensional image and g denotes a 2-dimensional kernel. Then, Equation 2.43 can be rewritten as " [DN20].

$$(f * g)(i, j) = \sum_{m} \sum_{n} f(m, n) g(i - m, j - n) \tag{2.44}$$

In equation 2.44 i,j denotes the coordinate in 2-D input.

Conex Methodology

Conex is defined as :

$$\phi(s, p, o) = \mathcal{H}(< e_s, e_p, e_o >; \mathcal{V}(e_s, e_p)) \tag{2.45}$$

"where $e_s, e_p, e_o \in \mathbb{C}^d$ and $\mathcal{V}$ denotes a 2D convolution layer with ω kernel over an input of $\mathbb{R}^{4\times d}$ followed by a linear transformation to project the feature map $\mathcal{T} \in \mathbb{R}^{c\times m\times n}$ into $\mathbb{R}^d$ and c denotes the number of 2D feature maps with dimensions m and n." [DN20]. $\mathcal{V}$ is defined as

$$\mathcal{V}(e_s, e_p) = f(vec(f([Re(e_s), Re(e_p), Im(e_s), Im(e_p)] * \omega))W) \tag{2.46}$$

where f denotes rectified linear units, ω is a kernel, vec is a flattening operation, and W is a projection matrix. Re and Im are the real and imaginary parts of a complex number, respectively.

$\mathcal{H}$ is the composition of $\mathcal{V}$ with Hermitian inner product of complex-valued vectors. $\mathcal{H}$ is defined as: [DN20]

$$\mathcal{H}(< e_s, e_p, e_o >; x) = \sum_{k=1}^{d} x_k Re(e_s)_k Re(e_p)_k Re(\bar{e_o})_k \tag{2.47}$$

$$\begin{aligned}\mathcal{H}(< e_s, e_p, e_o >; x) =& < x, Re(e_s), Re(e_p), Re(e_o) > \\ &+ < x, Re(e_s), Im(e_p), Im(e_o) > \\ &+ < x, Im(e_s), Re(e_p), Im(e_o) > \\ &+ < x, Im(e_s), Im(e_p), Re(e_o) >\end{aligned} \tag{2.48}$$

where $\bar{e_o}$ is the congjugate of e_o.

Training the data for Conex

A row-vector look-up operation is first applied to obtain e_s, e_p and e_o. Next the real and imaginary parts of e_s and e_p were stacked. Then a 2D convolution is applied with ω to obtain a feature map $\mathcal{T} \in \mathbb{R}^{c\times m\times n}$. Thereafter, $\mathcal{T}$ is flattened with vec(.) and transformed into $\mathbb{R}^d$ by applying a linear transformation W. Then Hermitian inner product is computed on e_s, e_p and e_o. In the last step logistic sigmoid function $\hat{y} = \sigma(\phi(s, p, o))$ is applied to compute the score of (s,p,o) and a loss function minimizes the following cross-entropy: [DN20]

$$\mathcal{L}(y, \hat{y}) = -(y \cdot log(\hat{y})) + (1 - y) \cdot log(1 - \hat{y})) \tag{2.49}$$

where $y = 1$ if $(s, p, o) \in \mathcal{G}$, otherwise $y = 0$.

Optimization

During training Conex follow $1 - N$ scoring regime (with $N = |\xi|$) for efficient training. $1 - N$ scoring regime has two advantages : "(1) the regime has an effect akin to batch normalization, and (2) faster convergence" [DN20]. Conex mentions "We also employ the Glorot initialization technique for parameters of CONEX, as using the logistic sigmoid activation often drives the top hidden layer into saturation provided that parameters are randomly initialized. The parameters are optimized using the Adam optimiser in mini-batch fashion. During loss minimization, we employ the early stopping technique. As a non-linearity in $\mathcal{V}$, we use ReLU for faster training. To lessen overfitting, we use the dropout technique on each layer and we apply label smoothing. Moreover, we perform batch normalisation after each layer to avoid internal covariate shift in the parameters." [DN20].

2.3 XML Parsing

An XML parser is a piece of code or package which validates the structure of XML and stores part or the whole XML body. Fig.2.11 depicts an XML parser which parses the intended tags from the XML dataset and stores it in a triple format (subject, predicate and object).

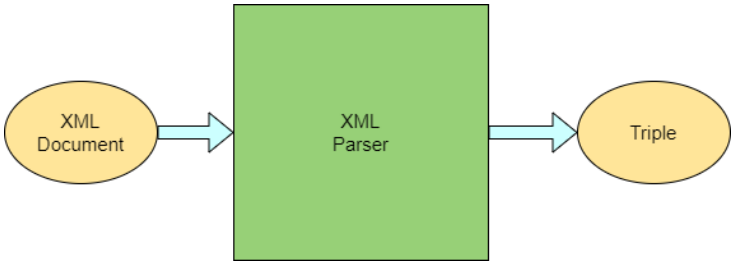

Figure 2.11: A sample XML parser

2.3.1 Types of XML Parser

There are two types of XML parser namely Document Object Model (DOM) and Simple API for XML (SAX) [Li09].

Document Object Model (DOM)

"DOM is a tree-based interface that models an XML document as a tree of various nodes such as elements, attributes, texts, comments, entities, and so on. A DOM parser maps an XML document into such a tree rooted at a Document node, upon which the application can search for nodes, read their information, and update the contents of the nodes." [Li09]

Simple API for XML (SAX)

"SAX is an event-driven interface. The application receives document information from the parser through a ContentHandler object. It implements various event handlers in the interface methods in ContentHandler, and registers the ContentHandler object with the SAX parser. The parser reads an XML document from the beginning to the end. When it encounters a node in the document, it generates an event that triggers the corresponding event handler for that node. The handler thus applies the application logic to process the node specifically." [Li09]

2.4 Types of Recommender Systems

This thesis is about recommender systems. Therefore, in the following sections, we discuss five different categories of recommender systems [Bur02].

Collaborative Recommender System

The collaborative system thinks if a person X likes something which another person Y also likes, then it is more likely their opinion matches on other issues [Col20]. Thus, collaborative filtering "collaborates" the tastes of multiple people to predict the choice of a person. For example, Amazon uses item to item collaborative filtering technique [LSY03] to recommend an item. Suppose both the person X and Y bought an item A previously. If person X buys an item B, then Amazon recommends item B to person Y as well (Fig.2.12).

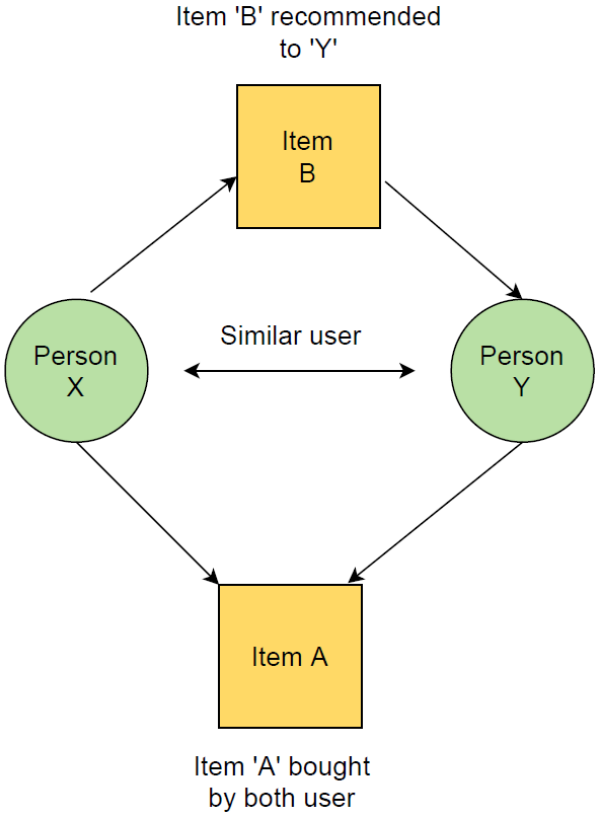

Figure 2.12: Collaborative filtering

Content Based Recommender System

This mechanism is based on the item description and user profile [Bur02]. This technique describes items with keywords. User profile provides the necessary information to identify user-item interaction, Which helps to predict the best-suited item for a user. This method is useful if specific information (e.g. name, description) about the item is available. Netflix is a well-known example of this approach. If a person watches or gives positive rating (e.g. voting, comments) for a movie which is labelled as "action", then Netflix predicts more "action" movies for that user (Fig.2.13).

Demographic Recommender Systems

This approach depends on the different demographics' classification characteristics (e.g. age, gender, cultural background or other personal characteristics). The derived model then matches with the stereotype user catalogue to predict an outcome. This approach does not need profile data. That is why it can solve the new user problem. When a user uses a website for the very first time, it's interests are unknown. Thus, a prediction is possible based on its available demographic data [LKH14].

Utility-Based Recommendation

These systems calculate the usability of each item for a user to predict the best item (e.g. item with most usable score). These algorithms do not use user rating. The main problem for this approach is to define the usability factor, which, in most cases, is done by different user satisfaction techniques [Bur02].

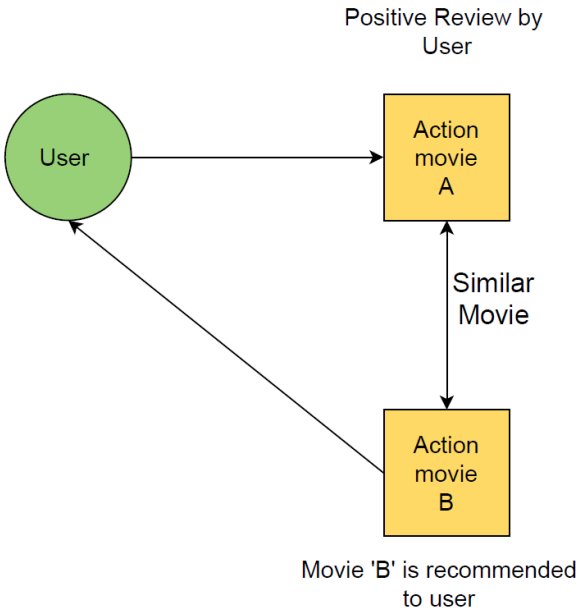

Figure 2.13: Content-based filtering

Knowledge-based Recommender System

These types of recommender systems try to identify information which tells the user needs. This information is available most of the time in user profile [Bur02]. We can think every kind of recommender system is a knowledge bases system because it works on the intuition that there is some information available which tells us what a customer wants [CNHAVGGS12].

Hybrid Recommender System

Above mentioned approaches have their strengths and weaknesses. To overcome the shortcomings and improve the performances in different scenarios, some researchers have combined two or more techniques. These combined approaches are called Hybrid system. Some of the Hybrid systems are Weighted recommender, Switching recommender, Mixed recommender etc. As the detailed discussion about Hybrid Recommender system is out of scope for this topic, we skip this part. The detailed discussion on this topic can be found at [Bur02].

3 Approach

Several studies[MA09, JMSM10, CHCH08] have proposed various algorithms for recommender systems based on customer profile data or item details. We have the same purpose of recommending an item to a customer, but we want to do this with the help of knowledge graph embedding algorithms. The key idea of our approach is to exploit the different properties of an item and predict the next product based on these properties. This chapter describes the theoretical foundations of our approach, while the implementation details of the process will be discussed in Chapter 4. For data security issues, some letters in the code and data are replaced by *'s.

3.1 Proposed Approach Overview

This section provides a brief overview of our approach. Fig.3.1 depicts the workflow diagram for the proposed method. The detailed discussion on each step is carried out in the next section.

The available dataset consists of 700000 transactions with 150000 different items as well as additional information which is discussed in section 3.2.1. The transaction data are in XML format, so we use XSLT (Extensible Stylesheet Language Transformations) to capture the data relevant for our task.

The last step produces RDF graph. In this step, a knowledge graph embedding algorithm represents the items as vectors in n-dimensional vector space. "Knowledge graph embedding aims to represent each predicate/entity in a KG as a low-dimensional vector, such that the original structures and relations in the KG are preserved in these learned vectors." [HZLL19]. To represent the RDF graph into embedding space, we use Pyke or Conex algorithm. Pyke is a physical embedding model which keeps linear complexity for high-quality embedding generation [DN19]. Conex learns complex-valued vector representation by combining 2D convolution operation with a Hermitian inner product [DN20].

The goal of the last step is to predict an item, which a customer may buy. A customer can purchase items of a varying quantity. Thus, we work with a recurrent neural network model which can deal with such situations. We use a 50-unit vanilla LSTM model with SGD optimizer (Neural Network Layers in Fig.3.1). The final layer of this neural network is a cosine similarity calculation which predicts the items.

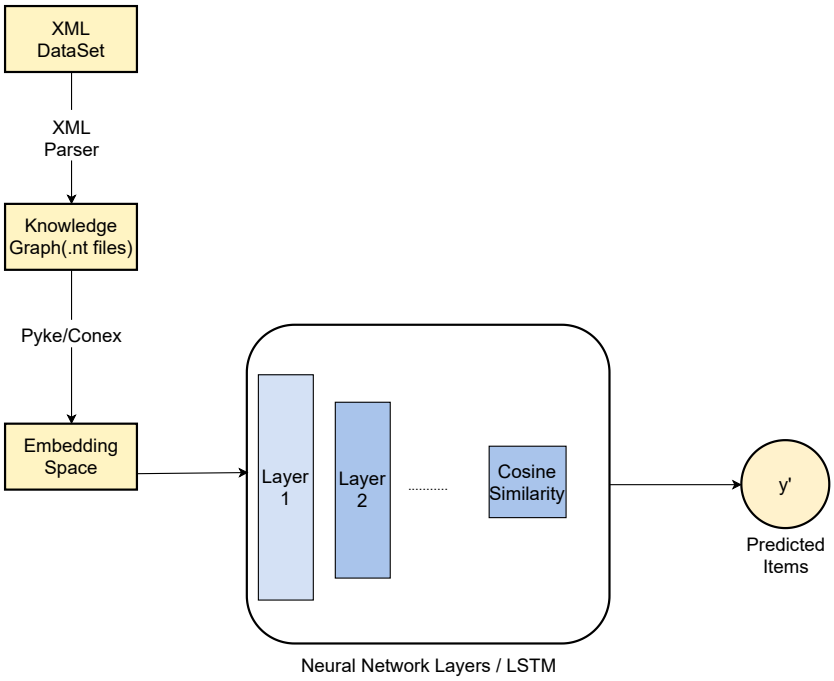

Figure 3.1: Work flow diagram

3.2 XML Parsing

XML parsing is the first data pre-processing step of the proposed approach. Therefore, the know-how of the raw data is necessary to understand the intents of all pre-processing steps. The following section explains the data in detail.

3.2.1 Data

The transaction data are collected from retail stores across the United States. The dataset has around 700000 transaction details for 150000 different items. But the dataset contains much extra information. Table 3.1 shows different categories of information present in the dataset.

Tag	Contents
Header	This section has details like cashier name, workstation group id, retail store id, date and time of system sign in, city name etc.
Statistics	This section contains information related to workstation like station id, operator id etc.

Emplifiles	This section relates cashier details with workstation details.
Sign_on	Section stores the sign-on information of cashier like time, workstation-id, crate number, channel number, employee id etc.
Sign_off	Section stores the sign-off information of cashier like time, workstation-id, crate number, channel number, employee id etc.
Art_sale	This section stores sold item details like discount-details, size-code, colour-code, item description, quantity, price, item tax group id, item class, item category, size name, colour name etc.
Tax	This section stores all the related information of goods tax. It has several sections depending upon the country and states.
Tax_excluded	This section stores items details that are excluded from general taxation criteria.
Total	As the name suggests this section stores details of how much a customer has paid to the cashier, their payment modes, currency details etc.
Media	This section stores drawer information, cashier session payment information etc.
Footer	This section stores extra information like client name, client id, printer code, external id etc.

Table 3.1: Dataset contents

The data related to the items are needed to build the recommender system. Therefore, *Art_sale* is the section of interest for the mentioned approach. Fig.3.2 depicts the part of a sample XML dataset which comes under *Art_sale* tag. The dataset shows features of an item that come along with the transaction data.

3.2.2 Data Cleansing and Feature Engineering

The data available for this thesis are in XML format, and we have decided to parse the data with XSTL. The goal of this XML parsing subtask is to produce triples so that knowledge graph embedding algorithms can use it. Fig.3.3 depicts that we have approximately 1484280 files at the beginning. After the data cleansing operation, the number of files reduces to 718484. The parser has extracted a few features directly while combining others to maintain its uniqueness

```
<bTotalDiscountAllowed>-1</bTotalDiscountAllowed>
<szSizeCode>176-004</szSizeCode>
<szColorCode>09</szColorCode>
**********
**********
<dPRPackingUnitPriceAmount>17.9900</dPRPackingUnitPriceAmount>
<dPRPriceEntryLowAmount>0.0100</dPRPriceEntryLowAmount>
<dPRPriceEntryHighAmount>999999999999999.0000</dPRPriceEntryHighAmount>
**********
<szPRDateValidFrom>20180901000000</szPRDateValidFrom>
<szPRDateValidTo>29991231235959</szPRDateValidTo>
<szPOSItemID>**713428001188004</szPOSItemID>
<lRetailStoreID>1*****</lRetailStoreID>
<szItemID>713428001188004</szItemID>
<szDesc>Jersey *****</szDesc>
<szDescription>Other top Black, M</szDescription>
**********
<szItemCategoryTypeCode>1</szItemCategoryTypeCode>
<lMerchandiseStructureID>1</lMerchandiseStructureID>
<szMerchHierarchyLevelCode>1</szMerchHierarchyLevelCode>
<bDiscountFlag>-1</bDiscountFlag>
<szPieceUnitOfMeasureCode>**</szPieceUnitOfMeasureCode>
**********
**********
<bUseSizeAndColor>-1</bUseSizeAndColor>
<dPieceQuantity>1.0000</dPieceQuantity>
<szItemTaxGroupID>*</szItemTaxGroupID>
<szTypeCode>NRML</szTypeCode>
<bVoidAllowed>1</bVoidAllowed>
<sz**ItemType>Stock</sz**ItemType>
<sz**ProductID>0713428</sz**ProductID>
<sz**BlockedStartDate>20000101</sz**BlockedStartDate>
<sz**BlockedEndDate>29991231</sz**BlockedEndDate>
**********
<szOriginalTaxGroup>1</szOriginalTaxGroup>
<szSizeName>M</szSizeName>
<szColorName>Black</szColorName>
<sz**CodeType>NOOS14</sz**CodeType>
<sz**ItemLookupCode>03155883024012</sz**ItemLookupCode>
<sz**Category>00_A10</sz**Category>
<sz**Class>00_A</sz**Class>
<szOrderNumber>31558</szOrderNumber>
<sz**Season>8</sz**Season>
```

Figure 3.2: Sample dataset

and co-relation with similar features. The extraction methodology and the reason behind the creation of combinatorial features are discussed in the next section.

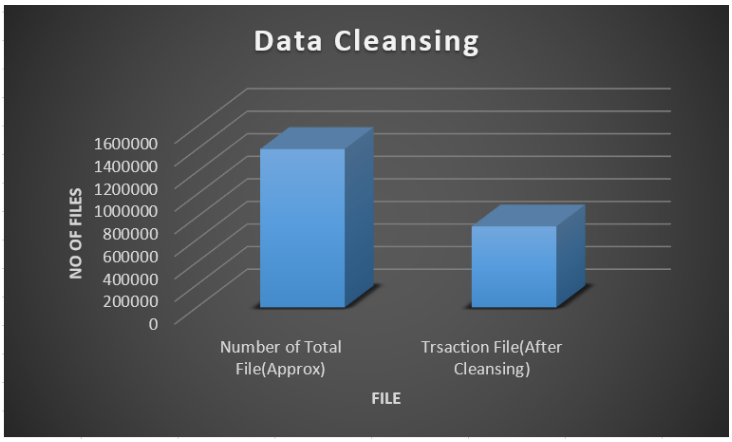

Figure 3.3: Data cleansing

The developed XML parser search for item details inside the database. It is observed that *item return* receipts are similar to *item sell* receipts, means it contains item details. We are only interested in *item sell* details. Therefore, a program is developed which can identify the *item return* receipts and filter it out. Machine learning model needs transactions with at least two items to learn. Thus, the parser has another function which can label out the transaction with a single product. An identifier is needed to identify all products bought by a single customer.

Therefore, another implementation is added, which generates a unique identifier for each transaction and adds this id to all the products sold on that single transaction to identify it later. The implementation details of all these logics are discussed in Chapter 4.

3.3 Physical Embedding Model

The previous procedure produces .nt files which contain information of items along with its features. The goal of this step is to represent these items in the embedding space. Therefore, we have chosen knowledge graph embedding algorithms Pyke and Conex for this task.

3.3.1 Pyke

In this section, we discuss the reason behind selecting Pyke. We also discuss the changes made in the dataset to adapt it to Pyke.

Reason

Pyke has a linear space complexity with close to linear run time complexity. Pyke can scale graphs containing up to millions of triples. Pyke has also achieved cluster purity up to 26% better than the existing state of the art. Pyke is 22 times faster than the existing embedding solutions in the best-case scenario [DN19]. As we are dealing with a large dataset in this thesis, we, therefore, consider Pyke as the best candidate for the proposed approach.

Adapting Pyke

One of the shortcomings of Pyke is that it can not work with the literals. Therefore, we have changed the literals into IRIs so that Pyke can access it.

```
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasPosId> <http://**2.com/pos#**759871002199003> .
<http://**2.com/article#759871002199003> <https://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://**2.com/class#Article> .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasTransactionId>  "12010119020190313124526" .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasDescription>  "Basic 1"@en .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasBigDescription>  "Tank top Black, S"@en .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasDepartment>  "1643" .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasPrice>  "2.9900" .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasColour> <http://**2.com/colour#Black09> .
<http://**2.com/colour#Black09> <http://www.w3.org/2000/01/rdf-schema#colourLabel> "Black"@en .
<http://**2.com/colour#Black09> <http://**2.com/vocabulary#colourCode> "09" .
<http://**2.com/colour#Black09> <https://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://**2.com/class#Colour> .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasItemType> "Stock"@en .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasTypeCode> "NRML"@en .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasSize> <http://**2.com/size#S176-003> .
<http://**2.com/size#S176-003>  <http://www.w3.org/2000/01/rdf-schema#sizeLabel> "S" .
<http://**2.com/size#S176-003>  <http://**2.com/vocabulary#sizeCode> "176-003" .
<http://**2.com/size#S176-003>  <https://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://**2.com/class#Size> .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasItemCatagory/> "1" .
<http://**2.com/article#759871002199003> <http://**2.com/vocabulary#hasClass/> <http://**2.com/**class#00_D> .
<http://**2.com/**class#00_D> <http://**2.com/vocabulary#has**Catagory/> "00_D15" .
<http://**2.com/**class#00_D> <https://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://**2.com/class#**> .
```

Figure 3.4: Sample triples for Pyke

For example, the dataset has features with colour name and colour code. At first, these features were extracted as literals, but Pyke could not process that information. Then, we combined these two features (e.g. colour name and number both pointing towards the same attribute) to make it an IRI (Fig.3.4). With this same idea, we have combined the sizeCode, sizeLabel as well as product class, product category into single IRIs.

Below diagram (Fig.3.5) shows the output from Pyke. Pyke represents each item along with its features in a 50-dimensional vector space. To make the embeddings visible, we show only seven dimensions (seven columns for each row) in this diagram. The next step (i.e. LSTM learning) requires item embeddings to train a machine learning model. Therefore, we need to

extract only the item embeddings from this embedding dictionary. So, a program is developed which extricate the item embeddings from the dictionary and transfer it to the LSTM model for further process.

A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7
http://***.com/article#569978020199004	0.260000363413862	0.285158456308968	0.710647528238192	0.54514165867959	0.415058203389439	0.053933452260284	0.863886147997126	0.668347761217192
http://***.com/vocabulary#hasPosId	0.260128648227633	0.285154781561248	0.710665672701845	0.545153012353966	0.41513868598337	0.054994838665411	0.863904746890076	0.668501696192927
http://***.com/pos#HM569978020199004	0.260123225148688	0.285162897310038	0.710655816930064	0.545142312439281	0.415049078537208	0.053949766048194	0.863892812288081	0.66836497671374
https://www.w3.org/1999/02/22-rdf-syntax-ns#type	0.260109774153014	0.285171157533926	0.710408636673635	0.545163177076337	0.415142090171392	0.053882137556971	0.863911813916598	0.668209144350086
http://***.com/class#Article	0.260141697674295	0.285154801988408	0.710623836957871	0.545153791557796	0.415146824190716	0.053934492819013	0.863995322361834	0.66830538526406
http://***.com/vocabulary#hasColour	0.260185965132867	0.285155474230079	0.710629942168243	0.545146621864218	0.415158028425348	0.053962836161505	0.863923510527381	0.668328230826088
http://***.com/colour#Blue78	0.260149533921008	0.28515874934468	0.710616993264012	0.545154448541166	0.415134442215028	0.053932986707256	0.863883171451042	0.668328013557798
http://***.com/class#Colour	0.260157789384558	0.285157780366833	0.710634179595553	0.545134322988588	0.415144364102565	0.053936468206042	0.863901127232535	0.668287835626179
http://***.com/vocabulary#hasSize	0.260131505956152	0.285161974976042	0.710645173957822	0.545149516443402	0.415141811755195	0.0539320195511	0.864103483028593	0.668325959185335
http://***.com/size#L191-004	0.260142805033661	0.285154537424252	0.710639908602124	0.545156236788992	0.415162301966149	0.053940497539369	0.863913276261471	0.668350462950222
http://***.com/class#Size	0.260140365417212	0.285156749467115	0.710610398426489	0.545227483889939	0.415199672878185	0.053935087462178	0.863888261795837	0.668330053002957
http://***.com/vocabulary#hasClass/	0.260151862732109	0.285155387924257	0.71063965336092	0.5451624106404	0.415160429323482	0.053934007877149	0.863915617624551	0.668493856188831
http://***.com/hmclass#00_F	0.260136980189723	0.285162087190312	0.710656039469844	0.545148747567996	0.415138953460511	0.054067903634352	0.863906641227837	0.668362089009938
http://***.com/class#HM	0.260157183806914	0.285160410619154	0.71062514257478	0.545157296033358	0.415086527056549	0.053928401975714	0.863947421815918	0.668458392137872
http://***.com/article#685816013199003	0.260122989183315	0.285157317405601	0.710632368170165	0.545245387501483	0.415020444427706	0.053939622183549	0.863949249047153	0.668252553418185
http://***.com/pos#HM685816013199003	0.260130583107675	0.285157316691647	0.71063028695023	0.545172491451841	0.415122266532886	0.053954160721635	0.863913251221575	0.669101701958182
http://***.com/colour#Turquoise87	0.260093180355077	0.285157610167049	0.710633375548147	0.545148750317238	0.415116065308863	0.05394321038505	0.863884347905236	0.668312381181057
http://***.com/size#M186-003	0.260143730832926	0.285155077883852	0.710652492632777	0.545168362110492	0.415143648946619	0.053930286854253	0.863917191752632	0.667555864707474
http://***.com/article#685816016199003	0.260149880470921	0.28515706243625	0.710240804031716	0.54516218289434	0.415139786206999	0.053938638261883	0.863905021522218	0.668268493512227
http://***.com/pos#HM685816016199003	0.260135126879589	0.285157000216324	0.71058006352706	0.545161725633834	0.415153812546305	0.0539286905764	0.86392552072113	0.668381727613714
http://***.com/colour#Orange37	0.260128250340266	0.285156928617956	0.710596387061977	0.545165072109408	0.415129936355049	0.053918079931817	0.863875539332448	0.668184782771982
http://***.com/article#612334006188003	0.26009302525903	0.284707294723634	0.710609247860328	0.545153270164055	0.415144350220535	0.053958148384382	0.863934020961822	0.668283807002644

Figure 3.5: Sample embeddings

3.3.2 Conex

Conex is a knowledge graph embedding algorithm which has achieved superior performance [DN20]. Therefore, we use Conex for the proposed approach.

Adapting Conex

The input data for Conex requires a slightly different format than Pyke. For Conex, the triples need a "tab" space between subject, predicate and object. Also, the input data need to be divided into train, test and validation sets. Fig.3.6 depicts a sample input dataset for Conex. So, we use a regular expression tool *Grepwin* to transfer the format of the dataset from Fig.3.4 to Fig.3.6. We have searched four patterns and replaced them with *Grepwin*. Table 3.2 shows those patterns with corresponding replacements in a sequential manner.

Searched Pattern	Replacement
>{0,1} \.$	
>[ ]*	\t
<	
"	

Table 3.2: Replacement patterns

3.4 LSTM

This is the final step of our proposed approach. This step retrieves the item embeddings from the previous step. The goal of this subtask is to feed the item embeddings into a recurrent neural network (RNN) so that RNN can learn and start predicting the next item a customer can buy. In the following subsections, this thesis discusses the reason behind choosing LSTM, the model architecture, and the final layer (similarity calculation) of LSTM.

Figure 3.6: Sample triples for Conex

3.4.1 Model Selection

In an e-commerce setting customers can buy products of a variable quantity. For example, customer *A* buys ten items, where customer *B* buys six items. Therefore, a suitable model should handle inputs of different length. An RNN algorithm is ideal for such a scenario. LSTM is a kind of RNN which is popularly used for developing recommender systems [LLJZ16, YZC+18]. Therefore, we choose to use LSTM for this approach.

3.4.2 LSTM Architecture

We defined our model with 50 LSTM units hidden layer and with 50 neuron output (Fig.3.7). The model inputs nine vectors each of 50-dimensional length where the output is a 50-dimensional vector. The output vector is passed on to the cosine similarity calculation stage, which in turns produce the final prediction. The model was first tested with Adam optimizer, but the performance was low. Therefore, we tested the model with SGD optimizer along with Nesterov momentum and received better performance. We will discuss this topic further with relevant implementation details in chapter 4.

Similarity Calculation

Cosine similarity is a metric used to find the similarity among vectors. For example, we have three sentences. *Bob is a Master's student*, *Hary is a PhD student*, *Rody is a professor*. Suppose we represent *Bob*, *Hary* and *Rody* as vectors in three-dimensional space and try to find the similar

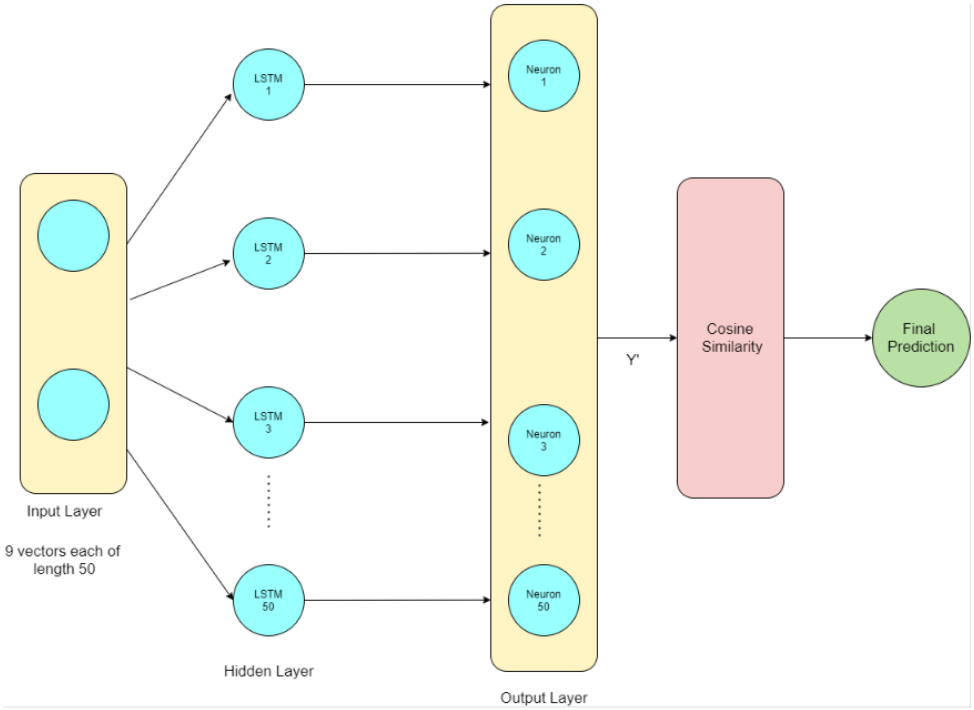

Figure 3.7: LSTM architecture

vectors. The required formula for this calculation is :

$$Cos\theta = \frac{\vec{a} \cdot \vec{b}}{\| \vec{a} \| \| \vec{b} \|} \tag{3.1}$$

Where $\vec{a}$ and $\vec{b}$ are the two different vectors (i.e. *Bob, Hary*) and this formula tries to calculate the angle between these two vectors. So, the similar vectors have lesser angels between them compared to the dissimilar once.

Fig. 3.8 shows *Bob*, and *Hary* both are a student; therefore, they are similar vectors (i.e. the angle between them is lesser) compared to *Rody*. Cosine similarity is very popular to calculate the similarity among documents. The main advantage of cosine similarity over other similarity calculations (e.g. Euclidian distance) is that two similar documents/vectors (like the word *student* comes hundred times in one document and ten times in the other document) can be far apart in the vector space because of their respective size. However, the angle between them is still less.

We have used the above idea to find top "N" possible items a customer can buy. So this step calculates the cosine similarity between y' [a prediction received from LSTM model] with all the possible items in the catalogue (e.g. around 150000 different items) to predict the outcome.

3.5 Baseline Algorithm

Baseline calculates the centroid of each transaction inputs in place of an LSTM model used by the proposed model (Fig. 3.9). So the data pre-processing steps (XML parsing, knowledge graph embedding) of the baseline are similar to the proposed model. The item embeddings of each transaction are fed into the centroid calculation stage. The output (i.e. calculated centroid)

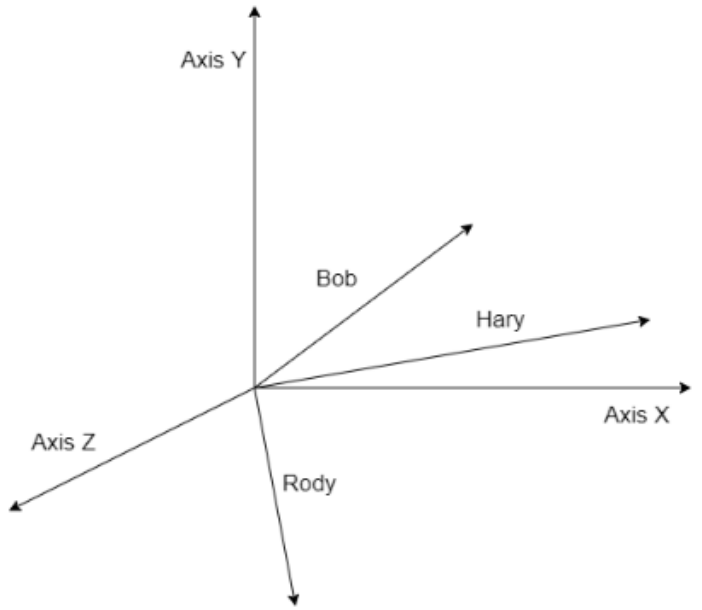

Figure 3.8: Cosine similarity example

is passed onto a cosine similarity stage where a cosine similarity is calculated between centroid with all other catalogue elements to find out top "N" most similar items.

Why we choose this model?

We check the efficiency of the proposed machine learning (LSTM) model by comparing with the baseline (without machine learning) model. If the prediction from both models (proposed model and baseline) are low, we conclude the data pre-processing steps were not sophisticated. If prediction from the proposed model is inferior compared to the baseline, we infer the LSTM model needs modification.

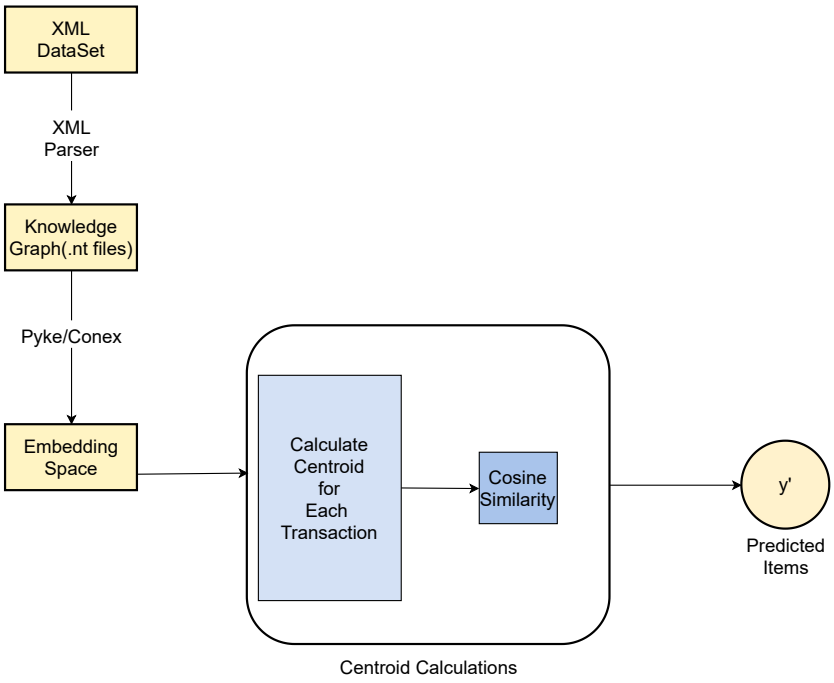

Figure 3.9: BaseLine algorithm workflow

4 Implementation

This chapter describes the design and implementation of the proposed approach, along with the baseline algorithm. It first describes the techniques and the technologies used; then we will focus on the individual components in detail as well as how the parts integrate. For data security issues, some letters in the code are replaced by *'s.

4.1 Overview

4.1.1 Technologies and Packages

As mentioned earlier, the very first step of this thesis is XML parsing, the rest of the work (Pyke/Conex, LSTM Learning/ centroid calculation, similarity calculation) is performed with Python with the help of many standard libraries. All the libraries used in this thesis are listed at table 4.1.

Package Name	Use/Description
NumPy	Used for math calculations, string manipulation
TensorFlow	Used to build LSTM
Sklearn train_test split	Used to split data among test and train datasets
pickle	Used for storing and uploading the dataset. Easily transport data.
random	Used for the random shuffle of transactions
intertools	Used to slice the large dictionaries and arrays to a smaller size for testing
tqdm	Helps to visualize a program execution time

Table 4.1: List of packages used

4.1.2 General Workflow

The overview of the proposed approach is already discussed in section 3.1. Before the in-depth discussion about the implementation of each component, a brief overview of the entire workflow is given. As Fig.3.1 represents, an XML parser converts the unstructured knowledge into a structured format (i.e. generate triples). Later a knowledge graph embedding algorithm (i.e. Pyke and Conex) maps the triples in the embedding space. Then an LSTM model is trained on these item embeddings to predict the item a customer may buy. The baseline algorithm calculates the centroid of each transaction in place of LSTM. Fig.3.9 depicts the workflow of the baseline algorithm.

4.2 System Design and Architecture

4.2.1 Triples Generation

Fig.4.1 depicts the interaction between different components in our proposed model. The following sections discuss each part in detail.

Fig. 4.1 shows all the files that are required to generate the triples out of the XML dataset. A dataflow diagram (Fig. 4.2) better describes the flow of data among the different subcomponents of this task. The starting point of this task is to have the input data. All the data available for this thesis are referred to as *Input Data*. The details of the dataset were provided in section 3.2.1. *XML parsing logic* contains all the functions necessary to parse the data. *Execution file* is an .exe file where the input data location and output location are mentioned. The .exe file calls *XML parsing logic* file to produce the final output.

The following four steps explain the *XML parsing logic*. The steps can be broadly divided into two categories, namely selection and generation procedure. The selection procedure identifies the intended section/ items in the XML dataset while the later approach generates the triples. The first two steps perform the selection operations, and the generation operations are performed by the last two steps. The codes are written in XSLT language for the mentioned steps.

Step 1:

As mentioned in section 3.2.2, a function is built to identify all items bought in a single transaction. So, four features of a transaction are identified and concatenated to generate a unique id. This unique id is later attached to all the items bought in a single transaction. Listing 4.1 depicts a code snippet along with the necessary comments.

Listing 4.1: Generate unique identifier

```
1  <xsl:variable name="vals1" select="//FOOTER/Hdr/lTaCreatedRetailStoreID"/>
2  <xsl:variable name="vals2" select="//FOOTER/Hdr/lTaCreatedWorkstationNmbr"/>
3  <xsl:variable name="vals3" select="//FOOTER/Hdr/lTaCreatedTaNmbr"/>
4  <xsl:variable name="vals4" select="//FOOTER/Hdr/szTaCreatedDate"/>
5  <xsl:variable name = "val">
6
7     <!-- store the unique number in a variable -->
8     <xsl:value-of select="concat($vals1,$vals2,$vals3,$vals4)"/>
9
10 </xsl:variable>
```

The id (*val* in Listing 4.1) is assigned as a predicate to each item (Listing 4.2).

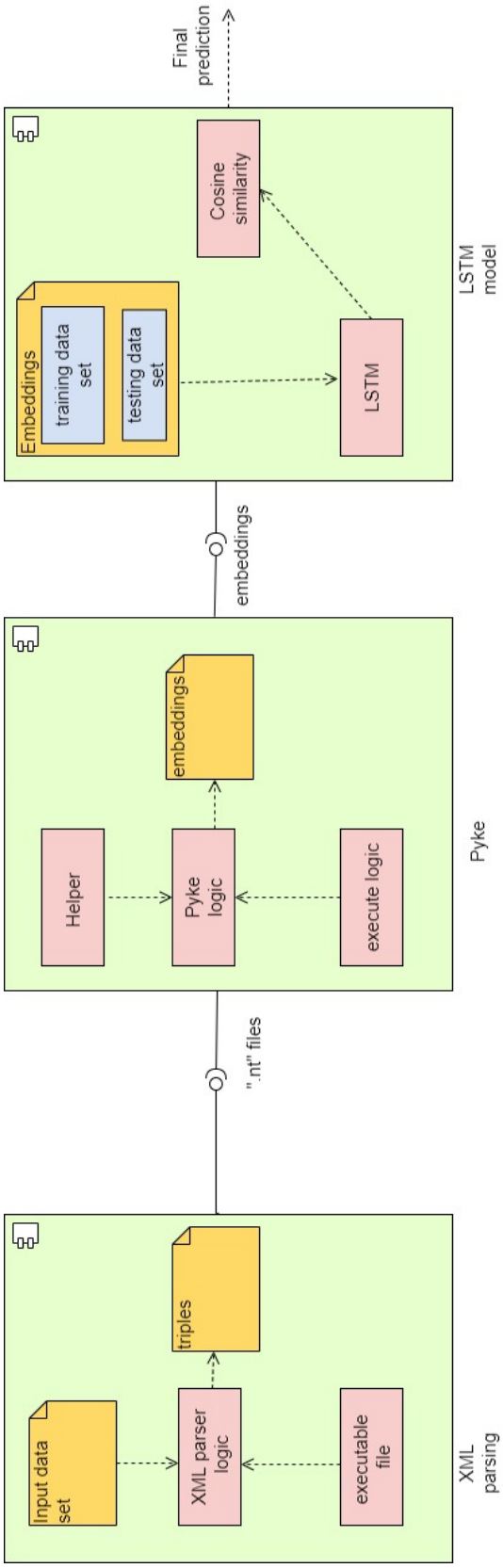

Figure 4.1: Component diagram of proposed model with Pyke

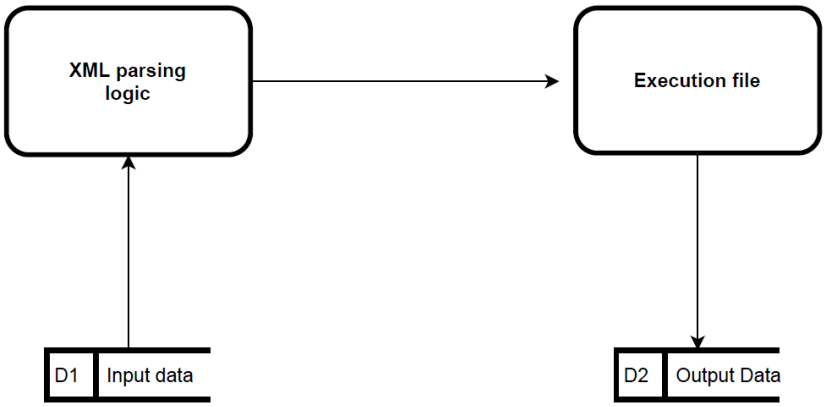

Figure 4.2: Data-Flow diagram for XML parsing

Listing 4.2: Assigning unique id to each item

```
1 &lt;http://***.com/article#<xsl:value-of select="szItemID"/>&gt; &lt;http://***.com/
      vocabulary#hasTransactionId&gt;  "<xsl:value-of select="$val"/>".
```

Step 2:

Another implementation checks if an item is valid (Listing 4.3). As mentioned earlier, the data have many similar sections which come along with identical item features (i.e. item sell, item return). So, we have identified a unique feature named *Unit price* in the dataset, that always comes along with an item sell. The *Unit price* also comes with the item return. So, a function checks if the *Unit price* comes from item sell only. If it comes from item sell, the program carries on with the next operations. Otherwise, the item is considered invalid and ignored.

Listing 4.3: Valid item check

```
1  <!-- if the item not in item_return -->
2  <xsl:if test= "not(//ART_RETURN)">
3    <xsl:if test ="$counter &gt; 1" >
4    <!-- if the item comes from other place -->
5      <xsl:for-each select="//ARTICLE">
6      <!-- logic for valid item check -->
7        <xsl:variable name = "validitem">
8          <xsl:value-of select="dPRPackingUnitPriceAmount"/>
9        </xsl:variable>
10
11       <xsl:choose>
12       <!-- if item is valid print the desired element -->
13           <xsl:when test="$validitem !=''">
```

Step 3:

After an item is identified as valid; the program parses several features of that item. Pyke cannot process the literals in the triples. As mentioned in Chapter 3, the dataset has some complementing characteristics which point to the same feature (i.e. *colour_name* and *colour_code*, *size_name* and *size_code*, etc.). A code snippet concatenates these complementing characteristics into a single feature and stores it as an IRI in place of literals. Listing 4.4 shows the

implementation where *colour_name* and *colour_code* are concatenated (see line no.:1).

Listing 4.4: Feature concatenation

```
1 &lt;http://***.com/article#<xsl:value-of select="szItemID"/>&gt; &lt;http://***.com/
      vocabulary#hasColour&gt; &lt;http://***.com/colour#<xsl:value-of select="
      szColorName"/><xsl:value-of select="szColorCode"/>&gt; .
2 &lt;http://***.com/colour#<xsl:value-of select="szColorName"/><xsl:value-of select="
      szColorCode"/>&gt; &lt;http://www.w3.org/2000/01/rdf-schema#colourLabel&gt; "<
      xsl:value-of select="szColorName"/>"@en .
3 &lt;http://***.com/colour#<xsl:value-of select="szColorName"/><xsl:value-of select="
      szColorCode"/>&gt; &lt;http://***.com/vocabulary#colourCode&gt; "<xsl:value-of
      select="szColorCode"/>" .
```

Step 4:

Other features are parsed and stored in a standard triple format. Listing 4.5 depicts a typical code snippet for a triple generator. For generating a triple the left-hand side of a URI is typecasted while the right-hand side is fetched on the fly from the XML dataset. For example(see Listing 4.5) "< http://***.com/article#" is type casted part of the URI and "<xsl:value-of select="szItemID"/>gt" part fetches the data on the fly from XML dataset. Table.4.2 shows the extracted features of an item from the given XML dataset.

Listing 4.5: Triple generation

```
1 &lt;http://***.com/article#<xsl:value-of select="szItemID"/>&gt; &lt;http://***.com/
      vocabulary#hasPosId&gt; &lt;http://***.com/pos#<xsl:value-of select="szPOSItemID"
      />&gt; .
```

Item Features
ItemID
POSItemID
transaction identifier
short description
long description
POSDepartmentID
Price
ColorName
ColorCode
**ItemType
TypeCode
SizeName
SizeCode
ItemCategoryTypeCode
**Class
**Category

Table 4.2: List of item features

Pyke needs all its input files to be stored under a single directory. As the generated triples are stored in different folders, so we used a shell script (Listing 4.6) to copy files from these

folders and paste it under a single folder.

Listing 4.6: Command for folder re-structure

```
find ./input -name '*.nt' -exec mv {} /media/sami/70AE0846AE080776/folder2 \;
```

Listing 4.6 shows a shell script command where the first parameter (*./input -name '*.nt' -exec mv*) is source folder location (here we search for all .nt files) and the second parameter (*/media/sami/70AE0846AE080776/folder2*) is destination folder location. Thus, as Fig.4.1 shows, this step concludes with the generation of nt files.

4.2.2 Vector representation of triples

Pyke

The triples were transformed into embedding space with the help of Pyke. As described in Chapter 3, Pyke is a knowledge graph embedding algorithm which is used in this thesis. As Fig.4.1 depicts, Pyke comes with a helper class and an execute logic class. These classes call the main logic of Pyke to generate the necessary embeddings. Pyke generates embeddings in a *DataFrame*. So, one cannot identify the vector representation of an item. Fig.4.3 depicts this problem which only shows the vectors. Therefore, it is important to generate a *Dictionary* where every vector is associated with an object (Fig.3.4). In short, we have replaced a *DataFrame* with a *Dictionary* to store the embeddings in Pyke.

Standard	Standard	Standard	Standard	Standard	Standard	Standard	Standard
0	1	2	3	4	5	6	7
0.2600003634138624	0.28515845630896813	0.7106475282381925	0.5451416586795896	0.4150582033894386	0.05393345226028392	0.8638861479971256	0.6683477612171915
0.2601286482276332	0.2851547815612477	0.710665672701845	0.5451530123539661	0.41513868598336995	0.05499483866541072	0.8639047468900758	0.6685016961929268
0.2601232251486878	0.2851628973100374	0.7106558169300636	0.5451423124392809	0.4150490785372077	0.05394976604819424	0.863892812288081	0.6683649767137396
0.2601097741530138	0.2851711575339254	0.7104086366736354	0.5451631770763373	0.4151420901713916	0.05386213755697083	0.8639118139165979	0.6682091443500856
0.2601416976742950	0.2851548019884076	0.7106238369578713	0.5451537915577958	0.4151468241907164	0.05393449281901265	0.8639953223618342	0.6683053852640599
0.2601859651328672	0.2851554742300792	0.7106299421682427	0.5451466218642178	0.4151580284253475	0.05396283616150469	0.8639235105273807	0.6683282308260885
0.2601495339210082	0.2851587493467983	0.7106169932640115	0.5451544485411661	0.4151344422150279	0.05393298670725579	0.8638831714510421	0.6683280135577979
0.2601577893845579	0.2851577803668327	0.710634179595553	0.545134322985883	0.4151443641025652	0.0539364682060413	0.863901127232535	0.6682878356261794
0.2601315059561525	0.2851619749760421	0.7106451739578216	0.5451495164434024	0.4151418117551955	0.05393201955109966	0.8641034830285925	0.6683259591853352
0.2601428050336614	0.2851545374242519	0.7106399086021241	0.5451562367889924	0.4151623019661489	0.05394049753936857	0.8639132762614713	0.6683504629502218
0.2601403654172121	0.2851567494671149	0.7106103984264895	0.5452274838899392	0.4151996728781857	0.0539350874621774	0.8638882617958369	0.6683300503029571
0.2601518627321086	0.2851553879242571	0.7106396533609204	0.5451624106404	0.4151684293234817	0.05393400787714858	0.8639156176245513	0.6684938561888306
0.2601369801897235	0.2851620871903124	0.7106560394698439	0.5451487475679959	0.4151389534605108	0.05406790363435149	0.8639066412278369	0.6683620890099377
0.2601571838069144	0.2851604106191538	0.7106251425747799	0.545157296033358	0.4150865270565489	0.05392840197571343	0.8639474218159184	0.6684583921378723
0.2601229891833150	0.2851573174056009	0.710632368170165	0.5452453875014835	0.4150204444277064	0.05393962218354859	0.8639492490471531	0.6682525534181852
0.2601305831076749	0.2851573166916746	0.7106302869502297	0.5451724914518411	0.4151222665328859	0.0539541607216348	0.8639132512215751	0.6691017019581815
0.2600931803550775	0.2851576101670485	0.7106333755481472	0.5451487503172381	0.4151160653088633	0.05394321038505033	0.8638843479052356	0.6683123811810577
0.2601437308329263	0.2851550778838852	0.7106524926327773	0.5451683621184925	0.4151436489466193	0.05393828685425308	0.8639171917526318	0.6675558647074745
0.2601498804709214	0.2851570624625033	0.7102408040317162	0.5451621828943398	0.4151397862069994	0.05393863826188278	0.863905021522218	0.6682684935122267
0.2601351268795893	0.2851570002163243	0.7105800635270595	0.5451617256338341	0.415153812546305	0.05392869057639991	0.863925520721113	0.6683817276137142
0.2601282503402660	0.2851569286179564	0.7105963870619765	0.5451650721094082	0.4151299363550494	0.0539180799318170	0.8638755393324479	0.6681847827719818
0.2600930252590299	0.2847072947236340	0.7106092478603278	0.5451532701640546	0.4151443502053536	0.05395814838438151	0.8639340209618219	0.6682838070026443
0.2601373017844648	0.2851547307422048	0.7106316147643279	0.5451360054017985	0.4151410771224939	0.05394199498582424	0.8639105964963234	0.6683312856262021
0.2601388232800999	0.2851568197655845	0.7106252220608016	0.5450846537362407	0.4151391127937098	0.0539412760419091	0.863845843847274	0.6683573502452
0.2601394076549051	0.2851545641421876	0.7106068409845308	0.5451912804947482	0.4151388161621062	0.05394153679116145	0.8639040081953411	0.6683404225854693
0.2601369535283981	0.2851569295932486	0.7106172457678658	0.5451691034586063	0.4151395575801100	0.05395408384742609	0.8639022931862676	0.6683732609754766
0.2601159113758611	0.2851574989940762	0.7106433807408536	0.5447035520444128	0.4151413297710224	0.0539369081434704	0.8638861122344578	0.6683003793932493
0.2601428336157664	0.2851525711874321	0.7106455556424299	0.5451071070814582	0.4150999245188491	0.05393494336268372	0.8639063728658398	0.6683657257488379
0.2601427057855355	0.2851624551093867	0.7105316033083556	0.5451742195817415	0.4152109464395631	0.05393825929782075	0.8639323815434798	0.6679768052219944

Figure 4.3: Original Pyke embeddings

Conex

As discussed in Chapter 3, Conex is a knowledge graph embedding algorithm which is used in this thesis to represent the items as vectors in n-dimensional vector space. Conex generates real and imaginary vector for every resource. Therefore, we concatenate this real and imaginary vectors for every item to generate the final embedding. Fig.4.4 depicts the component diagram of the proposed model with Conex.

4.2.3 Prediction Generator

As Fig.3.5 shows, the previous steps produce a dictionary with the objects as the keys and the corresponding vector representations in its values. The goal of this subtask is to generate the final predictions. Therefore, the following eight steps carry out the prediction operation. The code listings presented in these steps are written in Python language.

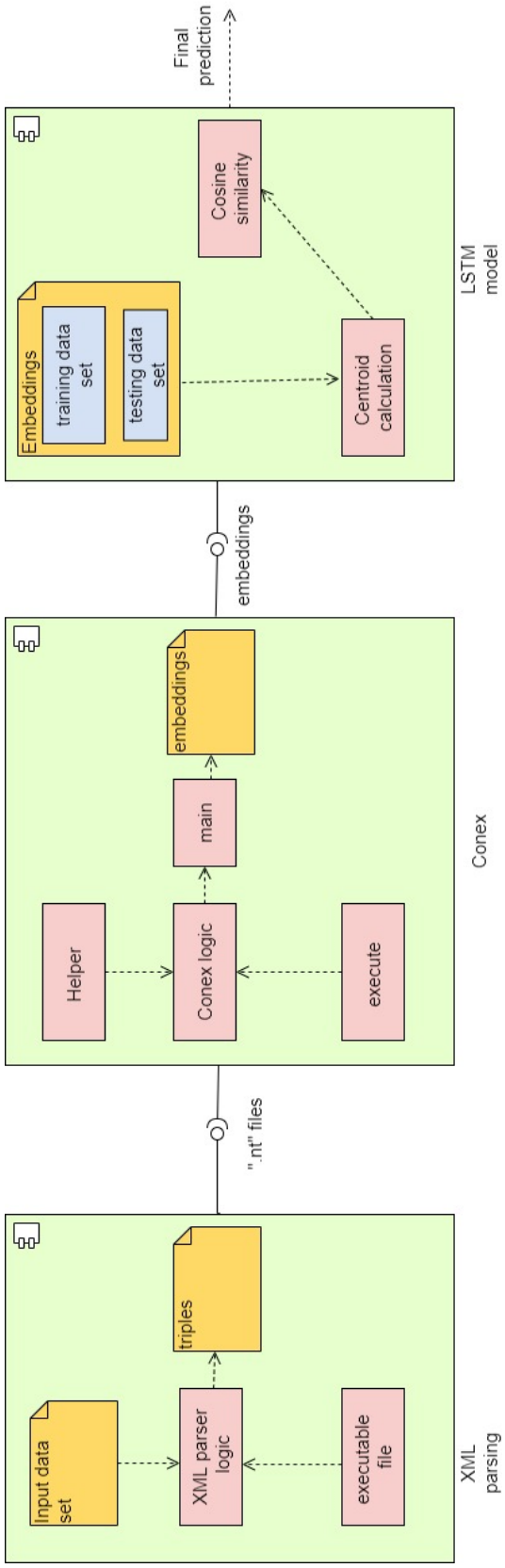

Figure 4.4: Component diagram of proposed model with Conex

Step 1:

The LSTM model needs to be trained on the item embeddings from the transaction details. Therefore, it is necessary to identify the vector representations of each item. So a list of lists is created which stores all the item identifiers from the triples. Here the inner lists store all the items bought in a single transaction. Thus, the outer list contains the lists of all transactions.

Listing 4.7: Implementations for item identification

```
## this cell extracts the url from url list which contains the embeddings for an item

## .nt files location
path_root = '/media/sami/70AE0846AE080776/folder2'

all_urls = []
for root, dirs, files in os.walk(path_root):
    for file in files:
        links = []
        to_Read = os.path.join(root, file)
        f = open(to_Read,"r")
        urls = f.readline()
        while(urls):
            links.append(re.findall('^<http:\/\/hm2\.com\/article#\d{1,18}>', urls))
    ## identify the items inside triples
            urls = f.readline()
        # convert list of list to list
        merged = list(itertools.chain.from_iterable(links))
        # remove duplicates
        unique_urls = list(set(merged))
        # remove special charecter < ,>
        removetable = str.maketrans('', '', '<>')
        rmv_specialchar = [s.translate(removetable) for s in unique_urls]
        f.close()
        # store in final list
        all_urls.append(rmv_specialchar)
        # empty the lists
        del links[:]
        del merged
        del unique_urls
        del rmv_specialchar

pickle.dump( all_urls, open(storage_path+"/all_urllist.p", "wb" ) )
```

Listing 4.7 depicts that the item identifiers are stored in *all_urls*. Here a small experiment is performed to check the length of each transaction (number of items bought per transaction). The data shows that the maximum number of items bought in a single transaction is 268, where most of the transactions are comprised of less than ten items. Therefore, a vertical bar chart is plotted to represent the item frequencies per transaction.

Fig.4.5 depicts the statistics of transactions where the x-axis shows the number of items in a transaction, and the y-axis represents the number of transactions. As the diagram shows, the transactions that comprise of more than ten items are negligible in quantity. Therefore, we decide to train the LSTM model with the transactions that contain up to ten items and skip the rest of the transactions. Machine learning model cannot learn from transactions with one item. Therefore, the transactions that contain a single item are labelled out. Listing 4.8 shows that these transactions are stored in a list named *intended_urls*.

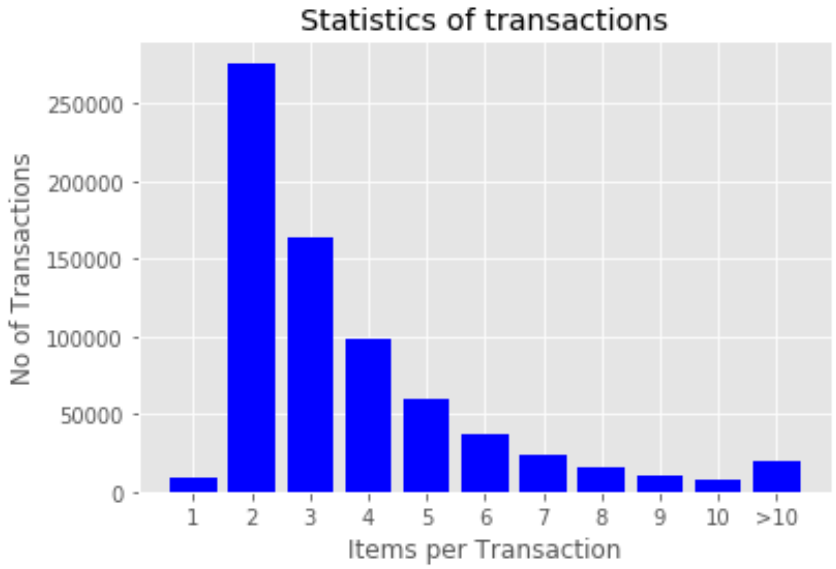

Figure 4.5: Statistics of transactions

Listing 4.8: Implementations to store transactions up to ten items long

```
## store transactions up to 10 products long

unusual_length=[]
intended_urls = [i for i in all_urls if len(i)< 11 and len(i)>1] # transaction length
    = 10, that is why <11
```

Step 2:

So *intended_urls* is a list of lists where the inner lists store transactions (item identifiers) that contain up to ten items. The next task is to identify the corresponding vector form of each item and keep it in a list. Listing 4.9 depicts that these vectors are stored in a list named *final_list*.

Listing 4.9: Implementation to identify Vector forms of items

```
## this cell extracts the embeddings of the corrosponding items

final_list = []

for items in intended_urls:

    new_list = []
    for key in items:


        if key in final_embeddings:  ## final_embeddings received from Pyke/Conex

            new_list.append(final_embeddings[key])
    final_list.append(new_list)

pickle.dump( final_list , open(storage_path+"/extracted_embedddings.p", "wb" ) )
```

The implementation shows that the item identifiers are searched against the key values of dictionary received from knowledge graph algorithm (dictionary name: *final_embeddings*). If a match is found the corresponding vector is stored in a list named *new_list*. This *new_list* is

kept inside another list named *final_list*.

Step 3:

We want all the transactions to have a uniform length of ten. Therefore, if a transaction length is less than ten, it requires to left pad with 0's[DR19]. Listing 4.10 shows the implementation where the vector forms of transactions are stored in a list named *a*.

Listing 4.10: 0-padding of vectors

```
1  ## this cell does the left padding for each transaction
2
3  a = [None] * len(final_list)
4
5  # hardcoded the maximum no of product in the transaction list as 10
6  for i in range(0,len(final_list)):
7      if len(final_list[i])<11:
8          pad = 10 - len(final_list[i])
9          a[i] = [[0]*50]*pad+(final_list[i])
10
11 pickle.dump( a, open(storage_path+"/padded_embedding.p", "wb" ) )
```

Step 4:

The previous step provides all the transactions in vector forms. So, the next task is to divide the data into the train and test set. With the help of the *sklearn* library, the data is divided into 80 : 20 ratio. After this, the data inside the train and test sets need to be divided into input streams and labels. As each transaction is of length ten, we identify the tenth item as the label and the first nine items as the input stream.

Step 5:

As Fig.3.7 shows we develop an LSTM with 50 LSTM units and the output layer has 50 neurons (see line 20 and 21 in Listing 4.11). We configure the LSTM to take input of 9 items, and every single vector is of length 50 (see line no. 12 and 15 in Listing 4.11). Listing 4.11 depicts the entire implementation details of this LSTM model. Preliminary results showed that SGD optimizer performs better with this model than Adam optimizer. Therefore, we choose to work with SGD optimizer. As the implementation shows, we store the predictions of this model in a list named *output_embeddings*.

Listing 4.11: Implementation of LSTM model

```
## univariate lstm
## with sgd optimizer
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras import optimizers

output_embeddings = []

# number of items in a single input
n_steps = 9

# reshape from [samples, item number] into [samples, item number, vector length]
n_features = 50
x = num_x_train.reshape((num_x_train.shape[0], num_x_train.shape[1], n_features))
y = num_y_train
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(50))
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='mse')
# fit model
model.fit(x, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = num_x_test
x_input = x_input.reshape((num_x_test.shape[0], n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
output_embeddings.append(yhat)
```

Step 6:

Fig.3.7 shows that the final step of this LSTM model does a cosine similarity calculation. As mentioned in Chapter 3, the cosine similarity is calculated between y' (a prediction from the LSTM model) and the list of all available items. Therefore, the primary task is to generate a dictionary which contains the item embeddings as the values and the item identifiers as the keys. Listing 4.12 depicts the implementation details of the mentioned task. As the implementation shows the logic loops through the items of the dictionary *entire_embeddings* received from Pyke. This logic tries to identify the keys which start with specific characters. If a match is found, then it inserts that dictionary item to a new dictionary called *item_embeddings*.

Listing 4.12: Creation of item embedding dictionary

```
## Creating a dictionary with only the item embeddings

# dictionary of item embeddings
item_embedddings = {}
for key, val in entire_embeddings.items(): ## entire_embeddings received from Pyke/
    Conex
    if key.startswith('http://hm2.com/article#'):

        item_embedddings.update( {key : val} )

print(len(item_embedddings))
pickle.dump( item_embedddings, open( "item_embedddings.p", "wb" ) )
```

Step 7:

The next step loops through the y' s (predictions from LSTM) and do cosine calculations. Then the top "N" predictions are stored in a list to check how many predictions are right. Listing 4.13 shows the implementation details of this logic.

Listing 4.13: Implementation of Cosine-similarity calculation

```
1  from tqdm import tqdm
2
3  ## this cell calculates the cosine similarity among y and y_hats
4  final_cosine = []
5  ranks = []
6
7  for yhat in tqdm(new_output_embeddings):
8      local_cosine = {}
9      sorted_cosine = {}
10     rank_list = []
11     sliced_dict = {}
12     yhat_norm = np.linalg.norm(yhat)
13     original_item_embedding_values = list(item_embedddings.values())
14     count = 0
15     for k,v in nomalized_item_embedddings.items():
16
17         # manually compute cosine similarity
18         dot = np.dot(yhat, original_item_embedding_values[count])
19         count +=1
20         result = dot / (yhat_norm * v)
21         local_cosine.update({k:result})
22
23     sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -
       item[1])}
24     sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50
       predictions for each item
25
26     rank_list = list(sliced_dict.keys())
27     with open('ranks_index.p', 'ab') as f:
28         pickle.dump(rank_list, f)
29     del original_item_embedding_values
```

One of the goals of this thesis is to develop an efficient (i.e. processing time, memory) solution. The preliminary results show that the processing time is faster if the vectors are normalized before doing the cosine calculation. Listing 4.14 shows the normalization calculations.

Listing 4.14: Implementation of items normalization

```
1  ## this cell stores the normalized item_embeddings
2  nomalized_item_embedddings = {}
3
4  for k,v in item_embedddings.items():
5      v_norm = np.linalg.norm(v)
6      nomalized_item_embedddings.update({k:v_norm})
```

For a particular y' (yhat in Listing 4.13) the program iterates through all the possible item embeddings to calculate the cosine similarity. Then the top "N" predictions (see line no. 24 in Listing 4.13) are stored in a list.

Improvements in the Code

We have made two improvements in the code to come up with the final version (Listing 4.13). The improvements are discussed in the following sections.

First: Listing 4.15 depicts the first version of cosine calculation. As the implementation

shows, all the item vectors are normalized repeatedly (see line no. 16 in Listing 4.15), which cost extra time. In the improved version, the item embeddings are normalized at once (Listing 4.14), which significantly reduces the runtime. The improvements are evaluated in Chapter 5.

Listing 4.15: First version of Cosine calculation

```
1  from tqdm import tqdm
2
3  ## this cell calculates the cosine similarity among y and y_hats
4  final_cosine = []
5  ranks = []
6
7  for yhat in tqdm(sample_output_embeddings):
8      local_cosine = {}
9      sorted_cosine = {}
10     rank_list = []
11     sliced_dict = {}
12     original_item_embedding_values = list(item_embedddings.values())
13     count = 0
14     for k,v in nomalized_item_embedddings.items():
15
16         result = 1 - spatial.distance.cosine(yhat, v)
17         local_cosine.update({k:result})
18
19     sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -
       item[1])}
20     sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50
       prediction for each item
21
22     rank_list = list(sliced_dict.keys())
23     with open('ranks_index.p', 'ab') as f:
24         pickle.dump(rank_list, f)
25     del original_item_embedding_values
```

Second: Listing 4.13 (line no. 24) shows that the *sliced_dict* stores only top 50 predictions for each iteration. The previous version (see line no. 17 in Listing 4.16) stored all the predictions. This procedure consumes extra memory. Chapter 5 evaluates and discusses the benefits of this approach.

Listing 4.16: Memory inefficient version of Cosine calculation

```
1  from tqdm import tqdm
2  from scipy import spatial
3  ## this cell calculates the cosine similarity among y and y_hats
4  final_cosine = []
5  ranks = []
6
7  for yhat in tqdm(sample_output_embeddings):
8      local_cosine = []
9      sorted_cosine = {}
10     rank_list = []
11     sliced_dict = {}
12     original_item_embedding_values = list(item_embedddings.values())
13     count = 0
14     for k,v in nomalized_item_embedddings.items():
15         result = 1 - spatial.distance.cosine(v, yhat)
16         local_cosine.append(result)
17     final_cosine.append(local_cosine)
18
19     with open('ranks_index.p', 'ab') as f:
20         pickle.dump(final_cosine, f)
21     del original_item_embedding_values
```

Step 8:

For the preparation of the evaluation, we have written the following implementation (Listing 4.17). This step checks for how many cases the expected item is present within the top "N" predictions. Therefore, we take a ratio between the successful prediction to the total number of cases. Chapter 5 discusses the results and related analysis.

Listing 4.17: Implementation to find accuracy of prediction

```
## This cell is to generate the matrix for final learning
# so if original label is present in top n prediction labels , increase the count by 1
y_arr = [None]* len(y)
y_hat_arr = [None]* len(y_hat)
y_arr = y
y_hat_arr = y_hat

i = 0
count = 0
while(i<len(y_arr)):
    if (y_arr[i] in y_hat_arr[i]):
        count +=1
    i +=1
Ratio = count/len(y_arr)
print('Learning Ratio is : ',Ratio)
```

4.3 Baseline Prediction Generator

Fig.4.6 depicts the component diagram of the baseline algorithm. The only difference between the baseline and the proposed approach is that the baseline algorithm calculates centroid for each transaction where the proposed model generates prediction from LSTM (Compare Fig.4.1 with Fig.4.6). As the data pre-processing steps are the same, the following section only discusses *Centroid Calculation*.

Listing 4.18: Implementation of centroid calculations for each transaction

```
## this cell calculates the centroid for each x's and store it in tuples along with x's
for i in num_x_test:
    centroid = np.mean(i, axis=0)
    new_tuple = (i,centroid)        # remember the order in the tuple for later use
    with open('all_centroids.p', 'ab') as f:
        pickle.dump(new_tuple, f)
```

As the implementation of Listing 4.18 shows, *num_x_test* is the test dataset which contains the input streams (vector representations of nine items). So, the centroids are calculated and stored alongside the vectors of these nine items in a tuple. Later these centroids are normalized and used to calculate cosine similarity against all possible items. Finally, the recommendations are made based on these cosine similarity scores. Then the top "N" most similar items are stored in a list to check how accurate the predictions are. These cosine similarity calculations are precisely identical to the proposed approach. Chapter 5 discusses the results and the related analysis of the baseline algorithm. Listing 4.19 depicts the implementations of the baseline algorithm. As the implementation shows, the top "N" predicted items are dumped into *baseline_ranks_index* for prediction accuracy calculation in later steps.

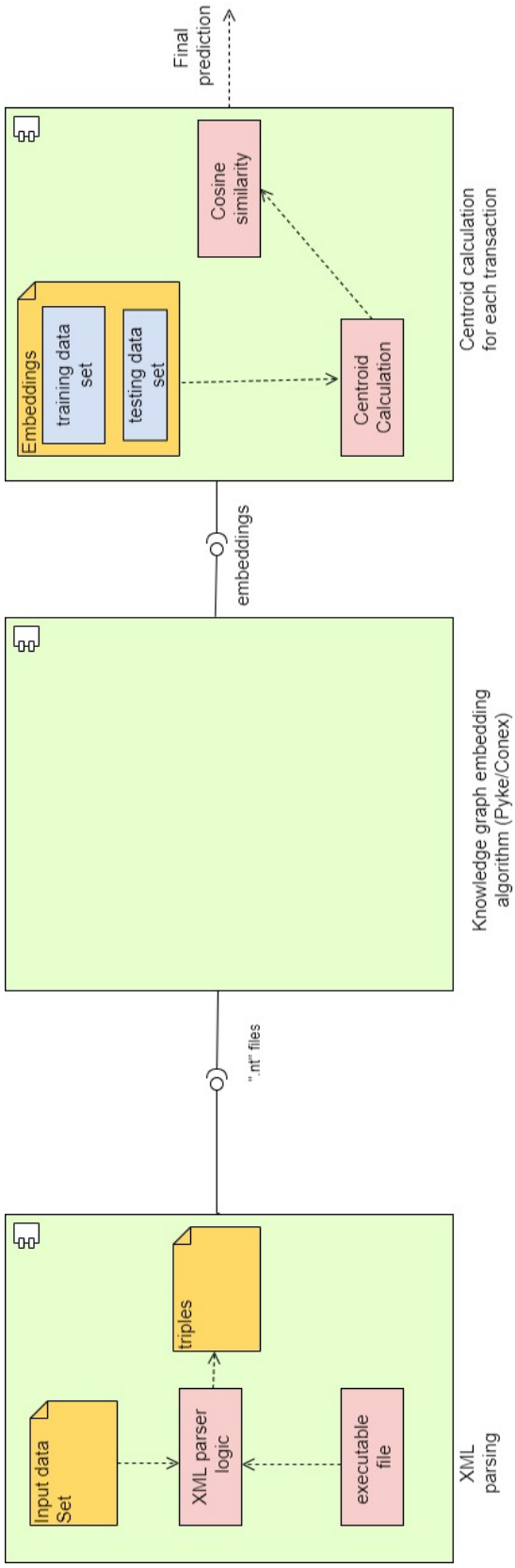

Figure 4.6: Component diagram of the BaseLine algorithm

Listing 4.19: Implementation of Cosine-similarity calculation for the BaseLine algorithm

```
## this cell calculates the cosine similarity among y and y_hats
final_cosine = []
ranks = []

for tuples in tqdm(centroid_list):
    local_cosine = {}
    sorted_cosine = {}
    rank_list = []
    sliced_dict = {}
    inputs,centroid = tuples
    centroid_norm = np.linalg.norm(centroid)
    original_item_embedding_values = list(item_embedddings.values())
    count = 0
    for k,v in nomalized_item_embedddings.items():

        # manually compute cosine similarity
        dot = np.dot(centroid, original_item_embedding_values[count])
        count +=1
        result = dot / (centroid_norm * v)
        local_cosine.update({k:result})

    sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -
    item[1])}
    sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50
    prediction for each item

    rank_list = list(sliced_dict.keys())
    with open('baseline_ranks_index.p', 'ab') as f:
        pickle.dump(rank_list, f)
    del original_item_embedding_values,inputs,centroid
```

5 Evaluation

This chapter shows and discusses the collected results from our experiments. It starts with the evaluation objectives. Then, it introduces the environment setup and the design of experiments. Later, this chapter shows the results from experiments. Finally, we analyse and interpret the results.

5.1 Evaluation Objectives

The primary objective of these experiments is to compare the prediction performance of the proposed model with the baseline. We have already discussed the difference between the models lie in prediction methodology where the proposed model uses LSTM architecture and baseline uses centroid calculation. Therefore, if the performance of the proposed model is comparatively less than that of the baseline, then it indicates that the LSTM architecture should be improved. If both models' predictions are low, then we need to investigate the vectors received from the knowledge graph embedding algorithm. Besides the primary objective, we have compared the item embeddings generated from Pyke and Conex as well as the performance of cosine similarity calculation.

5.2 Preparation

5.2.1 Data

As mentioned earlier, the proposed model is built with the transaction data from the retail domain. The details of the data are provided in section 3.2.1 and 4.2.3. As Fig.4.5 suggests most of the transactions comprise up to ten items. Therefore, the proposed model is developed and tested upon the transactions that contain up to ten items. The entire dataset was divided between train and test dataset in 80 : 20 ratio. We do not use any other dataset for the experiment.

5.2.2 Environment Setup

We have used two computers: a local system and a virtual machine. The hardware specifications of these systems are listed in Table 5.1.

Besides the mentioned hardware, we have used the following software during the experiment (Table 5.2).

Local PC	
CPU:	Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
RAM:	DDR3 7.6G
Virtual Box	
CPU:	Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
RAM:	DRAM 15G

Table 5.1: Hardware configaration

Local PC	
OS:	Ubuntu 18.04.5 LTS
Python:	Python 3.7.6
Python Interpreter:	Jupyter Notebook 6.0.3, Pycharm Community Edition 2020.1
Others:	Shell Script, XSLT
Virtual Box	
OS:	Ubuntu 18.04.5 LTS (Bionic Beaver)
Python:	Python 3.7.6
Python Interpreter:	Jupyter Notebook 6.0.3

Table 5.2: List of softwares

5.2.3 Experimental Design

We have performed four experiments in this thesis. The details of those experiments are given below.

1. **Pyke Analysis:** We want to note the processing time of Pyke for the number of RDF triples. Pyke takes close to linear runtime w.r.t the size of the knowledge graph. Therefore, this study can provide an estimate of how much time Pyke takes to process a large dataset.

2. **Prediction Performance Comparison:** As the name suggests, the experiment compares the prediction performance between the proposed approach and the baseline algorithm. We have used a standard metric, *recall@N* from information retrieval domain to perform this comparison. *recall@N* compares model performance with the help of the generated top "N" recommendation list.

   Recall@N

   "This measure describes how often the recommended item appears in the top-N places of the list of predicted items relative to all relevant items" [Fal18]. This metric is defined as [Dom08]:

$$recall@N = \frac{|\{\text{relevant items}\} \cap \{\text{predicted items on first N ranks}\}|}{|\{\text{relevant items }\}|} \tag{5.1}$$

   This metric is independent of order means it just checks if the actual items are within

the top "N" recommended items. That is why we choose different top "N" values (i.e. 1,3,20,50) in our experiment to check the accuracy of prediction.

3. **Vector Analysis:** This experiment analyses the item embeddings received from the knowledge graph embedding algorithm. It shows if the knowledge graph embedding algorithm can generate distinct vectors for unique items.

4. **Code Improvement Analysis:** As discussed in Chapter 4, we have improved the implementation, which calculates cosine similarity. This experiment compares the performance between the old code and the improved version.

5.3 Experiment Results

5.3.1 Pyke Analysis

Pyke comes with inbuild implementations which can track different measurements (i.e. no of triples, processing time etc.). Table 5.3 shows the related statistics collected from Pyke.

Number of RDF triples	22367362
Number of vocabulary terms	303239
Number of subjects	152063
Constructing Inverted Index took	1058.74 seconds
Calculation of PPMIs took	97.17 seconds
Pre-processing took	1157.01 seconds
Generating Embeddings: took	82035.81 seconds

Table 5.3: Statistics from Pyke

5.3.2 Prediction Performance Comparison

In this section, we discuss all the four scenarios that are considered for the experiment. Both approaches (i.e. proposed and baseline) are tested in each scenario to compare their performance. Table 5.4 depicts the results of the comparison in these scenarios. These results are calculated on the basics of metric *recall@N* discussed above. In the following sections, we show results from each scenario with the help of bar charts.

Metric	Proposed Model	BaseLine
recall@1	7.246481833070044e-06	7.246481833070044e-06
recall@3	1.4492963666140088e-05	4.347889099842027e-05
recall@20	0.0001304366729952608	0.00021739445499210133
recall@50	0.00042754242815113265	0.0005579791011463935

Table 5.4: Prediction performance comparison

recall@1

This experiment checks how accurately both approaches (proposed and baseline) can predict the next item. Fig.5.1 depicts the comparison of prediction performance between the methods. The Y-axis shows the prediction values where the X-axis represents the method names. As the diagram suggests the prediction quality for the *top* item is low from both methods.

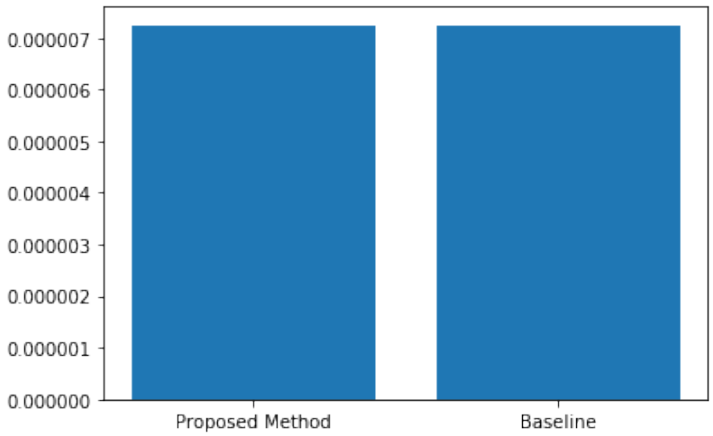

Figure 5.1: Proposed model vs BaseLine prediction performance(recall@1)

recall@3

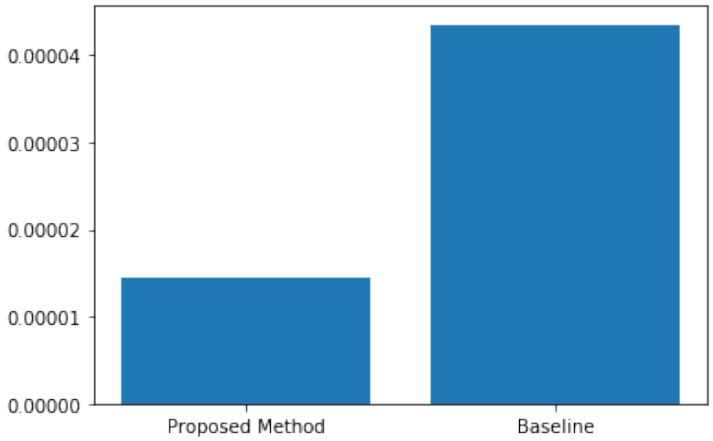

Figure 5.2: Proposed method vs BaseLine prediction performance (recall@3)

Fig.5.2 depicts methods prediction performance for reacll@3. It means the metric checks whether the actual item (y) lies within the *top 3* predicted items. Like the previous scenario, the method performance is shown by Y-axis, while the X-axis represents the method names. In the diagram baseline model outperformed our proposed method, but actually, both methods performances are low.

recall@20

In Fig.5.3 Y-axis represents the method performance while X-axis shows the method names. This diagram compares method performance for the *top 20* predicted items. Like the previous scenarios, both methods' performance is low.

recall@50

Fig.5.4 shows methods performance comparison for *top 50* predicted items. In the diagram, Y-axis shows the prediction score, while the X-axis shows the method names. Fig.5.4 depicts predictions are low for both methods.

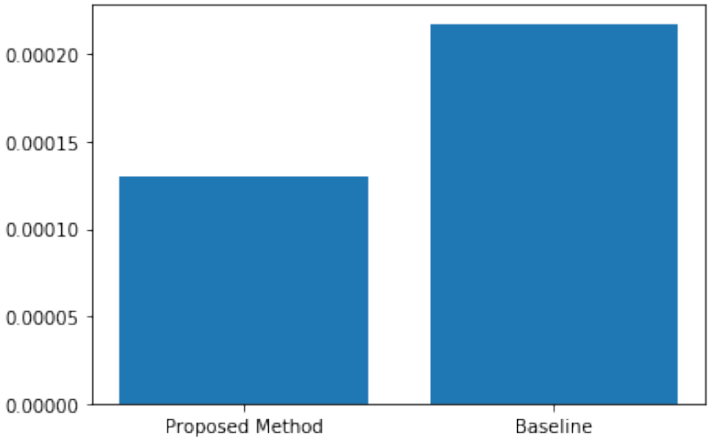

Figure 5.3: Proposed method vs BaseLine prediction performance(recall@20)

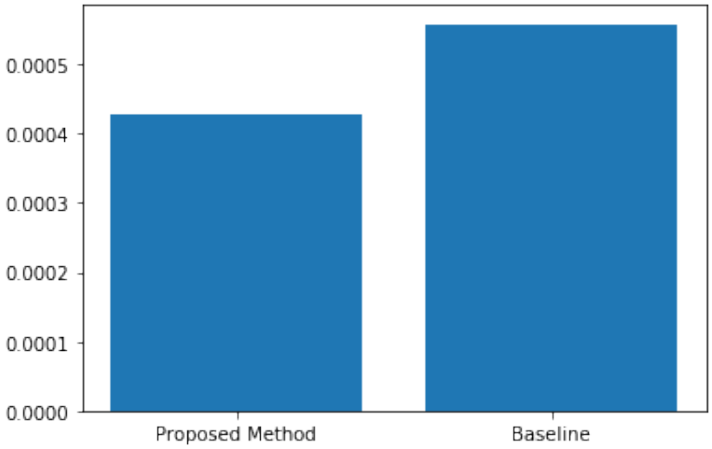

Figure 5.4: Proposed method vs BaseLine prediction performance (recall@50)

5.3.3 Vector Analysis

Prediction Vector Analysis

This experiment calculates cosine similarities for a sample of fifteen predicted items (y')s. The cosine similarity calculation was performed between the predicted vectors (y')s and the list of all possible item embeddings. Fig.5.5 depicts the mean of cosine similarity values of all the predicted vectors are identical.

Vector Comparion between Conex and Pyke

This experiment calculates cosine similarities for a sample of fifteen items (y). The cosine similarity calculation was performed between the sample vectors (y) and the list of all possible item embeddings. Fig.5.6 and Fig.5.7 depict the mean and standard deviation of cosine similarity values for Pyke and Conex algorithm, respectively.

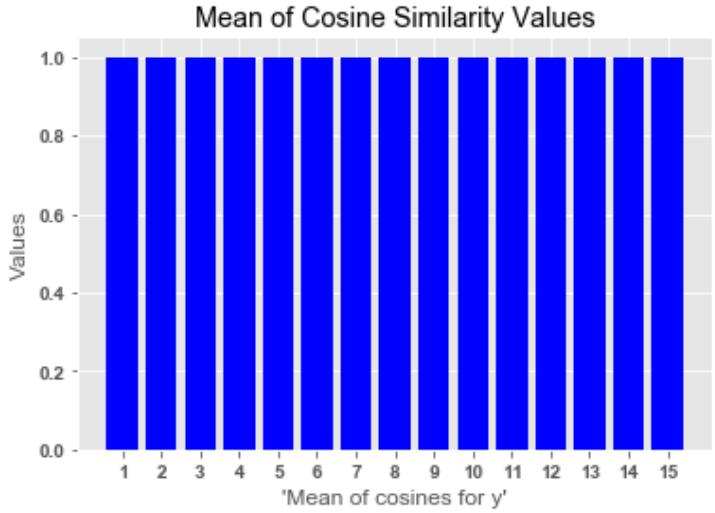

Figure 5.5: Vector analysis

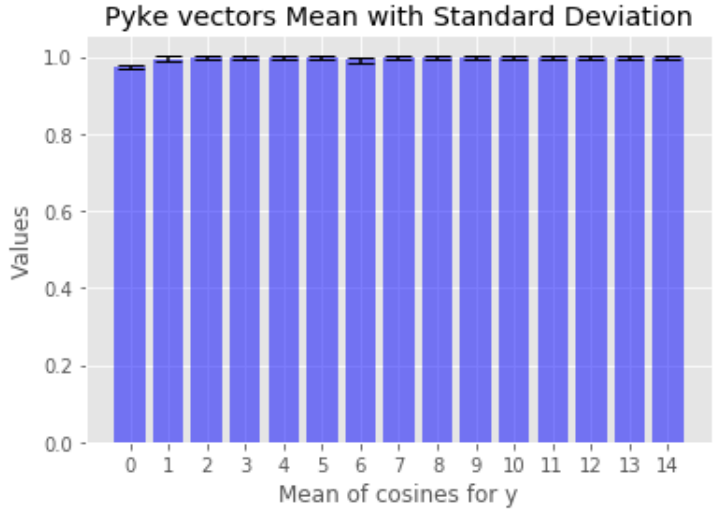

Figure 5.6: Pyke vector analysis

5.3.4 Code Improvement Analysis

Processing Time Improvement

We have made some changes in cosine similarity calculations for better performance. Fig.5.8 and Fig.5.9 depict the old code and the improved version respectively. This experiment compares the performance of the modified program with the older one. The performance comparison is shown in Fig.5.10.

Memory Efficiency

This experiment compares the memory usage between the old code snippet (Fig.5.11) and improved version (Fig.5.12) for cosine calculation. Fig.5.13 and Fig.5.14 shows the memory usage of old and new code snippet where Fig.5.15 and Fig.5.16 shows their respective percentage of memory usage. Fig.5.17 depicts the comparison of memory usage.

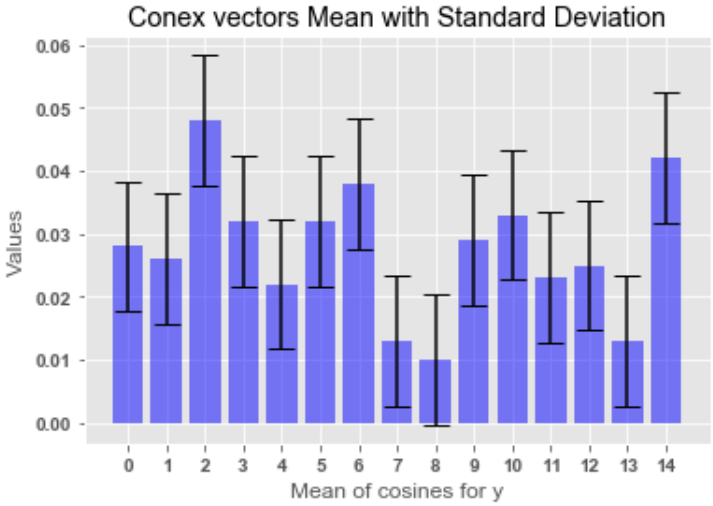

Figure 5.7: Conex vector analysis

```
from tqdm import tqdm
from scipy import spatial

## this cell calculates the cosine similarity among y and y_hats
final_cosine = []
ranks = []


for yhat in tqdm(sample_output_embeddings):
    local_cosine = {}
    sorted_cosine = {}
    rank_list = []
    sliced_dict = {}
    yhat_norm = np.linalg.norm(yhat)
    original_item_embedding_values = list(item_embedddings.values())
    count = 0
    for k,v in item_embedddings.items():
        [redacted]
        result = 1 - spatial.distance.cosine(yhat, v)
        local_cosine.update({k:result})

    sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -item[1])}
    sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50 prediction for each item

    rank_list = list(sliced_dict.keys())
    with open('ranks_index.p', 'ab') as f:
        pickle.dump(rank_list, f)
    del original_item_embedding_values
```

```
100%|██████████| 25000/25000 [08:07<00:00, 51.29it/s]
```

Figure 5.8: First version of Cosine calculation

5.4 Discussion

Many knowledge-graph-embedding algorithms [RSA20, WRL+19] need an excessive amount of runtime for processing a large dataset. Pyke shows that it can process a large dataset (e.g. with 152063 entities and 303239 relations) within 22 hours with the mentioned local PC configuration. The results from the experiment are supported by Pyke, which claims that it requires a close to linear runtime [DN19]. Therefore, we can state that Pyke requires a reasonable runtime to process a large dataset.

The prediction comparison experiment compares the model performance using *recall@N* metric with different "N" values. As shown in the figures (Fig.5.1,5.2,5.3,5.4), the prediction performance of both models' is low in all the cases. The data pre-processing steps (XML parsing, Pyke) are similar for these models. Therefore, we analyse the results received from the item embeddings to identify the reason behind this low prediction.

To analyse the vectors received from Pyke, we took a list of fifteen predictions (fifteen y' s)

```
from tqdm import tqdm

## this cell calculates the cosine similarity among y and y_hats
final_cosine = []
ranks = []


for yhat in tqdm(sample_output_embeddings):
    local_cosine = {}
    sorted_cosine = {}
    rank_list = []
    sliced_dict = {}
    yhat_norm = np.linalg.norm(yhat)
    original_item_embedding_values = list(item_embedddings.values())
    count = 0
    for k,v in nomalized_item_embedddings.items():

        # manually compute cosine similarity
        dot = np.dot(yhat, original_item_embedding_values[count])
        count +=1
        result = dot / (yhat_norm * v)
        local_cosine.update({k:result})

    sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -item[1])}
    sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50 prediction for each item

    rank_list = list(sliced_dict.keys())
    with open('ranks_index.p', 'ab') as f:
        pickle.dump(rank_list, f)
    del original_item_embedding_values
```

```
100%|██████████| 25000/25000 [01:23<00:00, 300.69it/s]
```

Figure 5.9: Improved Cosine calculation

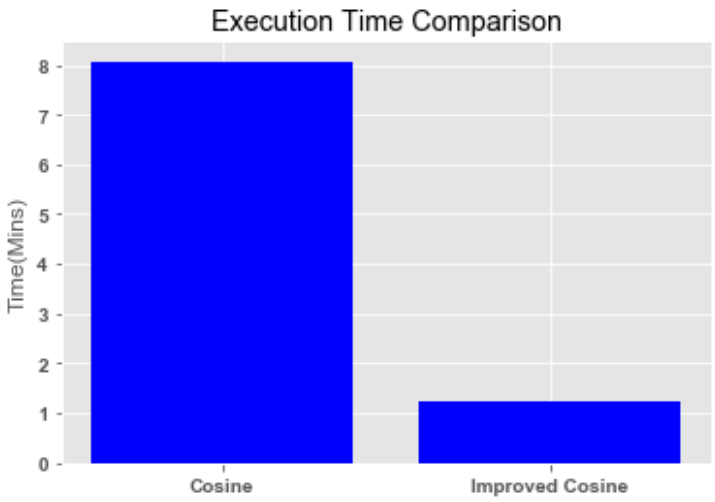

Figure 5.10: Processing time comparison

obtained from the LSTM. We calculate the cosine similarity between these predictions and the list of all possible items. Then arithmetic means of these cosine similarities are calculated to check how different these vectors are to one another. We also calculate the standard deviation of these arithmetic means to get an overview of the outliers. If the arithmetic means are high, then it means the vectors are very similar to each other. We calculate the mean of cosine similarities for each of the prediction (y') present in a list of fifteen predictions. So, we end up having a list of fifteen arithmetic means which are represented in forms of bars in Fig.5.5. The diagram depicts that for every set of cosine similarity calculation, the arithmetic means are identical (i.e. 1 in this case). The standard deviation is 0 among the arithmetic means. This is only possible when all the item vectors are similar.

Suppose an identical vector represents two different items A and B. Our proposed approach after the cosine calculation predicts B as the most similar item where the actual item is A. It is possible because the cosine similarity calculation produces 1 for every item. This is the reason behind the low *recall@N* values received from both methods.

```
from tqdm import tqdm
from scipy import spatial
## this cell calculates the cosine similarity among y and y_hats
final_cosine = []
ranks = []


for yhat in tqdm(sample_output_embeddings):
    local_cosine = []
    sorted_cosine = {}
    rank_list = []
    sliced_dict = {}

    original_item_embedding_values = list(item_embedddings.values())
    count = 0
    for k,v in nomalized_item_embedddings.items():
        result = 1 - spatial.distance.cosine(v, yhat)
        local_cosine.append(result)
    final_cosine.append(local_cosine)


    with open('ranks_index.p', 'ab') as f:
        pickle.dump(final_cosine, f)
    del original_item_embedding_values
```

Figure 5.11: Memory inefficient version of Cosine calculation

```
from tqdm import tqdm

## this cell calculates the cosine similarity among y and y_hats
final_cosine = []
ranks = []


for yhat in tqdm(sample_output_embeddings):
    local_cosine = {}
    sorted_cosine = {}
    rank_list = []
    sliced_dict = {}
    yhat_norm = np.linalg.norm(yhat)
    original_item_embedding_values = list(item_embedddings.values())
    count = 0
    for k,v in nomalized_item_embedddings.items():

        # manually compute cosine similarity
        dot = np.dot(yhat, original_item_embedding_values[count])
        count +=1
        result = dot / (yhat_norm * v)
        local_cosine.update({k:result})

    sorted_cosine = {k: v for k, v in sorted(local_cosine.items(), key=lambda item: -item[1])}
    sliced_dict = dict(itertools.islice(sorted_cosine.items(), 50))  ## take top 50 prediction for each item

    rank_list = list(sliced_dict.keys())
    with open('ranks_index.p', 'ab') as f:
        pickle.dump(rank_list, f)
    del original_item_embedding_values
```

Figure 5.12: Memory efficient version of Cosine calculation

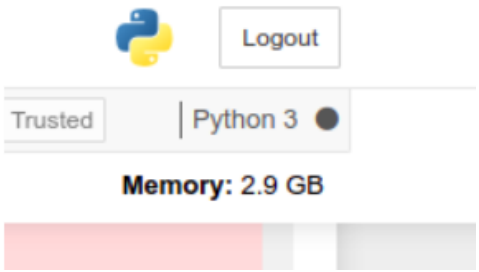

Figure 5.13: Memory usage for old version

Through the last experiment, we have identified that similar vector representations for different items are the reason behind the low final prediction. Therefore, the comparison of vectors generated through Pyke and Conex can show which algorithm can differentiate items better.

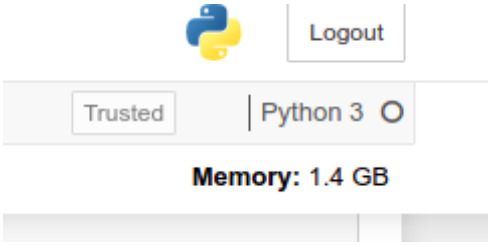

Figure 5.14: Memory usage for improved version

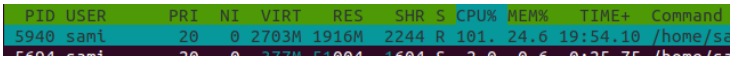

Figure 5.15: Percentage of memory usage for old version

Figure 5.16: Percentage of memory usage for improved version

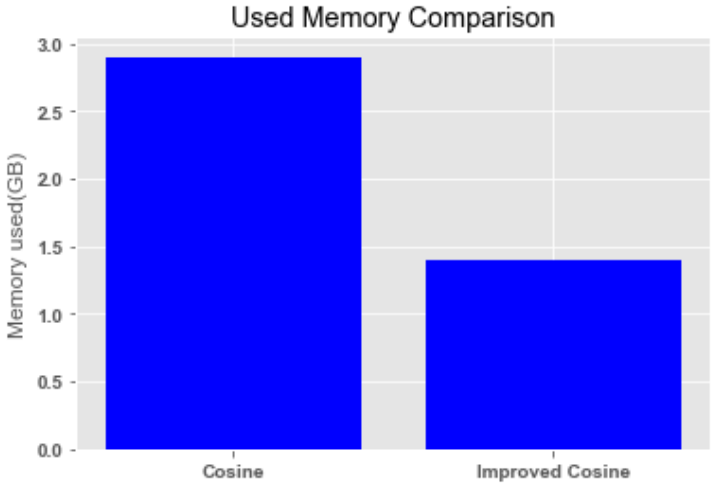

Figure 5.17: Memory usage comparison

Conex needs more extended time than Pyke to generate item embeddings. Because of limitations in time, we can not generate the embeddings of all possible items through Conex. Therefore, we randomly draw 100 transactions. We generate embeddings for the items of these transactions using both–Pyke and Conex. For Conex we divide 100 transactions among 80,10,10 for train, test and validation sets respectively. Like the previous experiment, we take an inventory of fifteen items and calculate the cosine similarity for them. 369 different items were present in those hundred transactions. Thus, cosine similarity is calculated between the list of chosen 15 fifteen items and the list of 369 different items. We compute the arithmetic mean for each set of cosine calculations. Therefore, we end up having 15 arithmetic means. Standard deviation is also calculated to get an idea about the outliers. For consistent comparison, the experiment is performed for the same set of fifteen items. Fig.5.6 supports our previous experiment result. This experiment calculates the cosine similarity between actual items. Therefore, the high mean (i.e. near to 1) of cosine values suggests the vectors are very similar to one another. Fig.5.6 shows almost all the means of cosine calculations are near to 1. Therefore, the standard deviation among the means is near to 0. It shows all the vectors are similar. Hence, this experiment shows

that it is difficult for the LSTM to distinguish between items because Pyke generates similar vectors. The results of the experiment are supported by Pyke, which states that one of its goals is to generate high purity vectors. It means "similar type vectors are located close to each other in the embedding space" [DN19]. Pyke generates similar vectors for different items because all of them come with the similar set of properties (i.e., all of a colour, size, etc.). That is why Pyke assumes that all of them have the same type. If we want to distinguish the items further, we need to have different features for different items. But LSTM predicts better if it uses the embeddings generated from Conex because the means of cosine similarity calculations provide variable values (Fig.5.7). In the diagram, the mean of most similar items is lower than .05, where the mean of least similar items is around .01. The standard deviation of this experiment is approximately .02. This low mean of cosine values, coupled with tiny standard deviations, prove Conex can generate different vectors for different items. Due to the lack of time, we cannot generate the final predictions from the vectors which are generated by Conex. Therefore, further research in this area is needed to mention the efficiency of Conex for the proposed approach.

One of the goals of this thesis is to produce a highly scalable and efficient (e.g. execution time, memory) solution. The dataset has around 150k different items. Therefore, the cosine calculation for all these items is time-consuming. Fig.5.8 shows the first version implementation of cosine calculation where Fig.5.9 shows the improved version of the implementation. In chapter 4, the difference between the implementations is already explained. Therefore, in this section, we will only focus on their respective evaluation time. To understand the benefit of processing time, we carried out the *Processing time comparison* experiment for a sample of 25000 different items. We calculate the cosine similarity for 25000 different items by the old code as well as the improved version and compare their processing time. Fig.5.10 depicts the comparison of processing time between two versions of the code. The bar-chart shows the improved version is nearly eight times faster than the older version.

To analyse the memory consumption we took a list of five hundred predictions (five hundred y') and calculated the cosine similarity by the old version of code snippet (Fig.5.11) as well as with the improved one (Fig.5.12). The implementation details were discussed in Chapter 4. Jupyter notebook comes with inbuilt feature which shows the memory used by the current program. This experiment uses Jupyter notebook as well as *htop* command on Ubuntu terminal to show the percentage of memory use. Fig.5.13 and Fig.5.14 depict the old version uses 2.9 GB of memory while the improved code needs only 1.4 GB of memory to execute the experiment. Therefore, Fig.5.17 shows that the enhanced code saves more than 50% of memory usage. We have also noted that the older version consumes 15 to 30% of memory where the newer code needs around 8.3% of memory. Fig.5.15 and Fig.5.16 show the percentage of memory usage by older version and improved version respectively.

In a nutshell, we developed a time and memory-efficient solution. The evaluation results from the experiments suggest that Pyke is a time-efficient knowledge graph embedding algorithm, but for our approach, Conex is more suitable.

6 Summary

This chapter recaps what has been achieved through our thesis. It discusses the significance and shortfall of this thesis. The chapter concludes with the prospects that can further extend the idea and relevance of our work.

6.1 Significance of Proposed Approach

The proposed approach uses the knowledge graph embedding algorithms for the recommendation task. The benefit of this approach is that it is highly scalable. Therefore, in future, if we want to train the model with an additional dataset where the dataset has similar invariant values as that of the original knowledge graph then, it requires no further implementations. The results from the experiments discussed in chapter 5 prove that our approach is efficient in terms of processing time and the memory consumption/usage.

After the Facebook- Cambridge Analytics scandal, the European Union (EU) introduced the General Data Protection Regulation, which empowers European citizens with their data. Since our approach does not use personal data of customers, it can be a solution to the current circumstances.

The proposed model is trained on extensive scale retail domain data (700k transactions with 150k items). Therefore, the results achieved from this model are highly significant.

6.1.1 Advantages

As mentioned earlier, the advantage of the proposed approach is that it is highly scalable. The method is built with Python language. Python supports many built-in libraries which make the development faster. The final solution is efficient (e.g. memory use, processing time). The approach is compliant with current EU data security standard as well as the large training dataset make the results more significant.

6.1.2 Limitation

Limitation in Dataset

The dataset possesses sixteen features for each item (Table 4.2). Out of these sixteen, one feature is department id which does not make any difference to current prediction settings because the proposed solution is not using any geographical location data. Some features are coupled with

each other like *ColorName-ColorCode*, *SizeName-SizeCode* and ***Class-**Category*. Therefore, the knowledge graph can be improved if we could extract more features per item. The enhanced knowledge graph can, in turn, boost the final predictions.

XML Parsing

The current version of XML parser can be improved. For example, the items have a feature named unit price. The working version parses this feature as a number and stores it as a literal. But item price range can be a primary determinant factor for a customer to choose a product. Therefore, the current version of parser has space for improvement.

Knowledge Graph Embedding Algorithm

As mentioned earlier, the proposed model's final prediction suffers because Pyke is unable to produce specific item embeddings for different items. Due to limitations in time, we could not generate all the item embeddings by Conex as well as the final predictions with these vectors. Therefore, in future, it will be interesting to see how accurate the predictions are with these Conex vectors.

LSTM Architecture

The processing time of the LSTM architecture can be improved. Currently, the proposed model uses a 50 unit *Vanilla* LSTM (single layer LSTM) where the final layer needs to do a separate cosine similarity calculation. This procedure is time-consuming since the cosine similarity calculation needs to wait until all the predictions (all the y' s) from the LSTM are collected. A separate code snippet needs to run manually on these collected data to do the cosine similarity calculation. The final prediction (starts from model training to final prediction) can process faster if it can be run on a single shot without human intervention in between.

6.2 Future Work

The limitation of a smaller number of features per item can be tackled by crawling more data from the webstore. For this idea to work, a properly maintained webstore data for some period is required. In this thesis, we are using transaction data from the past year. As retail stores change items frequently (e.g. new items come and old items disappear), this web crawled data can provide us with more UpToDate items. Therefore, the larger dataset with more items helps the proposed model to train better.

We can improve the existing knowledge graph so that embedding algorithms can use all the features. For example, knowledge graph embedding algorithms like Pyke can not make use of literals. Therefore, we can transform the literals into resources by using established techniques like binning. A good example is a price where we could introduce the categories, namely low, medium, and high. Suppose, these categories represent price ranges like 1-50$,51-200$,201-1000$, respectively. If a customer buys a t-shirt with the price tag of 15$, then XML parser should store "low" in place of digit 15.

An essential future task of this thesis is to generate all the item embeddings by Conex algorithm and check how accurate the final predictions are with these vectors. Besides this, we can improve the exiting version of LSTM architecture. For example, one can think of a more sophisticated form of LSTM (like stacked LSTM-multilayer LSTM). But this improvement is only required if the current version of LSTM doesn't perform well (e.g. less than 90% correct predictions) with the vectors generated by Conex algorithm.

6.3 Conclusion

This thesis has successfully proposed a recommender system with the help of a knowledge graph embedding algorithm. Unlike many other studies in this domain, which mainly focus on recommendation outcome, this thesis provides an initial blueprint to a highly scalable solution which follows different data protection measures. Therefore, regardless of limitations, this thesis has the potential to make the next generation recommender system in future.

Finally, this study is an initial effort to build a recommender system with the help of knowledge graph embeddings. This is a theme, which is still not being researched very precisely and has a lot of unrealized potent, that needs to be pointed out. In a nutshell, if in future some other researchers or scientists wanted to explore this theme more concretely, they would not have to start from scratch. This thesis could serve as a base for their future endeavours in this domain.

Bibliography

[ASAAA18] Belal Ayyoub, Ahmad Sharadqh, Ziad Alqadi, and Jamil Al-Azzeh. Simulink based rnn models to solve lpm. 5:49–55, 01 2018.

[BBL08] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. Team submission, W3C, January 2008.

[BBLP12] Y. Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, 12 2012.

[BG14] Dan Brickley and R.V. Guha. Rdf schema 1.1. W3C Recommendation, W3C, February 2014.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[BLC11] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable rdf syntax. Team submission, N3, March 2011.

[BLN09] James Bennett, Stan Lanning, and Netflix Netflix. The netflix prize. 01 2009.

[BOHG13] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109 – 132, 2013.

[Bot12] Léon Bottou. *Stochastic Gradient Descent Tricks*, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Bur02] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12, 11 2002.

[BYR17] Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLOS ONE*, 12(7):1–24, 07 2017.

[CHCH08] Long-Sheng Chen, Fei-Hao Hsu, Mu-Chen Chen, and Yuan-Chia Hsu. Developing recommender systems with the consideration of product profitability for sellers. *Information Sciences*, 178(4):1032 – 1048, 2008.

[CNHAVGGS12] Walter Carrer-Neto, María Luisa Hernández-Alcaraz, Rafael Valencia-García, and Francisco García-Sánchez. Social knowledge-based recommender system. application to the movies domain. *Expert Systems with Applications*, 39(12):10990 – 11000, 2012.

[Col20] Collaborative filtering. Collaborative filtering — Wikipedia, the free encyclopedia, 2007, [Online; accessed 03-September-2020]. http://en.wikipedia.org/w/index.php?title=Estimation_lemma&oldid=375747928.

[CS04] Jeremy J Carroll and Patrick Stickler. Rdf triples in xml. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 412–413, 2004.

[CVW11] Pablo Castells, Saúl Vargas, and Jun Wang. Novelty and diversity metrics for recommender systems: Choice, discovery and relevance. *Proceedings of International Workshop on Diversity in Document Retrieval (DDR)*, 01 2011.

[Dav20] David W. Richar, C. and L. Markus. Rdf 1.1 concepts and abstract syntax, 2014, [Online; accessed 04-September-2020]. http://www.w3.org/tr/rdf11-concepts/.

[DCJ19] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. Are we really making much progress? a worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 101–109, 2019.

[DN19] Caglar Demir and Axel-Cyrille Ngonga Ngomo. A physical embedding model for knowledge graphs. In *Joint International Semantic Technology Conference*, pages 192–209. Springer, 2019.

[DN20] Caglar Demir and Axel-Cyrille Ngonga Ngomo. Convolutional complex knowledge graph embeddings. *arXiv preprint arXiv:2008.03130*, 2020.

[Dom08] Sándor Dominich. *Vector Space Retrieval*, pages 157–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[DR19] Mahidhar Dwarampudi and N. S. Reddy. Effects of padding on lstms and cnns. *ArXiv*, abs/1903.07288, 2019.

[DYDA11] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2011.

[EJL+18] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 world wide web conference*, pages 1775–1784, 2018.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179 – 211, 1990.

[Fal18] Jonas Falkner. *Designing a Recommender System based on Generative Adversarial Networks*. PhD thesis, Institute of Information Systems, 2018.

[FCB16] Orhan Firat, Kyunghyun Cho, and Yoshua Bengio. Multi-way, multilingual neural machine translation with a shared attention mechanism. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 866–875, San Diego, California, June 2016. Association for Computational Linguistics.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GDBJ10] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. Beyond accuracy: evaluating recommender systems by coverage and serendipity. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 257–260, 2010.

[GNOT92] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[Gra12] Alex Graves. Sequence transduction with recurrent neural networks. *CoRR*, abs/1211.3711, 2012.

[HDY+12a] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N. Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[HDY+12b] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[Hea18] Jeff Heaton. Ian goodfellow, yoshua bengio, and aaron courville: Deep learning. *Genetic Programming and Evolvable Machines*, 19(1–2):305–307, June 2018.

[HK18] Balázs Hidasi and Alexandros Karatzoglou. Recurrent neural networks with top-k gains for session-based recommendations. *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, Oct 2018.

[HKBT16] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks, 2016.

[Hof04] Thomas Hofmann. Latent semantic models for collaborative filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1):89–115, 2004.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HZLL19] Xiao Huang, Jingyuan Zhang, Dingcheng Li, and Ping Li. Knowledge graph embedding based question answering. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 105–113, 2019.

[JKH07] Se Won Jang, Simon Kim, and J Ha. Graph-based recommendation systems: Comparison analysis between traditional clustering techniques and neural embedding, 2007.

[JMSM10] Mehrdad Jalali, Norwati Mustapha, Md Nasir Sulaiman, and Ali Mamat. Webpum: A web-based recommendation system to predict user future movements. *Expert Systems with Applications*, 37(9):6201–6212, 2010.

[JPC+20] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

[Kar15] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 21:23, 2015.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[KGB14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 655–665, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[Li09] Chengkai Li. XML parsing, SAX/DOM. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3598–3601. Springer US, 2009.

[LKH14] Blerina Lika, Kostas Kolomvatsos, and Stathes Hadjiefthymiades. Facing the cold start problem in recommender systems. *Expert Systems with Applications*, 41(4):2065–2073, 2014.

[LLJZ16] Yang Li, Ting Liu, Jing Jiang, and Liang Zhang. Hashtag recommendation with topical attention-based lstm. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3019–3029, 2016.

[LM14] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[LR+14] I Liu, Bhiksha Ramakrishnan, et al. Bach in 2014: Music composition with recurrent neural network. *arXiv*, pages arXiv–1412, 2014.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[MA09] A. Merve Acilar and Ahmet Arslan. A collaborative filtering method based on artificial immune network. *Expert Systems with Applications*, 36(4):8324 – 8332, 2009.

[Mar04] Benjamin M Marlin. Modeling user rating profiles for collaborative filtering. In *Advances in neural information processing systems*, pages 627–634, 2004.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[MHDB13] Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech*, pages 3771–3775, 2013.

[MKB10] T. Mikolov, M. Karafi'at, and L. Burget. Cernocky, J., and Khudanpur, S. *Recurrent neural network based language model*, 2010.

[Ngu18] Pham Thuy Sy Nguyen. Mimic real-world rdf graph data using multiple distributions. Master thesis, Universität Paderborn, November 2018.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[NMTG15] 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.

[PDS+16] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 24(4):694–707, April 2016.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[PWD16] Hamid Palangi, Rabab Ward, and Li Deng. Distributed compressive sensing: A deep learning approach. *IEEE Transactions on Signal Processing*, 64(17):4504–4518, Sep 2016.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[QKHC17] Massimo Quadrana, Alexandros Karatzoglou, Balázs Hidasi, and Paolo Cremonesi. Personalizing session-based recommendations with hierarchical recurrent neural networks. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, pages 130–137, 2017.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RIS+94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, 1994.

[Rob94] Anthony J. Robinson. An application of recurrent nets to phone probability estimation. *IEEE transactions on Neural Networks*, 5(2):298–305, 1994.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[RSA20] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Force2vec: Parallel force-directed graph embedding, 2020.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[SFR+06] K Shyong, Dan Frankowski, John Riedl, et al. Do you trust your recommendations? an exploration of security and privacy issues in recommender systems. In *International Conference on Emerging Trends in Information and Communication Security*, pages 14–29. Springer, 2006.

[SKR99] J Ben Schafer, Joseph Konstan, and John Riedl. Recommender systems in e-commerce. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, 1999.

[SLH14] Yue Shi, Martha Larson, and Alan Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798, 2007.

[SR14] Guus Schreiber and Yves Raimond. N-triples. W3C Recommendation, W3C, June 2014.

[VSC16] Flavian Vasile, Elena Smirnova, and Alexis Conneau. Meta-prod2vec: Product embeddings using side-information for recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 225–232, 2016.

[WRL+19] Cunxiang Wang, Feiliang Ren, Zhichao Lin, Chenxu Zhao, Tian Xie, and Yue Zhang. Domain representation for knowledge graph embedding. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 197–210. Springer, 2019.

[XY09] Liang Xiang and Qing Yang. Time-dependent models in collaborative filtering based recommender system. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 450–457. IEEE, 2009.

[YZC+18] Libin Yang, Yu Zheng, Xiaoyan Cai, Hang Dai, Dejun Mu, Lantian Guo, and Tao Dai. A lstm based model for personalized context-aware citation recommendation. *IEEE access*, 6:59618–59627, 2018.