

# **Hierarchical Structure and Ordinal Features in Class-based Linear Models**

by

Wan Shing Martin Wang

B.S. Cornell University, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES  
(Computer Science)

The University of British Columbia  
(Vancouver)

February 2021

© Wan Shing Martin Wang, 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Hierarchical Structure and Ordinal Features in Class-based Linear Models**

submitted by **Wan Shing Martin Wang** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Science**.

**Examining Committee:**

David Poole, Computer Science  
*Supervisor*

Giuseppe Carenini, Computer Science  
*Supervisory Committee Member*

# Abstract

In many real world datasets, we seek to make predictions about entities, where the entities are in classes that are interrelated. A commonly studied problem, known as the reference class problem, is how to combine information from relevant classes to make predictions about entities.

The intersection of all classes that an entity is a member of constitutes the most specific class for that entity. When seeking to make predictions about such intersection classes for which we have not observed much (or any) data, we would like to combine information from more general classes to create a prior.

If there is no data for the intersection, we would have to rely entirely on the prior. However, if data exists but is scarce, we seek to balance the prior with the available data.

We first investigate a model where we assign weights to classes, and additively combine weights to make predictions. The use of regularisation forces generalisation; the signal gets pushed up to more general classes. To make a prediction for an unobserved intersection of classes, we would use the weights from the individual classes that comprise the intersection. We introduce several variants that average the predictions, as well as a probabilistic mix of these variants. We then propose a bounded ancestor method, which balances the creation of an informed prior with observed data for classes varying amounts of observations.

When dealing with ordinal properties, such as shoe size, we can dynamically create new classes and subclasses in ways that are conducive to creating more informative priors. We do this by splitting the ordinal properties. Throughout, we test on the MovieLens and UCSD Fashion datasets.

We found that a combination of the three bounded ancestor method variants resulted in the best performance, and the best combination varied between datasets. We found that a simple model that assigns weights to classes and additively makes predictions slightly outperformed the bounded ancestor method for supervised classification. For the bounded ancestor method, we found that splitting ordinal properties in different ways had minimal impact on the error metrics we used.

# Lay Summary

When making predictions involving entities and relationships between them, we often want to incorporate prior knowledge to help us, whether from existing domain knowledge, or from entities that are similar to the entities of interest. We investigate how to combine such data to help us make predictions in cases where when the data we are interested in is scarce or unavailable. We propose and evaluate several simple methods for combining data, and we test our hypotheses on a Movie prediction dataset and two Fashion datasets. We also investigate data that has an ordering, and look at ways to structure this data to help us make better predictions.

# Preface

The parameter sharing model in Chapter 2 was developed in joint work with Ali Mohammad Mehr. The parameter sharing model was designed by Dr. David Poole. In his thesis, Ali Mohammad Mehr applies the parameter sharing model to water pollution prediction, and develops and proves several theorems about the parameter sharing model.

This thesis focuses on classification on the MovieLens and Fashion datasets. Implementation and testing for the experiments in this thesis were completed by the author.

The bounded ancestor model and its variants were proposed by Dr. David Poole, and implemented and tested by the author. For the work on constructing hierarchies based on splitting ordinal properties, Dr. David Poole suggested recursively splitting on information gain. The hierarchies and experiments for the ordinal experiments in Chapter 5 were designed and implemented by the author.

# Table of Contents

<b>Abstract</b> . . . . .	<b>iii</b>
<b>Lay Summary</b> . . . . .	<b>iv</b>
<b>Preface</b> . . . . .	<b>v</b>
<b>Table of Contents</b> . . . . .	<b>vi</b>
<b>List of Tables</b> . . . . .	<b>ix</b>
<b>List of Figures</b> . . . . .	<b>x</b>
<b>Acknowledgments</b> . . . . .	<b>xi</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Hypotheses . . . . .	3
1.2 Chapter Roadmap . . . . .	3
<b>2 Parameter Sharing Model</b> . . . . .	<b>5</b>
2.1 Basic definitions . . . . .	5
2.1.1 Properties, Attributes, & Classes . . . . .	5
2.2 Predictions and Parameters . . . . .	7
2.2.1 Regression . . . . .	7
2.2.2 Error function: Sum of squares . . . . .	7
2.3 Discrete outcomes (Classification) . . . . .	8
2.3.1 Binary outcomes . . . . .	8
2.3.2 Three or more outcomes . . . . .	8
2.3.3 Error function: Negative log likelihood . . . . .	9
2.4 Regularisation . . . . .	9
<b>3 Relational Parameter Sharing Model</b> . . . . .	<b>12</b>
3.1 Tuple Classes, Functions & Hierarchies . . . . .	12
3.1.1 Tuple Classes and functions on tuples . . . . .	12

3.1.2	Hierarchies . . . . .	13
3.2	Datasets . . . . .	14
3.2.1	MovieLens . . . . .	14
3.2.2	Fashion datasets . . . . .	16
<b>4</b>	<b>Prior and Posterior Prediction for Classes . . . . .</b>	<b>19</b>
4.1	Background . . . . .	19
4.2	Bounded ancestor method . . . . .	20
4.2.1	Notation . . . . .	20
4.3	Bounded ancestor method for trees . . . . .	21
4.3.1	Mixing signals from above and below . . . . .	23
4.3.2	Parameters . . . . .	23
4.4	Bounded ancestor method for graphs . . . . .	24
4.4.1	Multiple parents . . . . .	24
4.4.2	Variant 1: Sibling data . . . . .	25
4.4.3	Variant 2: Class mixing . . . . .	25
4.4.4	Variant 3: Parameter adding . . . . .	26
4.4.5	Numerical example . . . . .	27
4.5	Combination tests . . . . .	29
4.5.1	Experimental setup . . . . .	30
4.5.2	Error Metrics . . . . .	31
4.5.3	Results . . . . .	33
4.5.4	Interpreting the weights . . . . .	34
<b>5</b>	<b>Ordinals . . . . .</b>	<b>36</b>
5.1	Classes from ordinals . . . . .	36
5.1.1	Simple intervals . . . . .	37
5.1.2	Recursive binary splitting . . . . .	38
5.1.3	Subclass Hierarchies . . . . .	39
5.2	Experiments . . . . .	40
5.2.1	Hierarchy construction and experiment setup . . . . .	41
5.2.2	Results . . . . .	42
5.2.3	Negative log likelihood loss on classes . . . . .	43
5.2.4	Testing values of k . . . . .	44
<b>6</b>	<b>Conclusion and Future Work . . . . .</b>	<b>46</b>
6.1	Results . . . . .	46
6.1.1	Hypothesis 1 . . . . .	46
6.1.2	Hypothesis 2 . . . . .	47
6.1.3	Hypothesis 3 . . . . .	47

6.1.4	Future work . . . . .	47
<b>Bibliography</b>	<b>. . . . .</b>	<b>49</b>



# List of Tables

Table 3.1	Example MovieLens 100K [6] data. . . . .	15
Table 3.2	Example User from MovieLens 100K [6] user information. . . . .	15
Table 3.3	Example movie from MovieLens 100K [6] movie information. . . . .	16
Table 3.4	Example datapoint from “Renttherunway” [14] . . . . .	17
Table 3.5	Example datapoint from “ModCloth” [14] . . . . .	18
Table 4.1	Signal from below for the classes in our numerical example. Class $A$ is (Administrator, Musical), Class $A_{p_1}$ is Administrator, and Class $A_{p_2}$ is Musical. . . . .	28
Table 4.2	A comparison of the variants on an example, (Administrator, Musical) . . . . .	29
Table 5.1	Distribution of points: Simple intervals . . . . .	37
Table 5.2	The L2 regularised parameter sharing model and the bounded ancestor method on the test hierarchies, for “ModCloth”. For both metrics, lower is better. . . . .	42
Table 5.3	The L2 regularised parameter sharing model and the bounded ancestor method on the test hierarchies, for “Renttherunway”. For both metrics, lower is better. . . . .	42

# List of Figures

Figure 3.1	Example Hierarchy involving the pair class ( $\text{Male} \cap \text{Administrator}, \text{Animation}$ ), used in MovieLens[6]. We can see that the more general classes are parent classes. . . .	14
Figure 4.1	We are interested in class $A$ , which has a single parent $A_P$ , which has parent $A_Q$ . . .	22
Figure 4.2	We are interested in class $A$ , which has two parents $A_{P1}$ and $A_{P2}$ . . . . .	24
Figure 4.3	Example subset lattice from MovieLens 100K [6] . . . . .	30
Figure 5.1	The part of the class hierarchy for the <i>Hip size</i> property, if we use simple intervals.	39
Figure 5.2	The part of the class hierarchy for the <i>Hip size</i> property, if we use binary splitting. The structure is the same for both midpoint splitting and information gain splitting.	40
Figure 5.3	Part of the class hierarchy for the experiments for binary splitting. The ordinal properties are split into classes that form binary trees whose root is a child of the global class. . . . .	41

# Acknowledgments

I would like to give a huge thank you to my supervisor, Dr. David Poole, for the guidance, help and encouragement throughout the research process and when I was writing this thesis. I would also like to thank Ali Mohammad Mehr for his helpfulness when we worked together. I would like to thank my family for their support.

Thanks to NSERC for providing funding to me through Professor David Poole's discovery grant, and thanks to Compute Canada for providing me with computing resources.

# Chapter 1

## Introduction

We are interested in making predictions about relations involving tuples of entities, where these entities can be categorised into a hierarchical class structure, or classes that are interrelated. Classes of entities induce classes of tuples of entities. We consider the problem of determining how to use observed data in prediction making for relations. This is related to the reference class problem [2], which is concerned with selecting one or more relevant classes to use for a prediction. In the reference class problem, the classes relevant to a prediction are known as reference classes. We are interested in the problem of not only identifying and selecting reference classes, but also how to combine information from them to make a prediction. In particular, we look at how to make predictions for classes with few or no observations.

One of the datasets we test on is the MovieLens 100K [6] movie prediction dataset. MovieLens 100K data takes the form of (User, Movie, rating, timestamp) quadruples, where specific users give ratings to specific movies, and information about users and movies. We chose MovieLens because we can construct an interesting hierarchical class structure from the user and movie information. MovieLens is also interesting because of the cold start problem. The cold start problem involves predicting the rating scores for users and movies that have no rating data. Cold start is an interesting motivation because it is related to our problem of trying to predict for entities without rating data.

Much of the existing work on MovieLens [6] focuses on collaborative filtering: making predictions about unseen user-movie ratings based on previous ratings made by the user, and the ratings made by other users. An example of an existing collaborative filtering work is Marlin [13]. A classic paper that does collaborative filtering for Netflix movie predictions is Koren and Bell [9]. Collaborative filtering methods don't work well on the cold start problem [19]. This thesis does not deal with collaborative filtering. Instead, we make use of MovieLens as a test case in our work on priors for classes that are arranged hierarchically. As we discuss in the future work section, our work could be potentially be used in conjunction with collaborative filtering.

When dealing with the cold start problem for MovieLens, we can use properties of the user and movie entities involved. These properties are modelled in terms of classes. Deciding which properties to use

and how to combine information from those properties relates to the reference class problem, which involves identifying which classes to use to make a prediction. Those classes that are used in the prediction are the reference classes.

For example, suppose that when making a prediction about the rating a user gives a movie in MovieLens 100K [6], we know that the user is an administrator and the movie is an action film. Some reference classes relevant to the prediction would be the set of all (user, movie) pairs, the set of such pairs where the user is an administrator, the set of pairs where the movie is an action movie, and the set of pairs with an administrator and an action movie.

An issue with real world datasets such as MovieLens 100K [6] is that data for certain classes can be nonexistent or scarce. More specific classes have fewer observations. For instance, in the MovieLens 100K dataset, the class of (user, movie) pairs where the user's occupation is administrator has 7479 members while the class of (user, movie) pairs where the user is male and an administrator and the movie is an animation film has 128 members. (MovieLens 100K has 100,000 datapoints in total).

When making predictions about classes with highly constrained descriptions, for example male administrators and animated drama documentaries, the usefulness of the data from those classes is limited if very few male administrators have rated animated drama documentaries. To predict for the constrained class, we seek to combine information from more general classes, which are the reference classes for making predictions about the class with few ratings. To do so, we seek a principled prior from the more general classes to help us.

The scarcity problem involving classes with highly constrained descriptions can be subdivided into these two cases.

- When there is no data for a specific class, for example, no male administrators have rated animation drama documentaries, we want to directly utilise information from more general classes for prediction making. Examples of general classes that could be of use in obtaining a prior are the class of (user, movie) pairs where the user is male, or the class of (user, movie) pairs where the movie is a documentary. Since more general classes are more likely to have rating data, the goal would be to combine information from these more general classes to create an informed prior for the more specific class.
- When rating data for a class is scarce but not nonexistent, if we only use the limited amount of data for the class that we have, we would overfit on that data. To avoid overfitting, we could use a hybrid approach that combines an informed prior with the data that we do observe for the class. We need to balance the prior with the observed data, by using a mixture of both.

We first investigate a model where weights are assigned to classes, and prediction making is done by additively combining weights. We call this the **parameter sharing model**. L2 regularisation can be used when learning weights. In the parameter sharing model, we make predictions for an entity by adding the weights of more general classes. L2 Regularisation forces classes with no observations to have weights

of zero. For example, if no data for administrators and action movies are observed, the prediction for administrators and action movies will include the sum of the weights for only administrators and the weights for only action movies. The weight for the class whose members are both administrators and action movies will have a value of zero.

## 1.1 Hypotheses

This thesis investigates how to combine information from general classes when data for a more tightly constrained class is scarce or nonexistent.

The simplest and obvious approach of using L2 regularisation with the parameter sharing model for learning weights doesn't work well for learning about classes with very few or no datapoints.

We introduce the bounded ancestor method for combining information from priors with information from observed data. We also introduce several variants for learning about a class from more general classes. We are interested in the effectiveness of these variants for combining data from more general classes to get priors for a more constrained class.

1. We hypothesize that the bounded ancestor method will be more effective than L2 regularisation for learning about classes with few or no datapoints.
2. We hypothesize that a weighted combination of these variants is more effective in creating priors than any individual variant.
3. We hypothesize that for ordinal data, dynamically creating new classes by splitting existing classes can help us make better predictions.

We test our hypotheses on three real world datasets: The MovieLens [6] movie prediction dataset, the “ModCloth” [14] fashion dataset, and the “Renttherunway” [14] fashion dataset. All three datasets involve making predictions about **relations** involving pairs of entities. We introduce these datasets in Chapter 3.

## 1.2 Chapter Roadmap

In Chapter 2, we introduce the basic parameter sharing model. This model was developed in joint work with Ali Mohammad Mehr [15], who proves theoretical properties about the model in his thesis, and applies the model to water pollution prediction. We also introduce important terms and pieces of notation regarding entities, classes, and hierarchies.

In Chapter 3, we show how the parameter sharing model can work with entities and relations involving tuples of entities, and how when working with tuples of entities, we can work under the same framework as the basic parameter sharing model. This is important because our three test datasets involve relations on pairs of entities.

In Chapter 4, we investigate the first and second hypotheses. We propose the bounded ancestor method, where we combine information from more general classes to create informed priors for more specific unobserved classes. We then combine the prior with observed data. Using the example above, we would combine limited observed data for male administrators and animated drama documentaries with a prior obtained individually from male ratings, administrator ratings, animated films, and so on. A tradeoff exists between emphasizing the prior or observed data: too much emphasis on the observed data causes overfitting.

We take into account the importance of observed data by bounding the effect of the prior. If no data is observed, we use the prior exclusively. As more data is observed, the data starts to take over. Building on this method, we then introduce several variants that outline different ways to combine information from individual classes to create a prior. We then investigate and compare their performance on several test datasets.

In Chapter 5, we investigate the third hypothesis, by looking into different ways of creating new classes by splitting existing classes based on ordinal properties, and we compare these splitting methods along with the parameter sharing model and the bounded ancestor model.

## Chapter 2

# Parameter Sharing Model

In this chapter, we introduce the **parameter sharing model**. The original version, which we call the basic version of the parameter sharing model was developed in a project with Ali Mohammad Mehr [15]. The basic model performs regression and classification on individual entities. Regression takes in an entity as input and produces a numerical output. Classification produces a discrete outcome as output, and the parameter sharing model outputs the probabilities of these outcomes.

An example regression application is making predictions for water pollution, which Mohammad Mehr investigates in his thesis [15]. In Chapter 3, we show how the parameter sharing model can be used for relations on tuples of entities to predict discrete outcomes.

We begin this chapter by introducing key definitions about properties, attributes and classes. We then discuss class hierarchies, and we transition to a general discussion about learning and regularisation under the basic parameter sharing model. Since MovieLens and fashion involve tuples of entities, we leave discussion of those datasets to the subsequent chapters when we formally introduce tuple classes.

## 2.1 Basic definitions

### 2.1.1 Properties, Attributes, & Classes

We define a **property** as a function that takes in an entity and outputs a value. Example properties are “height” and “shoe size”. All the properties we work with are functions. We convert non-functional properties to Boolean functions. For example, “Movie Genre” can have multiple outputs such as “Action” and “Drama”. We convert these to “Genre: Action” and “Genre: Drama”, both of which are Boolean functions with True and False as possible values. Note that relations can’t be represented like this: they are discussed in Chapter 2.

We define an **attribute** as a Boolean function that compares a property to a value for an entity. An example attribute is “height(x) > 180cm”. This attribute compares the “height” property on entity x



with the value 180 centimeters, and outputs True or False. An attribute involving equality is “shoe size( $x$ ) = 10”, where the “shoe size” property on entity  $x$  is compared with the shoe size of 10.

A **class** is defined as a set of entities. Classes are described by Boolean functions of attributes. Examples are:

- The class “height > 180cm” consists of all entities  $x$  for which the attribute “height( $x$ ) > 180cm” returns True.
- The class “(height > 180cm)  $\cap$  (shoe size = 10)” consists of all entities  $x$  for which the conjunction of attributes “(height( $x$ ) > 180cm) and (shoe size( $x$ ) = 10)” returns True.

In the basic parameter sharing model, we start with a fixed set of classes, which we call the **represented classes**. An interesting problem is deciding which classes to use as the represented classes, which is explored in Chapter 5. Central to the basic parameter sharing model is the class hierarchy. The classes are interrelated, and the relationships between these classes forms a class hierarchy.

$C_1$  is a **subclass** of  $C_2$  if  $C_1 \subseteq C_2$ .  $C_1$  is a **proper subclass** of  $C_2$  if  $C_1 \subset C_2$ . The inverse relationship of a subclass is a **superclass**. Similarly,  $C_1$  is a **proper superclass** of  $C_2$  if  $C_2 \subset C_1$

The **global class** is the class containing all entities. We denote the global class using  $\top$ .

The relationships between classes can be depicted as a graphical structure, which we refer to as a **class hierarchy**. This is important for visualising the relationships between the classes.

In the context of a hierarchy, we can refer to subclasses and superclasses as **descendant** and **ancestor** classes respectively. Additionally, we introduce **parent and child classes**. For two classes  $A$  and  $B$ :

- **Definition: Parent class:** We say that  $A$  is a parent of  $B$ , if  $B \subset A$  and there does not exist any represented classes  $C$  such that  $C \subset A$  and  $B \subset C$ .
- **Definition: Child class:** If class  $A$  is a parent class of class  $B$ , then  $B$  is a child class of  $A$ .

Since the parent and child class definitions build on the definitions of proper superclasses and subclasses, the class hierarchy is a directed acyclic graph (DAG).

### Cities example

Consider an example about cities. Suppose we have three represented classes of cities: *Western Canadian City*, *Canadian City*, and *City*. Here, *Western Canadian City*  $\subset$  *Canadian City*  $\subset$  *City*. Thus we say that *City* is a parent of *Canadian City* which is a parent of *Western Canadian City*.

For classes  $A$ ,  $B$  and  $C$ ,  $A$  and  $B$  are **sibling classes** if  $C$  is a parent class of both  $A$  and  $B$ . This is symmetric:  $A$  is a sibling class of  $B$  implies  $B$  is a sibling class of  $A$ .

## 2.2 Predictions and Parameters

The basic parameter sharing model can be used for both regression and classification. Regression produces a real number as the prediction, while Classification tries to predict a discrete outcome, which is a member of a fixed set of outcomes. Mohammad Mehr [15] applies the parameter sharing model to regression for water pollution.

A model consists of a set of classes and a parameter for each class. We write  $\sigma_j$  as the real numbered parameter for class  $C_j$ . To make a prediction for the entity, we add up the parameters of the classes that the entity is a member of.

For example, all predictions involving entities in the class “height > 180cm” use the parameter for “height > 180cm”. An important property of the basic model is that parameter sharing, along with the use of regularisation, results in generalisation in the sense that signal is being pushed up to more general classes in the hierarchy. We discuss regularisation in Section 2.4.

### 2.2.1 Regression

A dataset is a set of (x,y) pairs, where x is an entity and y is a value. We refer to the training set as  $D_{TR}$  and the test set as  $D_T$ . Let  $C_x$  be the set of classes that the entity  $x$  is a member of. We define  $\hat{y}(x)$  as the prediction for entity  $x$  by:

$$\hat{y}(x) = \sum_{j \in C_x} \sigma_j \quad (2.1)$$

Since every entity is a member of the global class, the global class parameter is used in the summation of **every** prediction. Intuitively one can think of the global class as the most general reference class.

### 2.2.2 Error function: Sum of squares

To evaluate the parameters for the represented classes, we use the **sum of squares error**. The sum of squares error for dataset D is:

$$SSE(D) = \sum_{(x_i, y_i) \in D} \left( y_i - \sum_{j \in C_{x_i}} \sigma_j \right)^2 \quad (2.2)$$

The sum of squares error function is a widely used error function in machine learning. The function penalises the square of the difference of a prediction from the actual label: hence the name “sum of squares”. We want to minimise equation 2.2 (specifically, to find the parameter values  $\sigma_j$  that results in this minimum) for the test set.

## 2.3 Discrete outcomes (Classification)

Sometimes we want to categorise the entity into one of several discrete outcomes. For example, when predicting whether users like movies, the outcome could be one of “dislikes” and “likes”.

### 2.3.1 Binary outcomes

For the binary case, we have two outcomes, for example “likes” and “dislikes”. For simplicity, we refer to them as “0” and “1”. We want to predict the probability distribution of those outcomes. Since there are only two outcomes, it is sufficient to maintain just the probability of just one of those outcomes, such as  $p_1$ , the probability of “likes”. Since  $p_0 + p_1 = 1$ , we can obtain  $p_0$  (the probability of “dislikes”) by subtracting  $p_1$  from 1.

Prediction with two discrete outcomes is also known as binary classification. For this, we take the standard approach (Jurafsky and Martin [8]) where we utilise a sigmoid function: For any real number  $k$ ,  $\text{Sigmoid}(k) = \frac{1}{1+e^{-k}}$ , which “squeezes” the numerical value into a probability in the range  $[0,1]$ . As before, we have a entity  $x$ , and  $C_x$  is the set of classes that the entity  $x$  is a member of.

Define a predictor  $\hat{y}(x)$ , which is the probability that the model assigns the output “1”, by:

$$\hat{y}(x) = \text{Sigmoid} \left( \sum_{j \in C_x} \sigma_j \right) \quad (2.3)$$

### 2.3.2 Three or more outcomes

For more than two outcomes (specifically,  $M$  outcomes), a prediction consists of a vector of length  $M$ :  $[p_0, p_1, \dots, p_{M-1}]$ . As before,  $\sum_{i=0}^{M-1} p_i = 1$ . A classification task with multiple outcomes is known as multi-class classification.

For two outcomes, it was possible to obtain  $p_0$  through  $1 - p_1$ , since there were only two outcomes involved. For  $M$  variables, we could obtain any one outcome’s probability from the other  $M-1$  outcome probabilities. We could also learn an output vector of length  $M$ , and normalise to get the probabilities.

A simple single variable linear regression with  $n$  weights produces a single numerical output using a length  $n$  weight vector. A multiple linear regression problem [4] outputs multiple numerical values, using a length  $n$  weight vector for each output value.

Our approach is similar to multiple linear regression [4]. For each class  $j$ , instead of a single parameter  $\sigma_j$ , we maintain a parameter vector which we call  $S_j$ , where  $S_j = [\sigma_{j0}, \sigma_{j1}, \dots, \sigma_{j(M-1)}]$ .  $\sigma_{ji}$  is the numerical parameter associated with the  $i$ th outcome for class  $j$ . For example, if there are three prediction outcomes, the parameter for class  $A$  would be  $S_A = [\sigma_{A0}, \sigma_{A1}, \sigma_{A2}]$ .

We define the Softmax function as follows [5]. Let  $X$  be a vector with  $M$  elements:  $X = [x_0 \dots x_{M-1}]$ . The softmax function takes in this vector, and produces another vector that is normalised.

$$\text{Softmax}(X) = \left( \frac{e^{x_0}}{\sum_j e^{x_j}}, \frac{e^{x_1}}{\sum_j e^{x_j}}, \dots, \frac{e^{x_{M-1}}}{\sum_j e^{x_j}} \right) \quad (2.4)$$

The Softmax is applied to the input vector, so that the vector becomes normalised: the components sum up to 1. It is useful for converting regression outputs into valid probability distributions.

Utilising the Softmax, the prediction function for M classes is as follows. We effectively have several simple linear regressions, one for each outcome class.

$$\begin{pmatrix} p_0 \\ p_1 \\ \dots \\ p_M \end{pmatrix} = \text{Softmax} \begin{pmatrix} \sum_{j:x \in C_j} \sigma_{0j} \\ \sum_{j:x \in C_j} \sigma_{1j} \\ \dots \\ \sum_{j:x \in C_j} \sigma_{Mj} \end{pmatrix} \quad (2.5)$$

### 2.3.3 Error function: Negative log likelihood

For discrete prediction outcomes, we use the **negative log likelihood loss**. We show the equation for the binary case here: the three-outcome version is a simple extension of the two-outcome version. The equation we use is from Jurafsky & Martin [8].

In equation 2.6, the outer summation is over all  $(x_i, y_i)$  datapoints in the dataset D, where  $y_i$  is the classification output for the  $i$ th datapoint. Since this is for two-outcome discrete prediction (binary classification),  $y_i$  is either 0 or 1.

For any dataset D:

$$\text{Loss}(D) = -\frac{1}{|D|} \sum_{(x_i, y_i) \in D} [y_i \log \hat{y}(x) + (1 - y_i) \log(1 - \hat{y}(x))] \quad (2.6)$$

Note that we divide by the number of datapoints in the test set to obtain a normalised value. We want to minimise Equation 2.6 on the test dataset  $D_T$ .

## 2.4 Regularisation

Regularisation is a technique commonly employed in machine learning to combat overfitting. Overfitting occurs when a model over-learns on the training data, resulting in a poor ability to generalise to the test set. A model that is “memorizing” training data is an extreme example of an overfitting model. For example, a linear model that considers each datapoint to be its own singleton class could then have a weight for each such class.

Implementation-wise, recall that we sought to obtain parameters that minimised the error function on the test set. We did so by minimising the same error function but on the training set, to obtain parameters

that could be generalised to the test set. Regularisation adds an extra term to the training error function. A popular regulariser for simple linear models is the L2 regulariser, also known as ridge regression when combined with a linear model. We use L2 regularisation in the parameter sharing model. For more details on L2 regularisation, see James et al: [7].

In his thesis, Mohammad Mehr [15] states and proves several theorems regarding the effects of L2 regularisation on the simple linear model. Here, we show how to add a L2 regulariser to the sum of squares and negative log likelihood functions.

Adding a L2 regularisation term to equation 2.2, we get: (Let  $D_{TR}$  represent the training set).

$$\text{Regularised SSE}(D_{TR}) = \left[ \sum_{(x_i, y_i) \in D_{TR}} \left( y_i - \sum_{j \in C_{x_i}} \sigma_j \right)^2 \right] + \lambda \sum_{j \in \mathbf{C} \setminus \{\top\}} \sigma_j^2 \quad (2.7)$$

This function is trained (minimised) on the **training set**, to enable generalisation to the test set. The test set error function is still equation 2.2 (page 7). Here  $\lambda$  is a regularisation parameter, which controls how large the regularisation effect is. The regularisation summation is over all classes except the global class. We denote this by the set of classes  $\mathbf{C} \setminus \{\top\}$ . Since we are seeking to minimise equation 2.7, the extra term is a penalty on the size of the class parameters.

To minimise the regularised sum of squares, we take the derivative with respect to the parameters  $\sigma_j$ . Notice that the error function is quadratic. Since the derivative of a quadratic is linear, the optimisation problem that arises from minimising equation 2.2 (page 7) results in finding the zeros of a linear system, enabling one to use fast solvers like QR decompositions or other numerical linear algebra tools. For an overview of linear system solvers, see Wright and Nocedal [17].

For discrete outputs, we add the same term to the loss function. We show the binary output case here. All variables are the same as in equation 2.6.

$$\text{Regularised Loss}(D_{TR}) = \left[ -\frac{1}{|D_{TR}|} \sum_{(x_i, y_i) \in D_{TR}} \left( y_i \log \hat{y}(x) + (1 - y_i) \log(1 - \hat{y}(x)) \right) \right] + \lambda \sum_{j \in \mathbf{C} \setminus \{\top\}} \sigma_j^2 \quad (2.8)$$

Just like in regression, the regularisation acts as a penalty term that prevents overfitting, and reduces the size of the parameters.

In the context of the basic parameter sharing model, we see that by virtue of the class hierarchy and parameter sharing, signal gets pushed up the hierarchy to more general classes if we use L2 regularisation. To see how generalisation in the parameter sharing model works, consider the following example.

Suppose there is a parent class  $A$ , with three disjoint child classes:  $B_1, B_2, B_3$ . Since the parent class is a superclass of each child class, an entity who is a member of  $B_1, B_2, B_3$  will also be a member

of  $A$ . Since prediction involves adding the parameters associated with each class, a prediction sum that includes  $\sigma_{B1}, \sigma_{B2}$  or  $\sigma_{B3}$  will also include  $\sigma_A$ .

If we decrease a child class parameter by an arbitrary constant  $\delta$ , and increase the parent class parameter by  $\delta$ , the prediction for an entity that is a member of both classes remains unchanged. For example, if we decrease  $\sigma_{B1}$  by  $\delta$  and increase  $\sigma_A$  by  $\delta$ , the sum of the two will still be  $\sigma_{B1} + \sigma_A$ . Intuitively, L2 regularisation forces information to be pushed towards the superclasses.

Mohammad Mehr’s thesis [15] introduces several important theorems regarding the effects of L2 regularisation. In particular, Theorem 2.1 [15] of Mohammad Mehr’s thesis shows that in an L2 regularised parameter sharing model (fully trained), for any parent class with children, the parent’s parameter is equal to the sum of the child parameters. This is true for all parent classes with children apart from the global class  $\top$ . We don’t regularise the global parameter  $\top$ .

In this chapter, we introduced the basic parameter sharing model, that performs regression and classification on simple entities by adding parameters that are assigned to classes. In the next chapter, we show how the model works with pairs of entities because pairs are also entities. We also introduce our test datasets.

## Chapter 3

# Relational Parameter Sharing Model

The previous chapter introduced the basic parameter sharing model for entities. The parameter sharing model can also be applied to tuples of entities, since a tuple of entities is also an entity. The first part of this chapter shows how the parameter sharing model works with tuples of entities. We then introduce the test datasets, MovieLens [6] and Fashion [14].

### 3.1 Tuple Classes, Functions & Hierarchies

#### 3.1.1 Tuple Classes and functions on tuples

**Definition: Tuple class:** A tuple class is a tuple of classes. A tuple of entities  $(E_1, E_2, \dots)$  is a member of a tuple class  $(C_1, C_2, \dots)$  if entity  $E_i$  is a member of class  $C_i$  for all  $i$ .

We start with a fixed set of tuple classes.

A **function on tuples** is defined as a function that takes in a tuple of entities, and outputs a value. This extends the definition of relations to have a more general range, where a relation is a function from tuples to Booleans, or equivalently, a set of tuples [3]. Both of our test datasets, MovieLens [6] and Fashion[14], involve functions from pairs to discrete values.

- For MovieLens [6], for a user entity  $U$ , and a movie entity  $M$ , we are interested in the function  $likes(U, M)$ , where  $likes(U, M) = \text{True}$  means that the rating user  $U$  gives movie  $M$  is greater than 3, and  $likes(U, M) = \text{False}$  means that user  $U$  gives movie  $M$  a rating of 3 or less.
- For Fashion [14], for a user entity  $U$ , and a clothing item entity  $I$ , we are interested in the function  $Fit(U, I)$ , where  $Fit(U, I)$  gives one of three possible outcomes depending on how well clothing item  $I$  fits user  $U$ : “Small”, “Fit”, or “Large”.

An example tuple class is (Male, Action): consisting of pairs of user entities and movie entities, where the user is Male and the movie is an Action film. We can write this in set notation as:

$$(\text{Male}, \text{Action}) = \{(U, M) : U \in \text{Male} \cap M \in \text{Action}\} \quad (3.1)$$

For instance, let Bob, Brian, and John be two users that are Male, and “Superman” and “Avengers” be two examples of action movies. Then, (Bob, Avengers), (Brian, Superman), (Bob, Superman), (Brian, Avengers), (John, Avengers), (John, Superman) are all members of the pair class (Male, Action).

Note that the set of tuples of classes (sets) is more restrictive than the set of tuples. For example, a pair of classes cannot represent the set of ratings where the user is male or the movie is an action film.

### 3.1.2 Hierarchies

The subclass and superclass properties defined for classes of entities also hold for classes of entity tuples. Similarly, the definitions for parent, child and sibling classes are the same. The tuple  $(A_1, A_2, \dots)$  is a subclass of  $(B_1, B_2, \dots)$  iff  $A_i$  is a subclass of  $B_i$  for all  $i$ .

All entities are members of the global class:  $\top$ . For tuple classes, the global class is the tuple  $(\top, \top, \dots, \top)$ . For example, all the (User, Movie) pairs (2-tuples) are members of the global class  $(\top, \top)$ , since by definition, all Users are in  $\top$  and all Movies are in  $\top$ . In subsequent sections, we abbreviate tuples involving  $\top$ . For example, (Administrator,  $\top$ ) is abbreviated to Administrator.

We show the lattice that can be constructed using the hierarchy formed from the relationships between the tuple classes in an example.

Consider an example of male Administrators rating animation movies. In the notation of tuple classes, (User, Movie) tuples with these properties would be in the following pair:

$$(\text{Male} \cap \text{Administrator}, \text{Animation}) \quad (3.2)$$

A (user, item) pair in this class would have the user be a member of  $\text{Male} \cap \text{Administrator}$ , and the item be a member of Animation.

The pair class described in Equation 2 is 3.2 is a child class of:

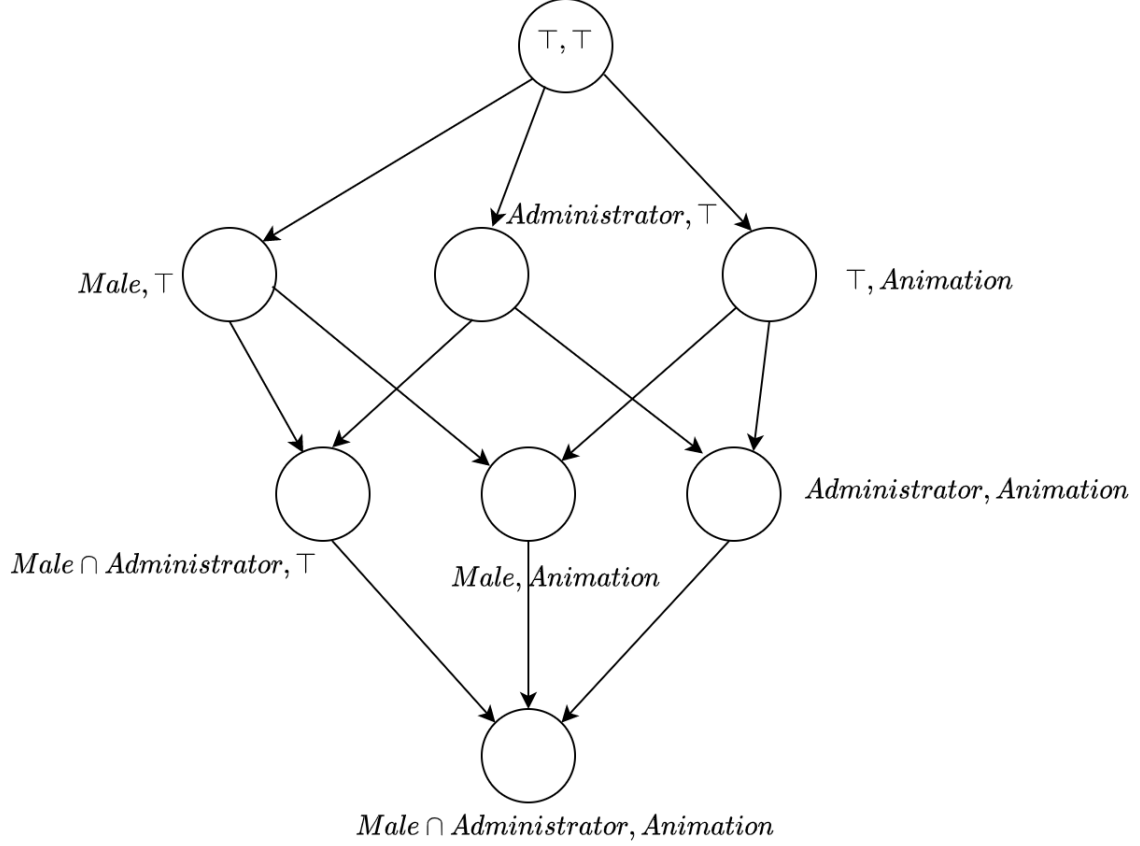
- $(\text{Male} \cap \text{Administrator}, \top)$
- $(\text{Male}, \text{Animation})$
- $(\text{Administrator}, \text{Animation})$

$(\text{Male} \cap \text{Administrator}, \top)$  is a child class of  $(\text{Male}, \top)$  and  $(\text{Administrator}, \top)$ .  $(\text{Male}, \text{Animation})$  is a child class of  $(\text{Male}, \top)$  and  $(\top, \text{Animation})$ .  $(\text{Administrator}, \text{Animation})$  is a child class of  $(\text{Administrator}, \top)$  and  $(\top, \text{Animation})$ .

The classes  $(\text{Male}, \top)$ ,  $(\text{Administrator}, \top)$ , and  $(\top, \text{Animation})$  are all child classes of  $(\top, \top)$ .



We illustrate the hierarchical relationships between these classes in Figure 3.1.



**Figure 3.1:** Example Hierarchy involving the pair class  $(\text{Male} \cap \text{Administrator}, \text{Animation})$ , used in MovieLens[6]. We can see that the more general classes are parent classes.

## 3.2 Datasets

We use the MovieLens 100K dataset as the binary outputs test set, and the Fashion datasets as the multi-output test set.

### 3.2.1 MovieLens

MovieLens [6] consists of several movie prediction datasets where users assign rating scores to movies. The datasets range in magnitude from 100,000 datapoints to 25 million datapoints. We use the MovieLens 100K and 1M datasets, which consists of 100,000 and 1 million datapoints respectively. We chose MovieLens 100K and 1M because these datasets contain information about the properties of users, whereas the 25 million sized dataset does not.

In MovieLens 100K and 1M, the data is provided in raw form as (User, Movie, Rating, Timestamp) quadruples. Properties of Users and Movies are also provided: Age, Gender, Occupation and Zip code

for users, and movie title, release date, whether the film was released for video, IMDB URL, and Genre information for Movies (IMDB is a website containing information about movies). Table 3.1 gives example quadruples, and Tables 3.2 and 3.3 gives examples of user and movie properties.

The rating score is a number in the set  $\{1,2,3,4,5\}$ , with 1 being the lowest score and 5 being the highest score. MovieLens is commonly used for testing recommendation systems, a class of model that tries to optimally recommend items to users. As mentioned in the introduction, this thesis does not deal with collaborative filtering: instead, it introduces methods that can be used in conjunction with, and as a prior for collaborative filtering.

Since there are 5 rating scores, we convert them into two outcomes: we denote scores of  $> 3$  as “likes” and scores of  $\leq 3$  as “dislikes”. We do this because we want to use MovieLens as a test set for two outcome (binary classification) prediction.

The timestamp denotes the time at which the rating was made. Since in real life, customers buy products and watch movies in real time, models that want to take this into account could utilise the timestamp. In our experiments, we don’t utilise the timestamp.

In Table 3.1, we show some example data, randomly selected from MovieLens 100K [6].

User ID	Item ID	Rating	Timestamp
275	473	3	880313679
896	73	3	887159368
629	276	5	880116887
394	658	3	880889159
856	310	3	891489217
892	238	4	886608296
279	166	4	879572893

**Table 3.1:** Example MovieLens 100K [6] data.

In Table 3.2, we show a randomly selected user from the user information for MovieLens 100K.

Property	Value
User ID	6
Age	42
Gender	M
Occupation	Executive
Zip code	98101

**Table 3.2:** Example User from MovieLens 100K [6] user information.

In Table 3.3, we show a randomly selected movie from the movie information for MovieLens 100K. The rows "Unknown" through to "Western" indicate genre: 1 indicates that the movie is of the genre, and 0 indicates that it is not. The URL property is not shown here, since it provides an IMDB link and is not used in this thesis.

Property	Value
Movie ID	899
Movie Title	Winter Guest, The (1997)
Release date	01-Jan-1997
Video release	NaN
URL	-
Unknown	0
Action	0
Adventure	0
Animation	0
Children's	0
Comedy	0
Crime	0
Documentary	0
Drama	1
Fantasy	0
Film-Noir	0
Horror	0
Musical	0
Mystery	0
Romance	0
Sci-Fi	0
Thriller	0
War	0
Western	0

**Table 3.3:** Example movie from MovieLens 100K [6] movie information.

### 3.2.2 Fashion datasets

The "fashion datasets" consist of two datasets: the "ModCloth", and "Renttherunway" datasets. Both were compiled by a team from UCSD (University of California, San Diego), and involve predicting the fit of articles of clothing, based on information about the user (for example, height, feet size), and the item of clothing. Three outcomes are possible: {Small, Fit, Large}. The fashion datasets are discussed in more detail in the creators' original paper [14]. "ModCloth" contains around 82,000 datapoints, while

Renttherunway contains around 192,000 datapoints.

In raw form, each fashion dataset is given as a JSON file with a dictionary for each datapoint. Each dictionary contains property value pairs. The first entry in the dictionary is “fit”, the output label. In both fashion datasets, properties are sometimes not reported for certain datapoints. For example, in the “ModCloth” dataset, of the 82790 datapoints, only 27914 of them have Shoe size specified.

In both “ModCloth” and “Renttherunway”, “fit” is the most common target outcome. In “ModCloth”, 68.6 % of datapoints have “fit” as the label, and 73.7 % of datapoints in “Renttherunway” have “fit” as the label.

In Table 3.4 we show an example datapoint from the “Renttherunway” dataset. The review text property is not shown due to its length: it consists of a review paragraph that isn’t used in this thesis.

Property	Value
Fit	Fit
User ID	420272
Bust size	34d
Item ID	2260466
Weight	137lbs
Rating	10
Rented for	Vacation
Review text	-
Body type	hourglass
Review summary	“ So many compliments!”
Category	romper
Height	5’8
Size	14
Age	28
Review date	April 20 2016

**Table 3.4:** Example datapoint from “Renttherunway” [14]

The “user\_id” indicates the user, and “item\_id” indicates the item entity. As part of our pre-processing, we identify which properties belong to the item and which properties belong to the user. There are a few properties that don’t belong to either, and we put them under “other properties”. The properties are as follow:

- User Properties: User ID, Age, Body type, Bust size, Height, Weight.
- Item Properties: Item ID, Category, Size.
- Other Properties: Rating, Rented for, Review date, Review summary, Review text.

Similarly, we show in Table 3.5 an example datapoint from “ModCloth”. The username property is omitted since we don’t use it in this thesis.

Property	Value
Item ID	123373
Waist	27
Size	11
Cup size	c
Hips	41
Bra size	36
Category	New
Length	Just right
Height	5 ft 4 in
User name	-
Fit	small
User ID	162012

**Table 3.5:** Example datapoint from “ModCloth” [14]

Just as in “Renttherunway”, we pre process by extracting the user, item, and “other” properties. The properties are as follows:

- User Properties: User ID, user name, Bust, Bra Size, Cup size, Height, Shoe size, Shoe width, waist.
- Item Properties: Category, Length, Quality.
- Other Properties: Review summary, review text.

Notice that the example datapoint in Table 3.5 has several properties missing, for example “shoe size” and “shoe width”.

In this chapter we described how since tuples of entities are also entities, the basic parameter sharing model applies. Learning and regularisation are the same as in the basic parameter sharing model. In the next chapter, we introduce the bounded ancestor method.

## Chapter 4

# Prior and Posterior Prediction for Classes

This chapter investigates the first and second hypotheses from the introduction.

- For the first hypothesis, we introduce and explore the bounded ancestor method for combining priors with observed data. We then introduce three variants for combining multiple parents, and we compare these methods on the fashion datasets. The results of the comparison are shown in Chapter 5.
- For the second hypothesis, we conduct a grid-search experiment on examples of multiple parents from all three datasets, to determine whether a weighted combination of the three variants produces a better prior than an individual variant.

### 4.1 Background

When we observe few or no datapoints for a class, we want to obtain a prior for the class. If known information is from related or more general classes, we want to use information from these general classes to create an informed prior. We then want to combine the prior with information from observed data to get a posterior for a class. Having a prior is important to avoid overfitting when data is scarce.

In Bayesian statistics the prior probability for a random variable represents the prior knowledge about it. The prior probability is combined with information from the observed data through multiplication with the likelihood function. If we haven't observed any data about the random variables we are interested in, we use the prior probability for inference. As the amount of observed data increases, the data starts to dominate the prior. For an introduction to Bayesian statistics in the context of machine learning, see Murphy [16].

When dealing with classes of entities or classes of entity pairs, sometimes all we know about an entity is that it is in a particular class. It would be very helpful to have a prior for that class. If there are no observations for that class, the prior for is all we have. As the amount of data for the class increases, the effect of the data should take precedence.

Our two test datasets, MovieLens 100K [6] and Fashion [14] both involve discrete target outcomes. We are interested in a probability distribution over these outcomes. A sensible choice of distribution to use as the prior for the probability of the target outcomes is the Dirichlet distribution [1].

$$p(\mu|\alpha) \propto \prod_{i=1}^W \mu_i^{\alpha_i-1} \quad (4.1)$$

Here,  $\mu_i$  are random variables that have real values between 0 and 1, and  $\sum_{i=1}^W \mu_i = 1$ . The  $\mu_i$  random variables are mutually exclusive, representing a set of probabilities of  $W$  outcomes of a single random variable  $Y$ . For each outcome  $i$  of the random variable  $Y$ ,  $\mu_i$  is a random variable representing the probability of that outcome.

The  $\alpha_i$  are the parameters of the Dirichlet distribution. There are two ways of parameterising a Dirichlet distribution.

- A positive real number  $\alpha_i$  assigned to each outcome  $i$ . For  $W$  target outcomes we write  $[\alpha_1 \dots \alpha_W]$ .
- A probability  $p_i$  associated with each outcome  $i$ , together with a positive number  $N$ . For  $W$  target outcomes, we write  $[p_1 \dots p_W]$  and  $N$ .

To map between the two parameterisations, we can do: [10]

- To get to the second parameterisation from the first,  $[p_1 \dots p_W] = [\frac{\alpha_1}{\sum_{i=1}^W \alpha_i}, \dots, \frac{\alpha_W}{\sum_{i=1}^W \alpha_i}]$  and the positive number is  $N = \sum_{i=1}^W \alpha_i$ .
- To get to the first parameterisation from the second,  $\alpha_i = p_i N$  for all  $i$ .

In this thesis, we use the second parameterisation for Dirichlet distributions. We refer to the real numbers  $[\alpha_1 \dots \alpha_W]$  and  $N$  as pseudocounts when dealing with Dirichlet distributions.

## 4.2 Bounded ancestor method

We use the bounded ancestor method for binary and multi-target classification. The bounded ancestor method outputs a probability distribution over the target outcomes. For instance, in Chapter 3, we said that each MovieLens 100K [6] entity pair is also an entity, and we defined a function that mapped each such entity to a target outcome: “dislikes” or “likes”. The bounded ancestor method gives a probability distribution over [“dislikes”, “likes”].

The bounded ancestor method combines an informed prior with observed data about entities in particular classes to get posterior distributions over the target outcomes for those classes. Both the prior and posterior distributions are in the form of Dirichlet distributions.

### 4.2.1 Notation

For a class  $A$ , the signal from below is information from observed data about entities in class  $A$ .

- For a target outcome  $v$ ,  $p_A^\uparrow(v)$  is the probability that an entity in class  $A$  has target outcome  $v$ , for the signal from below. That is,  $p_A^\uparrow$  is a function that takes in a target outcome and outputs the probability of that outcome.
- $N_A^\uparrow$  denotes the total number of entities in class  $A$ .

The signal from above is information from the prior for class  $A$ .

- For a target outcome  $v$ ,  $p_A^\downarrow(v)$  is the probability that an entity in class  $A$  has target outcome  $v$ , for the signal from above. That is,  $p_A^\downarrow$  is a function that takes in a target outcome and outputs the probability of that outcome.
- $N_A^\downarrow$  denotes the total number of entities assumed in the signal from above for class  $A$ .

To get the posterior distribution for class  $A$ , we mix the signals from above ( $p_A^\downarrow, N_A^\downarrow$ ) and below ( $p_A^\uparrow, N_A^\uparrow$ ). The mixing is a standard mixing of two Dirichlet distributions.

- Let  $p_A$  be the posterior probability distribution for class  $A$ . For target outcome  $v$ ,

$$p_A(v) = \frac{p_A^\downarrow(v)N_A^\downarrow + p_A^\uparrow(v)N_A^\uparrow}{N_A^\downarrow + N_A^\uparrow} \quad (4.2)$$

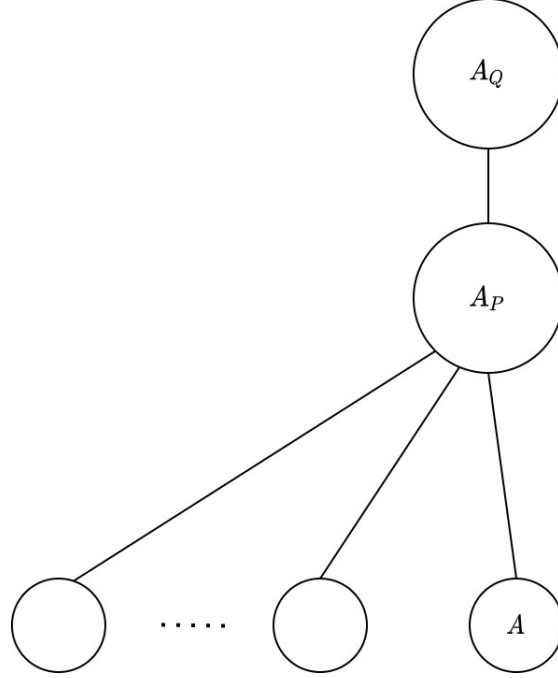
- Let  $N_A$  be the combination of the number of entities in class  $A$  and the number of entities assumed from the signal from above. For class  $A$ ,  $N_A = N_A^\downarrow + N_A^\uparrow$

### 4.3 Bounded ancestor method for trees

We first illustrate how the bounded ancestor method works for trees. In a tree hierarchy, each class (except the global class  $\top$ ) has exactly one parent.

Suppose we are interested in the distribution over target outcomes for a class  $A$ . Since we are in a tree, class  $A$  has a single parent which we call  $A_P$ . Class  $A$  can also have number of sibling nodes, who share the same parent  $A_P$ . This structure is illustrated in Figure 4.1. We first need to obtain the signals from above and below for class  $A$ .





**Figure 4.1:** We are interested in class  $A$ , which has a single parent  $A_P$ , which has parent  $A_Q$ .

The signal from below is a pair of two items  $p_A^\uparrow$  and  $N_A^\uparrow$ , both of which can be obtained directly from the members of class  $A$ . We obtain them by counting all the members of class  $A$ . The total number of members of  $A$  is  $N_A^\uparrow$ .  $p_A^\uparrow$  maps from the set of target outcomes to the proportion of members of  $A$  with those outcomes.

The signal from above is the prior. This is also a pair of two items:  $p_A^\downarrow$  and  $N_A^\downarrow$ . We get this prior by looking at the posterior for the class of entities that are in the  $A$ 's parent but not in  $A$ . These are entities in the set  $A_P - A$ , consisting of the set of all the entities not in  $A$  but in  $A$ 's parent. For a tree structured class hierarchy, we write:

$$p_A^\downarrow = p_{A_P - A} \quad (4.3)$$

$$N_A^\downarrow = k \quad (4.4)$$

$k$  is a numerical constant called the bounding constant. In Equations 4.3 and 4.4,  $p_{A_P - A}$  and  $N_{A_P - A}$  represents the posterior for the class  $A_P - A$ . class  $A_P - A$  consists of the entities that are in  $A$ 's parent but not in class  $A$ . As the amount of observed data for a class increases, we want to use the observed data and not rely too much on the prior. We try to achieve this by limiting the effect of the prior by bounding  $N_A^\downarrow$ , by  $k$ . This bounding is where the name “bounded ancestor model” comes from.

The prior for the global class  $\top$  always consists of a uniform probability distribution along with  $k$

pseudocounts. For  $W$  target outcomes, the prior would be  $p_{\top}^{\downarrow} = [\frac{1}{W}, \dots, \frac{1}{W}]$  and  $N_{\top}^{\downarrow} = k$ . For all classes  $A$ , we use a fixed pseudocount of  $k$  for the prior of  $A$ .

### 4.3.1 Mixing signals from above and below

Once we have the signal from above and the signal from below for a class, we “mix” these two signals to get a posterior distribution for that class.

At this point, we’ve computed the prior  $p_A^{\downarrow}$ ,  $N_A^{\downarrow}$ , and the signal from below  $p_A^{\uparrow}$  and  $N_A^{\uparrow}$ . The posterior distribution consists of a direct mix of the signals from above and below. We use Equation 4.2 (page 21), which is a standard mixing of Dirichlet distributions. For a target outcome  $v$ :

$$p_A(v) = \frac{p_A^{\downarrow}(v)N_A^{\downarrow} + p_A^{\uparrow}(v)N_A^{\uparrow}}{N_A^{\downarrow} + N_A^{\uparrow}}$$

The posterior count is obtained by adding the pseudocounts from above, and the number of observed counts.

$$N_A = N_A^{\downarrow} + N_A^{\uparrow} \tag{4.5}$$

Before proceeding, there are two things worth mentioning about the method.

- Since the global parameter has no siblings or parent, the “downward” count for the global class is a user-defined prior. We choose a uniform prior consisting of a uniform probability distribution along with a pseudocount of  $k$ .
- We conjecture that the bounding constant  $k$  should be chosen to be relatively small, on the order of 0-100. This is because in our test datasets such as MovieLens 100K [6], there are examples of classes with very few observations (less than 100). Since we want the data to be emphasized over the prior,  $k$  should be limited. If  $k$  is too large, the prior instead pulls the prediction towards information from other classes. We test this conjecture in Chapter 5.

### 4.3.2 Parameters

By mixing the signals from above and below, we obtained the posterior for a class  $A$ , which we denoted using  $p_A$  and  $N_A$ .

From the posterior for a class  $A$ , we can get the parameter for the class. For a class  $A$ , let  $\sigma_A$  be the parameter for the class.  $\sigma_A$  is a function that takes in a target outcome, and outputs a real number.

To obtain the parameter  $\sigma_A$  for a class  $A$ , we apply an inverse softmax function to  $A$ ’s posterior probability  $p_A$ , and then subtract the parameters of all the superclasses of  $A$ .

## 4.4 Bounded ancestor method for graphs

### 4.4.1 Multiple parents

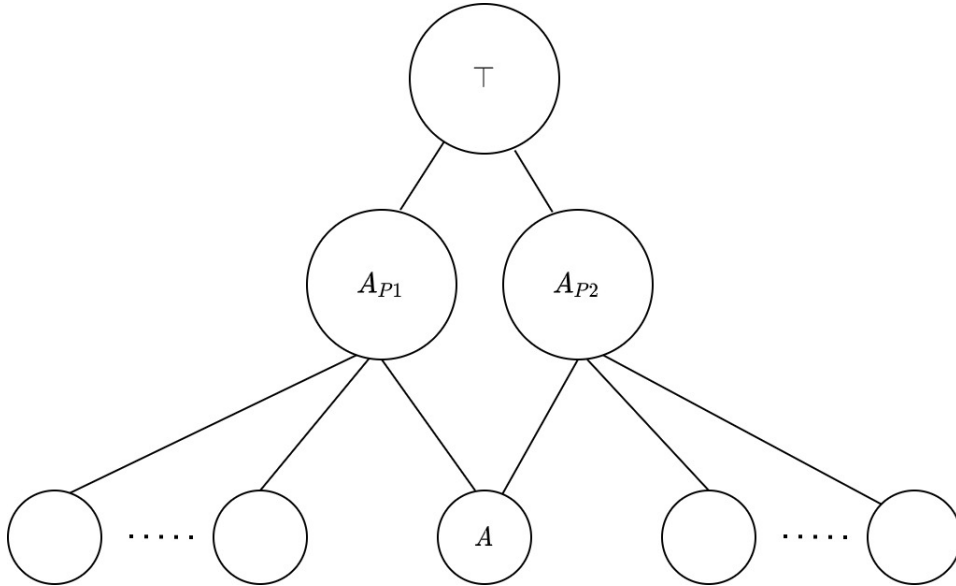
In many real-world class hierarchies such as the one constructed from MovieLens 100K [6], classes can have multiple parents. For a class in a tree, we showed how the signal from above is obtained by considering entities that are in the parent class but not in the class of interest. If the class we are interested in has multiple parents, we want ways to combine information from these parents to get the signal from above.

As a motivating example, consider the hierarchy shown in Figure 4.2, which we will return to throughout this chapter. Now, instead of class  $A$  having a single parent  $A_P$ , class  $A$  now has two parents,  $A_{P1}$  and  $A_{P2}$ .

Previously when  $A$  had a single parent  $A_P$ , we used the posterior of the set  $A_P - A$ . For two parents, we want to use information from the sets  $A_{P1} - A$  and  $A_{P2} - A$  in some way.

- $A_{P1} - A$  is the set of entities that are in  $A$ 's first parent but not in  $A$ .
- $A_{P2} - A$  is the set of entities that are in  $A$ 's second parent but not in  $A$ .

Note that the sets  $A_{P1} - A$  and  $A_{P2} - A$  don't have any common entities. This is because  $A$  is the intersection of  $A_{P1}$  and  $A_{P2}$ . For an entity to be in both  $A_{P1} - A$  and  $A_{P2} - A$ , it would have to be in the intersection of  $A_{P1}$  and  $A_{P2}$ , and not in  $A$ , which is impossible.



**Figure 4.2:** We are interested in class  $A$ , which has two parents  $A_{P1}$  and  $A_{P2}$

We now present three variants that combine information from multiple parents to get the signal from above. We show how they work with  $M$  parents. For two parents, we can set  $M = 2$ .

#### 4.4.2 Variant 1: Sibling data

In this variant, we use data from entities that are in parents of  $A$  but not in  $A$ . We call these siblings because they typically come from siblings of  $A$ . If any entities are in  $A$  but not in  $A$ 's children, we can construct a dummy class of those entities.

If  $A$  has  $M$  parents,  $A_{P1} \dots A_{PM}$ , then this variant combines information from the posteriors of  $A_{P1} - A$ ,  $A_{P2} - A$ , up to  $A_{PM} - A$ . Just like in the tree case, when working out the posteriors of  $A_{Pi} - A$  for all parents  $i$ , the signal from below excludes the entities in  $A$ , while the prior for  $A_{Pi} - A$  is the prior for  $A_{Pi}$ , since excluding entities in  $A_{Pi}$  also excludes entities in  $A$ .

$p_A^\downarrow$  would be, for a target outcome  $v$ :

$$p_A^\downarrow(v) = \frac{\sum_{i=1}^M (p_{A_{Pi}-A}(v) N_{A_{Pi}-A})}{\sum_{i=1}^M N_{A_{Pi}-A}} \quad (4.6)$$

Like in the tree structured case, because we use a uniform prior with  $k$  pseudocounts for the global class, the total number of entities in  $A$ 's prior is just the bounding constant  $k$ .

$$N_A^\downarrow = k \quad (4.7)$$

If the number of observations for entities in one parent class greatly exceeds the number of observations for entities in the other parents, this variant prioritises the impact of information from the parent with more data, because the total pseudocount involves adding the number of counts from each parent class.

#### 4.4.3 Variant 2: Class mixing

In this variant, we average the signal from each parent. The posteriors of  $A_{P1} - A \dots A_{PM} - A$  are averaged.

For a class  $A$  with  $M$  parents  $A_{P1} \dots A_{PM}$ , for  $p_A^\downarrow$ , for a target outcome  $v$ , we have:

$$p_A^\downarrow(v) = \frac{1}{M} \sum_{i=1}^M p_{A_{Pi}-A}(v) \quad (4.8)$$

The number of entities in  $A$ 's prior is  $k$ :

$$N_A^\downarrow = k \quad (4.9)$$

This variant could be potentially beneficial when we do not observe many observations for entities from a particular parent, but we think that (through prior domain knowledge or otherwise) the parent should still be strongly weighted.

#### 4.4.4 Variant 3: Parameter adding

In parameter adding, we add the parameters of all the superclasses of the class of interest to get the signal from above.

Suppose class  $A$  has  $M$  parents  $A_{P1}$  to  $A_{PM}$ . Let the set of superclasses of class  $A$  be represented by  $E_A$ . Then, to obtain  $p_A^\downarrow$  for a target outcome  $v$ :

$$p_A^\downarrow(v) = \text{Softmax}\left(\sum_{j \in E_A} \sigma_j(v)\right) \quad (4.10)$$

When obtaining the parameter for class  $i$ , we apply the inverse of the softmax function to  $i$ 's posterior probability  $p_i$ , and then we subtract the parameters of all superclasses of  $i$ . Then, to get a prior probability distribution for class  $A$  by adding the parameters of  $A$ 's superclasses, we add the parameters and apply a Softmax function to the sum.

To obtain the parameters of  $A$ 's superclasses to work out  $A$ 's prior, we would have to work out the posteriors of  $A$ 's superclasses. To avoid double counting, we subtract the observations of entities in  $A$  from the signals from below of all of  $A$ 's superclasses. The signals from above of  $A$ 's superclasses are the priors for those classes. Since we always exclude data from a class when computing the class' prior, the priors for  $A$ 's superclasses already exclude information from  $A$  because entities in  $A$  are also in  $A$ 's superclasses, so the priors do not require recomputing.

For all of  $A$ 's superclasses, we recompute their posteriors by mixing the signals from below that exclude  $A$ 's data, with the priors for those classes. After this mixing, for each of  $A$ 's superclasses we recompute their parameter by recursively subtracting the parameters of their superclasses. The summation in Equation 4.10 then adds these newly computed parameters and applies the softmax function.

Shown below is brief pseudocode for applying variant 3. When iterating through all the classes in a hierarchy, we have to do so in a topological ordering, where ancestors come before descendants. That

is, to compute the prior for class  $A$ , we have to have already computed the priors for  $A$ 's superclasses.

---

**Algorithm 1:** Using variant 3 to compute the prior for all classes

---

```

C = set of all classes in the hierarchy;
// priors stores the priors p, N for each class in C ;
priors = Empty hashset;
for  $c$  in  $C$  (topological ordering) do
     $S_c$  = set of superclasses of  $c$ ;
    // temp_parameters stores the parameters of  $c$ 's superclasses to use in obtaining  $c$ 's prior;
    temp_parameters = Empty hashset ;
    for  $s$  in  $S_c$  (topological ordering) do
         $S_s$  = set of superclasses of  $s$  ;
        //Subtract entities in  $c$  from  $s$ ;
         $p_{s-c}^\uparrow = \frac{p_s^\uparrow N_s^\uparrow - p_c^\uparrow N_c^\uparrow}{N_s^\uparrow - N_c^\uparrow}$  ;
         $N_{s-c}^\uparrow = N_s^\uparrow - N_c^\uparrow$  ;
        // Access the prior for  $s$  ;
         $p_s^\uparrow = \text{priors}[s]$ ,  $N_s^\uparrow = k$  ;
        Mix prior and posterior to get Posterior( $s-c$ ) ;
        // Recompute parameter for  $s$  by subtracting  $s$ 's superclass parameters, stored in
        temp_parameters, from posterior for  $s$  ;
         $\sigma_{s-c} = \text{InvSoftmax}(\text{Posterior}(s-c)) - \sum_{i \in S_s} \text{temp\_parameters}[i]$ 
        Store parameter for  $s$  in temp_parameters
    priors[ $c$ ] = Softmax( $\sum_{i \in S_c} (\text{temp\_parameters}[i])$ ) ;

```

---

We use a uniform probability distribution and a pseudocount of  $k$  for the prior of the global class, so the number of entities for  $A$ 's prior is  $k$ .

$$N_A^\downarrow = k \quad (4.11)$$

Intuitively, we can consider variants 1 and 2 to be interpolating between parents, while variant 3 involves adding the information from the parents, since variant 3 adds the parameters of all superclasses of the class of interest  $A$ .

#### 4.4.5 Numerical example

We show how the three variants work through a numerical example from MovieLens 100K [6]. Our example follows the hierarchical structure from Figure 4.2 (page 24), with two target outcomes, “dislikes” and “likes”. We are interested in predicting the target outcome for entities in the class (Administrator, Musical), which has two parents: Administrator and Musical. Let  $A$  denote the class (Administrator, Musical), and let  $A_{P1}$  denote Administrator and  $A_{P2}$  denote Musical.

This example will show, given observation data for classes  $A$ ,  $A_{P1} - A$ ,  $A_{P2} - A$ , and  $\top$ , how to obtain the signal from above for class  $A$ .

From MovieLens 100K [6], suppose we observe the following data about classes  $A$ ,  $A_{P1}$ ,  $A_{P2}$  and  $\top$ . These are observed counts, provided as a Dirichlet parameterisation with the probability and total count.

Class $i$	$N_i^\uparrow$	$[p_i^\uparrow(0), p_i^\uparrow(1)]$
$\top$	100000	[0.446, 0.554]
$\top - A$	99646	[0.450, 0.550]
$A_{P1}$	7479	[0.412, 0.588]
$A_{P2}$	4954	[0.460, 0.540]
$A_{P1} - A$	7125	[0.411, 0.589]
$A_{P2} - A$	4600	[0.462, 0.538]
$A$ (Ground truth)	354	[0.432, 0.568]

**Table 4.1:** Signal from below for the classes in our numerical example. Class  $A$  is (Administrator, Musical), Class  $A_{P1}$  is Administrator, and Class  $A_{P2}$  is Musical.

From the data in Table 4.1, we apply the three variants as follows. We use  $k = 5$ .

For variant 1, we need the posteriors of  $A_{P1} - A$  and  $A_{P2} - A$ .

The posterior for  $A_{P1} - A$  combines the signal from below for  $A_{P1} - A$  with the prior for  $A_{P1}$ . The prior for  $A_{P1}$  is the posterior for  $\top - A_{P1}$ . Similarly, the posterior for  $A_{P2} - A$  combines the signal from below for  $A_{P2} - A$  with the prior for  $A_{P2}$ . The prior for  $A_{P2}$  is the posterior for  $\top - A_{P2}$ . To get the posteriors for  $\top - A_{P1}$  and  $\top - A_{P2}$ , we mix the observations for  $\top - A_{P1}$  and  $\top - A_{P2}$  with the prior for  $\top$ , which is a uniform probability distribution with a pseudocount of  $k$ .

Once we have the posteriors for  $A_{P1} - A$  and  $A_{P2} - A$ , for variant 1 we mix the posteriors according to Equation 4.6 (page 25), and for variant 2 we average the posteriors according to equation 4.8 (page 25) and we get:

$$\text{Variant 1: } p_A^\downarrow = [0.431, 0.569]$$

$$\text{Variant 2: } p_A^\downarrow = [0.436, 0.564]$$

For Variant 3, we subtract the global parameter  $\sigma_\top$  from the posteriors of  $A_{P1} - A$  and  $A_{P2} - A$  to get  $\sigma_{P1}$  and  $\sigma_{P2}$ , and we add the parameters  $\sigma_\top$ ,  $\sigma_{P1}$  and  $\sigma_{P2}$  together to use as  $A$ 's prior.  $\sigma_\top$  is just the posterior of  $\top - A$ : mixing the signal from below of  $\top - A$  with the prior for  $\top$ . Note that when computing parameters for any class  $i$ , we always apply an inverse softmax to  $i$ 's posterior, before subtracting the parameters of  $i$ 's superclasses. Then, after adding the parameters of  $\top$ ,  $A_{P1}$  and  $A_{P2}$ , we apply a softmax function to the sum.

$$\text{For Variant 3, we get: } p_A^\downarrow = [0.426, 0.574]$$

Table 4.2 shows the prior produced by each example, alongside the ground truth, which is the actual distribution of the target outcomes of observed entities in the class (Administrator, Musical). We show  $P(\text{likes})$ , which is  $p_A^\downarrow(1)$ , for simplicity.

Variant	Probability (likes) for entities in (Administrator, Musical)
Ground truth	0.568
1: Sibling data	0.569
2: Class mixing	0.564
3: Parameter adding	0.574

**Table 4.2:** A comparison of the variants on an example, (Administrator, Musical)

While this example worked well in combining the two parents to form a prior, there are lots of examples where the individual variants do very poorly. To address this, we try to look at a large number of pairs of two parents to see if a weighted combination of the three variants produces a better prior.

## 4.5 Combination tests

In the sections above, we introduced three variants of the bounded ancestor method for obtaining priors for entities in classes with multiple parents. In the introduction, we hypothesized that a weighted combination of the three variants might be better for creating priors than any individual variant.

In this section we conduct an experiment that investigates whether a weighted combination creates more effective priors than an individual variant. We define three weights:  $w_1, w_2, w_3$ , where  $w_1 + w_2 + w_3 = 1$ . Let the prior produced by Variants 1, 2, and 3 be denoted by  $Output_1, Output_2$ , and  $Output_3$  respectively. The weighted prior is composed of a combination of the three variants:

$$\text{Weighted Prior} = w_1(Output_1) + w_2(Output_2) + w_3(Output_3)$$

Our goal to find the weights  $w_1, w_2, w_3$  that will produce an accurate of a prior as possible for entities in classes with multiple parents.

We treat the experiment as a supervised learning problem, where the parameters that we are trying to learn are the weights  $w_1, w_2, w_3$ . The training data consists of triples  $(A, A_{P1}, A_{P2})$ , where  $A_{P1}$  and  $A_{P2}$  are parents of  $A$ . The classes are selected from properties of the MovieLens 100K [6] and Fashion [14] datasets. A separate training set is constructed for each of the three datasets. The inputs to the learning problem are the observations of entities from the classes  $A, A_{P1}$  and  $A_{P2}$ , and the output is the prior probability distribution for class  $A$ .

We measure the accuracy of the prior for class  $A$  using the KL Divergence and the Euclidean distance. We call the actual distribution of the observations of entities in class  $A$  the ground truth distribution. For

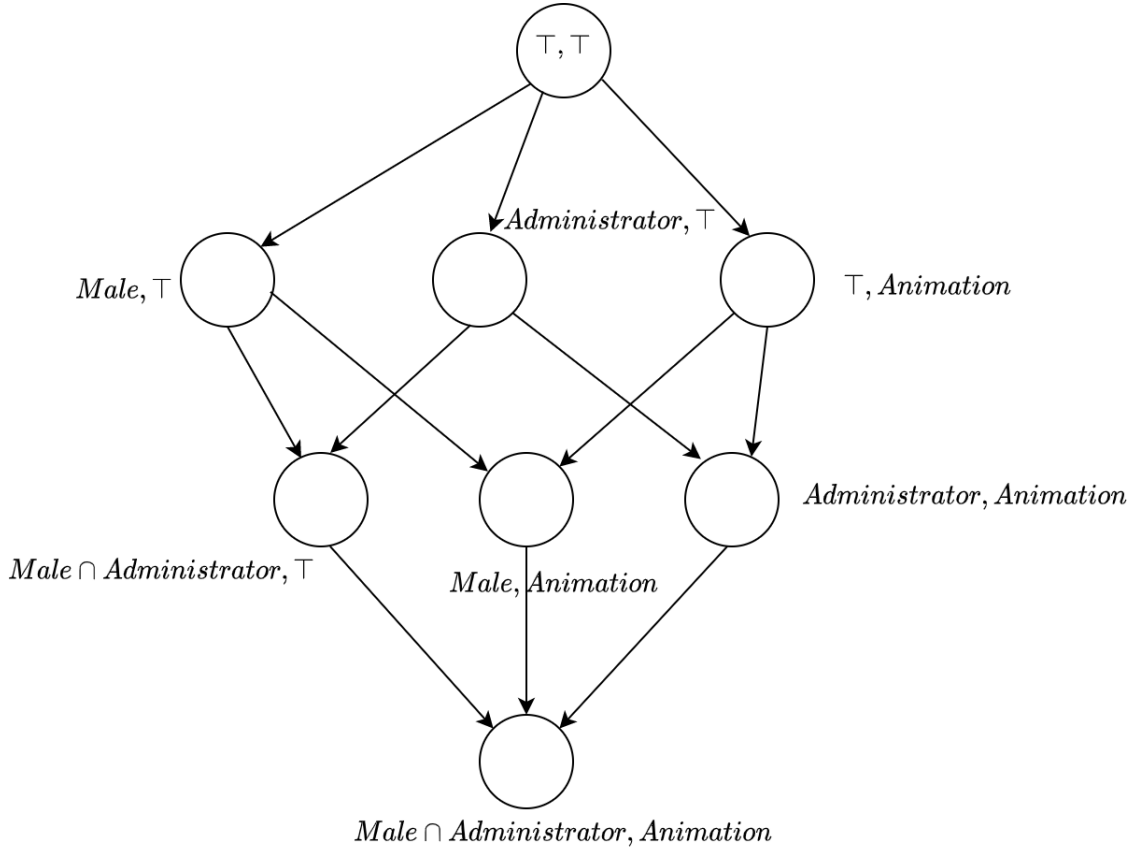


each class, we compare the prior obtained through weighting the three variants to the ground truth using the KL Divergence and Euclidean distance.

#### 4.5.1 Experimental setup

In our learning problem, we seek weights  $w_1, w_2, w_3$  that sum to 1. Our experiment uses a grid search over possible weights, to determine the weighting that minimises the KL divergence and Euclidean distance. We chose a grid search for simplicity. We only have three weights, allowing for a simple search over all weight combinations, and a grid search allows us to control the increment that we use to vary the weights.

We create classes out of individual user, movie, item properties. For example, consider the subset lattice shown in Figure 4.3 from MovieLens 100K [6].



**Figure 4.3:** Example subset lattice from MovieLens 100K [6]

An example training triple of the form  $(A, A_{P1}, A_{P2})$  from Figure 4.3 is  $((Male \cap Administrator, \top), (Male, \top), (Administrator, \top))$ . We abbreviate this as  $((Male, Administrator), Male, Administrator)$ . For properties with real values, such as shoe size, in this experiment, classes are created by splitting the value range into discrete equidistant intervals.

Of the  $(A, A_{P1}, A_{P2})$  triples, some are not useful for testing the effectiveness of the priors we obtain, so we remove them from the training set, based on the following criteria:

- In some cases, we don't have access to any observations of class  $A$ . In the bounded ancestor method, a prior for class  $A$  would have been used. However, for the purposes of training for this experiment, we need to know the ground truth observations for  $A$  to compare with the prior we create.
- If all entities in class  $A_{P1}$  are in  $A$ , the class  $A - A_{P1}$  would be empty. Similarly, if all entities in  $A_{P2}$  are in  $A$ , class  $A - A_{P2}$  would be empty. While we could still get a prior from the other parent, the purpose of this combination experiment is to determine what weights work best when working with multiple parents, so we remove these.

The number of triples that make up the training set for the weighted combination experiments that we do for each dataset are as follows:

- **MovieLens**: In MovieLens, we have 50 classes that we use as the parents. Since this experiment involves pairs of parents with a single joint child, by pairing classes up, we get 1225 possible triples. We remove some triples according to the criteria above, and end up with 826 triples.
- **Fashion, "ModCloth"**: In "ModCloth", we start with 118 classes, and pairing them up gives us 5995 possible triples. After removing some triples according to the criteria above, we end up with 4599 triples.
- **Fashion, "Renttherunway"**: In "Renttherunway", we start with 147 classes, and pairing them up gives us 10731 possible triples. After removing some triples according to the criteria above, we end up with 4954 triples.

In our experiments for all three of MovieLens, "ModCloth" and "Renttherunway", we perform 10-fold cross validation. To do so, we divide the training sets into 10 sections, and repeat the experiment 10 times. In each repeat, one section, which consists of 10% of the triples is the test set, with the remaining 9 sections consisting of 90 % of the triples as the training set.

In training, we conduct grid searches across different weights  $w_1, w_2, w_3$  for the three variants, in 0.01 increments. (For example, [0.31, 0.09, 0.6] is a potential combination of weights) and all combinations are exhausted: 5050 different weights are considered in total. For each combination, we measure the effectiveness of the weighted prior using our error metrics, shown in the next section.

#### 4.5.2 Error Metrics

We use two metrics to determine the effectiveness of the prior for a class. We use the KL divergence and the Euclidean distance. This section describes each in more detail.

##### KL divergence

The KL divergence was originally introduced by S. Kullback and R. A. Leibler [12]. An in depth introduction can be found in S. Kullback’s textbook [11]. The KL divergence is a distance measure between two probability distributions that share the same domain. Mathematically, the divergence from  $P_2$  to  $P_1$  is given by:

$$D_{KL}(P_1(x), P_2(X)) = \sum_{x \in \text{Dom}(X)} P_1(X) \log \left( \frac{P_1(X)}{P_2(X)} \right) \quad (4.12)$$

In our application, the two probability distributions we want to compare are:

- The prior obtained through weighting the three variants.
- The ground truth probability distribution.

The domain of both of these probability distributions consists of two outcomes (simplified to 0,1) for the MovieLens pairs of classes, and three outcomes for fashion (simplified to 0,1,2).

Intuitively, one can think of the KL divergence in terms of bits. If we consider  $P_1$  to be the ground truth, then the KL divergence measures the extra bits needed to encode  $P_1$  using  $P_2$ : in other words, how many more bits does one need if one chooses to use the approximation  $P_2$  in place of  $P_1$ ? Since we are approximating the ground truth with the weighted prior, we are measuring how many extra bits we need to use for the prior to approximate the ground truth. This means that in terms of direction, we compute the KL divergence from the weighted prior to the ground truth.

We sum the KL divergence from each  $(A, A_{P_1}, A_{P_2})$  triple, and divide by the number of training triples (averaging). For the logarithm, we use base 2. When working with probabilities, the KL divergence is more sensible choice compared to the Euclidean distance. The Euclidean distance is included just for comparison.

### Euclidean distance

The Euclidean distance is a classic measure of the distance between two vectors: here we include this as a baseline check, in addition to the KL divergence.

Let  $P_1$  and  $P_2$  be two probability distributions, where  $P_1 = [p_{10}, p_{11}, p_{12}]$ ,  $P_2 = [p_{20}, p_{21}, p_{22}]$ . Then, the Euclidean distance is as follows:

$$D(P_1(x), P_2(x)) = \sqrt{(p_{10} - p_{20})^2 + (p_{11} - p_{21})^2 + (p_{12} - p_{22})^2} \quad (4.13)$$

For both metrics, we compute the **average** across the pairs in the relevant training dataset.

### 4.5.3 Results

Here we show the optimal weight combinations from all 10 folds in the 10-fold cross validation experiments that minimise the KL divergence and Euclidean distance. For each of  $w_1$ ,  $w_2$ ,  $w_3$ , we report the range, mean and median for all three datasets.

#### MovieLens 100K

Optimising for KL divergence:

- Ranges:  $w_1 = [0.38, 0.49]$ ,  $w_2 = [0, 0]$ ,  $w_3 = [0.51, 0.62]$ .
- Median:  $w_1, w_2, w_3 = 0.475, 0, 0.525$
- Mean:  $w_1, w_2, w_3 = 0.465, 0, 0.535$

Optimising for Euclidean distance:

- Ranges:  $w_1 = [0.27, 0.37]$ ,  $w_2 = [0, 0]$ ,  $w_3 = [0.63, 0.73]$ .
- Median:  $w_1, w_2, w_3 = 0.32, 0, 0.68$
- Mean:  $w_1, w_2, w_3 = 0.321, 0, 0.679$

#### “ModCloth”

Optimising for KL divergence:

- Ranges:  $w_1 = [0.33, 0.46]$ ,  $w_2 = [0, 0]$ ,  $w_3 = [0.54, 0.67]$ .
- Median:  $w_1, w_2, w_3 = 0.37, 0, 0.63$
- Mean:  $w_1, w_2, w_3 = 0.371, 0, 0.629$

Optimising for Euclidean distance:

- Ranges:  $w_1 = [0.15, 0.33]$ ,  $w_2 = [0, 0]$ ,  $w_3 = [0.67, 0.85]$ .
- Median:  $w_1, w_2, w_3 = 0.2, 0, 0.8$
- Mean:  $w_1, w_2, w_3 = 0.208, 0, 0.792$

#### “Renttherunway”

Optimising for KL divergence:

- Ranges:  $w_1 = [0, 0.08]$ ,  $w_2 = [0, 0.07]$ ,  $w_3 = [0.92, 0.97]$ .
- Median:  $w_1, w_2, w_3 = 0.035, 0.01, 0.95$
- Mean:  $w_1, w_2, w_3 = 0.032, 0.02, 0.948$

Optimising for Euclidean distance:

- Ranges:  $w_1 = [0, 0.05]$ ,  $w_2 = [0, 0]$ ,  $w_3 = [0.95, 1]$ .
- Median:  $w_1, w_2, w_3 = 0.01, 0, 0.99$
- Mean:  $w_1, w_2, w_3 = 0.011, 0, 0.989$

Notice that the weighting for “Renttherunway” differs significantly from the weightings for MovieLens and “ModCloth”.

#### 4.5.4 Interpreting the weights

By considering how the variants work, we can draw some insights from them.

Variant 1 directly mixes the counts from both parents. For two parents, if one parent has significantly more observations than the other parent, the distribution of the counts of the computed prior will heavily skew towards the distribution of counts of the dominating parent class.

Variant 2 avoids this by forcing an even split in the weighting of the parents, effectively neglecting the significance of a large number of observations for a particular parent class. Thus, one can consider the weighting optimisation as determining the extent to which the skewing of the datapoints in the parents affects the distribution for a shared child. By skewing, we refer to the case where one parent has significantly more observed data than another parent. Variant 3 relies on the addition of parent and superclass parameters to compute the prior. These parameters are dependent on the distribution of counts of the classes that they represent.

We hypothesized that a weighted combination would do better than any individual variant. This was the case, but surprisingly, there are weightings of 0 for all three datasets. For MovieLens and “ModCloth” for both metrics the optimal weighting of the second variant was 0.

The results for “Renttherunway” are also surprising. For both metrics, almost all the weighting is on variant 3, with very little in the first two variants. This was the case across most of the folds in 10-fold validation. The variation between folds is very small: for example, for Euclidean distance, in “Renttherunway”, the second weight is always zero while the first weight is close to zero.

We now try to consider reasons for why “Renttherunway” differs significantly from MovieLens and “ModCloth”. First, we looked at the sets of  $(A, A_{P1}, A_{P2})$  triples used for the grid search experiment for MovieLens, “ModCloth” and “Renttherunway”. In particular, at the average difference in the number of observations of each parent  $A_{P1}$  and  $A_{P2}$ .

We looked at the differences in the number of observations between parents for the three datasets, since the first and second variants take into account the relative numbers of observations of the parents in different ways. We looked at the  $(A, A_{P1}, A_{P2})$  triples of classes that were constructed as the training sets for the combination test and looked at the average difference in number of datapoints between

parents  $A_{P1}$  and  $A_{P2}$  for the three datasets. For MovieLens the average difference in the number of entities between the two parents selected is 15208 datapoints, for “ModCloth” it was 14521, and for “Renttherunway” it was 32407. The average difference in number of datapoints between parents for “Renttherunway” is over two times that for MovieLens and “ModCloth”.

The weights for MovieLens and “ModCloth” both weighted the first and third variants positively with almost no weight on the second variant, while “Renttherunway” has similarly low weights for the first two variants. It could be that the difference in the number of points between the parents in “Renttherunway” could have affected the results.

We looked at the distribution of target outcomes “ModCloth” and “Renttherunway”. In both datasets (over the entire datasets), “fit” is the most common target outcome. In “ModCloth”, 68.6 % percent are “fit”, and in “Renttherunway”, 73.7 % are “fit”. However, since these two values are similar, it does not seem to explain the zero weight for “Renttherunway”.

Finally, the training set of  $(A, A_{P1}, A_{P2})$  triples of classes for “Renttherunway” might not represent the whole dataset, for two reasons. Firstly, as described in Chapter 3, “Renttherunway” contains missing data: certain attributes are missing. Secondly, we removed a number of training triples from the grid search where for the triple of classes  $(A, A_{P1}, A_{P2})$ , all the entities in  $A_{P1}$  are in  $A$  or all entities in  $A_{P2}$  are in  $A$ . We removed these because they wouldn’t be helpful for learning the weights, but information might have been lost because of it. For the triples removed, most of them were due to the child class  $A$  having no data. For MovieLens, 399 of the 1225  $(A, A_{P1}, A_{P2})$  triples either had no data in the joint child  $A$ , or had no data for one of the sibling sets  $A_{P1} - A$  or  $A_{P2} - A$ . Likewise, for “ModCloth”, 1396 out of 4995 had no such data, and for “Renttherunway” 5777 out of 10731 had no such data.

Since the parents  $A_{P1}, A_{P2}$  in the triples were all selected from classes constructed from properties of the three datasets, we could reason that had we had observations for the removed triples, we would have a more complete picture of the effects of multiple parents on their joint child in those datasets. In particular, in “Renttherunway”, close to half the triples were removed, which potentially means that the set of triples we use is not fully representative of the dataset.

## Chapter 5

# Ordinals

This chapter investigates the third hypothesis. We try to construct new classes from ordinal properties, and we test to see the impact of different class hierarchies on prediction making for our test datasets.

The chapter consists of two parts:

- **Ordinals:** We explore different ways of splitting ordinal properties to construct new classes. We first look at simple ways to split the range of ordinal properties into intervals. Then, we consider recursive binary splitting, by splitting on the midpoint of a range and by splitting on information gain.
- **Experimental results:** We evaluate the parameter sharing model and the bounded ancestor model on the Fashion datasets. We evaluate on the negative log likelihood loss and sum of squares error.

### 5.1 Classes from ordinals

Since classes are sets of entities, there are many possible classes. Selecting which classes to assign parameters to (known as the modelled classes) is an important problem. This chapter investigates how to select classes for ordinal properties. The Fashion datasets contain properties that are ordinal, such as shoe size, where it makes sense to consider not just the numerical values, but the “ordering” of a value relative to other values. In this section we investigate different ways in which we can split these properties and construct new classes.

Consider the property *Hip size* from the “ModCloth” fashion dataset. Of the 82790 datapoints in “ModCloth”, 56064 of them have *Hip size* specified. *Hip size* ranges from 30 to 60. In ModCloth, the input to the relation “Fit(U,I)” is a pair of entities: A (User, Item) pair.

We investigate the following ways to split this property.

1. **Intervals:** This does not take ordering into account. We can define classes based on discrete non-overlapping intervals: “Hip size 30-35”, “Hip size 35-40”, and so on. Each (User, Item) pair is a

member of only one of these classes, since the intervals are by definition mutually exclusive.

2. **Ordinal: Recursive binary splitting:** We can split the ordinal property at a single point, creating two classes. We can recursively split each new class into two more classes, creating a binary tree class hierarchy. For example, for *Hip size*, if we choose the splitting index to be 35, we get: “Hip size  $\geq 35$ ”, and “Hip size  $< 35$ ”. Each of these new classes can then be further divided recursively. We choose the splitting point using one of two ways:

- The simplest way is to choose the midpoint between the maximum and minimum value of range of the ordinal property. For example, if *Hip size* ranges from 10 to 50, we select 30 as the splitting point.
- We can also choose the splitting point resulting in two intervals that maximises the information gain. Information gain is a concept used often in decision trees in machine learning. We properly introduce this in Section 5.1.2.

### 5.1.1 Simple intervals

This is the simplest approach that does not take ordering into account. Given the minimum and maximum numerical values of *Hip Size*, we divide the range into 10 equidistant brackets. We would create 10 new classes, each described by a *Hip Size* range.

The following distribution of points can be seen in Table 5.1.

Class	Number of points
Hip Size: 30-32	2142
Hip Size: 33-35	9777
Hip Size: 36-38	12952
Hip Size: 39-41	11861
Hip Size: 42-44	8047
Hip Size: 45-47	4282
Hip Size: 48-50	3279
Hip Size: 51-53	1745
Hip Size: 54-56	1086
Hip Size: 57-59	893

**Table 5.1:** Distribution of points: Simple intervals

In this method, the datapoints are placed into classes defined on equal sized value intervals. However, splitting in this way has several issues. Such a split does not take into account the distribution of points. Splitting to create equal intervals also does not take ordering into account, which is important for ordinals.



An alternative way to create simple intervals is to use the cumulative distribution of points. We call this simple thresholding. Like before, we take the maximum and minimum numerical values of *Hip size*, and now we divide the range into 10 equidistant threshold values. Now, a (User, Item) pair is a member of one of these classes if the User’s hip size is greater than the threshold (and thus greater than all preceding thresholds). We don’t test on simple thresholding. Instead, we focus on testing interval splitting and recursive binary splitting.

### 5.1.2 Recursive binary splitting

Instead of splitting an ordinal property into multiple intervals in a single split, and creating classes from the intervals, we could select a single **splitting point** that divides the range of the property into two parts. Let the range of the property (e.g. Hip size) have minimum and maximum values  $s_{min}$  and  $s_{max}$  respectively. Then, if we split at point  $p$ , the resulting classes represent intervals:  $[s_{min}, p]$  and  $[p, s_{max}]$ . We can then choose another midpoint recursively from the intervals of each of the newly created classes to further split.

Splitting based on the midpoint is simple: The midpoint is:

$$p = s_{min} + \frac{s_{max} - s_{min}}{2}$$

Splitting on the midpoint might not be the most useful for learning and generalisation, because we are splitting at the midpoint regardless of where the observations are distributed. A more principled way of splitting would be to use the information gain. To introduce information gain, we first discuss the concept of **entropy**, which is commonly used in information theory and machine learning.

We use the definition and notation from J. R. Quinlan’s classic decision tree paper [18].

Let  $A$  be a set of points where the points are assigned outcomes of “0” or “1”. Let  $n_0$  and  $n_1$  be the number of “0”s and “1”s respectively. Then, the entropy is defined as: (Page 89, [18]. Quinlan uses  $p$  and  $n$ .  $n_0$  and  $n_1$  are used here to allow for extending to more than 2 outcomes).

$$\text{Entropy}(A) = -\frac{n_0}{n_0 + n_1} \log_2 \frac{n_0}{n_0 + n_1} - \frac{n_1}{n_0 + n_1} \log_2 \frac{n_1}{n_0 + n_1} \quad (5.1)$$

The entropy intuitively represents the amount of useful information we get from a particular collection of points. Next, we define information gain. The information gain is defined with respect to a set and a split. Let  $A$  be the original set, and  $C_1, C_2$  be the disjoint result of partitioning  $A$  into two sets. We call this split  $s$ . Let  $N_1$  and  $N_2$  be the number of points in  $C_1$  and  $C_2$ . Then, the information gain from this partition is: (Page 90, [18])

$$\text{Information Gain}(A, s) = \text{Entropy}(A) - \frac{N_1}{N_1 + N_2} \text{Entropy}(C_1) - \frac{N_2}{N_1 + N_2} \text{Entropy}(C_2) \quad (5.2)$$

We weight the entropy by the number of points in each set. This is for a particular split  $s$ . We consider all the potential splits, and we choose the split that maximises the information gain.

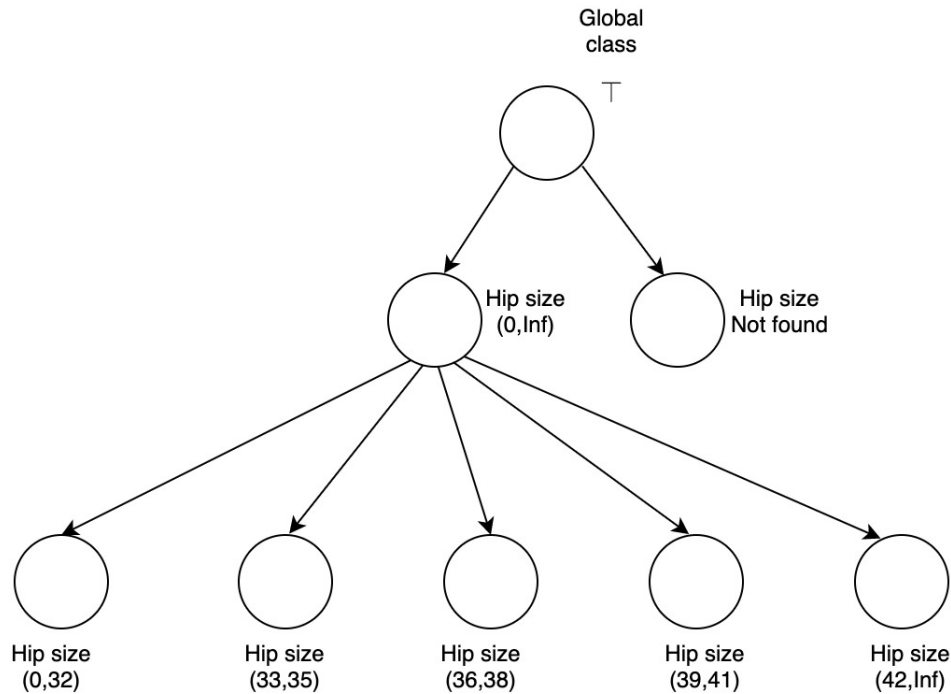
### 5.1.3 Subclass Hierarchies

The classes created through the splitting variants we discussed above results in distinct class hierarchies.

If we split a property into intervals and create classes out of these intervals, without considering ordering, all of these newly created classes will be disjoint, since any datapoint (entity tuple) can only be a member of a single one of these classes.

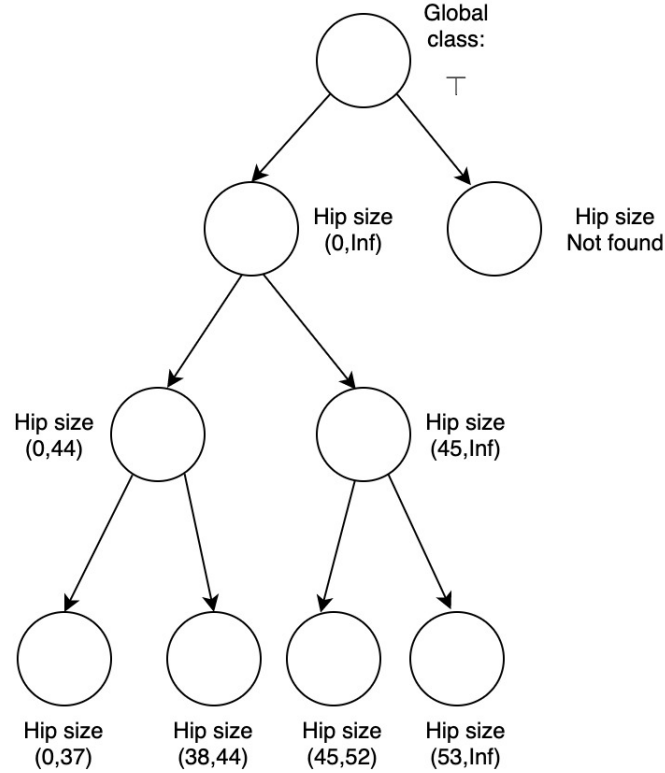
In the Fashion datasets [14], there is missing data: not all properties are reported. For instance, the property *Hip size* is not reported for all datapoints. To take this into account, we create a class whose members don't have *Hip size* reported.

The hierarchy resulting from splitting intervals for the *Hip size* property can be found in Figure 5.1.



**Figure 5.1:** The part of the class hierarchy for the *Hip size* property, if we use simple intervals.

If we choose to do recursive binary splitting, regardless of where we pick the splitting point, the hierarchy of the relationships between the created classes is a binary tree. We show this in Figure 5.2. Again, we have a separate class for the datapoints that don't report the property *Hip size*.



**Figure 5.2:** The part of the class hierarchy for the *Hip size* property, if we use binary splitting. The structure is the same for both midpoint splitting and information gain splitting.

Notice that the depth of the subclass hierarchy corresponds to the number of splits.

## 5.2 Experiments

Here, we investigate the third hypothesis: whether different ways of splitting ordinals improve test performance.

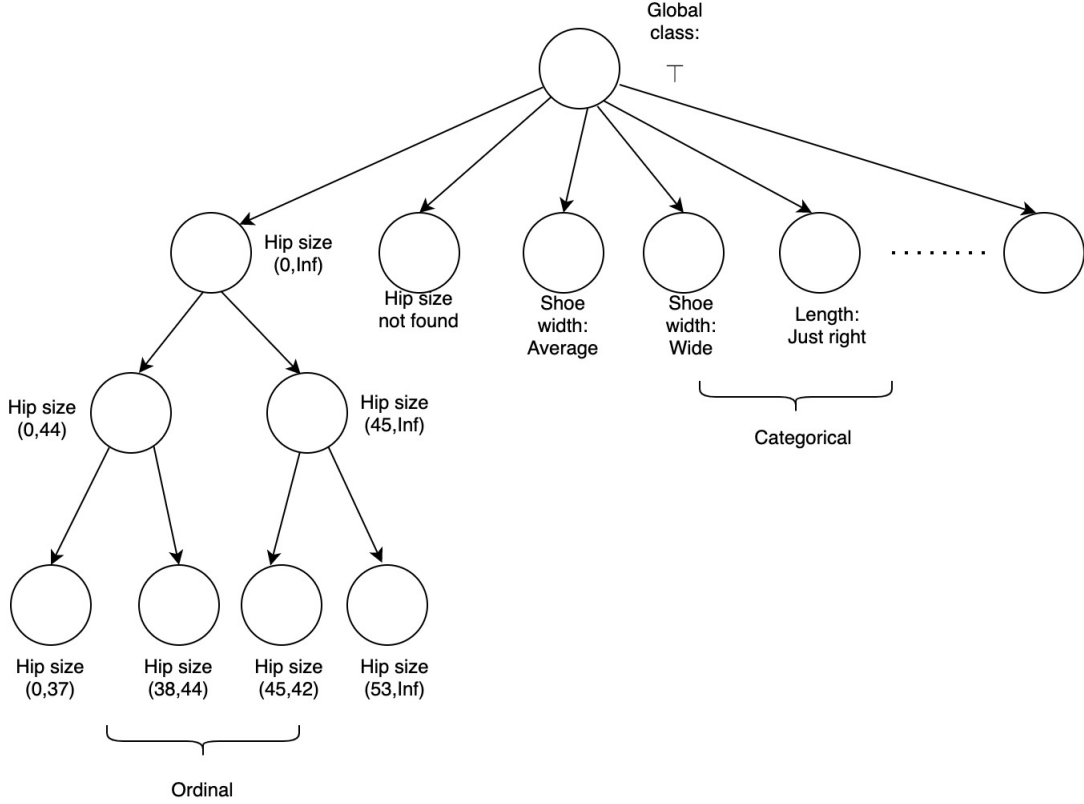
We want to see if splitting ordinals by recursively splitting on the midpoint or by recursively maximising information gain will improve prediction accuracy and negative log likelihood loss. In this experiment, we compare the performance of the regularised parameter sharing model and the bounded ancestor method on three hierarchies that differ on how the classes for ordinal properties are constructed.

- Simple intervals: like in Figure 5.1. This is the baseline that we compare to.
- Binary splits on midpoints.
- Binary splits to maximise information gain.

We test on the Fashion datasets “ModCloth” and “Renttherunway”, since both datasets have a number of ordinal properties in addition to categorical properties. We construct hierarchies for both datasets in the same way.

### 5.2.1 Hierarchy construction and experiment setup

The categorical properties have one class for each possible value. For example, for *Shoe width*, two possible values are “Average” and “Wide”. (The classes “Shoe width: Average” and “Shoe width: Wide” correspond to the attributes “Shoe width = Average” and “Shoe width = Wide”). We create new classes for the ordinals based on binary splitting. Figure 5.3, shows a subset of this hierarchy. In it, we show *Hip size* as an example ordinal, and the two values of “Shoe Width” are shown.



**Figure 5.3:** Part of the class hierarchy for the experiments for binary splitting. The ordinal properties are split into classes that form binary trees whose root is a child of the global class.

The hierarchy looks the same for both midpoint binary splitting and information gain splitting.

We use a train test split where 80% of the dataset is used for training and 20% for testing. The training testing split is done randomly, and the resulting training and test sets are set aside for the experiments. Python is used, with the Pytorch library used for the regularised parameter sharing model, and the negative log likelihood loss. (The Cross entropy loss function in Pytorch is used). For the L2 regularised parameter sharing model, the training set is further divided into a 90/10 training/validation set split, in order to determine when to stop training.

We use the negative log likelihood loss and the sum of squares error for the comparisons in the experiments in this section. The negative log likelihood loss is as defined in Chapter 2. The sum of squares error is slightly modified from Chapter 2. We are comparing an output probability vector with a target

label. In this case, we compute the average sum of squares error between the probability of an outcome, and the number 0 or 1 depending on whether the outcome was the actual label.

### 5.2.2 Results

#### “ModCloth”

Table 5.2 shows the results for the sum of squares error and negative log likelihood for “ModCloth” for the L2 regularised parameter sharing model and bounded ancestor method. For the L2 regularised parameter sharing model, the validation set was used to decide when to stop training. The results shown here are the test set error values.

Model and Hierarchy	Sum of Squares	Negative log likelihood
L2 parameter sharing: Baseline	0.14551	0.76307
L2 parameter sharing: Midpoint splitting	0.14552	0.76323
L2 parameter sharing: Information gain splitting	<b>0.14485</b>	<b>0.75972</b>
Bounded ancestor method: Midpoint splitting	0.15108	0.78849
Bounded ancestor method: Information gain splitting	0.15116	0.78852

**Table 5.2:** The L2 regularised parameter sharing model and the bounded ancestor method on the test hierarchies, for “ModCloth”. For both metrics, lower is better.

#### “Renttherunway”

Table 5.3 shows the results for “Renttherunway” for the L2 regularised parameter sharing model and the bounded ancestor method. The results shown here are the test set error values.

Model and Hierarchy	Sum of Squares	Negative log likelihood
L2 parameter sharing: Baseline	0.13273	0.70977
L2 parameter sharing: Midpoint splitting	0.13280	0.70962
L2 parameter sharing: Information gain splitting	<b>0.13205</b>	<b>0.70396</b>
Bounded ancestor method: Midpoint splitting	0.13368	0.71675
Bounded ancestor method: Information gain splitting	0.13361	0.71584

**Table 5.3:** The L2 regularised parameter sharing model and the bounded ancestor method on the test hierarchies, for “Renttherunway”. For both metrics, lower is better.

For both “ModCloth” and “Renttherunway”, the bounded ancestor method seems to do slightly worse than L2 regularised parameter sharing for both metrics. Within hierarchies, the difference in performance across the different hierarchies is very small, suggesting that splitting ordinals using information gain or at the midpoint of intervals does not have a large effect on performance.

For “ModCloth”, midpoint splitting slightly outperforms information gain splitting for the bounded ancestor model, and for “Renttherunway” the opposite is true: information gain splitting gives a lower loss on both metrics. However, the difference is extremely small, suggesting that the construction of hierarchies for ordinal properties does not have a large impact, at least with respect to supervised classification.

Despite the choice of split for the ordinal properties having a lack of impact on the negative log likelihood and sum of squared errors, using different ways to split ordinals is still informative to us because it gives us insight regarding how to group entities into classes based on ordinal properties.

In the next section we look at the contributions of individual classes to the log likelihood loss. In particular, we compare this to the number of entities in those classes.

### 5.2.3 Negative log likelihood loss on classes

To further analyse the results of running the bounded ancestor method, we looked at the value of the negative log likelihood loss function on the represented classes of our hierarchies. In particular, we look at the classes for which the loss values are largest and smallest, and the number of entities contained within those classes.

We looked at the represented classes in the hierarchies for the information gain splitting experiment from above, for both “ModCloth” and “Renttherunway” so we can compare between them. We looked at the number of entities seen in the training set for each of the represented classes, alongside the contribution of those represented classes to the test negative log likelihood loss.  $k$  was selected to be 5.

For “ModCloth”, the 5 classes with the largest negative log likelihood losses are as follows. We show the number of entities in those classes alongside the log likelihood loss. We show the average negative log likelihood loss for entities in the sets.

- Quality not found, Loss = 1.3933, 59 entities.
- Length not found, Loss = 1.3261, 27 entities.
- Waist 39-40, Loss = 1.3179, 37 entities.
- Bust 58-59, Loss = 1.3087, 2 entities.
- Waist: 41-50, Loss = 1.3064, 140 entities.

The 5 classes with the smallest negative log likelihood losses are:

- Waist: 20-22, Loss = 0.2718, 15 entities.
- Waist: 20-21, Loss = 0.2718, 14 entities.
- Bust: 20-21, Loss = 0.4146, 8 entities.

- Height: 86-91, Loss = 0.4848, 3 entities.
- Height: 55-58, Loss = 0.4999, 46 entities.

For “Renttherunway”, the 5 classes with the largest negative log likelihood losses are:

- Category: overalls, Loss = 3.6717, 5 entities.
- Category: skirts, Loss = 1.7221, 5 entities.
- Category: caftan, Loss = 1.7025, 2 entities.
- Category: sweatershirt, Loss = 1.5520, 3 entities.
- Height: 76-78, Loss = 1.4881, 17 entities.

The 5 classes with the smallest negative log likelihood losses are:

- Category: legging, Loss = 0.1082, 72 entities.
- Category: cami, Loss = 0.1135, 11 entities.
- Category: duster, Loss = 0.1382, 9 entities.
- Category: tee, Loss = 0.2844, 17 entities.
- Category: henley, Loss = 0.3002, 6 entities.

For both “ModCloth” and “Renttherunway”, the classes with the largest and smallest negative log likelihood losses tend to be smaller classes with fewer datapoints. The difference in loss values is also more extreme for “Renttherunway”, ranging from 0.30 to 3.67. One explanation is that classes with fewer entities tend to have more variance in the loss value, which we see reflected in the variation in loss.

Another possible interpretation of the variation in loss value for small classes is that the small  $k$  value might have a proportionally much larger effect on the small classes, because the prior count would be comparable in size to the number of entities in those classes. In the next section, we test the effect of varying the value of  $k$  on log likelihood loss and the sum of squares error.

#### 5.2.4 Testing values of $k$

In Chapter 4 we conjectured that smaller values of  $k$  might be more effective as a pseudocount for the prior of a class. Here, we test the effects of the value of  $k$  on the hierarchies that we use for our experiments. Here, we look at the hierarchy constructed by splitting on information gain, for “ModCloth”. We try out the following values of  $k$ : [5,10,25,50,75,100,200,300,400,500,600,700,800,900,1000]. Here, NLL stands for negative log likelihood loss, and SSE stands for the sum of squares error.

- $k = 5$ : NLL = 0.7885, SSE = 0.15116
- $k = 10$ : NLL = 0.7884, SSE = 0.15113

- $k = 25$ : NLL = 0.7881, SSE = 0.15107
- $k = 50$ : NLL = 0.7876, SSE = 0.15096
- $k = 75$ : NLL = 0.7872, SSE = 0.15087
- $k = 100$ : NLL = 0.7868, SSE = 0.15078
- $k = 200$ : NLL = 0.7856, SSE = 0.15048
- $k = 300$ : NLL = 0.7847, SSE = 0.15024
- $k = 400$ : NLL = 0.7840, SSE = 0.15007
- $k = 500$ : NLL = 0.7836, SSE = 0.14994
- $k = 600$ : NLL = 0.7833, SSE = 0.14986
- $k = 700$ : **NLL = 0.7832**, SSE = 0.14982
- $k = 800$ : NLL = 0.7833, SSE = **0.14980**
- $k = 900$ : NLL = 0.7834, SSE = 0.14982
- $k = 1000$ : NLL = 0.7837, SSE = 0.14987

The value of  $k$  seems to have a small effect on overall performance accuracy. It seems that a larger value of  $k$ , 700 for negative log likelihood loss, and 800 for sum of squares error, is the most effective out of the values we tested.

For the prior to have a significant impact on the posterior of a class,  $k$  would have to be of comparable size to the size of the class. However, as  $k$  increases, the prior would out scale and dominate the smaller classes. For example, a prior count of  $k = 500$  would result in a very large signal from above for a class with 50 entities. One possible explanation for why a prior of around 700 pseudocounts performs well is that smaller class sizes with few entities (less than 100) might have higher variance, and would benefit from a large prior.

In the context of the reference class problem, it is worth emphasizing that part of the purpose of the prior in the bounded ancestor method is to provide insight into a entities in a class using information from more general classes. The overall performance of the method on supervised classification is only part of the picture.



## Chapter 6

# Conclusion and Future Work

### 6.1 Results

We tried to address several outstanding issues with the parameter sharing model with respect to obtaining priors and combining information from multiple parents. We also explored the effectiveness of constructing new classes from ordinal properties.

We investigated three hypotheses, shown in the Introduction. We investigated them through testing the MovieLens, “ModCloth” and “Renttherunway” datasets.

#### 6.1.1 Hypothesis 1

We hypothesized that the bounded ancestor model would be more effective than regularisation when it comes to making predictions on classes with few or no datapoints.

It was found that in supervised classification tasks on entire datasets, for example on “ModCloth” and “Renttherunway” in Chapter 5, the bounded ancestor method did not perform as well as L2 regularised parameter sharing.

With respect to classes with few datapoints, we looked at how closely the priors produced by the variants matched the ground truths. While in some cases the priors are reasonable when compared to the ground truth, there are many instances of priors being created that do not seem to match the ground truth. However, for predictions involving classes that are not seen in the training set, having a prior obtained through the bounded ancestor method is significantly better than having weights of zero for those classes from regularisation.

The bounded ancestor model and the three multiple parent variants also have the benefit of being very simple to work with, being based on Dirichlet distributions. The variants also allow us to explore different ways to combine multiple parents.

### 6.1.2 Hypothesis 2

We hypothesized that a weighted combination of the three variants would be more effective than any individual variant in producing priors. We measured effectiveness of a prior by comparing the prior distribution with the “ground truth” actual distribution of counts for a class. We did so by considering pairs of classes and how their counts can be combined to get a prior for their joint child class.

We found that a weighted combination of the three variants did work best, but it depended on the dataset. MovieLens and “ModCloth” seemed to favor a combination of variants 1 and 3, while “Renttherunway” seemed to favor combining variants 2 and 3. However, we found that due to outliers and the closeness of the KL divergence of the various combinations, it was difficult to have a combination that works for all pairs of parents. However, since we can quantify the closeness of priors through the KL divergence, we reported on the combinations that minimises this based on 10-fold cross validation.

We looked at the distribution of observations for the parents in the grid search experiment conducted, and found that for MovieLens and “ModCloth”, for two parents, the number of observations in each parent were closer together than in “Renttherunway”.

### 6.1.3 Hypothesis 3

We hypothesized that splitting ordinal properties to create new classes would help us make better predictions. One can also consider this the problem of how to choose the modelled classes (classes represented with parameters) to help with prediction making.

We looked at the binary splitting of ordinal properties for “ModCloth” and “Renttherunway”. We considered splitting on the midpoint of the range of properties, and splitting to maximise information gain, to construct a new class hierarchy. We looked at how this new hierarchy affects the sum of squares and negative log likelihood loss of the regularised parameter sharing model on the entire “ModCloth” and “Renttherunway” datasets.

We found that the results for both error metrics did not vary significantly across hierarchies. While the overall prediction accuracy did not meaningfully improve, the newly constructed classes based on information gain splitting are still very informative. One key benefit of having a class hierarchy is explainability, as described by Mohammad Mehr [15]. By constructing new classes based on splits that maximise information gain, we are able to identify which intervals of property values are most important to a prediction.

### 6.1.4 Future work

We tried to obtain priors for classes with few or no datapoints by combining information through observations from the parents and siblings of those classes. Future work could potentially look into incorporating domain knowledge alongside this. For instance, in movie predictions, known habits about particular age demographics might be incorporated alongside priors obtained from data.

Future work could also investigate using the methods for obtaining priors and combining parents alongside collaborative filtering methods for MovieLens and other movie prediction datasets. Collaborative filtering methods often utilise latent embeddings that are initialised to random values or zero. The variants for combining classes could potentially be adapted to combine embeddings to obtain priors for other embeddings.

Future work could also utilise our work alongside approaches for the reference class problem. Other open problems include how to better combine information from priors with observations from data: in this thesis we computed a weighted mix, but other methods could be investigated. Furthermore, future work could also consider more sophisticated ways of combining multiple parents.

# Bibliography

- [1] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006. → page 20
- [2] M. Chiang and D. Poole. Reference classes and relational learning. *International journal of approximate reasoning*, 53(3):326–346, 2012. → page 1
- [3] E. F. Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002. → page 12
- [4] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001. → page 8
- [5] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016. → page 8
- [6] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015. → pages ix, x, 1, 2, 3, 12, 14, 15, 16, 20, 23, 24, 27, 28, 29, 30
- [7] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013. → page 10
- [8] V. Keselj. Speech and language processing daniel jurafsky and james h. martin (stanford university and university of colorado at boulder) pearson prentice hall, 2009, xxxi+ 988 pp; hardbound, isbn 978-0-13-187321-6, 2009. → pages 8, 9
- [9] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. → page 1
- [10] S. Kotz, N. Balakrishnan, and N. L. Johnson. *Continuous multivariate distributions, Volume 1: Models and applications*. John Wiley & Sons, 2004. → page 20
- [11] S. Kullback. *Information theory and statistics*. Courier Corporation, 1997. → page 32
- [12] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951. → page 32
- [13] B. M. Marlin. Modeling user rating profiles for collaborative filtering. *Advances in neural information processing systems*, 16:627–634, 2003. → page 1
- [14] R. Misra, M. Wan, and J. McAuley. Decomposing fit semantics for product size recommendation in metric spaces. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 422–426, 2018. → pages ix, 3, 12, 16, 17, 18, 20, 29, 39

- [15] A. Mohammad Mehr. *Prediction And Anomaly detection In Water Quality with Explainable Hierarchical Learning Through Parameter Sharing*. UBC Thesis, 2020. → pages 3, 5, 7, 10, 11, 47
- [16] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012. → page 19
- [17] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006. → page 10
- [18] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. → page 38
- [19] M. Vartak, A. Thiagarajan, C. Miranda, J. Bratman, and H. Larochelle. A meta-learning perspective on cold-start recommendations for items. In *Advances in neural information processing systems*, pages 6904–6914, 2017. → page 1