



# Recommending recipes along with healthier alternatives using multi-headed attention and pre-trained SBERT embeddings

SPHC7<sup>1</sup>

MSc Data Science & Machine Learning

Internal Supervisor: Brooks Paige, UCL

External Supervisor: Francesca Giordano, ZOE

Submission date: 20 09 2022

<sup>1</sup>**Disclaimer:** This report is submitted as part requirement for the MSc degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged

# Code listing

The code used to produce the report is available at this [hyper-link](#). The Readme file in the repository contains links to the data, pre-trained models and results.

## **Abstract**

In this work, we tackle the recipe recommendation problem with an added health component. Currently, numerous websites provide a database of recipes to users worldwide. To keep the users interested, they recommend various recipes from their website for people to try out. The recommendations are often based on their past interactions with other recipes. To facilitate this, we build a novel model using multi-headed attention to recommend users relevant recipes. Our model shows an improvement over several baselines used for other recommendation tasks. Secondly, we take on the challenge of adding a constraint on the healthiness of the recipe that we are recommending. While suggesting users new recipes, we have an opportunity to make an impact on their health. However, it is difficult to change people’s eating habits. Our approach was to recommend them healthier versions of recipes that they have consumed before. To this end, we designed a novel method for recommending healthy recipes using recipe embeddings from a pre-trained Sentence BERT model. To test the utility of the method, we conducted a user study that shows that our approach could allow for recommending healthy recipes that users will find relevant.

# Contents

<b>List of Figures</b>	<b>2</b>
<b>1 Introduction</b>	<b>4</b>
1.1 Motivation . . . . .	4
1.2 Research Goals . . . . .	5
1.3 Thesis outline . . . . .	5
<b>2 Background &amp; Related Work</b>	<b>6</b>
2.1 Recommender Systems . . . . .	6
2.1.1 Problem setting . . . . .	6
2.1.2 Goals of a recommender system . . . . .	7
2.1.3 Types of Recommender systems . . . . .	8
2.1.4 Neural Networks for recommendation . . . . .	9
2.2 Neural Network Introduction . . . . .	10
2.2.1 Bias-Variance Trade off . . . . .	11
2.2.2 Gradient Descent . . . . .	11
2.2.3 Linear Layers . . . . .	13
2.2.4 Rectified Linear Units (ReLU) . . . . .	13
2.2.5 Dropout . . . . .	13
2.2.6 Multi-Layer-Perceptron . . . . .	13
2.2.7 Residual Connections . . . . .	14
2.2.8 Binary Cross Entropy . . . . .	14
2.3 Embeddings . . . . .	15
2.4 Attention . . . . .	15
2.5 Multi-headed Attention . . . . .	17
2.5.1 Scaled Dot-Product Attention . . . . .	17
2.5.2 Multi-headed attention module . . . . .	18
2.5.3 Self-Attention . . . . .	19
2.5.4 Transformer Encoder . . . . .	19
2.6 BERT . . . . .	20
2.7 Sentence-BERT . . . . .	21
2.8 Recipe Recommendation . . . . .	22
2.8.1 CF methods . . . . .	22

2.8.2	Content-based methods . . . . .	22
2.8.3	Hybrid-based methods . . . . .	23
2.8.4	Neural Network based methods . . . . .	23
2.8.5	Health-based methods . . . . .	23
<b>3</b>	<b>Methods</b>	<b>27</b>
3.1	Data . . . . .	27
3.1.1	Recipe Information . . . . .	27
3.1.2	Interaction Information . . . . .	27
3.1.3	Data-Exploration & Filtering . . . . .	28
3.2	Data-Split . . . . .	32
3.3	Rating Model . . . . .	32
3.3.1	Matrix Factorization (MF) . . . . .	32
3.3.2	Matrix Factorization with biases . . . . .	34
3.3.3	Content-boosted Matrix factorization . . . . .	34
3.3.4	Neural Collaborative filtering (NCF) . . . . .	35
3.3.5	Two-tower Model . . . . .	36
3.3.6	Recommendation with AutoInt . . . . .	37
3.3.7	Our model . . . . .	39
3.4	Health Model . . . . .	41
3.5	Combining Rating & Health . . . . .	44
3.5.1	New Approach . . . . .	44
3.5.2	Recipe Embedding . . . . .	44
3.5.3	Recipe Similarity . . . . .	45
3.5.4	Embedding Visualization . . . . .	45
3.5.5	Similar recipes . . . . .	46
3.6	Combining Similarity, Rating & Health . . . . .	47
<b>4</b>	<b>Results &amp; Analysis</b>	<b>50</b>
4.1	Rating model . . . . .	50
4.1.1	Evaluation metric . . . . .	50
4.1.2	Experiments . . . . .	51
4.1.3	Analysis of our model . . . . .	53
4.1.4	Ablation . . . . .	54
4.1.5	Failure Analysis of our model . . . . .	55
4.2	Recommending Healthy Recipes . . . . .	57
4.2.1	Quantitative Results . . . . .	57
4.2.2	User Study & Qualitative Evaluation . . . . .	58
4.3	Limitations . . . . .	59
<b>5</b>	<b>Conclusion &amp; Future Work</b>	<b>60</b>
5.1	Summary . . . . .	60
5.2	Future Work . . . . .	60



# List of Figures

2.1	User-Item Matrix $S$ . The entries in the matrix indicate user-item interactions. . .	6
2.2	Example of top-K recommendations on Amazon. The item starting from the left is ranked first. Image from [9]. . . . .	7
2.3	Artificial Neural Network with 1 hidden layer (in blue). Each circle represents a node and the arrows represent connections between nodes. Image from [29]. . . . .	10
2.4	MLP as a chain of functions. . . . .	13
2.5	Residual connection between layers. Image from [39]. . . . .	14
2.6	Encoder-Decoder RNN with attention for English to French translation. Each cell in the encoder is represented in blue, while the decoders are represented in red. The attention mechanism is demonstrated for the first decoder state. Image from [43]. .	16
2.7	The figure illustrates scaled-dot product attention. This layer performs the operation $Attention(Q, K, V)$ from equation 2.15. Image from [5]. . . . .	17
2.8	Multi-headed attention module with $h$ heads represented by overlaid tiles. The output of this layer is given by $MultiHead(Q, K, V)$ described in equation 2.18. Image from [5]. . . . .	18
2.9	The Transformer encoder consists of a multi-headed attention, layer normalization & an MLP (feed-forward). The inputs to this model were words in a sentence. Therefore we notice a positional encoding in the image. Image from [5]. . . . .	20
2.10	This figure illustrates sentence BERT. The inputs are “Sentence A” & “Sentence B” to individual BERT models. The model is trained on similarities between the sentences with the goal of obtaining a fixed representation for each sentence. Image from [6]. . . . .	21
2.11	Traffic lights according to the FSA. The range for each macro-nutrient under green, amber & red traffic lights is shown in this figure. Image from [65]. . . . .	24
2.12	Traffic lights on the back of a typical product in the supermarket. Image from [64].	25
3.1	Distribution of calories amongst all recipes. <b>Left:</b> Unfiltered <b>Right:</b> Filtered under 2000 kcal . . . . .	28
3.2	Distribution of time taken amongst all recipes. <b>Left:</b> Unfiltered <b>Right:</b> Filtered under 300 minutes . . . . .	29
3.3	Ingredients word cloud. More frequent ingredients amongst the recipes are represented by a larger font in the figure. . . . .	29
3.4	Distribution of ratings for all the interactions. . . . .	30

3.5	Number of unique ratings given by all users. For instance, “1” on the x-axis indicates a user has rated all recipes with only one type of rating. . . . .	31
3.6	This figure illustrates the distribution of the number of recipes rated per user. . . .	31
3.7	The user-item matrix $S$ decomposed into matrices $U$ and $I^T$ . At inference, $S$ is reconstructed using an inner product and missing values are inferred. Image from [72]. . . . .	32
3.8	<b>a)</b> The user-item matrix $S$ <b>b)</b> The latent space that matrix factorization projected each user to. Each arrow represents a user, and the blue arrow indicates the two possible assignment of a new user $p_4$ . Image from [75]. . . . .	35
3.9	NCF architecture as illustrated by [75]. . . . .	35
3.10	Two-tower model. Each MLP is a separate “tower” that is used to create representations for the user and the item. . . . .	36
3.11	AutoInt model architecture from [79] used to predict click-through-rate using multi-headed attention. The input features $x$ contains features from both items & users. . . . .	37
3.12	Examples of features from a movie recommendation problem along with their associated attention weights in a self-attention layer. Image from [79]. . . . .	39
3.13	Model architecture. We use the two-tower approach and model the user & items separately. Firstly, ingredient sequences go through a self attention layer. Then, a multi-headed layer is used to generate the recipe vector from a weighted combination of ingredient embeddings. . . . .	40
3.14	Health Score vs Value of nutrient (grams) . . . . .	43
3.15	Log Frequency vs Sodium(grams) . . . . .	43
3.16	Illustration of recipe embedding for “Italian Fries”. The output is a 768 dimensional contextual embedding that captures various aspects of the recipe. . . . .	44
3.17	UMAP Projection of recipe embeddings in 3D. Each point in the figure represents a recipe. The colours indicate the rounded health score. . . . .	45
3.18	A cluster of recipes from the 3D space illustrated in the bounded-box. We see that similar recipes cluster together from the annotations. . . . .	46
3.19	Combining Rating, Health & Similarity. From right to left: The process for generating recommendations for user #1 is detailed in the figure. . . . .	48
4.1	Confusion matrix . . . . .	51
4.2	ROC curve with the shaded region being the AUC. Image from [87]. . . . .	51
4.3	In this figure we have a heatmap of attention weights with the users on the y-axis & the ingredients of a recipe from the test set on the x-axis. From the highlighted recipe, we notice that the attention weights are providing importance to single ingredient. . . . .	53
4.4	Two recipes having similar attention weights for different users. . . . .	54
4.5	No of recipes where the model failed to predict interactions vs the frequency of the recipe in the train set. . . . .	56
4.6	(Health + Rating) score function visualization. . . . .	57



# Chapter 1

## Introduction

### 1.1 Motivation

With increasing content and traffic on websites, companies want to enrich user experience by recommending relevant content to them. As the number of users and data generated scales, automatically recommending content becomes possible through various machine learning techniques. This has prompted research on using machine learning for recommender systems. By definition, recommender systems can be thought of as the task of predicting user preferences for certain items over others in a larger list of items. Items here could mean a multitude of things. For instance, the video recommendations that we receive on Youtube[1], playlist/music recommendations on Spotify[2], and item recommendations on Amazon have some of these algorithms working behind the scenes.

The area that this thesis focuses on is recipe recommendation. But what are the practical reasons to choose such a problem? Today, we have several apps where we can log our food, count calories, and keep a record of what we ate in the past. Additionally, some websites like food.com and allrecipes.com have a large database of recipes and users. In both of these cases, user-recipe interactions are commonly stored by the app. Interactions could be clicks, ratings, or even reviews. Ideally, as with all recommender systems, we want to recommend the users items or, in this case, meals that they would like to keep them using the app or the website. However, there are several challenges to recipe recommendation. A variety of factors come in when someone chooses a recipe on one of these apps/websites. How does the recipe image look? Is it difficult to cook? How long does it take to cook? Are they familiar with the recipe? Have they tried something similar before? Are there some ingredients in the recipe that they are put off by? Some people might be health conscious and look at calories/nutrient information. Some would only care about the taste.

Another area of focus in thesis is making sure that the recommended recipes are healthy. Obesity has become a significant concern in today's society [3]. According to the world health organization (WHO) obesity in the world has tripled since 1975 [4]. In 2016 WHO found over 650 million people in the world are obese. Obesity leads to shorter life expectancy, increased risk of heart disease, and various other health risks [3]. The fact is that processed food and fast

foods are typical in today’s day and age since they are easily accessible at a relatively low price. This has led to a shift in food habits for the worse. However, the platforms (websites/apps), through their recommendations, have the opportunity to suggest their users healthier recipes or even healthier alternatives to their favourite dishes. Thus our main problem is formulated as follows: **Recommend healthy recipes to users that they would like.**

## 1.2 Research Goals

The research goal of the thesis are follows: To design an appropriate machine learning model for the task of recipe recommendation, and recommend healthier recipes combining the healthiness of a recipe along with user preferences. In more details, the hypotheses can be summarised as follows:

1. Using ingredient information is beneficial to the task of recipe recommendation. For this, we show ablations that prove that adding recipe information as a whole (instead of using ingredients to form a representation of the recipe) deteriorates model performance.
2. Multi-headed attention [5] can be used to derive weights for each ingredient in a recipe in a way that is beneficial to the task of recipe recommendation. To this end, we design a novel model that achieves higher metrics than several baselines. Additionally, we visualize the attention weights to interpret the results.
3. A pre-trained sentence BERT model [6] can be used to derive similar recipes to ones that users already consumed, in turn, to find healthier versions of recipes that users like. To achieve this, we show that a pre-trained sentence BERT model can be used to construct meaningful recipe embeddings that can be used to recommend similar recipes.

## 1.3 Thesis outline

The rest of the thesis is organized as follows:

1. Chapter 2 focuses on relevant background information to understand the contents of the thesis. We review recommender systems (section 2.1) & neural networks (sections 2.2). Then, we review the attention mechanism (section 2.4) going all the way up to transformers (section 2.5) & BERT (sections 2.6, 2.7) The chapter also contains a literature review of previous methods used for recipe recommendation.
2. Chapter 3 contains information about the various methods used in the thesis. We start with the data used (section 3.1), then the various types of models used for recommendation (section 3.3). Finally, we define the methods for recommending healthy recipes (sections 3.4, 3.5, 3.6)
3. Chapter 4 deals with the results of the various models and the analysis conducted, whereas Chapter 5 presents the conclusions and future work.

## Chapter 2

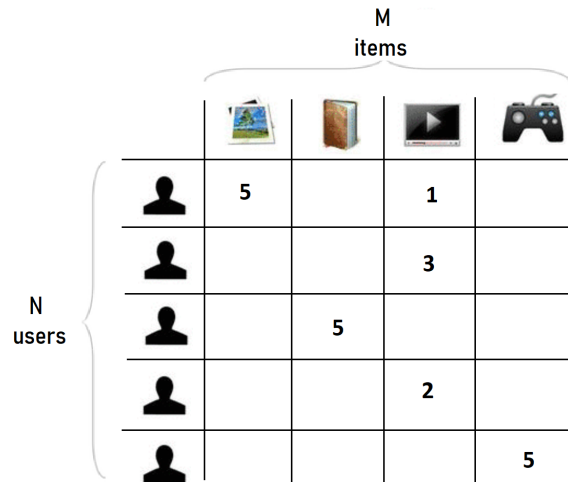
# Background & Related Work

### 2.1 Recommender Systems

#### 2.1.1 Problem setting

In a typical recommender system problem we have a set of  $N$  users,  $U = \{U_1, U_2 \dots U_N\}$  and  $M$  items  $I = \{I_1, I_2 \dots I_M\}$ . Additionally, for each user in  $U$  we have a subset of items  $I$  that they have interacted with. These interactions denoted as  $R$  can be explicit and take values in  $r \in \mathbb{R}$ . This could be in the form of categorical ratings ( $r \in \{1, 2, 3, 4, 5\}$ ), binary ( $r \in \{1, 0\}$ ) or even implicit ones such as clicks where the positiveness of the interaction is inferred [7]. The problem can then be arranged into two types [8]:

1. **Prediction of ratings:** Ideally, we would have a function  $F$  defined as  $f : UX I \rightarrow R$  that estimates the interaction  $R$  for every user-item pair. This can be viewed as the completion of sparse user-item matrix  $S$  as shown in Figure 2.1. Usually, most of the entries in the matrix are empty, and the goal of this problem formulation is to complete the missing values.



The diagram illustrates a User-Item Matrix  $S$ . It features a grid with 5 rows and 4 columns. The columns are labeled 'M items' at the top, with icons representing different items: a book, a box, a play button, and a game controller. The rows are labeled 'N users' on the left, with person icons representing different users. The matrix contains numerical ratings in specific cells: the first row has a '5' in the first column and a '1' in the third column; the second row has a '3' in the third column; the third row has a '5' in the second column; the fourth row has a '2' in the third column; and the fifth row has a '5' in the fourth column. All other cells are empty.

	M items			
N users	5		1	
			3	
		5		
			2	
				5

Figure 2.1: User-Item Matrix  $S$ . The entries in the matrix indicate user-item interactions.

2. **Ranking list of items** Instead of predicting the ratings, for each user, we rank the top-k items that can be recommended to them. The ranking problem is different from the prediction problem in the fact that we do not consider a user-item pair to predict one interaction  $r$ . Instead, we try to predict rank of items from most relevant to least relevant to the user. These are items that the users might find interesting to try out next. For instance, we see in figure 2.2 below some items on Amazon suggested to a particular user in the form of a list.

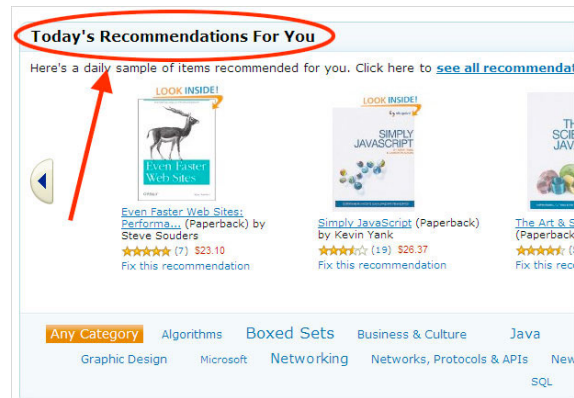


Figure 2.2: Example of top-K recommendations on Amazon. The item starting from the left is ranked first. Image from [9].

### 2.1.2 Goals of a recommender system

There can be a broad variety of goals for a recommender system depending on the use case. For websites & companies the main motive might be business goals that include increasing user retention, getting more clicks & increasing profits. However, to achieve those targets we need to attain some technical goals that would directly lead to an impact on the business. Some of these technical goals are listed below:

1. **Relevance & Accuracy:** We want our recommender system to be as accurate as possible with respect to unseen data. This means we want the model to have higher quantifiable metrics on unseen data. However, we should also make sure that the new items recommended to the users are relevant to them. Accuracy should not be confounded with relevance [10], and higher accuracy doesn't always equate to higher relevance. This is why user studies are usually conducted when deploying recommender systems in production - we want to find if the items recommended are something the users would click on or buy.
2. **Novelty:** Novelty of recommendations [11] ensures that the items recommended to the users are somewhat new to them. This will keep the user interested to use the website/service. This could also include the diversity of items recommended. But again, we also have to make sure that the items are relevant. For instance, recommending random items could be considered novel to the users, but not necessarily relevant.

3. **Scalability:** The model must be scalable with respect to inference time to provide low latency recommendations to customers if required. Additionally, the training time of the model must be quick enough so that turnaround times are higher if re-training of the model is required.

### 2.1.3 Types of Recommender systems

Recommender systems can fall into a variety of categories, depending on the features available in the data and the end goal of the system itself. Some of these types with their advantages and limitations are discussed below.

1. **Collaborative filtering(CF):** Collaborative filtering assumes that there are similar users to you who might interact with the items as you will. So, based on the items that you have liked/disliked we find other users with similar tastes. If they have liked some other items perhaps you would like them too. In this case, we do not care about the information about the item, only the interactions between the users and the items. There are two sub-varieties of CF models according to [12] that are as follows:
  - (a) **Memory-based methods** Memory-based methods like [13] is based on the similarity of any two users directly inferred from the user-item rating matrix. From Figure 2.1, if we consider a row in the matrix, we have single user  $u$  with a vector of length  $M$  containing the ratings for all  $M$  items. A memory-based method would involve comparing all of the other user vectors to  $u$  using a similarity measure. Then, we recommend  $u$  items based on the interactions of the most similar user to  $u$ .
  - (b) **Model-based methods** Model-based methods are those where a model is parameterized and trained to predict the ratings or rank the items in an ordered list. Previous work [14] has shown advantages of model-based methods over memory-based methods that include - better performance & handling the issue of sparsity. Thus, in this thesis we decided to use model-based CF methods instead of memory-based methods.

#### Limitations:

- (a) **Cold-start Problem:** The cold start problem [15] occurs when we do not sufficient interactions for new users or new items. CF suffers from both **new-item** & **new-user** recommendations since we depend on similarity between users. For a new user, we simply won't have enough ratings to calculate the similarity to other users. Similarly, for a new item, not enough people would have rated them, and they won't likely be recommended.
2. **Content-based filtering(CB):** In content-based filtering, we take into account the descriptions of the items while making our recommendation. If a user has liked an item in the past, they might like other items that are similar to that particular item.

#### Limitations:

- (a) **Homogeneity:** Purely content-based filtering suffers from the lack of novelty in the items recommended to the user. This is because we are only taking into account the

preference of one user and recommending similar items to the set that they have already interacted with. Therefore recommendations may be not diverse enough to keep the user interested.

- (b) **Cold-start Problem:** Since we are taking into account the item features, content-based filtering does not suffer in **new-item** recommendations, since we have feature information even for newer items. However, it does fail with **new-users** where we do not have enough items that a single user has rated.
- 3. **Knowledge-based filtering:** Knowledge-based recommender systems [16] allow for domain knowledge about the problem to be incorporated into the recommendations. For instance, constraint-based filtering accounts [17] for a filter on the items that a user can specify about the items that they want.
- 4. **Hybrid-based recommender systems:** Hybrid recommender systems [18] [19] often combine two or more of the methods described above. This is especially helpful when a variety of inputs are available, and various aspects of each method can be combined to overcome the disadvantages of a single method. According to [18] hybrid recommendations can be of several types. Some examples are as follows:
  - (a) **Weighted:** Here, two separate models of various types are trained and combined. The final prediction could be a voted classifier or even a linear combination [20] [21].
  - (b) **Monolithic/Feature Combination:** In this case, we have content information incorporated into one unified model along with a collaborative model [22]. This usually helps in avoiding the cold-start problem for new items since incorporating content information has that advantage over purely CF models.
  - (c) **Mixed:** For mixed hybrids, typically recommendations from several individual models are shown to the users.
  - (d) **Meta-level** Usually, an output of a particular model will be used as the input of another one to produce the final result.

From the discussion above, we frame our methodology in later chapters considering the limitations and advantages of each method.

#### 2.1.4 Neural Networks for recommendation

Neural Networks have formed the basis for the state of the art for several machine learning tasks to do with images [23], audio [24], video [25] and language [26]. Similarly, most research in recommender systems has been focused around leveraging neural network models [27]. Neural networks can be designed to exploit the inherent structure in the data. Thus, it is especially useful with content-based systems. Since we often have multi-modal data - text (reviews, descriptions), images (recipe images, social media posts), neural network models allows us to learn representations for them end-to-end without the need to design specific features. Even for the CF case, when there are only user-item interactions as features, neural networks have shown improved performance than traditional methods such as matrix

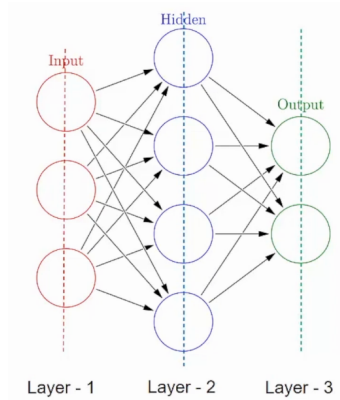


Figure 2.3: Artificial Neural Network with 1 hidden layer (in blue). Each circle represents a node and the arrows represent connections between nodes. Image from [29].

factorization [28]. Usually, with methods like matrix factorization (described in section 3.3.1), there is an assumption of linearity between the user-item interactions. Instead, as we will see in the following sections, neural networks can model non-linear relationships by using non-linear activations. Neural networks have been known to be universal function approximators [21], and it can thus model very complex user-item interactions. However, they also have limitations such as interpretability and the requirement for large amounts of data. The following subsections includes a discussion of various components of neural networks.

## 2.2 Neural Network Introduction

In traditional machine learning methods, clever feature engineering & domain knowledge were required to build a highly accurate model. However, with the advent of neural networks, the features were automatically learned from the dataset. Originally inspired by the human brain Artificial Neural Networks (ANNs) contain a set of inter-connected nodes or “neurons”.

In figure 2.3 above, we see a typical example of an ANN. The structure consists of three layers - an input, a hidden layer and an output. The nodes are interconnected through a set of weights that determines the strength of the connection. Additionally, at the end of the hidden layer, a non-linearity is added to model complex relationships between the input and the output. Thus, the main advantage of neural networks comes when the relationship between the inputs and outputs is not truly linear. A loss function is defined to optimize the weights (or parameters), which determines how well the network performs for the task at hand. However, there is no closed form to optimizing a neural network, and iterative methods must be used. This is described in detail in the subsequent subsections.

Neural networks aim to approximate a function  $f^*$  that estimates the actual relationship between the input & the output. So we usually have,  $y = f^*(x; \theta)$  that maps an input  $x$  to a output  $y$ . Here,  $\theta$  represents the neural network parameters that must be optimized to produce the best function approximation for the mapping.

### 2.2.1 Bias-Variance Trade off

In the thesis, we compare models with a varying degree of assumptions between input and output, along with various number of parameters  $\theta$  (and hence representation capability). We describe the bias-variance trade-off to understand why one model would work over the other due to changes in assumptions.

Ideally, we want our machine learning model to perform well on unseen data. Therefore, we usually split our dataset into three parts: training, validation & a test set. First, the training set is used to train the model & optimize the parameters  $\theta$ . Next, to choose the best model, we use the validation set. Finally, we use the test set to measure the generalizability of our model. We assume the true relationship between the input  $x$  & the output  $y$  to be given by:

$$y = f(x) + \epsilon \quad (2.1)$$

with  $\epsilon$  being standard Gaussian noise with 0 mean and variance  $\sigma^2$ .

If our training set consists of a sample  $D = \{(x^1, y^1), (x^2, y^2) \dots (x^N, y^N)\}$ , where  $(x^i, y^i)$  is a single sample, our model (neural network or otherwise) must estimate a function  $f^*(x; D)$  that estimates  $f(x)$  well for all points in the training set as well as unseen data. We want the error  $(y - f^*(x; D))^2$  to be the minimum for all values of  $x$ . The expected error on an unseen sample  $x$  can be proven to be as follows:

$$E_D \left[ (y - f^*(x; D))^2 \right] = \left( \text{Bias}_D [f^*(x; D)] \right)^2 + \text{Var}_D [f^*(x; D)] + \sigma^2 \quad (2.2)$$

where,

$$\text{Bias}_D [f^*(x; D)] = E_D [f^*(x; D)] - f(x) \quad (2.3)$$

$$\text{Var}_D [f^*(x; D)] = E_D \left[ (E_D [f^*(x; D)] - f^*(x; D))^2 \right]. \quad (2.4)$$

The bias term accounts for the assumptions that the model we are using makes. If the function  $f(x)$  is truly non-linear but our model assumes  $f^*(x)$  is linear, the bias could be high. This is also called **under-fitting**. The variance term (Var) could be thought of as how far the predictions  $f^*(x)$  move about its mean. The variance is high when the model is complex (has more parameters or has a non-linear assumption), but the underlying function is not. In other words, the model **overfits** the training data, and small fluctuations in the training set could lead to varied predictions. There is usually a trade-off between bias & variance - where we can reduce variance by increasing the model's bias and vice-versa.

### 2.2.2 Gradient Descent

To understand how we optimize a neural network's weights, we look at the most commonly used algorithm - gradient descent. Similar to the previous section, we have the inputs  $x$  and



some known labels  $y$  to form the training set  $D = \{(x^1, y^1), (x^2, y^2) \dots (x^N, y^N)\}$ . To measure the performance & train the network, we must define a loss function.

$$J(\theta) = L(f^*(x^i; \theta), y^i) \quad (2.5)$$

This compares the prediction of the network  $f^*(x)$  to the label  $y$ . To optimize using gradient descent, we compute the gradient of the loss function with respect to the parameters. This gives us the direction of the steepest ascent.

$$g = \nabla_{\theta} J(\theta) \quad (2.6)$$

To minimize the loss function, we must move opposite to this direction, thus following the loss function down the slope until we find a (local) minimum. This is an iterative procedure where we perform the following gradient update at each iteration and thus optimize the parameters.

$$\theta = \theta - \eta J(\theta) \quad (2.7)$$

$\eta$  corresponds to the hyper-parameter learning rate, which denotes how big a step we take towards the minimum. However, we only take the step with gradient descent after summing all the gradients from all the  $N$  training examples. This might become computationally expensive and slow when  $N$  is really large. Thus, in practice, we use **mini-batch gradient descent**, where we sample a batch of examples from the training set and update it right after the batch. The batch size, like the learning rate, is a hyper-parameter. Hyper-parameters are model configurations that are “tuned” and chosen manually using the validation set.

Other than the update step mentioned in equation 2.7, several other optimizers provide better convergence with respect to the loss function. For instance, the optimizer Adam [30] uses an adaptive learning rate for different parameters depending on the first & second moment of the gradients calculated. In this thesis, we use **AdamW** [31], that is, Adam with an improved implementation of weight-decay [32]. Weight decay is a regularisation technique that modifies the loss function with a term depending on the sum of L-2 norms of the weights. In the optimization process, this aids in penalization of large weights, thus contributing to sparsity & avoiding overfitting.

In neural networks, we first perform a **forward pass** - where we pass the input through the neural network to get a prediction. Then, we perform a **backward pass** that updates all the parameters using gradient descent & back-propagation [33] [34]. During back-propagation for a neural network, we compute the gradient with respect to the loss function to each parameter using the chain rule, one layer at a time, iterating from the final layer to the first. Thus, using the gradient, we can update every parameter using the update equation 2.7. In practice, the back-propagation process is automated using automatic differentiation [35] that updates the weights of the neural network using computational graphs that are constructed during the forward pass.

### 2.2.3 Linear Layers

To understand the sub-components of the ANN in figure 2.3, we take a look what comprises of a hidden layer. Firstly we have a linear layer. A linear layer is a fully connected layer (all nodes are interconnected) which takes an input in  $\mathbb{R}^M$  to  $\mathbb{R}^N$ . They are given by:

$$F(X) = WX + b \quad (2.8)$$

where  $F(X)$ ,  $X$  is the output and input respectively.  $W$  is the weight matrix of shape  $\mathbb{R}^{M \times N}$  and  $b$  is the bias term of shape  $\mathbb{R}^N$ . Thus, for the linear layer, we have  $W$  and  $b$  as the parameters.

### 2.2.4 Rectified Linear Units (ReLU)

As we discussed in section 2.2 hidden layers comprises of a linear layer followed by a non-linear activation function. ReLU [36] is the choice of non-linear activation used in this thesis. The ReLU has known to be better for the vanishing gradient problem (gradients become exceedingly small during the backward pass and the weights don't update) [37] than other non-linear activations such as sigmoid or tanh. It is defined as:

$$ReLU(X) = \max(0, X) \text{ where } X \text{ is the input to the ReLU function} \quad (2.9)$$

### 2.2.5 Dropout

Dropout is standard technique introduced in [38] that prevents over-fitting in neural networks. Some nodes of the linear layer along with it's connections are randomly dropped at random during training time with a probability parameter  $p$  (usually set to  $\approx 0.5$ ).

### 2.2.6 Multi-Layer-Perceptron

Apart from the visual analogy with the brain, an ANN, or Multi-layer perceptron could be looked at like a chain of functions as shown in equation 2.10. The diagram in figure 2.3 contained only 1 hidden layer. However, the number of hidden layers is actually a hyper-parameter that decides the representation capability of the network. As we add more nodes and layers, the representation capability of the model increases. However, this can lead to high variance or over-fitting. The choice of network size (number of nodes), number of hidden layers depends on the data, and is generally found using hyper-parameter tuning.

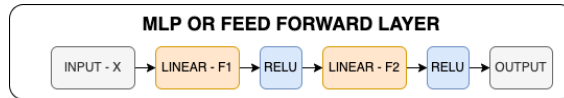


Figure 2.4: MLP as a chain of functions.

Figure 2.4 shows an example of an MLP with two hidden layers. If we have two hidden layers comprising of  $L$  nodes and an input  $X$  in  $\mathbb{R}^M$  then a forward pass in the MLP would be composed as:

$$MLP(X) = ReLU\left(F_2\left(ReLU(F_1(X))\right)\right) \quad (2.10)$$

The MLP would have an output in  $\mathbb{R}^L$ . Then the output would be converted to a vector or a scalar (via another linear layer) depending on the task, passed through a loss function and the parameters of the MLP would be updated using back-propagation.

### 2.2.7 Residual Connections

Some models in the thesis uses “residual connections” between two layers. Originally introduced by [39], residual connections were introduced to solve the degradation problem in deep neural networks (one with several layers), compared to shallower ones. With increasing depth, the training accuracy of the network gets saturated and degrades rapidly. To solve this, the authors introduced skip connections between layers. As we see in figure 2.5 the input  $X$  is directly added to the output of the layer  $F(X)$ . This allows the model to learn identity mappings (passing the output  $X$  of the shallower layer directly) whenever required. This also aids in gradient flow and leads to improved optimization for the network.

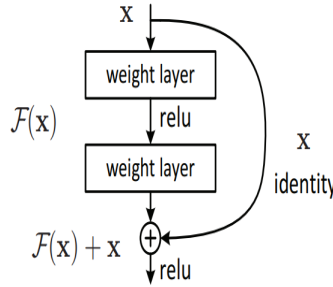


Figure 2.5: Residual connection between layers. Image from [39].

### 2.2.8 Binary Cross Entropy

We use the loss function binary cross entropy in the thesis. Binary cross entropy is a loss function that is used in neural networks when the task is binary classification. If we have labels  $y \in \{0,1\}$  and  $N$  examples and the neural network predicts a probability score  $\hat{y}$  between 0 & 1 for each of the  $N$  examples, the loss is formulated as follows:

$$Loss = -\frac{1}{N} \sum_{i=1}^N (y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)) \quad (2.11)$$

## 2.3 Embeddings

The inputs to a neural network model aren't always numerical. However to use neural networks & optimizers all input features must be so. We often have text or even categorical data to process as input. For these cases, we use a  $d$ -dimensional numerical vector representation of **each category or word** to feed into our model. The representations are called **embeddings**. The embeddings are learnable model parameters that are updated during training. This way they capture the representation of the word/category that is useful for the task. Leveraging this idea, several types of models have been built to capture semantic relationships between words. These are termed as **language models**. For instance, Continuous bag of words (CBOW), Skip-Gram models [40] used word embeddings to either predict a centre word given its context (the words surrounding the centre word in the local window) or predict the context words given the centre word. These models yielded representations of words that were meaningful and had properties that allowed similar words to have similar embeddings.

## 2.4 Attention

This thesis uses embeddings for sentences generated from Sentence Bidirectional Encoder Representations from Transformers (SBERT) [6]. To understand how SBERT works, we provide a description of attention - the core mechanism behind the architecture.

The attention mechanism [41] was developed in conjunction with sequence modelling with recurrent neural networks (RNNs) [42]. RNNs are a type of neural network that is usually used for sequence modelling. Sequences could range from words, audio or even frames of video. RNNs can be used for a multitude of tasks - many-to-one (Sentiment Analysis), many-to-many (Language Translation), or one-to-many (Poem Generation from the title). At each time step in the sequence, one RNN cell receives a hidden state from the previous time step (forming a recurrence relation) along with an input from the original sequence. It then produces a hidden state that has context from all the previous time steps. The hidden state  $h^t$  at time step  $t$  with an input  $x^t$  is given as

$$h^t = g_1(W_{hh}h^{t-1} + W_{hx}x^t + b_h) \quad (2.12)$$

To produce an output  $y^t$  from the RNN cell, we have,

$$y^t = g_2(W_{yh}h^t + b_y) \quad (2.13)$$

Here,  $g_1, g_2$  are non-linear activation functions and  $W_{hh}, W_{hx}, W_{yh}, b_h, b_y$  are parameters. We take the example of language translation from an English sentence  $X_1, X_2, \dots, X_N$  to a French sentence  $Y_1, Y_2, \dots, Y_N$  to demonstrate the attention mechanism. To this end, we illustrate an encoder-decoder architecture composed of two stacked RNNs that perform the translation task in figure 2.6. The encoder (in blue) takes each English word at a time step as input and

produces a context vector which summarizes the entire English sentence. In regular RNNs, the decoder (in red) takes the context vector and produces an output french word at each time step. However, the entire context of the English sentence has to be represented by one vector. Therefore, for longer sentences, the context is lost. This proved to be a bottleneck and the attention mechanism was developed to overcome this problem. For the simplest form of attention, the decoder at each time step “attends” to every time step in the input sequence to weigh how important that word is for the current time step in the decoder.

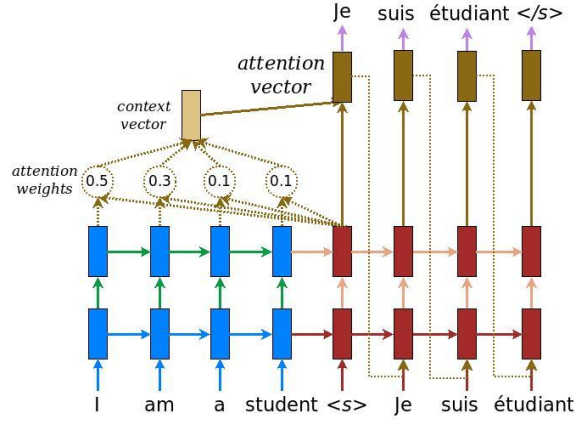


Figure 2.6: Encoder-Decoder RNN with attention for English to French translation. Each cell in the encoder is represented in blue, while the decoders are represented in red. The attention mechanism is demonstrated for the first decoder state. Image from [43].

From figure 2.6, if we look at the decoder at time step 1, when it receives the start token  $< s >$  (indicating the beginning of a sentence), the particular hidden state attends to all of the time steps in the encoder. This is illustrated using the arrows from the red hidden state to all the blue ones. We see from the numbers in the circle, that the attention weights are **higher for “I”** while computing the **context vector to translate to “Je”**. To understand how we compute this, we must have a notion of similarity between “I” and “Je”. Therefore, we introduce a function  $F_{att}$  that takes the decoder’s hidden state at time step 1 and calculates how similar it is to the hidden states of the encoder. Then, to calculate normalized weights, we use the softmax function (described in subsequent sections) so all the weights sum to 1. If we have the current decoder state  $h^d$  and any encoder state  $h^e$ , we calculate the attention weight  $\alpha_{d,e}$  between them as:

$$\alpha_{d,e} = \frac{\exp(F_{att}(h^d, h^e))}{\sum_{e=1}^N \exp(F_{att}(h^d, h^e))} \quad (2.14)$$

where  $N$  denotes the number of hidden states in the encoder. Then the context vector is computed with the linear combination of the encoder hidden states weighted by the attention weights. Thus, to get the context for the word “Je”, we get a linear combination of the encoder hidden states weighted by the attention weights for each of the words for “I am

a student” individually. We note that the through optimization the model automatically learns that the weight for “I” to produce “Je” should be higher.

## 2.5 Multi-headed Attention

The RNN has the disadvantage of sequential computation that precludes parallelization. To avoid this, the authors of [5] introduced **transformers** that capture input-output dependencies only via attention. The main component of this module is the “Multi-headed Attention” layer. Firstly, the attention mechanism is similar to the one described in section 2.4. However, to generalize, the authors introduce **queries, keys & values**, analogous to the query, key & values used in information retrieval systems. We take the example of an online search to explain the analogy. The query would be what we typed in the search bar. The keys would be what the query is matched to, and finally, the values are what we will be shown with respect to the best matches.

From the example in section 2.4, the query would be the decoder hidden state  $h_d$ . The keys and values would be the hidden state of the encoder  $h_e$ . Thus,  $\alpha_{d,e}$  are the weights obtained when we check the similarity of the keys & queries. And finally, the values  $h_e$  are weighted by this weight to obtain the final context vector. For  $F_{att}$  (introduced in the previous section), the function that computes the “similarity” the authors used a scaled-dot product explained in the next section.

### 2.5.1 Scaled Dot-Product Attention

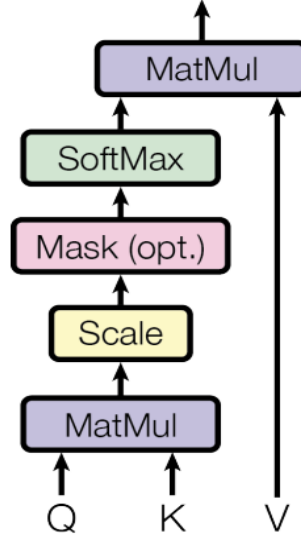


Figure 2.7: The figure illustrates scaled-dot product attention. This layer performs the operation  $Attention(Q, K, V)$  from equation 2.15. Image from [5].

In figure 2.7, we see the mechanism for the scaled dot-product attention. This is given by,

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (2.15)$$

$$Softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (2.16)$$

- Q, K, V stands for queries, keys & values.
- The Softmax function is the same as the one used in section 2.4 to normalize the attention weights.  $z_i$  is the input to the function.
- $d_k$  is the dimensionality of the query, key & value.

From equation (2.15), we notice that  $F_{att}$  is simply a dot product. However, the dot product is scaled by  $\sqrt{d_k}$ . The authors used the scaling factor ( $\sqrt{d_k}$ ) to avoid the problem of vanishing gradients since with large values of  $d_k$  the dot product grows larger in magnitude, pushing the softmax into a region where the gradients are small, thus preventing the network from learning. The expression  $Softmax(\frac{QK^T}{\sqrt{d_k}})$  gives us the attention weights. The final output from this layer is a weighted (by attention weights) combination of the values  $V$ .

### 2.5.2 Multi-headed attention module

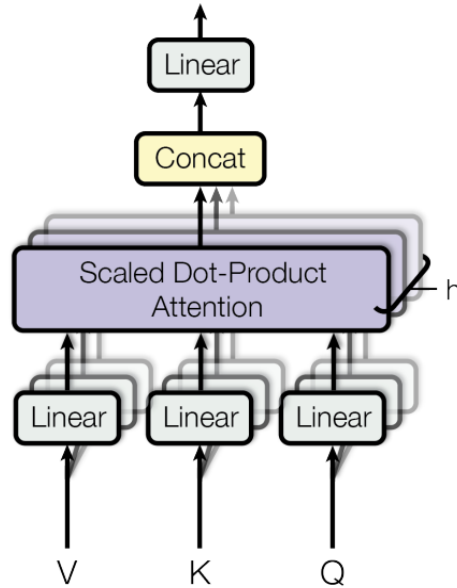


Figure 2.8: Multi-headed attention module with  $h$  heads represented by overlaid tiles. The output of this layer is given by  $MultiHead(Q,K,V)$  described in equation 2.18. Image from [5].

In the multi-headed attention module, instead of inputting the queries, keys & values to the scaled dot-product attention directly, we project the query, key and value vectors through

multiple linear layers. If we consider the dimensions of query, key & value to be  $d_k$ , then each of these linear layers takes an input in  $\mathbb{R}^{d_k}$  to  $\mathbb{R}^{d_{model}}$ , where  $\mathbb{R}^{d_{model}}$  denotes the dimensionality of the module. The “heads” module represents the number of times the scaled dot-product attention operation is repeated. This is denoted by  $h$ .

The authors argue that projecting the queries, keys & values to different subspaces multiple times allows each “head” to learn a separate task related to the structure of the sequence. After passing these projections through the scaled-dot production attention layers, the outputs from all the heads are concatenated and projected to  $d_{model}$  again through another linear layer. An obvious advantage is that this operation could be done in parallel across all the heads.

The output from each head,  $head_i$  is given as,

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \quad (2.17)$$

where  $(W_i^Q, W_i^K, W_i^V) \in \mathbb{R}^{d_{model} \times d_k}$  are the parameters of the first set of linear layers that are used to project  $Q, K, V$ . The overall output from the multi-headed attention module is given as,

$$MultiHead(Q, K, V) = Concat(head_1, head_2 \dots head_h)W^O \quad (2.18)$$

where  $W_O \in \mathbb{R}^{hd_k \times d_{model}}$  is the parameter for the final linear layer after the concatenation of all the heads.

### 2.5.3 Self-Attention

The authors used the term “self-attention” for the case where the query, key & the value refer to the same vector. This served the purpose of learning long-range dependencies in a sequence. Every input in the sequence can attend to itself. For instance, in the sentence, “The animal didn’t cross the river because it was too exhausted”, we want to know what “it” was referring to the river or the animal. Intuitively, and in practice, we see that the attention weights in this case for “it” in trained language models are higher for “animal”. This encoding process more computationally efficient than the encoders in RNNs since the dependencies could be calculated in parallel. The authors also found the self-attention mechanism to add interpretability to the model, unlike RNNs. The primary means of interpretability comes from visualizing the attention weights.

### 2.5.4 Transformer Encoder

Using the self-attention mechanism in the multi-headed attention module, the authors of [5] developed the transformer model. Originally, the transformer was made to replace RNN-based encoder-decoder architectures for language translation like the one described in section 2.4. The encoder and decoder in the transformer is made up of similar components. This is illustrated in figure 2.9. Using the encoder of the transformer, large language models were



built that achieved state of art results in several natural language processing (NLP) tasks. Since we use one of these language models in the thesis, we describe the encoder of the transformer in this section.

Instead of the encoder used in section 2.4, the RNN cells were replaced by the multi-headed attention module and an MLP, as seen in figure 2.8. We also notice that the self-attention mechanism makes the sequence permutation equivariant. That is, even if we shuffle up the sequence, there is no change in the output. Since the order of words in a sentence is important, the authors injected a positional encoding into this model. Additionally, there are residual connections (described in section 2.2.7) and layer normalization [44] in between the MLPs and the multi-headed attention as we see in figure 2.8. This whole unit is repeated  $N_x$  times to form the encoder.

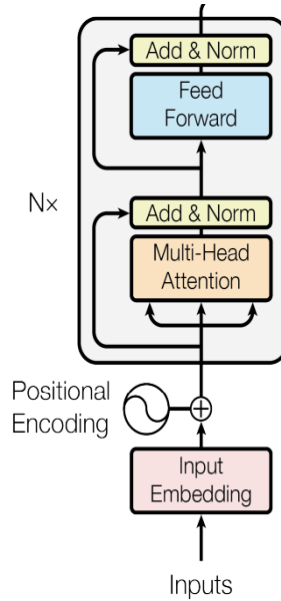


Figure 2.9: The Transformer encoder consists of a multi-headed attention, layer normalization & an MLP (feed-forward). The inputs to this model were words in a sentence. Therefore we notice a positional encoding in the image. Image from [5].

## 2.6 BERT

Using  $Nx = 12$  stacked transformer encoders of  $d_{model} = 768$  and  $h = 12$  mentioned in the previous section, [45] introduced Bidirectional Encoder Representations from Transformers (BERT). They used the process of “masked language modelling” where they masked out 15% of the words from a large corpus of the text and trained the model to predict the missing word. The multi-headed mechanism allowed the model to learn deep bi-directional contextual information since we have the current word attending to both the previous and subsequent words in the sentence. These contextual representations helped and provided state-of-the-art results in downstream NLP tasks. This introduced the idea of “pre-training” or transfer

learning [46] in NLP. This involved using the trained language model representations to further fine-tune BERT for other tasks such as question answering (given a question and a paragraph find the answer in the text), and many others involving language understanding.

## 2.7 Sentence-BERT

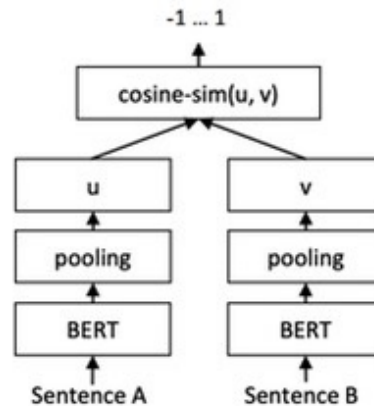


Figure 2.10: This figure illustrates sentence BERT. The inputs are “Sentence A” & “Sentence B” to individual BERT models. The model is trained on similarities between the sentences with the goal of obtaining a fixed representation for each sentence. Image from [6].

BERT set new benchmarks for the sentence textual similarity task (STS). In STS, we train the model to predict the similarity of a pair sentences. However, with BERT there was a bottleneck during inference. For instance, if we have 10,000 sentences and we need to check the sentence similarity for all of the pairs of sentences, we have to run the forward pass  $\binom{10000}{2}$  times. This would be 49,995,000 inference computations on the BERT model and could take over 65 hours on a modern GPU. This is because both sentences need to be fed into the same BERT model simultaneously. Another issue is how we derive sentence embeddings from BERT. When we input sentences to BERT, it is first broken down to tokens that are smaller units of the sentence. Each token in the sequence is represented by an embedding. In BERT, before every sentence, there is a [CLS] token that is appended to it. This represents the token for the whole sentence. Thus, to have sentence embeddings, we take the embedding of the [CLS] token. However, this sometimes leads to inaccurate sentence embeddings.

To alleviate the problems mentioned, [6] came up with sentence BERT based on a Siamese network where we have two individual towers (BERT models), one for each sentence. The overall model is illustrated in figure 2.10. The “pooling” layer after the BERT model in each tower determines the strategy to extract the sentence embedding. Instead of using the [CLS] token, the authors experimented with two other approaches: taking the mean of

all the word embeddings and using the maximum value over time (each dimension of the embedding). Ultimately for this model, the mean strategy worked the best. Once we feed in a pair of sentences, we obtain fixed representation  $u$  &  $v$  for both sentences. Then, the cosine similarities ( $\frac{u \cdot v}{\|u\| \|v\|}$ ) of both vectors give us a score of how similar the two sentences are. A squared error objective (squared difference between labels and predicted value) is used to train the model. The main difference lies in the fact that we are not feeding both of the sentences to the same BERT model. In comparison for the 10000 sentences, the computation takes only 5 seconds compared to the 65 hours. Additionally, training the model on cosine similarities allows for more useful sentence embeddings since we are training the model to learn the similarities between two sentences.

## 2.8 Recipe Recommendation

We now discuss previous work done in recipe recommendation which is the main focus of this thesis. Typically, the problem statement is to recommend users new recipes based on their previous ratings of other recipes [47][48][49] or their browsing history [50]. The authors of [51] presents the various recommendation algorithms used in the food recommendation systems. In terms of the data, for food recommendations, we have the recipes as items and some content information like steps to complete the recipe, a list of ingredients, and recipe tags.

We now review some of the methods in practice and elaborate on some methods that we use in some shape or form in the thesis.

### 2.8.1 CF methods

CF methods described in section 2.1.3 utilizes the user-item matrix to recommend recipes based on similarities between users. The authors of [48] tried a nearest neighbour approach based on the user-item matrix. Whereas, [52] used Singular Value Decomposition [53] which outperformed the nearest neighbour approach. [51] tried a variety of CF methods including Latent Dirichlet Allocation (LDA) [54] & Matrix factorization [55]. For the thesis we use matrix factorization as a baseline for a CF-based method.

### 2.8.2 Content-based methods

In content-based methods (described in section 2.1.3), we use information about the recipe to make new recommendations. As for previous content-based approaches in recipe recommendation, [56] used ingredient networks which pit frequently co-occurring ingredients together for substituting ingredients in recipes based on a users suggestions for the modification of a recipe. There are approaches based on food images as [57] notes that consumers often base their meal consumption on images they saw online. Based on that, [58] used image features such as brightness, sharpness & colourfulness to distinguish between user preferences for the recipes that they chose.

### 2.8.3 Hybrid-based methods

As we noted in section 2.1.3, hybrid-based systems that incorporate content information, as well as ratings often improve some of the shortcomings of purely CF & content-based methods. In that respect, [59] used matrix factorization that uses recipe tags along with ratings to slightly improve on the purely rating-based matrix factorization baselines. The authors of [47] used ingredient information as content information within the matrix factorization objective function to show superior results from baselines. They termed this as “content-boosted matrix factorization”. Instead of directly using categorical recipe embeddings (where each recipe is represented by an embedding like the one described in section 2.3), they used a linear combination of ingredient embeddings. As a result, this paper showed that representing recipes as a combination of ingredients could be useful for recipe recommendation. We use this idea later on in the thesis, and the method is described in detail in subsequent sections.

All approaches discussed till now were based on traditional methods of recommendation system algorithms. However as we discussed in section 2.1.4, the state of art for recommendation systems has moved on to neural networks.

### 2.8.4 Neural Network based methods

The use of neural networks have been limited in recipe recommendation. Some examples from previous research are listed in this section. The authors of [60] used convolutional neural networks (specialized neural networks that take into account the structure of images) on pictures of ingredients to detect the ingredients in a picture. At inference, they use the ingredients to make recommendations based on the closest recipe to the particular set of ingredients detected in the picture. [49] used a graph based attention model. They had similarities between ingredient pairs, recipe pairs, ingredient-recipe pairs and their relative importance to the recommendation problem were sought out using attention. However, research on improving on neural recommendation baselines on recipe recommendation that are universally used in other recommendation tasks have been limited. Therefore a sub-goal of our thesis was to compare neural network approaches for recipe recommendations to machine learning models that have been previously used for the task. To this end, we implement 3 variants of matrix factorization (section 3.3.1, section 3.3.2, section 3.3.3), and then focus on constructing common neural network baselines (section 3.3 for generic recommendation systems). Finally, we improve the neural network architecture to fit the particular task of recipe recommendation (section 3.3.7).

### 2.8.5 Health-based methods

Incorporating health into the recommender system to improve people’s diets has been an area of focus in the food recommendation niche. Normally balancing out health requirements reduces the chances that the user will like the recipe. There are methods which linearly combine the rating model and a separate health score for each recipe. Therefore, we have a

filter based on health after the rating model has generated user preferences. This could be denoted as an example of the “knowledge-based” recommender system described in section 2.1.3. For instance, [61] formulates their “health-aware recommendation system” as follows:


$$score(U, I) = W_R * RatingModel(U, I) + W_H * HealthScore(U, I) \quad (2.19)$$

where,

- RatingModel is the model used to predict the rating for any user-item combination.
- HealthScore is quantified for any user-recipe combination using a difference of calories that the user needs and the number of calories in a recipe.
- $W_R$  &  $W_H$  is the weighting between user preference and the health component. The authors optimized these weights with a focus on balancing the health component with the rating component.

The combined score,  $score(U, I)$  is then sorted in descending order and used to generate recommendations per user. However, this was only based on calories and did not consider macro-nutrients. By definition, macro-nutrients are the nutrients we require in the most significant amounts: fats, protein & carbohydrates. The concept of “calories in, calories out” [62] tells us if we consume more calories than we expend, we gain weight. More recent research has demonstrated that food quality and macro-nutrients influence a person’s health and weight more than simply calories. For instance, [63] shows that calories from carbohydrates contribute to rising obesity prevalence far more than total calorie intake.

The Food Standard Agency (FSA) in the UK devised the traffic light system [64] to inform consumers about how healthy a food is depending on the quantity of some macro-nutrients that represent how nutritious a food is. This includes fat, saturated fat, sugars & salt.



	LOW	MEDIUM	HIGH	
	Per 100g	Per 100g	Per 100g	Per portion
<b>Fat</b>	3.0g or less	3.0g - 17.5g	More than 17.5g	More than 21g
<b>Saturates</b>	1.5g or less	1.5g - 5.0g	More than 5.0g	More than 6.0g
<b>(Total) Sugars</b>	5.0g or less	5.0g - 22.5g	More than 22.5g	More than 27g
<b>Salt</b>	0.3g or less	0.3g - 1.5g	More than 1.5g	More than 1.8g

Figure 2.11: Traffic lights according to the FSA. The range for each macro-nutrient under green, amber & red traffic lights is shown in this figure. Image from [65].

As we see in figure 2.11, there are cut-off values for each macro-nutrient that is demarcated as green, amber or red. The idea is that consumers can directly see the colour of these traffic lights for a particular product and judge its “healthiness”. Figure 2.12 shows an example of a product with a red light for sugars, amber for salt and green for fat & saturated Fats. Numerically, in research the **FSA score** is calculated by assigning labels - 0, 1 or 2 to each traffic light and adding the score for each component. For instance the score for the product

(observing the colour coding) in figure 2.12 would be (0: Fat) + (0: Saturated Fat) + (2: Sugar) + (1: Salt) = 3.

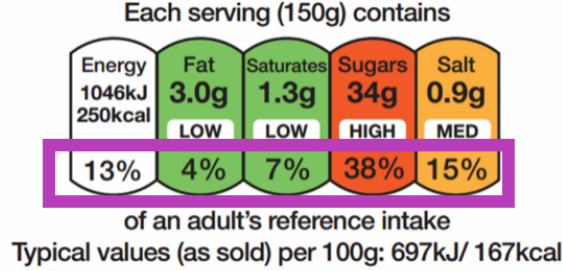


Figure 2.12: Traffic lights on the back of a typical product in the supermarket. Image from [64].

Another recognized nutrition score is by the world health organization, termed the WHO score [66]. This is calculated from 0-7, stratified by the quantity of macro-nutrients. Using the FSA score, the WHO score and a rating model, [67] came up with a scoring method for all users and items. The scores were given by:

$$score(U, I) = RatingModel(U, I) \cdot (WHO_{score}(I) + 1) \quad (2.20)$$

$$score(U, I) = RatingModel(U, I) \cdot (13 - FSA_{score}(I)) \quad (2.21)$$

However, they found optimizing the trade-off between the RatingModel & the FSA score challenging. This is because the distribution of the outputs of the rating model is different than that of the health scores of the recipes. They also found the quality of the recommendations declined when using this particular scoring method. For instance, using this scoring method, a lot of the users were getting recommended the same set of popular recipes in the dataset that had a good health score. So the model kept recommending the same collection of recipes to most users, and thus the recommendations were not very personalized.

Another approach considered in health-aware systems are temporal methods. For example, we can consider a users daily/weekly consumption and base the recommended recipe on that constraint [68].

Finally, the other option is to have an ingredient or **recipe substitution**. The problem can be framed as follows: for a recipe, nudge the user towards a healthy recipe. To this end, [69] used Positive Pointwise Mutual Information (PPMI) between a recipe and its context. For the “context”, the authors extracted MyFitnessPal logs of foods for individual users. They hypothesized that recipes co-occurring together in the same meal would be suitable substitutes for each other. The PPMI between a recipe  $f_i$  and it’s context  $c_j$  is given as:

$$PPMI_{ij} = \max(\log \frac{\#(f_i, c_j) |D|}{\#(f_i) \#(c_j)}, 0) * \sqrt{\max(\#f_i, \#c_j), 0} \quad (2.22)$$

In equation 2.22,  $D$  indicates the set of all food context pairs,  $\#(f_i, c_j)$  is the number of times  $f_i$  and  $c_j$  occur together in  $D$ , while  $\#(f_i)$  &  $\#(c_i)$  is the number of times that they individually occur in  $D$ .

However, there were some limitations to this approach. The main one being that online recipe names are not always accurate. It is possible to have the same food item logged with entirely different names. For instance, chicken sandwich & Tesco’s chicken sandwich indicate the same food, but they are logged differently in this case. The authors argued that word-based similarity models could help mitigate this problem. A very simple approach would be to measure the number overlapping words between two recipes normalized by the length of the recipe title.

We extend this idea in our methods by using sentence BERT embeddings for similarities between recipes & nudging the user to a healthier recipe based on what they have liked previously. This would allow us to eliminate the problem of mismatching names without any pre-processing. We hypothesised that a pre-trained sentence BERT embedding can capture the similarity between any two recipes using the contextual description of the recipe. This will allow us to incorporate all the content information we have about the recipe in the similarity metric. The overall method is described in section 3.6. We compare our method to [67] (FSA score & Rating combination) and try to overcome the personalization problem of the recommendations that they faced. In other words, our primary goal was ensuring that the recommended healthy recipes are considered relevant by the user.

# Chapter 3

## Methods

In this chapter, a detailed description of the approach taken to the problem is described.

### 3.1 Data

The data for this project was generated by [70]. The dataset consists of 17 years of user-recipe interactions on food.com - a recipe-sharing website. Overall, the number of interactions were 1,132,367 with 226,270 recipes & 231,637 users. There were two sources of information available - one describing the recipes and the other on the interactions.

#### 3.1.1 Recipe Information

The available recipe information were as follows:

1. Recipe-ID: A unique identifier for the recipe.
2. Nutrition Information : The total calories, total fat (PDV), sugar (PDV), sodium (PDV), protein (PDV), saturated fat (PDV), and carbohydrates (PDV) for each recipe was mentioned, where PDV represents the **percentage daily value** of the nutrient required for consumption for a regular adult.
3. Tags : A total of 552 Tag's like "Healthy", "15-mins-or-less" describing the recipe were available along with each recipe.
4. Description: A description submitted by the user who posted the recipe on the website.
5. Ingredients: A list of ingredients that makes up the recipe.
6. Steps for Recipe: Step-wise description of each recipe on how to cook it.
7. Cooking Time: The total cooking time for the recipe in minutes.

#### 3.1.2 Interaction Information

The available interaction information were as follows:



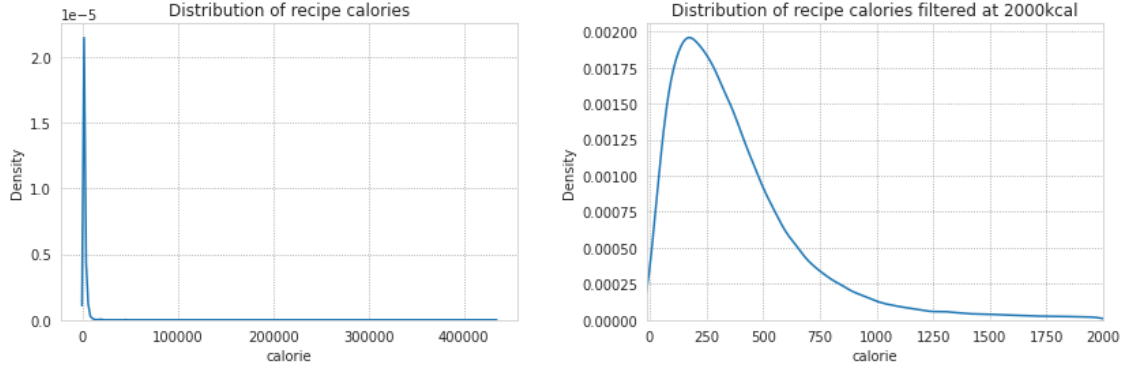


Figure 3.1: Distribution of calories amongst all recipes. **Left:** Unfiltered **Right:** Filtered under 2000 kcal

1. User-ID: The unique user-id for each user who interacted with a recipe
2. Recipe-ID: The unique recipe-id that constitutes the user-recipe interaction.
3. Rating: A numerical rating from 0-5 that the user gave a particular recipe.
4. Date: The date that the interaction was submitted.

### 3.1.3 Data-Exploration & Filtering

Since this dataset wasn't used widely for recommendation tasks in previous works, we performed some dataset exploration to see if the dataset was suitable for the recommendation task. Initially, to get an idea about what kind of recipes were there in the dataset, we created several summary statistics & plots that could help us filter out things that are not relevant to the task at hand.

Firstly, upon inspecting the recipes we found that a lot of the recipes have an abnormal amount of calories and subsequently a large cooking time.

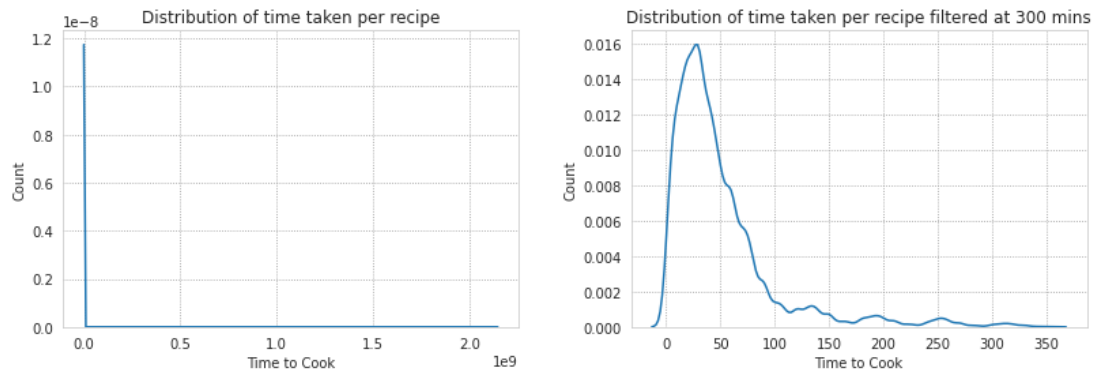
#### Calories

From figure 3.1 (left) it is visible that there are recipes in the dataset with a lot more calories than what is acceptable for a meal. Upon further inspection, it was seen that these were recipes in larger batches. Additionally, we had no information about serving size. Thus as pre-filtering step to visualize the distribution of calories we filter the recipes to only those under 2000 calories as seen in figure 3.1 (right). Amongst these recipes, we note that the mean of the number of calories in the recipes is  $\approx 382$ . This brings us to our first assumption and filter.

**Filter 1:** Since we have no information about serving size, we assume that recipes below **500 calories** are single serving. This also helps us with the health score computation described in section 3.4.

#### Time taken Per Recipe

Similar to the number of calories, there were recipes in the dataset that had an abnormally high amount of preparation time as seen in figure 3.2 (left).



**Filter 2:** Thus, to filter for time-taken, we assume that people would not like to be recommended recipes that took more than **5 hours (300 minutes)** to prepare. The final distribution of cooking time can be seen in figure 3.2 (right).

## Ingredients

Since we knew we were going to incorporate content information in the form of ingredients (as per *research goal-1*) into our models, we took a look at the various ingredients present in all the recipes.

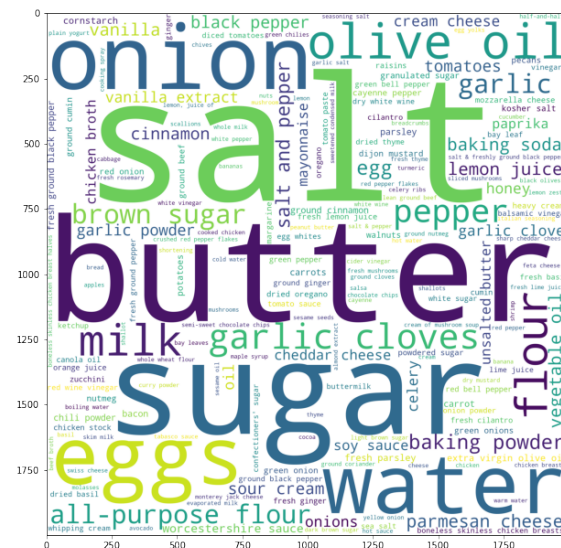


Figure 3.3: Ingredients word cloud. More frequent ingredients amongst the recipes are represented by a larger font in the figure.

To visualize all 14942 ingredients we use a word cloud as shown in figure 3.3. It is easy to see by observation that common ingredients like salt, butter & sugar occur very frequently. However

we found that there were some rarer ingredients that could be broken down into sub-parts. For instance, “**orange zest with chilli**” could be broken down into **orange zest** and **chilli** which are more frequently available in the list of ingredients. To facilitate this, we scraped base ingredients from the *world open food facts website* that contains a list of  $\approx 4000$  ingredients that are commonly found on food labels. Then we followed the following steps to break down ingredients:

### Algorithm to break down ingredients

1. Initialize a dictionary *rare-ingredients* that contains the occurrence of each ingredient amongst all the recipes in the dataset.
2. Find rare recipes from the dictionary where the occurrence is 1.
3. Initialize a loop through all the rare ingredients (occurrence=1).
4. Find any matches among the list of base ingredients scraped from the website mentioned above to the rare ingredients. If any such ingredients are found store them in another dictionary *rare-ing-to-sub* mapping the **rare ingredient to base ingredients**.
5. Initialize another loop through all the recipes.
6. If an ingredient in a recipe is in *rare-ingredients* add it’s base ingredients from *rare-ing-to-sub* to the recipe.

### Interactions

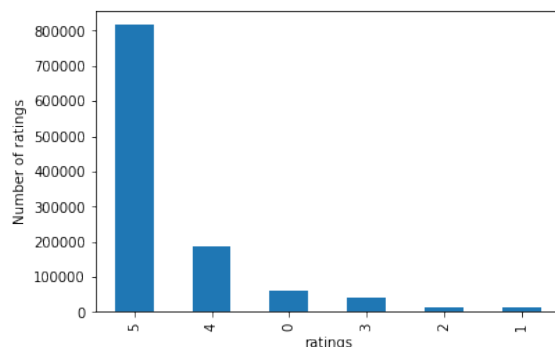


Figure 3.4: Distribution of ratings for all the interactions.

In figure 3.4 we visualize the distributions of the ratings given by the users in all their interactions with the recipes. We notice that users mostly give a rating of 5 for any recipe they rate. From this, we assume that if a user does not rate a particular recipe 5 or 4, they do not like it. We convert the rating prediction problem (discussed in subsequent sections) into a binary classification one - where “**like or 1**” indicates that a user has rated a recipe **5 or 4** and “**dislike or 0**” indicates a user has rated a recipe **3 or below**. This is akin to the binary recommendation problem setting as mentioned in section 2.1.1. However, from the distribution in figure 3.1 we see that the number of 5s & 4s are far higher than ratings of 3 or below. This makes the problem

highly imbalanced. Also, converting the problem into the mentioned structure can pose a problem. For example, if a user rated a recipe with only one unique kind of rating (only 5s, or only 0s), then we would only have one type of rating class available for each user.

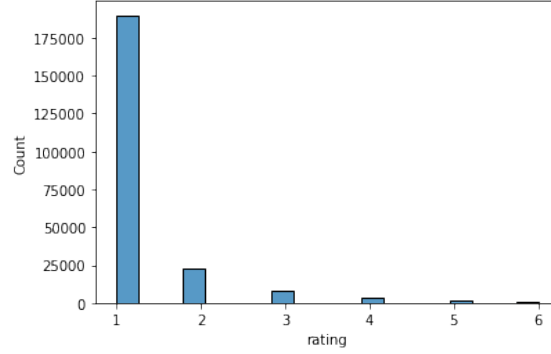


Figure 3.5: Number of unique ratings given by all users. For instance, “1” on the x-axis indicates a user has rated all recipes with only one type of rating.

If we look at the number of unique ratings for all users in figure 3.5 we see that this is indeed the case. There were two options to fix this. If “likes or 1s” dominate the dataset, which is the case here, negatives or 0s could be sampled from the rest of the dataset as shown in [1] for each user. In this case we simply decided to apply a filter that fixes this problem, since re-sampling caused the models to perform worse.

**Filter 3:** Choose users who have minimum 3 unique ratings amongst 0,1,2,3,4,5. This guarantees that we will have a disliked and liked recipes for each user.

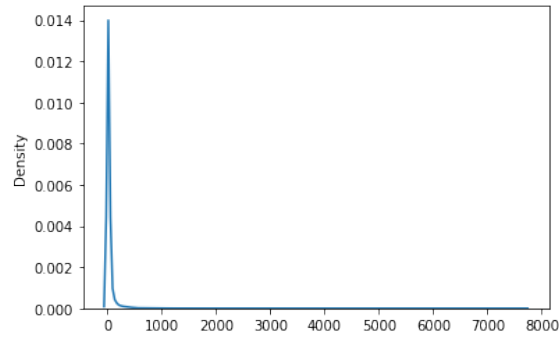


Figure 3.6: This figure illustrates the distribution of the number of recipes rated per user.

Additionally, from figure 3.6 that shows the number of recipes rated per user, we note that most users are not active. Or in other words, they rated very few recipes. We choose to only consider active users, thus arriving at filter 4.

**Filter 4:** Choose users who have rated a minimum of 20 recipes.

Finally, we get  $N_{users}=3705$  and  $N_{recipes}=45276$  with a total number of interactions of 336494.

## 3.2 Data-Split

Since there wasn't a standard split for validation, testing & training for the data, this was done manually. Usually for recommender system problems, we perform this split either by each user [71] or completely at random. We choose to group by user & sample 25% of their interactions from the entire set. We further sample 50% of the interactions from the 25% to form the test set & the rest are reserved for validation. These splits are done at random, 3 times over - so we have 3 sets of train, test & validation splits. All models henceforth are therefore trained thrice and metrics are reported with the average and standard deviation of the 3 test sets. To choose the best model, during training, the model with the best validation loss is saved and tested on the test set.

## 3.3 Rating Model

As mentioned in section 2.1 one problem statement of recommender systems involve finding interactions  $R$  for every user-item pair. In this case  $r \in \{1, 0\}$  since we converted the categorical ratings to binary values. One of the sub-goals of the thesis mentioned in section 2.8.4 was to study the effectiveness of neural networks for recipe recommendation. To facilitate this, we implement 3 variants of matrix factorization models to test the utility of a non-neural network based method for recipe recommendation. We then implemented 3 neural-network based baselines before coming to our own novel model.

Additionally, *research goal-1* as mentioned in section 1.2 is to test the importance of content information in form of ingredients for recipe recommendation. Thus, we compare collaborative filtering methods to hybrid/content-based methods for recipe recommendation. Finally, we study the utility of multi-headed attention for recipe recommendation with respect to *research goal 2* from section 1.2. The loss function used was binary cross entropy (as described in section 2.2.8) for all models. The learning rate was tuned using the validation set for each model. The optimizer used was AdamW as mentioned in section 2.2.2. All models were implemented using PyTorch, an open source library where the backward pass is automated for us. The models were trained and tested on an NVIDIA GeForce RTX 3060 GPU.

### 3.3.1 Matrix Factorization (MF)

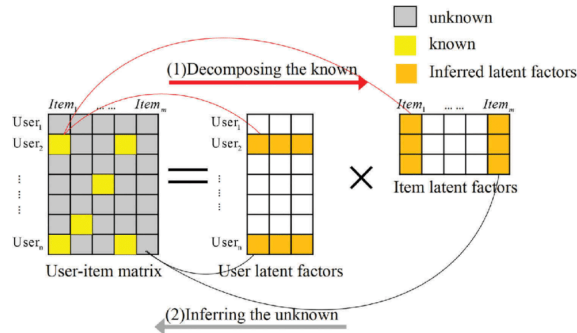


Figure 3.7: The user-item matrix  $S$  decomposed into matrices  $U$  and  $I^T$ . At inference,  $S$  is reconstructed using an inner product and missing values are inferred. Image from [72].

In matrix factorization, we have following the minimization problem,

$$\min_{U, I} ||S - UI^T|| \quad (3.1)$$

where the notation is defined as follows:

- $S$  is the user-item matrix with rating scores of size  $N_{users} \times N_{recipes}$
- $I$  is the embedding matrix of the recipe in a low dimensional space of size  $N_{recipes} \times K$ .
- $U$  is the embedding matrix of the user in a low dimensional space of size  $N_{users} \times K$

As illustrated in figure 3.7, we reconstruct the user-item matrix  $S$  using,

$$S \approx UI^T \quad (3.2)$$

To do this, we use equation (3.1) to minimize the reconstruction error. As described in section 2.1.1, each row of the  $U$  matrix represents a user  $u$ , and each row of the  $I$  matrix represents an item  $i$ . We notice that this approach is **purely collaborative** since all we do is use the user-item interactions in the form of ratings. From figure 3.7, we observe that matrix factorization represents each user and item into a **latent space of dimension  $K$** . The dot product of each user and item embedding gives us the predicted rating for that user-item combination. The predicted rating  $\hat{r}$  for any user  $u$  & item  $i$  is given as:

$$\hat{r} = e_u \cdot e_i \quad (3.3)$$

where  $e_u$  is the  $u^{th}$  row of the matrix  $U$  and  $e_i$  is the  $i^{th}$  row of the matrix  $I$

For our problem, since we assume binary values, we formulate the problem in the form of minimizing the binary cross-entropy loss instead of equation (3.1). This is given as

$$\min \sum_{u, i \in R} \left[ r_{ui} \log(\sigma(e_u \cdot e_i)) + p_s(1 - r_{ui}) \log(1 - \sigma(e_u \cdot e_i)) \right]. \quad (3.4)$$

where

- $r_{u,i}$  represents the true binary rating for the user-item combination (u-i).
- $e_u$  is the  $K$  dimensional representation for a particular user  $u$  while  $e_i$  is the  $K$  dimensional representation for an item  $i$ .
- $\sigma(x)$  is the sigmoid function that converts a raw score into values between 0 & 1, and is given by  $\sigma(x) = \frac{1}{1 + \exp(-x)}$
- Additionally, as we recall from the data exploration, there is a major class imbalance towards the “1” class. To mitigate this problem we decided to use a weighted binary cross entropy loss shown in equation (3.4).  $p_s$  in equation (3.4) gives us the weight of the positive class. In this case, we chose  $p_s = \frac{Numberofclass0}{Numberofclass1}$  for each training iteration. Therefore the contribution of the positive class in the loss is reduced by this weightage  $p_s$ . The loss function and the weighting stayed the same for all the models explored hereafter.

The dimension size  $K$  of both user & item embeddings  $I$  &  $U$  were kept at 256 (found using hyper-parameter tuning). The learning rate was 0.001 and the batch size used was 12. The model was trained thrice for ten epochs since the validation loss did not improve further.

### 3.3.2 Matrix Factorization with biases

The interaction between the user & item is captured using dot products in standard matrix factorization. However, there is the fact that some users rate a higher rating value than others on average. Or, some items are more popular and, as a result, are rated higher. To account for this, we incorporate two learnable bias terms that capture this -  $b_i$  for the items &  $b_u$  for the users. In the previous section, we expressed predicted rating  $\hat{r}$  as the dot product of the item & user embedding. Now we add both the user bias & the item bias to the score. This is shown below.

$$\hat{r} = e_u \cdot e_i + b_u + b_i \quad (3.5)$$

We replace the score ( $\hat{r}$ ) in the cross-entropy loss in equation (3.4). However, like the previous method, this is also purely collaborative since we are still not adding any content information to the model. The model configuration (learning rate, batch size, embedding dimension) is kept exactly the same as the baseline matrix factorization from section 3.3.1.

### 3.3.3 Content-boosted Matrix factorization

As discussed in section 2.8.2, [47] incorporated ingredient information in the standard matrix factorization algorithm. They did this by having a separate embedding matrix  $\phi$  of size  $N_{ing}$  x  $K$  for all ingredients, where  $N_{ing}$  is the total number of ingredients in the dataset, and each row signifies one ingredient embedding. The recipe embedding matrix  $I$  is then given as:

$$I = X\phi \quad (3.6)$$

where  $X$  is matrix of size  $N_{recipes}$  X  $N_{ing}$  where each item  $x$  in the matrix is populated as follows:

$$x = \begin{cases} 1, & \text{if ingredient is in the recipe,} \\ 0, & \text{otherwise.} \end{cases}$$

For example, if we have a recipe with a list of ingredients  $[ing_1, ing_2, ing_3]$ , then the recipe embedding is represented as  $e_{ing_1} + e_{ing_2} + e_{ing_3}$  ( $e_{ing_i}$  represents the embedding of the  $i^{th}$  ingredient), where each ingredient is represented by an embedding of size  $K$  from the  $\phi$  matrix.

So the reconstruction of the user-item matrix  $S$  similar to equation (3.2) becomes:

$$S \approx U\phi^T X^T \quad (3.7)$$

$\hat{r}$ , thus is given as  $e_u \cdot (e_{ing_1} + e_{ing_2} + e_{ing_3} \dots)$ . Then, we minimize the binary cross entropy loss function as described in previous sections. In the code implementation, we padded each ingredient sequence with a “pad” token so that sequences are of the same length. Additionally, we made sure that the embedding for the pad token were always set to 0. This way, the padding does not affect the recipe embedding. The embedding dimension (users and ingredients), learning rate & batch size was kept the same as the previous matrix factorization models.

### 3.3.4 Neural Collaborative filtering (NCF)

Now we move on the neural network based approaches - albeit purely collaborative. The first model we consider is Neural Collaborative Filtering (NCF). We chose NCF since this was a common neural network baseline used for a variety of other recommender system tasks [73][74][69].

The authors of [75] showed the disadvantage of matrix factorization with graphical illustration that we show in figure 3.8. Matrix factorization, as discussed in the previous section, maps users and items into a latent space. The similarity of two users can thus be measured by the same dot product between two latent vectors or the cosine angle between the two (assuming unit vectors). Now as an demonstration, we consider the Jaccard similarity between two users in the user-item matrix as the ground truth that the matrix factorization model needs to recover.

If  $R_u$  is the set of items that user  $u$  has interacted with, then the Jaccard similarity between user  $i$  and  $j$  is given as  $\frac{|R_i \cap R_j|}{|R_i \cup R_j|}$ .

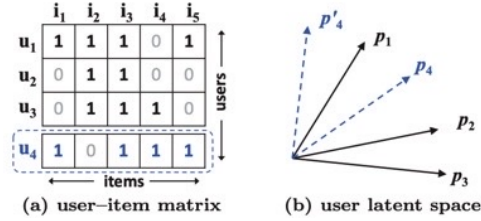


Figure 3.8: **a)** The user-item matrix  $S$  **b)** The latent space that matrix factorization projected each user to. Each arrow represents a user, and the blue arrow indicates the two possible assignment of a new user  $p_4$ . Image from [75].

In figure 3.8(b), we have 3 users  $u_1, u_2, u_3$  and their latent space vectors given by  $p_1, p_2$  &  $p_3$ . Now we add in a new user  $u_4$ . The ground truth similarity  $s$  between users is as follows:  $s_{41}(0.6) > s_{43}(0.4) > s_{42}(0.2)$ , which means that  $u_1$  is closest to  $u_4$ , followed by  $u_3$  and so on. However, as seen in figure 3.8(b), if the MF model places  $p_4$  closest to  $p_1$  in the latent space (the options are shown in the figure with dashed blue lines) it is closer to  $p_2$  instead of  $p_3$ . This shows that a simple inner product might not be enough to model user-item interactions.

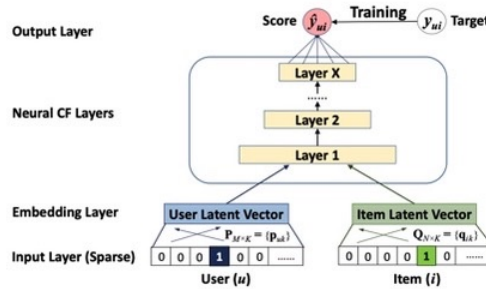


Figure 3.9: NCF architecture as illustrated by [75].



Instead of using the inner product of user & item embeddings, we now pass the concatenated user & item embedding into an MLP as illustrated in figure 3.9.

Here, we describe an example of a forward pass for the NCF model used in this thesis.

1. We initialize matrices for the user embeddings  $U$  and recipe embeddings  $I$ . Users and recipes are assigned a numerical value from 1 to  $N_{users}$  and 1 to  $N_{recipes}$  respectively. For the  $i^{th}$  recipe and the  $j^{th}$  user, the  $i^{th}$  entry of  $I$  and the  $j^{th}$  entry of  $U$  represents the respective recipe and user embedding. If we take a batch size of 12, we have vectors of size 12 x K for the user & item, respectively. For now, we assume these batched vectors to be denoted as  $e_u$  and  $e_i$ .
2. We obtain the vector  $X$  by concatenating the user & item embedding as  $X = e_u | e_i$ . (“|” is the pipe operators used to denote concatenation) Thus,  $X$  becomes a vector of size 12 x 2K.
3. Now, we follow the MLP structure as defined in section 2.2.6. Firstly, we have a linear layer that takes  $X$  from  $\mathbb{R}^{2K}$  to  $\mathbb{R}^{100}$ . This is followed by the non-linearity ReLU and a dropout layer with  $p = 0.3$ . Then another linear layer takes the vector from  $\mathbb{R}^{100}$  to  $\mathbb{R}^{50}$ . Post this, another ReLU is applied to the output of this layer.
4. Finally, a last linear layer takes the vector from  $\mathbb{R}^{50}$  to a single scalar value for each user-item combination in the batch of 12.
5. We then use the weighted binary cross entropy loss as previous sections to obtain a loss value.

The batch size remains 12, the learning rate was chosen to be 0.0001 and embedding size  $K$  was 256 (for users and recipes). The model was trained 3 times over the three training sets for 8 epochs each, at which point the validation loss did not show much improvement. Additionally, the **size of the linear layers for this model** and henceforth, were based on keeping the **number of parameters consistent** across all the models we tried out.

### 3.3.5 Two-tower Model

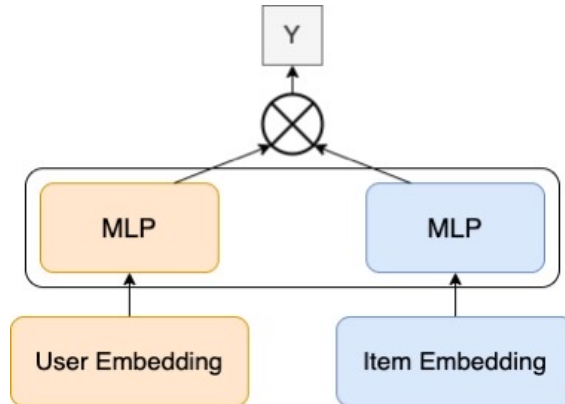


Figure 3.10: Two-tower model. Each MLP is a separate “tower” that is used to create representations for the user and the item.

Instead of having an MLP that takes in a concatenated version of the user and the item embeddings recent papers have shown **a separate MLP for the user & the item** could be used to generate different representations for the user and the item [76] [77] [78]. This class of models that have two similar towers are often known in literature as “Siamese Neural Networks” (the same structure as sentence BERT as explained in section 2.7). This approach allows us to custom build neural network architectures suitable for each tower. For instance, on the item side, content information can be put through different layers that best suit it. Till now, we kept the model purely collaborative and did not use any content information.

We have the forward pass as follows:

1. Like the NCF we start with the user embedding  $U$  and recipe embeddings  $I$ .
2. However we process each embedding separately using an identical MLP.
3. The structure of the MLP is as follows: A linear layer takes the embedding from  $\mathbb{R}^K$  to  $\mathbb{R}^{100}$ . This is followed by a ReLU and a dropout layer with probability  $p = 0.3$ . Similarly like the NCF, another linear layer takes the input now in  $\mathbb{R}^{100}$  to  $\mathbb{R}^{50}$ . After another ReLU, a final linear layer converts this into a scalar value.
4. We get two scalars for the user & the recipe that are then multiplied and passed through the loss function.

The configuration for training the model was kept exactly the same as the one used in NCF.

### 3.3.6 Recommendation with AutoInt

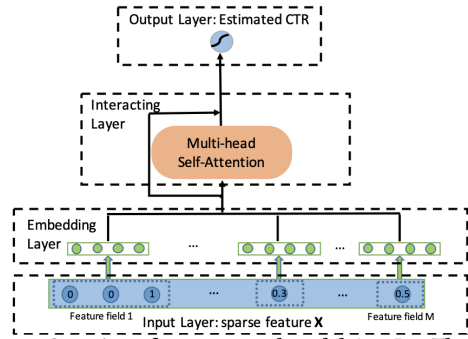


Figure 3.11: AutoInt model architecture from [79] used to predict click-through-rate using multi-headed attention. The input features  $x$  contains features from both items & users.

To incorporate content information into our model, we explored a self-attention based model that improved on NCF and two-tower methods for other recommendation tasks. The authors of [79] introduced AutoInt that explored automatic feature learning in recommender systems with self-attention. Their hypothesis was to do with learning multiple combinations of features that were helpful to the recommendation task with the self attention layer.

Firstly, if we have  $M$  features from the users and the items, they are concatenated as follows:

$$x = [x_1|x_2|...x_M] \quad (3.8)$$

In figure 3.11 above both numerical & categorical features are considered. We represent each feature as embeddings. This is given by

$$e_j = V_j X_j \quad (3.9)$$

where  $V_j$  is the embedding look up matrix,  $X_j$  represents the category of the embedding. This is exactly the same method of how we arrived at user, recipe & ingredient embeddings for previous methods. For our task, we concatenate the user & ingredient embeddings in the first step. So the sequence  $x$  becomes  $[e_u|e_{ing_1}|e_{ing_2}|e_{ing_3}...e_{ing_N}]$ . Here, the recipe consists for  $N$  ingredients  $[ing_1, ing_2...ing_N]$ . For now, we do not use the recipe rembeddings  $e_i$ . The reason for this is explained in the ablation study in section 4.1.4.

Now using the multi-headed attention block, for each head, we obtain attention weights for any two combination of features say  $ing_1$  &  $ing_2$  shown in the equation below.

$$\alpha_{ing_1, ing_2}^h = \frac{\exp(\psi^h(e_{ing_1}, e_{ing_2}))}{\sum_{n=1}^M \exp(\psi^h(e_{ing_1}, e_n))} \quad (3.10)$$

$$\psi^h(ing_1, ing_2) = \langle W_{query}^h e_{ing_1} W_{key}^h e_{ing_2} \rangle \quad (3.11)$$

where  $\psi^h(.,.)$  is the attention function that in original paper consists of a simple inner product as shown in equation 3.11. Here,  $h$  represents a single attention head. As explained in section 2.5, we project both  $e_{ing_1}$  and  $e_{ing_2}$  (query and key) through linear layers with parameters  $W_{query}$  and  $W_{key}$ . Here  $W_{query}, W_{key} \in \mathbb{R}^{d' \times K}$  and  $K$ , as usual, is the dimension of each embedding and  $d'$  is the space we project the embedding to. Since we derive the query, key & value from the same sequence  $x$ , we call this a self-attention layer (every feature in the sequence attends to itself). Finally, we use the values and attention weights  $\alpha_{ing_1, feature}^h$  to generate the representation  $e_{ing_1}^{\hat{h}}$  of the feature  $e_{ing_1}$ . This is done as follows:

$$e_{ing_1}^{\hat{h}} = \sum_{n=1}^M \alpha_{ing_1, n}^h (W_{value}^h e_n) \quad (3.12)$$

where  $W_{value}^h \in \mathbb{R}^{d' \times K}$  and  $N$  goes from 1 to  $M$  representing each of the  $M$  features in the sequence  $x$ . For our case, this would constitute of the user embeddings  $e_u$  and ingredient embeddings  $e_{ing_i}$ .

Therefore for each type of embedding all features in the sequence - the user embedding, recipe embedding, and ingredient embeddings are combined with attention weights to form the representation of the entire feature. Intuitively, this answers the question "For any feature in sequence which other feature is important?".

However till now, we only have the computation for one head of the multi-headed attention block. For  $h$  such heads we will receive  $h$  of the vectors  $e_{ing_1}^{\hat{i}}$  (where  $i$  receives values from 1 to  $h$ ). The authors hypothesised that each head might learn different combinations of features that are important to the final task.

To get the final output we must take the output from these  $h$  heads and concatenate them.

$$\hat{e}_m = \hat{e}_m^1 | \hat{e}_m^2 | \dots \hat{e}_m^h \quad (3.13)$$

Then, we have a final projection matrix  $W_0$  that takes  $\hat{e}_m$  from  $\mathbb{R}^{d'h}$  to  $\mathbb{R}^K$  where  $h$  is the number of attention heads. Also, the authors add a residual connection from the original set of features  $x$  to the final representation. The authors also experimented with adding another self-attention layer on top of the described layer. However, our experiments found this to reduce performance for our task. Hence, we only use one multi-headed attention block.

Finally, we pass the final representation of the features through another MLP similar to the one we used in section 3.3.4 to obtain the final output that we use to backpropagate and update the parameters similar to previous methods.

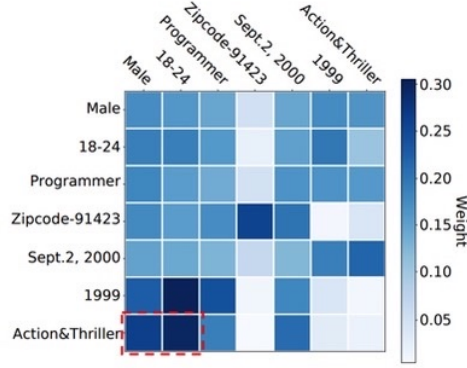


Figure 3.12: Examples of features from a movie recommendation problem along with their associated attention weights in a self-attention layer. Image from [79].

To provide intuition on how the model is useful, we show 3.12 visualizing the attention weights for feature combinations. Originally, the authors used the MovieLens-1M dataset containing information about users and the movies they rated. We can see in the heatmap of the attention weights from figure 3.12, the model captured that the male demographic from 18-24 has an affinity for action and thriller movies (denoted by dashed-red box in figure 3.12).

$d'$  or the model dimension for the self-attention module was kept the same as the embedding dimension 256 (for users & ingredients). The number of attention heads was chosen to be 8. The rest of the hyper-parameters were kept the same as the ones used in NCF.

### 3.3.7 Our model

Following *research goal - 2* in section 1.2, we wanted to design a model suited to the recipe recommendation task. Constructing more advanced neural network models has been building towards having assumptions about the structure of the task and the data itself. For instance, in images, we use convolutional neural networks that assume that nearby pixels are usually similar. So we drop a few of the connections from a fully connected (linear) layer to adjust for the structure

of the image. Here, we combine a few ideas from previous methods to make a set of assumptions that might be relevant to our problem.

1. Recipes embeddings can be built from a **linear combination of its ingredients**. This was inspired from “Content-boosted Matrix factorization” (as detailed in section 3.3.3) where a simple ingredient embedding sum was the recipe embedding.
2. Users might have **preference for specific ingredients**. We use the multi-headed attention mechanism to look for preferences that would help our model perform better on the rating prediction task. Another use of the self-attention layer could be to see which features in the sequence are essential for the task. So, are **some ingredients in the list of ingredients for a particular recipe redundant**? We also note that the order of the sequence of ingredients don’t matter. Thus, we do not add any positional encoding to the multi-headed attention module (which would be required if we were modelling words in a sentence).

For capturing user preference for ingredients within the multi-headed attention mechanism - we use the **user embedding as the query**, and the **ingredient embedding as the keys & the values**. Thus when the **query (users)** is “**matched**” to the **key (ingredients)**, we get a set of attention weights that could be interpreted as the preference of the user for the particular ingredient.

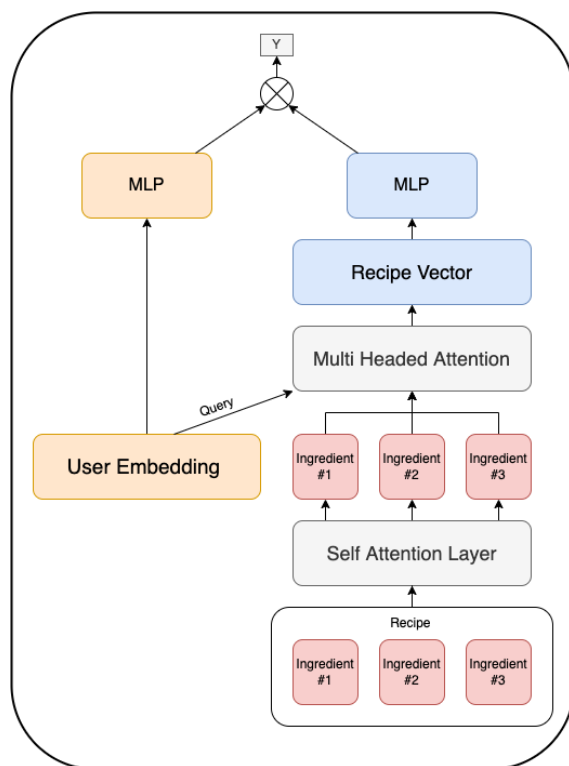


Figure 3.13: Model architecture. We use the two-tower approach and model the user & items separately. Firstly, ingredient sequences go through a self attention layer. Then, a multi-headed layer is used to generate the recipe vector from a weighted combination of ingredient embeddings.

Using our previous notation, we have user embeddings  $e_u$ , and a sequence of ingredient embeddings  $[e_{ing_1}|e_{ing_2}|e_{ing_3}..e_{ing_N}]$  for each user-recipe pair. As described earlier, we set  $e_u \in R^K$  as the query and ingredient embeddings  $e_{ing} \in R^K$  as the keys & values. To generate the attention weights for a sequence of ingredients  $[ing_1|ing_2|ing_3..ing_N]$  and a particular user  $u$  we have:

$$\alpha_{u,ing_i}^h = \frac{\exp(\psi^h(e_u, e_{ing_i}))}{\sum_{n=1}^N \exp(\psi^h(e_{ing_i}, e_{ing_n}))} \quad (3.14)$$

$$\hat{e}_i^h = \sum_{n=1}^N \alpha_{u,ing_n}^h (W_{value}^h e_{ing_n}) \quad (3.15)$$

where,

- $u$  represents a single user and  $\hat{e}_i$  is the embedding constructed for a recipe  $i$ .
- $\alpha_{u,ing_i}^h$  represents the attention weight for user  $u$ , an ingredient  $ing_i$  for a single attention head  $h$ . There will be an attention weight for each of the  $N$  ingredients in a recipe & a particular user. This weight is one that might provide interpretability to the model, answering the question, “For this particular user, how important is this ingredient?”
- Finally, we combine the ingredient embeddings to obtain an embedding for a recipe using equation 3.15 with the attention weights for a single head  $h$ . Representations from all the heads are concatenated and are passed through a linear layer to obtain the final recipe representation  $e_i$  (similar to equation 3.13 in section 3.3.6). Therefore, we consider the recipe to be a linear combination of the ingredient embeddings. However, in this case, they are weighted by the attention weights.

Before passing the user embedding and the ingredient sequence in the multi-headed attention module, we also use a self-attention layer similar to the AutoInt model (from the previous section 3.3.6). In this layer, the ingredient embedding is treated as the query, key & value. Intuitively, this layer signifies, “In these sequence of ingredients, are some ingredients more important than others?”.

This is illustrated in the model architecture in figure 3.13. We followed the two-tower model approach and modelled the user & the recipes differently (motivated by the results in section 4.1.2). After the recipe embedding is derived, we re-use the user embedding to obtain the final score using two separate MLPs (one for the user and the other for the recipe) similar to section 3.3.5.

The multi-headed attention configuration, batch size, learning rate, and number of epochs was kept the same as AutoInt in the previous section for comparability.

## 3.4 Health Model

To facilitate *research goal - 3* from section 1.2, we describe the process of generating a health score for every recipe in the dataset. This will allow us to achieve our goal of recommending healthier recipes that are relevant.

We follow the FSA guidelines for computing the health score, similar to previous work mentioned in section 2.8.5. However, we eliminate the “fat” category for scoring. Many papers [80] [81] have stated that all fats aren’t bad for you. For instance, unsaturated fats are known to be healthier than saturated fats. Therefore, we keep three categories for the health score - saturated fat, sugar and salt.

As mentioned before, the quantity of nutrition values available in the dataset is in Percentage Daily Values (PDV). Firstly, we convert each one of the values to grams (g) using

$$SaturatedFat(g) = \frac{Fat(PDV)}{100} * 20 \quad (3.16)$$

$$Sodium(g) = \frac{Sodium(PDV)}{100} * 2.3 \quad (3.17)$$

$$Sugar(g) = \frac{Sugar(PDV)}{100} * 50 \quad (3.18)$$

The conversion was based on daily recommended allowances for a 2000 kcal diet for fat, sodium & sugar by the Food & Drug Administration (FDA) [82]. As we noted in section 2.8.5, previous work had converted the traffic lights into categorical variables (green-0, amber-1, red-2) & added up the score. However, this method resulted in plenty of ties for health scores between recipes. This is because a lot of the recipes in the same range of the traffic lights received the same score. We decided to use the traffic lights but use a continuous (as opposed to discrete) value for the scores. To do this, we interpolate the score using the amount of the nutrient in grams. The calculation of the health score (HS), assuming  $x$  is the value of nutrient in grams is as follows:

$$HS_{Sodium}(x) = \begin{cases} \frac{x}{0.3} & \text{if } x \leq 0.3 \\ 1 + \frac{x}{1.5} & \text{if } 0.3 < x \leq 1.5 \\ 2 + \frac{x}{7.4} & \text{if } 1.5 < x \leq 7.4 \\ 3 & \text{if } x > 7.4 \end{cases} \quad (3.19)$$

$$HS_{Sugar}(x) = \begin{cases} \frac{x}{5} & \text{if } x \leq 5 \\ 1 + \frac{x}{22.5} & \text{if } 5 < x \leq 22.5 \\ 2 + \frac{x}{249.5} & \text{if } x > 22.5 \end{cases} \quad (3.20)$$

$$HS_{SaturatedFat}(x) = \begin{cases} \frac{x}{1.5} & \text{if } x \leq 1.5 \\ 1 + \frac{x}{5} & \text{if } 1.5 < x \leq 5 \\ 2 + \frac{x}{31.2} & \text{if } x > 31.2 \end{cases} \quad (3.21)$$

$$FSA_{Score} = HS_{SaturatedFat} + HS_{Sugar} + HS_{Sodium} \quad (3.22)$$

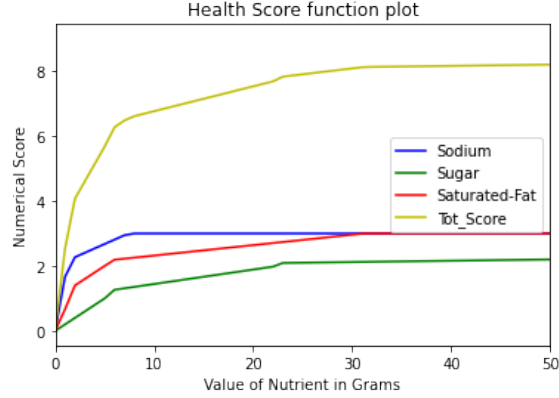


Figure 3.14: Health Score vs Value of nutrient (grams)

The ranges in equations 3.19, 3.21, 3.20 come from the **FSA traffic label cut off values** as described in section 2.8.5. However, we make some minor changes to adjust to the dataset. For sugar and saturated fat, the first range which corresponds to the green traffic light, is scaled by the value in the recipe - 0 at 0g & 1 at the maximum value of the green traffic light. Similarly, if a value is within the amber traffic light, we assign it a score between 1 & 2 scaled by the value of the nutrient in the dataset. For instance, considering sugar, the boundary point between the range for the amber traffic label lies between 5g & 22.5g. So if a value lies between those two points, we scale it by 22.5 to get a value between 0 & 1. Then we add 1 to that number to obtain a range between 1 & 2. Similarly, for the red traffic label, we scale the value by the maximum value of the nutrient in the dataset to obtain a range between 0 & 1 and add 2 to it. However, for sodium, the range of values in the dataset proved to have outliers as shown in the figure 3.15 below.

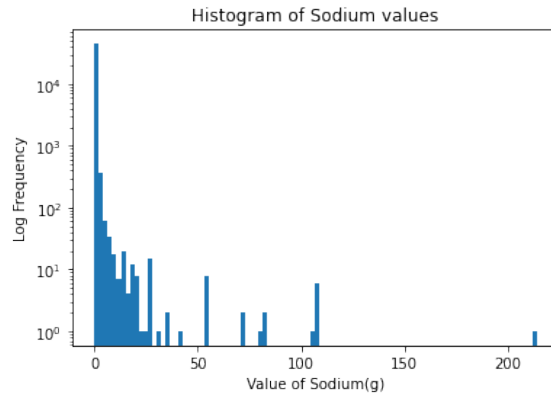


Figure 3.15: Log Frequency vs Sodium(grams)

Therefore, we did not scale by the maximum value for the red traffic light for sodium. Instead we took a value in the 90<sup>th</sup> percentile of the distribution of sodium in the dataset (7.4g) and values above that were automatically assigned a score of 3.



Figure 3.14 shows a graph of the health score vs the value of nutrients for sugars, sodium, saturated fat and the cumulative health score as the value of each nutrient increases.

## 3.5 Combining Rating & Health

As mentioned in section 2.8.5 various methods have been used to combine rating scores from a recommendation model and recipe health scores. First, we use the method from [67]. We recall that this is given by  $score(U, I) = RatingModel(U, I) \cdot (13 - FSA_{Score}(I))$ . This model wouldn't entail tuning separate weights for the health & rating and we would have a unified score without any optimization required. However, since in the  $FSA_{Score}$  we did not use fats, we modified this to:

$$score(U, I) = RatingModel(U, I) \cdot (8 - FSA_{Score}(I)) \quad (3.23)$$

We chose “8” instead of “13” since the maximum health score among all the recipes was 7.46.

### 3.5.1 New Approach

The main issue with adding the health score is that the recommendations may lose relevance (explained in section 2.1.2). Therefore, we add another constraint to the equation - **recipe similarity**. Recipe similarity would allow us to quantify how interchangeable two recipes are. We hypothesised that if a user likes a bunch of recipes, we can find similar recipes that are healthy & rated highly by them. In terms of stratification of recommender systems, this is akin to hybrid recommender systems. We have content information in the recipe from which we will derive similarities (similar to content-based recommendations from section 2.1.3). We also have domain-knowledge-based information from the health model and, finally, the rating model that will give us the probability of the user liking the recipe. By combining the three aspects - **the rating model, health score & similarities**, we believed we could achieve the goal of recommending recipes similar to the one a user ate, therefore relevant and healthy.

### 3.5.2 Recipe Embedding

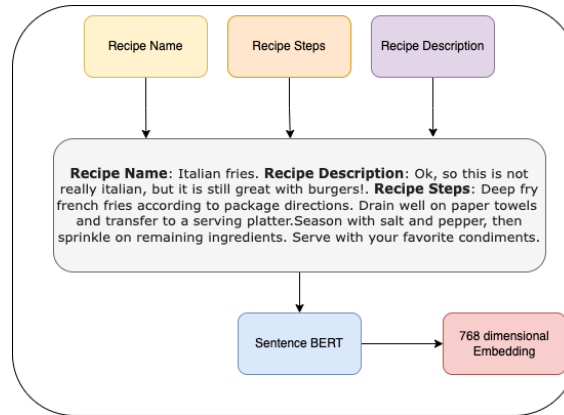


Figure 3.16: Illustration of recipe embedding for “Italian Fries”. The output is a 768 dimensional contextual embedding that captures various aspects of the recipe.

To find similar recipes, we use sentence BERT embeddings described in section 2.7. To get a contextual embedding for an entire recipe, we use the recipe name, description & the steps needed to complete the recipe. This way, we incorporate information about the ingredients & cooking methods of a recipe into one embedding. We concatenate the recipe name, description and steps, adding an identifier in front of each of them to obtain a large piece of text describing the entire recipe. We then pass the text through a pre-trained Sentence BERT model. This generates a 768-dimensional vector for each recipe in the dataset. We used “all-mpnet-base-v2” [83] from hugging face that was trained on over a billion sentence pairs for the task of sentence similarity. This method was inspired by [84] who used pre-trained BERT embeddings to generate recipe embeddings for recipe search. They showed that BERT embeddings were superior to word matching for search, especially regarding diversity of results. Figure 3.16 illustrates the process of generating the embedding for the recipe “Italian Fries”.

### 3.5.3 Recipe Similarity

To quantify the similarity between two recipe embeddings  $R_1$  &  $R_2$  we use the cosine similarity as follows:

$$Similarity(R_1, R_2) = \frac{R_1 \cdot R_2}{\|R_1\| \|R_2\|} \quad (3.24)$$

However, to store similarities for all the recipes, we would need a matrix of over  $45276 \times 45276$  ( $N_{recipes} \times N_{recipes}$ ). We would not be able to store this in memory for a vectorized operation. So instead, we looped over each recipe and calculated the similarity score against all of the recipes in the dataset & proceeded to store the top 150 in a dictionary. The point of storing only 150 was a choice so that we don’t stray too far from the original recipe while providing our recommendations.

### 3.5.4 Embedding Visualization

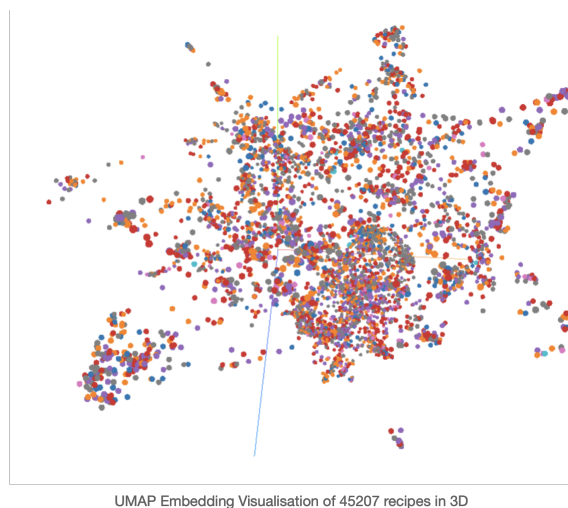


Figure 3.17: UMAP Projection of recipe embeddings in 3D. Each point in the figure represents a recipe. The colours indicate the rounded health score.

However, using similarities to generate recommendations would only work if there are sufficient “healthier” recipes in the database that are reasonably similar to the original recipe the user liked. To check whether this is the case, we used Uniform manifold approximation and projection (UMAP) for dimensionality reduction [85] to project the 768 dimensional embedding to 3 dimensions to visualize the recipes in that space. We then marked each recipe by a rounded version of the health score. This is illustrated in figure 3.17.

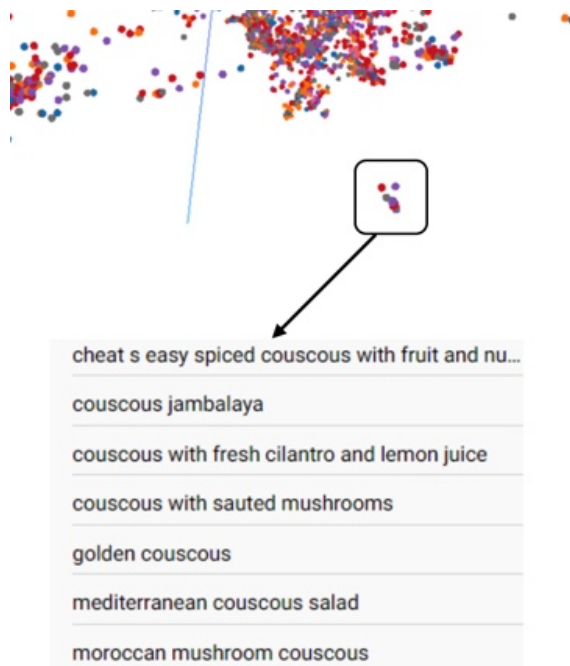


Figure 3.18: A cluster of recipes from the 3D space illustrated in the bounded-box. We see that similar recipes cluster together from the annotations.

From figure 3.17 we observe that there are several clusters of recipes that are close together in the 3D space. One such cluster is observed in figure 3.18. We see that they predominantly represent couscous recipes. Additionally, we see that several colours or several variations of recipes with different health scores are available in the cluster. So, it is reasonable to assume that we could find similar and healthier versions of recipes in this dataset. To further our claim, we provide a few more examples of the original recipes along with similar ones in the next section.

### 3.5.5 Similar recipes

To motivate why we chose sentence BERT based embeddings we provide a few examples of similar recipes with their health scores in table 3.1.

Table 3.1 contains the original recipe and the top 5 closest recipes with their health scores sorted by the similarity score as described in section 3.5.3. Firstly, we observe that we get five similar recipes to the original recipe of varying healthiness levels. Then, we notice some advantages of using a pre-trained SBERT model to construct the recipe embeddings. For instance, one of

Original Recipe Name	Similar Recipes	Similarity Score	Health Score
“Mennonite” corn fritters	Old fashioned corn fritters	0.849	0.443
	“Marmy’s” corn fritters	0.780	2.508
	Easy corn fritters	0.792	4.131
	Corn fritters and maple syrup	0.762	1.626
	Peppered corn fritters	0.760	1.260
“Blond” Minestrone	Italian Vegetable Soup	0.818	3.455
	Minestrone soup without pasta	0.812	2.802
	Tomato and Spinach soup	0.802	5.854
	Easy Creamy Tomato Soup	0.800	3.934
	Tomato and Pasta Soup	0.794	4.040
“Sin-free” Fries	Healthy Oven Fries	0.770	0.876
	Fat-free skillet home fries	0.764	1.948
	Tasty home fries	0.747	3.012
	Crispy sweet potato fries “weight-watchers”	0.746	2.932
	incredible french fries	0.745	2.667
Costa Vida green rice	Chipotle copycat lime rice recipe Fries	0.846	1.253
	Restaurant style Mexican rice	0.837	1.836
	Restaurant Spanish rice	0.833	2.616
	“Weight-watchers” core Mexican rice	0.824	2.634
	Home style Mexican rice	0.822	0.966

Table 3.1: Examples of the original recipe, corresponding similar recipes with health scores and cosine similarities. Similar recipes are sorted by cosine similarity.

the similar recipes for “Blond” Minestrone, an Italian dish, is “Italian Vegetable soup”. The Minestrone is a dish that usually consists of pasta, rice, and vegetables. We see this captured in the similar recipes “Tomato & Spinach soup” and “Tomato & Pasta soup”. Therefore, information about the ingredients incorporated in the recipe steps is reflected in the similarity metric. This is especially useful if we consider the example where the original recipe is “Sin-free” Fries. Ideally, we would have similar recipes that are healthier than regular fries (since they are “sin-free”). Since we are using pre-trained contextual embeddings from SBERT, similar recipes are automatically healthier versions of fries. And we can see that “Healthy Oven Fries” is the most similar recipe, with “Fat-free skillet home fries” coming in second. The pre-trained embeddings therefore, capture that “sin-free” corresponds to healthy. Finally, we have “Costa-Vida green rice” as the original recipe. Costa-Vida is a Mexican restaurant chain, and we find that the most similar recipe is a Chipotle (another Mexican restaurant chain) based recipe. The second and third most similar recipes are also restaurant-based recipes. Thus, pre-trained contextual embeddings allow us to capture similarities within the recipes that wouldn’t otherwise be possible without a lot of feature engineering.

### 3.6 Combining Similarity, Rating & Health

We now turn to the problem of optimizing the predicting rating, health score and similarity scores together. The means, standard deviations and ranges of all 3 variables are different: the health scores are between 0 and 8, while cosine similarities and ratings could range between 0 and

1. However, they could have completely different distributions. So instead of using the values directly, we create an **ordered ranking** for each of them and combine the three. The process is illustrated in figure 3.19 and described further in this section.

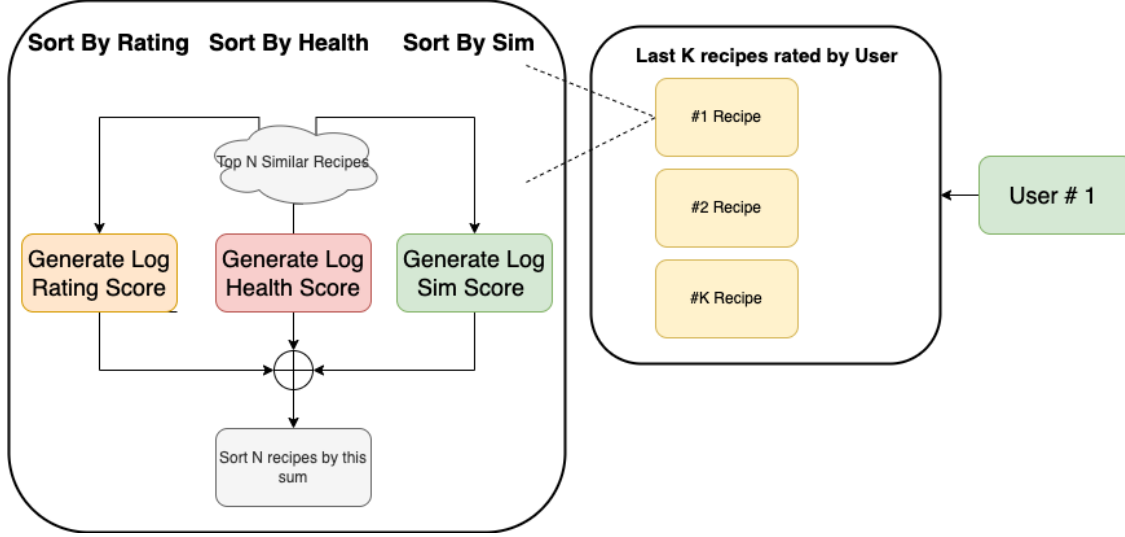


Figure 3.19: Combining Rating, Health & Similarity. From right to left: The process for generating recommendations for user #1 is detailed in the figure.

As described in section 2.1.3, for content-based recommendations, if a user likes a particular item, they might like similar items. So for any user, we have the last  $K$  recipes (by date that they rated) that they have rated. From the right, in figure 3.19, we demonstrate an example of the method. We have **user #1** and the **last  $K$  recipes that they rated**. From here we take the **top  $N$  similar recipes** for **each of the  $K$  recipes**. This is done using the pre-computed recipe-to-recipe similarity dictionary described in section 3.5.3. In the figure, we take the top  $N$  similar recipes of recipe #1 amongst the  $K$  recipes. The **predicted rating** for the user-recipe combinations of all  $N$  recipes is calculated using the trained rating model. To avoid users being recommended the recipes that they have already rated, the ratings for all the recipes that they have previously interacted with are set to -100. So, they do not show up in the recommendations since the rating component would be too low. We also have the **health scores** for all the  $N$  recipes from the health model. We sort the  $N$  recipes using similarities, health scores & rating predictions separately. The rating and the similarities score are sorted in descending order (since higher is better). In contrast, the health score is sorted in ascending order (since lower health values are better). This gives us **3 different lists** and hence **3 distinct rankings** of the same recipes sorted differently. To obtain scores using the ranking of the three lists, we use a part of the standard metric used in information retrieval - Discounted Cumulative Gain (DCG) [86]. In the DCG metric, there is a logarithm reduction with increasing rank. Thus, if we have a sequence sorted with ranks from 1 to  $N$ , the “Log score” of an item with rank  $i$  would be as follows:

$$LogScore(i) = \frac{1}{log(i+1)} \quad (3.25)$$

Therefore, if we have a recipe with ranks  $i, j, k$  for similarity, health & rating respectively, the final score for that recipe would be:

$$FinalScore_{recipe} = \frac{1}{log(i+1)} + \frac{1}{log(j+1)} + \frac{1}{log(k+1)} \quad (3.26)$$

For each of the N recipes, since we have a rank for the health, similarity & rating we have a “Final Score”. Finally, we choose the recipe with the highest final score to recommend to the user. This repeats of each of the K recipes that the user liked.

## Chapter 4

# Results & Analysis

This chapter focuses on the results for the methods described in the previous chapter.

### 4.1 Rating model

#### 4.1.1 Evaluation metric

When describing the goals of a recommender system in section 2.1.2, we mentioned “accuracy” in terms of a quantifiable metric as one of the key components. Therefore, we evaluated all the models with a common metric that can help us compute how accurate the models’ predictions are on unseen data. For the rating models, the metric used is the area under the curve (AUC) of the receiver operating characteristic curve (ROC). The ROC is a graph that shows us the performance of a classification model at various thresholds. Recalling from section 3.3 for our binary classification model, the output comes from a sigmoid function that converts a raw score into a probability value between 0 & 1. However our labels are binary  $\in (0, 1)$  (0 for “dislikes recipe” and 1 for “likes recipe”). Therefore we must choose a “threshold” that converts the probability into a binary label. This is often chosen as 0.5, where probabilities  $>0.5$  are kept as 1, else 0. The ROC curve is made up of two parts: the true positivity rate (TPR) & the false positivity rate (FPR).

$$TPR = \frac{TP}{TP + FN} \quad (4.1)$$

$$FPR = \frac{FP}{TN + FP} \quad (4.2)$$

TP,FP,TN,FN are defined as true positives, false positives, true negatives and false negatives. The definitions of each can be derived from the matrix in figure 4.1. This matrix is known as a “Confusion Matrix”. For instance, when the label is positive (1) & the predicted value is positive (1) we call them true positives. Whereas if the label is negative (0) and the predicted value is positive (1) we refer to them as false positives.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Figure 4.1: Confusion matrix

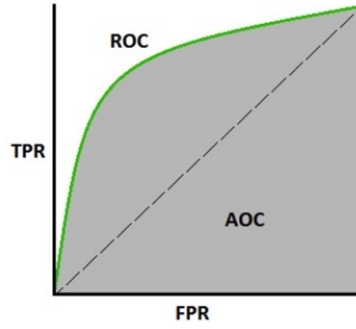


Figure 4.2: ROC curve with the shaded region being the AUC. Image from [87].

The ROC curve is shown in figure 4.2. The FPR is on the x-axis while the TPR is on the y-axis. The graph is plotted across various thresholds of converting the probabilities to binary values. The AUC is the area under the curve for this graph. AUC gives us the probability that a random positive sample is assigned higher probability than a random negative sample. An AUC of 0.5 represents the straight line in the ROC curve, which means the model distinguishes both the classes at random. A higher AUC represents a better classifier and an AUC of 1 means that the classifier is always correct.

### 4.1.2 Experiments

As mentioned in section 3.2, we have 3 sets of train, test & validation splits. Thus each model is trained 3 times and the model with best validation loss is saved. The model configuration and choices of hyper-parameters were mentioned in the description of the individual models in the previous chapter. The hyper-parameter search space is provided in Appendix A. The AUC reported in the following table represents the test-set AUC for each model across the 3 sets. We also show the time taken (in seconds) per epoch for each model during training. This pertains to the scalability goal of recommender systems discussed in section 2.1.2.

From table 4.1 we can make a few observations:

1. The baseline matrix factorization model (MF) performs the worst in terms of AUC ROC with an average of 0.6376. Therefore, the assumption that the interactions can be modelled only using an inner product from section 3.3.1 does not work out in this case. However,



Model Name	Time	Set-1	Set-2	Set-3	Average	S.D
MF	219.3	0.6378	0.6388	0.6361	0.6376	0.0014
MF with bias	220.1	0.6604	0.6581	0.7006	0.6730	0.0239
Content boosted MF	90.20	0.6669	0.6523	0.6524	0.6572	0.0084
NCF	195.2	0.7382	0.7370	0.7402	0.7385	0.0016
Two-tower	206.4	0.7310	0.7451	0.7443	0.7402	0.0079
Auto Int	200.1	0.7526	0.7407	0.7460	0.7464	0.0060
Our Model	222.3	<b>0.7580</b>	<b>0.7469</b>	<b>0.7482</b>	<b>0.7510</b>	0.0061

Table 4.1: AUC scores for 3 sets of test sets with the average and the standard deviation (S.D) for each model

adding content information & bias helps the classifier to improve (AUC of 0.6730 with bias and 0.6572 with content information). We note that the baseline MF model & the MF model with bias has a total number of parameters of  $\approx 12.5\text{M}$  (million), largely owing to size of the recipe embeddings that are of size (45276 x 256). The size of the embedding was optimal according to our validation set loss, so we did not reduce it. The content boosted model that does not have recipe embeddings has  $\approx 3.1\text{M}$  parameters. Even with such a reduction in parameters, and thus representation capability (as discussed in section 2.2.1) the content-boosted model performs better than the baseline MF model.

2. One of the sub-goals (from section 2.8.4) in the thesis was to benchmark the performance of common neural network baselines for recipe recommendation. We note that the neural network-based models outperform the matrix factorization models. There is an increase in the AUC between NCF and the two-tower model (Siamese model). Both the MLP-based model had  $\approx 12.6\text{M}$  parameters and were thus comparable. This prompted us to have the two-tower design while designing our model architecture.
3. Finally, we can see that the multi-headed attention-based models perform better than regular MLP-based models (Two-tower & NCF). **Our model performs the best out of all the models across all the three test sets with an average AUC ROC of 0.7510.** The number of parameters in AutoInt were  $\approx 4.5\text{M}$  whereas in our model we had  $\approx 3.70\text{M}$  parameters. Thus, the performance increase was not based on an increase in model complexity.
4. Concerning the time taken per epoch for the models, we notice for the most part, all of the models require  $\approx 200$  seconds per epoch. Only the content-boosted MF model has a substantial reduction in time taken (90.2 seconds). The difference between the original MF model & the content-based MF model is that we do not use recipe embeddings in the latter. The time delta could be because the size of the embedding matrix for the recipes ( $\approx 45000$  recipes x 256) is much larger than that of the ingredients ( $\approx 8000$  ingredients x 256). Thus, there might be a bottleneck in training for storing or lookup.

### 4.1.3 Analysis of our model

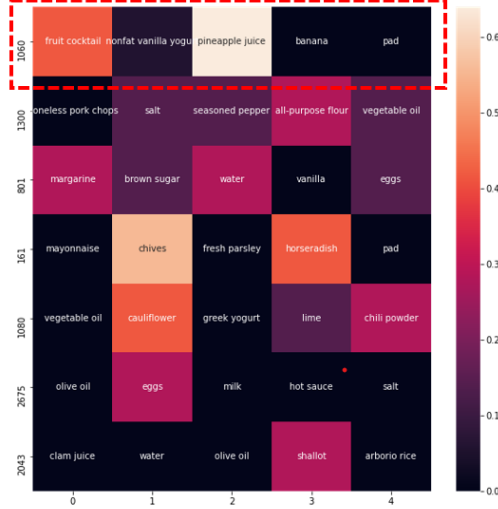


Figure 4.3: In this figure we have a heatmap of attention weights with the users on the y-axis & the ingredients of a recipe from the test set on the x-axis. From the highlighted recipe, we notice that the attention weights are providing importance to single ingredient.

This section looks at possible reasons for improved performance for our model and test research goal- 2 from section 1.2. The advantage of our model over other MLP based models others is that the analysis is only not limited to the quantitative metric. As shown in section 3.3.6, for AutoInt, the authors provide an interpretable view of the results for the multi-headed attention-based model. We recall from section 3.3.7 that in our model, we had  $\alpha_{u,ing_i}^h$  that represents an attention weight for a user  $u$  and an ingredient  $ing_i$  for one attention head  $h$ . Now averaging over all the heads gives us one single attention weight for each ingredient in a recipe for a user. It is using these weights that the recipe vector is constructed. We visualize these weights in a manner similar to the method used by [79] to see if we can find some interpretability within the model. To get the attention weights, we take a trained model checkpoint and conduct a forward pass on some examples from the test-set.

The figure 4.3 below is one of the examples of visualized attention weights for a batch of users and corresponding recipes from the test-set. On the y-axis, we have the unique user-id and on the x-axis, we have a list of ingredients for a particular recipe. For instance, in the first row, we have user #1060, and the recipe contains ingredients like “fruit-cocktail”, “non-fat vanilla yogurt”, “banana”. We can also see the appended “pad” tokens in the recipe that were added to keep length of ingredients in all the recipes the same. We already assigned 0 weight to that before training, and we observe that all pad tokens in the heatmap have 0 weight.



Figure 4.4: Two recipes having similar attention weights for different users.

One pattern observed was the model paying attention to one or two ingredients heavily in a recipe. For instance, for user #1060 the attention weight for “pineapple juice” in figure 4.3 is much higher than other ingredients. A possible explanation could come from looking at the recipes that user #1060 rated. We found that pineapple juice was reasonably common in the recipes that the user interacted with.

From the highlighted box in figure 4.4 above, we see that for some recipes, the model learns equal weights for the ingredients irrespective of the user. This could suggest if the users are similar enough, the model learns similar representations for recipes.

From figure 4.4 & figure 4.3 we also notice that a lot of weights for individual ingredients for a recipe is  $\approx 0$ . This means only a few ingredients contribute to the final representation of the recipe. Essentially, features that are not helpful to the task are automatically removed.

#### 4.1.4 Ablation

We conduct ablation studies on the attention-based models to understand how much content information helps for this task and answer *research goal - 1* from section 1.2. To do this, we test the attention-based models with and without a unique recipe embedding.

For the AutoInt model from section 3.3.6 our input was a concatenation of all the features. In the results from section 4.1.2 we had only used ingredient embeddings and user embeddings as features  $([e_u|ing_1|ing_2|ing_3..ing_n])$ . Now we add the recipe embedding  $e_i$  with it for the ablation study as  $[e_u|e_i|ing_1|ing_2|ing_3..ing_n]$ .

Our model (section 3.3.7) required an architectural modification to accommodate the recipe embedding. We add another user-recipe multi-headed attention module that takes in the user embedding as the query & the recipe embedding as the key & and the value. The output of the

Model Name	Time	Set-1	Set-2	Set-3	Average	S.D
Auto Int	200.1	0.7526	0.7407	0.7460	<b>0.7464</b>	0.0060
Auto Int with recipe	288.1	0.7460	0.7363	0.7421	0.7395	0.0029
Our Model	222.3	0.7580	0.7469	0.7482	<b>0.7510</b>	0.0061
Our Model with recipe	371.4	0.7386	0.7583	0.7397	0.7455	0.011

Table 4.2: AUC scores for 3 sets of test sets with the average and the standard deviation (S.D) for AutoInt & Our model with and without recipe embeddings

user-recipe attention module is passed through another MLP similar to the ones used for the user & the output of the user-ingredient attention module.

From table 4.2 we can see that both the AutoInt & Our model with recipe embeddings become worse in terms of mean AUC. This suggests that content information in the form of ingredients might be more important for our task compared to unique recipe-based embeddings (where each recipe is represented by a trainable embedding  $e_i$ ). We also observe the increase in time per epoch once we add in the recipe embeddings. The same increase was observed in section 4.1.2 for the MF model that included recipe embeddings versus content-boosted MF.

#### 4.1.5 Failure Analysis of our model

To see where our model fails we analyze one of the test set predictions. First we make a confusion matrix (figure 4.1). To do this we must convert the probabilities obtained from the model into binary ratings. To choose the threshold, we optimize based on the macro F1 score.

Firstly, precision of any class  $i$  is given by,

$$Precision_i = \frac{TP_i}{TP_i + FP_i} \quad (4.3)$$

The recall for any class  $i$  is given as,

$$Recall_i = \frac{TP_i}{TP_i + FN_i} \quad (4.4)$$

The F1 score of any class  $i$  is given by the harmonic mean of precision & recall of class  $i$

$$F1_i = \frac{2 * (Precision_i) * (Recall_i)}{Precision_i + Recall_i} \quad (4.5)$$

The macro F1 score is the unweighted average of the F1 score of all the classes (in our case, 2 of them). Therefore to choose the best threshold to optimize the score, we iterate through various thresholds and select the one that gives us the best macro F1 score.

After obtaining the best threshold (0.794) we obtain the following confusion matrix for predictions from one of the test sets.

		Predicted class		Total
		Positive (1)	Negative (0)	
True class	Positive (1)	37014	2367	39381
	Negative (0)	1854	724	2578
Total		38868	3091	41959

From the confusion matrix above, we observe high precision (0.939) & recall (0.952) for the positive class (1). However, the metrics for the negative class (0) need improvement. To calculate the precision and recall for the negative class, we use equation 4.3 and 4.4. However, the meaning of TP changes to label (0) and predicted value (0). The same can be inferred for TN and FP. We find that the precision (0.280) & recall (0.234) are quite low for the negative class. We saw in section 3.1.3 that class 1 dominates the dataset, and as a result, most of the training examples have been positive examples. Therefore, the problem could be addressed with more effective methods of handling class imbalance.

As discussed in section 2.1 recommender systems can often suffer from **“item” cold starts**. This is when the item does not have interactions to merit a correct prediction from the model. To analyze the performance of our model on rare items (with few interactions), we take all the failed test set predictions and see how many of these are present in the train set. We take all the recipes where the model failed in its predictions and find how often these recipes occur in the training set. If the model fails to predict the rating for a particular recipe, we want to see if the model has seen the recipe in training. The more times the recipe has occurred in the training set, the more likely that the test set prediction for the recipe would be correct.

From the ablation study, we know recipe embeddings are detrimental to the model’s performance. So we plot the failure’s for our model with and without recipe embeddings.

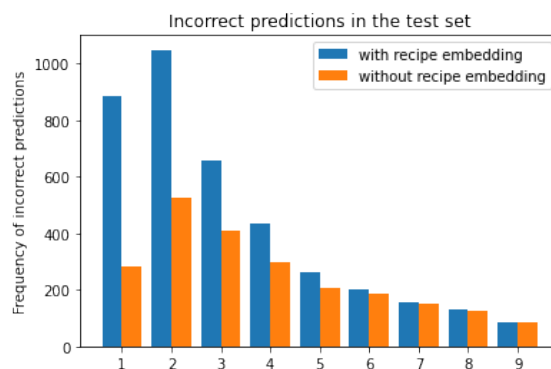


Figure 4.5: No of recipes where the model failed to predict interactions vs the frequency of the recipe in the train set.

On the x-axis of the plot above we have the number of occurrence in the training set for all the failed recipes. We can see that most of the failures occur when the recipe has only occurred once or twice in the training set. We notice that when we add the recipe embeddings, the failures increase substantially for rare recipes. This shows us that content information alone is in all probability, not only more important for this task - but it could also alleviate item cold start problems. This is also logical since ingredients are what constitutes a recipe. New recipes might have ingredients that the model has already learned a good representation for. So when there is a new recipe, using the ingredients to construct the recipe embedding is more helpful than initializing a separate embedding for the recipe.

## 4.2 Recommending Healthy Recipes

We compare the two methods of recommending healthy recipes described in section 3.5 quantitatively and qualitatively. The first method used the health score & the rating model output, while our approach we combine the health, rating & a similarity score. To compare the two models quantitatively, we use the health score & rating of the recommended recipes. However, since we added an additional numerical constraint - “similarity” in one of the models, we added qualitative evaluation to choose between the models. We recall the model from the literature [67] was based on the formula  $score(U, I) = RatingModel(U, I) \cdot (8 - FSA_{Score}(I))$ . We plot a contour plot to visualize the score function across a possible range of health and rating values. From the contour plot in figure 4.6 we see the trade-off between the health score and the rating score. For instance, a rating of  $\approx 0.6$  and health score of  $\approx 0$  has the same overall score as a rating of  $\approx 0.999$  and health score  $\approx 3.2$ .

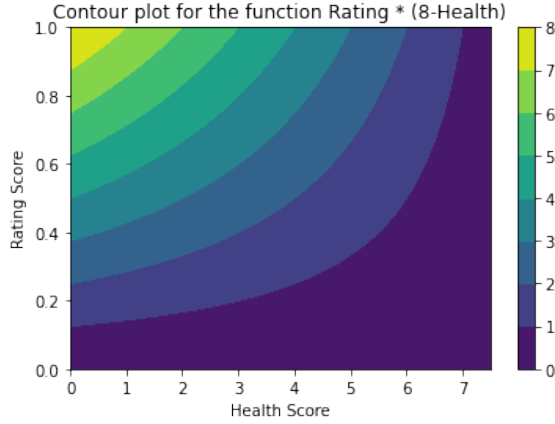


Figure 4.6: (Health + Rating) score function visualization.

### 4.2.1 Quantitative Results

Model	Average Health Value	Average Rating Value
Health and Rating	0.1061	0.9922
Heath, Rating and Similarity	2.7461	0.9691

Table 4.3: Quantatative evaluation for both methods for recommending healthy recipes

To compare both the methods quantitatively we use the mean rating value & health scores across predictions for a sample of users. The results are shown in table 4.3.

We notice that the average health score is lower & rating value is higher for the model from literature (that only uses the health score and the rating). However, upon inspection of the recommendations we find that the model from literature recommends recipes that might not be relevant to the user. As some recipes are popular, they will be rated highly by the model for a lot of the users. This leads to a set of popular & healthy recipes being recommended to them.

However, this is not the case for the model where we add similarity. To validate this, we conduct a user study to test the relevance of the recommendations in the next section.

#### 4.2.2 User Study & Qualitative Evaluation

Model with similarity, health & rating	Model with health & rating
0.694	0.340

Table 4.4: Percentage of recipes selected by N=12 users from recommendations.

User-Rated Recipes
Peachy Spinach Salad
Pumpkin Pie With Tofu
Thai Coconut Rice
Lemon Garlic Spinach
Brussels Sprouts in Garlic Butter
Strawberry Pavlova
Sweet Curry Chickpea Casserole
Super Pumpkin Pie Smoothie

Table 4.5: Recipes that user #3632 rated highly previously.

We mentioned relevance as one of the key components in the goals of a recommender system in section 2.1.2. To this end, to make sure the recommended items from our similarity based model is more relevant we conduct a user study where we ask users to select relevant recipes. We provided 12 people (test-users) with a form based on one of the users in the dataset. The form contained a particular user interaction history from the dataset.

We also provided them with recommended recipes from the model with similarity, health, rating & the ones with only rating & health scores. Then we asked them to tick off which recipes they find relevant.

An abridged example of one of the test-users submissions are shown in tables 4.6. Table 4.5 contains the recipes that user #3632 in the dataset had liked previously. The test-user was asked to use table 4.5 to understand this particular users taste in recipes. Then using this information, they ticked the recipes in table 4.5 that they thought that user #3632 will find relevant.

Model with Similarity,health & Rating	Model with health & Rating
Strawberry Spinach Salad ✓	Crab Cucumber ✓
Pumpkin pie shake ✓	Wondra Instant Noodles ✗
Ginger Coconut Rice ✓	Mexican sunset drink ✗
Lemony Spinach ✓	Lemon Tarragon Base ✓
Roasted Brussel Sprouts✓	Creamy Roast Veggie Pasta ✓
Pavlovas with Lemon Curd and Tropical Fruit ✓	Moroccan spicy mix ✗
Slow Cooked Madras Chicken Curry Crock Pot ✗	Lamb Chops Au Poivre ✗
Pumpkin shake ✓	Lavender, Lemon and Honey Tea ✗
Total: 7/8	Total: 3/8

Table 4.6: Recommendations from both of the models.

We see that the model with similarity in table 4.6 generates a similar set of recipes to the original recipes that the user rated in table 4.5. It is easy to understand that a user would prefer healthier alternatives to recipes that they have already tried out before, instead of a completely new recipes (that is outputted by the model from literature).

Thus, we expected the model with similarity to perform better on this test of relevance. We see in table 4.5 that the test-user chose 7/8 recipes from the similarity based model while only 3/8 from the model with only health and rating.

Table 4.4 shows the percentage of recipes cumulatively picked by the 12 test-users out of all the recipes recommended to them for each model. The model with similarity has a much higher fraction of recipes (0.694) that a user finds relevant compared to the model with only health and rating (0.350). Although the sample size is a limitation, we get an idea that using similarity, health & rating could help us recommend relevant & healthier recipes to users.

### 4.3 Limitations

The rating model still suffers from bias towards the majority class. The limitation for the method for recommending healthier recipes is that of novelty (as discussed in section 2.1.2). Since we are using similarity as one of the factors in our recommendation, this might albeit generate relevant recipes, but there is a chance of the user getting tired of the same thing in different variations over and over again. Also, due to the addition of similarity as a constraint, our approach is not at par with the model from literature in terms of average rating & health value of the recommendations. Additionally, since we are using the last K interactions from the user to compute the similarity, the information won't be available for new users. Some methods to address this problem is addressed in the next chapter under future work.



## Chapter 5

# Conclusion & Future Work

### 5.1 Summary

In this thesis, a novel architecture using multi-headed attention for recipe recommendation was developed that had a higher average AUC than standard neural recommendation baselines and another attention-based model from the literature. We analyze the patterns within the attention weights to provide interpretability to our model. We find that ingredient information benefits recipe recommendations in predicting rarer recipes' ratings. We observe that adding unique recipe embeddings makes the models less accurate and slower. Instead, constructing the recipe embeddings from the ingredients is more convenient and accurate. Additionally, a new method for recommending healthy recipes was developed, combining rating from a recommender system model, a pre-determined health score & a similarity score. We presented the advantages of using a pre-trained BERT model for computing the similarities between recipes. The health score that was calculated using interpolation between the traffic light scores gave each recipe a unique health score. This was lacking in previous work, where many recipes received the same categorical value. Finally, we qualitatively & quantitatively compare our method with existing ones. Quantitatively the model from the literature has a higher average rating and lower health values across the predictions. However, we find that recommendations from the model with similarity could be more relevant based on a user study. Although our method results in recipes that might be relevant to the user, we find that the recommendations could lack novelty.

### 5.2 Future Work

Regarding the rating model, the drawback was the bias towards higher ratings. Since our data was inherently unbalanced, we used a weighted loss function. But the performance could be improved by using sampling techniques such as SMOTE [88]. Furthermore, an ensemble approach [89] to combining multiple models could also be used to improve all the metrics further. Furthermore, our dataset did not have any features about the users. Therefore, user embeddings were initialized at random. With more information about the users, we hypothesize our model will perform better due to explicit modelling of the similarity between the user (the query) and the content information (the value).

For recommending healthy recipes, various approaches could be tried to mitigate the risk of losing novelty amongst the recommendations. Firstly, it could be possible to combine the recommendations from the rating model itself & the ones from the similarity model. For instance, we could have a portion of the recommendations only based on predicted rating instead of serving all of the recommendations based on a healthier version of recipes (mixed-hybrids from section 2.1.3). This would allow novel recommendations which then the user can then choose to interact with. Secondly, another option is to have a 2nd order similarity where we look at another level of similarities (find similar recipes to the ones generated in the first case). Finally, for newer users who don't have past recipe interactions to base the similarity on, we can simply use the rating model's outputs to generate the recommendations.

## Appendix A

# Appendix

We first tuned the learning rate for each of the models, followed by the other hyper-parameters. After tuning the learning rate, we looked at the validation loss and set the rest of the hyper-parameters for each model manually, using the search space listed below.

1. Learning Rate: 0.1, 0.01, 0.001, 0.0001, 0.00001
2. Batch Size: 6, 12
3. Embedding Size (Ingredients/Recipe/User): 32, 128, 256, 512
4. Multi-headed Attention model dimension (Auto Int & Our model) 4, 8

# Bibliography

- [1] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, et al. The youtube video recommendation system. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 293–296, 2010.
- [2] Kurt Jacobson, Vidhya Murali, Edward Newett, Brian Whitman, and Romain Yon. Music personalization at spotify. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 373–373, 2016.
- [3] George A Bray. Medical consequences of obesity. *The Journal of clinical endocrinology & metabolism*, 89(6):2583–2589, 2004.
- [4] Obseity. <https://www.who.int/news-room/fact-sheets/detail/obesity-and-overweight>.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [6] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.
- [7] Bradley N Miller, Joseph A Konstan, and John Riedl. Pockettlens: Toward a personal recommender system. *ACM Transactions on Information Systems (TOIS)*, 22(3):437–476, 2004.
- [8] Charu C Aggarwal et al. *Recommender systems*, volume 1. Springer, 2016.
- [9] Amazon. <https://bantrr.com/2013/12/03/marketing-automation-software-should-offer-personalized-re>
- [10] Sean M McNee, John Riedl, and Joseph A Konstan. Being accurate is not enough: how accuracy metrics have hurt recommender systems. In *CHI’06 extended abstracts on Human factors in computing systems*, pages 1097–1101, 2006.
- [11] Joseph A Konstan, Sean M McNee, Cai-Nicolas Ziegler, Roberto Torres, Nishikant Kapoor, and John Riedl. Lessons on applying automated recommender systems to information-seeking tasks. In *AAAI*, volume 6, pages 1630–1633, 2006.
- [12] John S Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. *arXiv preprint arXiv:1301.7363*, 2013.

- [13] Buhwan Jeong, Jaewook Lee, and Hyunbo Cho. Improving memory-based collaborative filtering via similarity updating and prediction modulation. *Information Sciences*, 180(5):602–612, 2010.
- [14] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009, 2009.
- [15] Xuan Nhat Lam, Thuc Vu, Trong Duc Le, and Anh Duc Duong. Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 208–211, 2008.
- [16] Robin Burke. Knowledge-based recommender systems. *Encyclopedia of library and information systems*, 69(Supplement 32):175–186, 2000.
- [17] Alexander Felfernig and Robin Burke. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*, pages 1–10, 2008.
- [18] Robin Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.
- [19] Erion Çano and Maurizio Morisio. Hybrid recommender systems: A systematic literature review. *Intelligent Data Analysis*, 21(6):1487–1524, 2017.
- [20] Mark Claypool, Anuja Gokhale, Tim Miranda, Paul Murnikov, Dmitry Netes, and Matthew Sartin. Combing content-based and collaborative filters in an online newspaper. In *Proc. of Workshop on Recommender Systems Implementation and Evaluation*, 1999.
- [21] Michael J Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial intelligence review*, 13(5):393–408, 1999.
- [22] Justin Basilico and Thomas Hofmann. Unifying collaborative and content-based filtering. In *Proceedings of the twenty-first international conference on Machine learning*, page 9, 2004.
- [23] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.
- [24] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara Sainath. Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219, 2019.
- [25] Dong Liu, Yue Li, Jianping Lin, Houqiang Li, and Feng Wu. Deep learning-based video coding: A review and a case study. *ACM Computing Surveys (CSUR)*, 53(1):1–35, 2020.
- [26] Amirsina Torfi, Rouzbeh A Shirvani, Yaser Keneshloo, Nader Tavaf, and Edward A Fox. Natural language processing advancements by deep learning: A survey. *arXiv preprint arXiv:2003.01200*, 2020.

- [27] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.
- [28] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.
- [29] ANN. [https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored\\_neural\\_network.svg/250px-Colored\\_neural\\_network.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored_neural_network.svg/250px-Colored_neural_network.svg.png).
- [30] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [31] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.
- [32] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*, 2018.
- [33] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.
- [34] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.
- [35] Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.
- [36] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.
- [37] Hong Hui Tan and King Hann Lim. Vanishing gradient mitigation with deep learning neural network optimization. In *2019 7th international conference on smart computing & communications (ICSCC)*, pages 1–4. IEEE, 2019.
- [38] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [41] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [42] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

- [43] RNN. <https://medium.com/syncedreview/a-brief-overview-of-attention-mechanism-13c578ba9129>.
- [44] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- [45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [46] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.
- [47] Peter Forbes and Mu Zhu. Content-boosted matrix factorization for recommender systems: experiments with recipe recommendation. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 261–264, 2011.
- [48] Jill Freyne and Shlomo Berkovsky. Intelligent food planning: personalized recipe recommendation. In *Proceedings of the 15th international conference on Intelligent user interfaces*, pages 321–324, 2010.
- [49] Yijun Tian, Chuxu Zhang, Ronald Metoyer, and Nitesh V Chawla. Recipe recommendation with hierarchical graph attention network. *Frontiers in big Data*, 4:778417, 2022.
- [50] Mayumi Ueda, Mari Takahata, and Shinsuke Nakajima. User’s food preference extraction for personalized cooking recipe recommendation. In *Workshop of ISWC*, pages 98–105, 2011.
- [51] Christoph Trattner and David Elsweiler. Food recommender systems: important contributions, challenges and future research directions. *arXiv preprint arXiv:1711.02760*, 2017.
- [52] Morgan Harvey, Bernd Ludwig, and David Elsweiler. You are what you eat: Learning user tastes for rating prediction. In *International symposium on string processing and information retrieval*, pages 153–164. Springer, 2013.
- [53] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.
- [54] Hamed Jelodar, Yongli Wang, Chi Yuan, Xia Feng, Xiahui Jiang, Yanchao Li, and Liang Zhao. Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey. *Multimedia Tools and Applications*, 78(11):15169–15211, 2019.
- [55] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE international conference on data mining*, pages 263–272. Ieee, 2008.
- [56] Chun-Yuen Teng, Yu-Ru Lin, and Lada A Adamic. Recipe recommendation using ingredient networks. In *Proceedings of the 4th annual ACM web science conference*, pages 298–307, 2012.
- [57] Milica Milosavljevic, Vidhya Navalpakkam, Christof Koch, and Antonio Rangel. Relative visual saliency differences induce sizable bias in consumer choice. *Journal of Consumer Psychology*, 22(1):67–74, 2012.

- [58] David Elswailer, Christoph Trattner, and Morgan Harvey. Exploiting food choice biases for healthier recipe recommendation. In *Proceedings of the 40th international acm sigir conference on research and development in information retrieval*, pages 575–584, 2017.
- [59] Mehdi Elahi, Mouzhi Ge, Francesco Ricci, Ignacio Fernández-Tobías, Shlomo Berkovsky, Massimo David, et al. Interaction design in a mobile food recommender system. In *CEUR Workshop Proceedings*. CEUR-WS, 2015.
- [60] Md Kishor Morol, Md Shafaat Jamil Rokon, Ishra Binte Hasan, AM Saif, Rafid Hussain Khan, and Shuvra Smaran Das. Food recipe recommendation based on ingredients detection using deep learning. In *Proceedings of the 2nd International Conference on Computing Advancements*, pages 191–198, 2022.
- [61] Mouzhi Ge, Francesco Ricci, and David Massimo. Health-aware food recommender system. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 333–334, 2015.
- [62] Scott Howell and Richard Kones. “calories in, calories out” and macronutrient intake: the hope, hype, and science of calories. *American Journal of Physiology-Endocrinology and Metabolism*, 313(5):E608–E612, 2017.
- [63] Daniel Riera-Crichton and Nathan Tefft. Macronutrients and obesity: revisiting the calories in, calories out framework. *Economics & Human Biology*, 14:33–49, 2014.
- [64] Fsa. [https://www.food.gov.uk/sites/default/files/media/document/fop-guidance\\_0.pdf](https://www.food.gov.uk/sites/default/files/media/document/fop-guidance_0.pdf).
- [65] Trafficlights. [https://www.abdn.ac.uk/rowett/documents/Traffic\\_light\\_guide.pdf](https://www.abdn.ac.uk/rowett/documents/Traffic_light_guide.pdf).
- [66] Joint Who and FAO Expert Consultation. Diet, nutrition and the prevention of chronic diseases. *World Health Organ Tech Rep Ser*, 916(i-viii):1–149, 2003.
- [67] Christoph Trattner, David Elswailer, and Simon Howard. Estimating the healthiness of internet recipes: a cross-sectional study. *Frontiers in public health*, 5:16, 2017.
- [68] David Elswailer and Morgan Harvey. Towards automatic meal plan recommendations for balanced nutrition. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 313–316, 2015.
- [69] Palakorn Achananuparp and Ingmar Weber. Extracting food substitutes from food diary via distributional similarity. *arXiv preprint arXiv:1607.08807*, 2016.
- [70] Bodhisattwa Prasad Majumder, Shuyang Li, Jianmo Ni, and Julian McAuley. Generating personalized recipes from historical user preferences. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5976–5982, Hong Kong, China, November 2019. Association for Computational Linguistics.
- [71] Zaiqiao Meng, Richard McCreadie, Craig Macdonald, and Iadh Ounis. Exploring data splitting strategies for the evaluation of recommendation models. In *Fourteenth ACM conference on recommender systems*, pages 681–686, 2020.



- [72] Ling Cai, Jun Xu, Ju Liu, and Tao Pei. Integrating spatial and temporal contexts into a factorization model for poi recommendation. *International Journal of Geographical Information Science*, 32:1–23, 11 2017.
- [73] Bo Song, Xin Yang, Yi Cao, and Congfu Xu. Neural collaborative ranking. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1353–1362, 2018.
- [74] Jesus Bobadilla, Abraham Gutiérrez, Santiago Alonso, and Ángel González-Prieto. Neural collaborative filtering classification model to obtain prediction reliabilities. *Int. J. Interact. Multimed. Artif. Intell*, 2021.
- [75] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW ’17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.
- [76] Ji Yang, Xinyang Yi, Derek Zhiyuan Cheng, Lichan Hong, Yang Li, Simon Xiaoming Wang, Taibai Xu, and Ed H Chi. Mixed negative sampling for learning two-tower neural networks in recommendations. In *Companion Proceedings of the Web Conference 2020*, pages 441–447, 2020.
- [77] Yantao Yu, Weipeng Wang, Zhoutian Feng, and Daiyue Xue. A dual augmented two-tower model for online large-scale recommendation. *DLP-KDD*, 2021.
- [78] Jianling Wang, Ainur Yessenalina, and Alireza Roshan-Ghias. Exploring heterogeneous meta-data for video recommendation with two-tower model. *arXiv preprint arXiv:2109.11059*, 2021.
- [79] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. AutoInt: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM ’19, page 1161–1170, New York, NY, USA, 2019. Association for Computing Machinery.
- [80] Mariasole Da Boit, Angus M Hunter, and Stuart R Gray. Fit with good fat? the role of n-3 polyunsaturated fatty acids on exercise performance. *Metabolism*, 66:45–54, 2017.
- [81] Michael E Symonds, Peter Aldiss, Mark Pope, and Helen Budge. Recent advances in our understanding of brown and beige adipose tissue: the good fat that keeps you healthy. *F1000Research*, 7, 2018.
- [82] FDA. <https://www.fda.gov/media/135301/download>.
- [83] HuggingFace. <https://huggingface.co/sentence-transformers/all-mpnet-base-v2>.
- [84] Diya Li, Mohammed J. Zaki, and Ching-Hua Chen. Nutrition guided recipe search via pre-trained recipe embeddings. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pages 20–23, 2021.

- [85] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.
- [86] Georges Dupret. Discounted cumulative gain and user decision models. In *International Symposium on String Processing and Information Retrieval*, pages 2–13. Springer, 2011.
- [87] ROC. <https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5>.
- [88] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
- [89] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.