Food Adulteration Detection Using Neural Networks

by

Youyang Gu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Regina Barzilay
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Tommi S. Jaakkola
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Food Adulteration Detection Using Neural Networks

by

Youyang Gu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In food safety and regulation, there is a need for an automated system to be able to make predictions on which adulterants (unauthorized substances in food) are likely to appear in which food products. For example, we would like to know that it is plausible for Sudan I, an illegal red dye, to adulter "strawberry ice cream", but not "bread". In this work, we show a novel application of deep neural networks in solving this task. We leverage data sources of commercial food products, hierarchical properties of substances, and documented cases of adulterations to characterize ingredients and adulterants. Taking inspiration from natural language processing, we show the use of recurrent neural networks to generate vector representations of ingredients from Wikipedia text and make predictions. Finally, we use these representations to develop a sequential method that has the capability to improve prediction accuracy as new observations are introduced. The results outline a promising direction in the use of machine learning techniques to aid in the detection of adulterants in food.

Thesis Supervisor: Regina Barzilay
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Tommi S. Jaakkola
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

The project was partially supported by FDA contract number HHSF223201310210C. This paper represents the views and perspectives of the writer and should not be viewed or acted upon as FDA policy. For official policy and guidance, consult http://www.fda.gov.

I would like to thank my advisers, Professor Regina Barzilay and Professor Tommi Jaakkola for giving me this amazing opportunity and for guiding me throughout the research process.

I also want to thank the MIT NLP group, especially Tao Lei, for the useful feedback and help you have given me. You guys will be missed.

To the members of the MIT FDA team, thank you for the work you have all put in. We have accomplished a lot this year, I hope our work will be impactful for many years to come.

To my friends, thanks for all of your support throughout my years at MIT. I am glad we were able to push each other to not only survive at MIT, but also succeed.

Finally, thank you Mom and Dad, for everything you have done to get me to where I am today. I am forever grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ensuring the safety of food being sold in a region is important for the well-being of a country and its citizens. In the United States, the Food and Drug Administration (FDA) is responsible for protecting public health by regulating all food products that are commercially sold. The FDA has a stringent list of requirements that need to be followed before any product sees its life on a store shelf. Unfortunately, not everyone follows the rules, and there have been an increasing number of incidents over the years involving *adulteration*, the addition of unauthorized substances to food products.

There have been hundreds of recorded incidents of large-scale economically motivated adulteration of food in the past 40 years [2]. Some incidents, such as the adulteration of olive oil with hazelnut oil, have less devastating effects than others. However, in the late 2000s, milk products imported from China have been found to contain melamine, an illegal substance added to boost protein content in regulatory testing. Hundreds of thousands of children in China were sickened; six died. The number of possibilities for adulterations are plentiful and difficult to pinpoint.

Current methods of detecting potential adulterants fall on the burden on domain experts, who do not have the resources and capabilities to check every all possible combinations. There is no centralized database containing relevant information about all the adulterants. Again, human experts are needed to independently check each adulterant and outline its properties. In some instances, even these experts would have trouble making appropriate predictions. For example, sildenafil, a drug com-

monly known as Viagra, has been documented in the adulteration of chewing gum. Most humans would not have thought of Viagara and chewing gum as a possible combination, and thus lies the difficulty in these qualitative assessments. Often times, by the time an illegal substance has been discovered in a food product, it already has reached thousands, perhaps even millions of consumers. Being able to stay one step ahead is essential in this field: prevention is key.

There has been a significant interest in having an automated system with the ability to make predictions of likely adulterations. Such a system would be greatly beneficial in tackling this problem, as this would allow domain experts to narrow their search to high-risk products, therefore increasing the chance of stopping potentially dangerous products from reaching consumers. The goal of this project is to make the first steps towards building such a system. More specifically, our goal is to develop a model that would 1) characterize various adulterants and 2) predict which food products are most likely to be affected by a given adulterant.

Neural Networks

We will explore the building of this system using neural network architectures. The rise in popularity of neural networks in recent years is partly due to their ability to learn latent features in a semi-supervised fashion, requiring minimal prior knowledge about the data and drastically reducing the time spent on feature selection. In natural language processing, using deep neural networks to generate word embeddings have been shown to perform significantly better than traditional N-gram models [9, 11]. In addition, they have also been used to achieve state-of-the-art results in areas such as sentiment classification [6], Go [13], and parsing [8].

There are many technical challenges in properly training neural networks. There is often a misconception that they can generate great results with little effort. Given a large database such as the ULMS Metathesaurus, it is unclear what information we can extract from it that will be valuable in helping us make predictions. Using text data poses a completely different challenge. Short text from a source such as

Wikipedia often contain a lot of noise, and it is unclear what the best way to train a model to extract information from this data. Furthermore, our training data is bounded by the number of articles we have for ingredients. Very few neural network models in literature are trained on a corpus of only a few thousand entries.

In this work, we try to tackle these challenges and apply our models to this novel task, namely that of predicting which food categories are likely to contain certain (possibly illegal) ingredients. We use two main types of inputs: hierarchical properties and text descriptions of ingredients. We show that neural networks are able to achieve good predictions on a variety of tasks involving ingredients, adulterants, and food products.

Related Work

As far as we are aware, we are the first to apply this technique to this problem. While neural network models are prevalent in the machine learning community, there has be no prior work done in applying these models to large-scale adulteration detection (to the best of our knowledge). [16] showed the generation of ingredient substitutions and alternate combinations of ingredients using online recipes. However, the set of ingredients available in recipes is much smaller than the set of ingredients used in commercial food products. Neural networks are uniquely equipped to handle large datasets, and it is of interest to see if they are applicable in this particular setting (they are).

Recurrent neural networks have been used to generate representation from the Wikipedia corpus [15]. Wikipedia text have also been used for text similarity classification [14]. These neural network models use the entire Wikipedia corpus, which differs from our approach of training on a specific subset of articles.

Narayanan [10] worked on building a database of adulteration instances from relevant news articles on the web. He applied various machine learning techniques such as the collaborative filtering classifier to determine whether an ingredient and a food product can co-exist.

Data

Most resources contain only a few hundred instances where a substance has adulterated a food product. The success of neural networks often depends on a large training set. The scarcity of data in the area of food adulterants leads us to explore alternative data sources to which we can train our models on. We relate adulterants to regular food ingredients (milk, sugar, strawberries, etc.) and view them as part of the same group of substances that can appear in a food product category. We assume that a model that can predict which product category a regular food ingredient can appear in will also be able to generalize to adulterants. Therefore, for the rest of this paper, we will use the term *ingredient* to also include adulterants.

We have access to a large database of food products sold in the United States and their ingredients, which we describe in detail in Chapter 2. We would like to take a systemic approach to characterize the ingredients and the food products. Can we predict the ingredient based on what products it appears in? Can we predict which food products an ingredient is likely to appear in? Can ingredients be clustered into groups based on their properties? What are likely substitutions for a given ingredient? These are all questions we hope to answer.

Of course, while analyzing common food ingredients can be interesting, our ultimate purpose is to generalize our results for adulterants. By narrowing the range of food products that a substance can adulter, regulators such as the FDA can focus their inspection efforts on a much lower subset of products, leading to higher efficiency and effectiveness.

Sequential prediction

An issue with many standard machine learning techniques lies in how they are trained. Once trained, many models are unable to adjust for new data points without having to retrain on the entire data set, which is often infeasible, whether it is due to time, cost, computational power, or all of the above. Online machine learning methods

address this issue by making their predictions one data point at a time, making them ideal in this particular scenario. Often times, we are given a small subset of adulterant/product category occurrences, and we must be able to adjust our model accordingly in order to make predictions about other product categories that this adulterant could occur in. In other words, if adulterant X has been found in food product category Y, what other categories is it likely to be found in? We present a model built on top of our neural network model that does sequential refinement based on new data points.

Code and data

We use Python to implement all the models described in this paper. The Numpy, Pandas, Theano, and Scikit-learn packages were instrumental to the success of our implementation. The code is available at: *https://github.mit.edu/yygu/adulteration* and *https://github.mit.edu/yygu/rcnn*.

Overview of the remaining sections

We will first describe the data we use in Chapter 2. Next, we present our first attempt at characterizing ingredients in food products using neural networks (Chapter 3). In Chapter 4, we describe a collaborative filtering method of predicting adulterant / food product pairs. The remaining chapters focuses on the following question: given an ingredient, what food product categories can it appear in? In Chapter 5, we leverage hierarchical properties of each ingredient to predict likely food categories. We introduce a recurrent neural network architecture in Chapter 6 to leverage Wikipedia articles of ingredients. Finally, we implement a model in Chapter 7 that can sequentially update its predictions with each additional observation.

Chapter 2

Data

Our data is divided into two main categories: 1) properties of ingredients and food products and 2) known cases of adulterant-food product pairs.

2.1 Properties of ingredients and food products

We can further divide this data into two subsections:

1. Product-level information of the ingredients. The *FoodEssentials LabelAPI*[1] gives us access to a majority of products being sold commercially in the US. For each product, we have information about its name, product category, list of ingredients, UPC code, and several other fields. See the Appendix for a sample entry. The majority of this paper focuses on this data set.

    - We extracted over 140,000 unique products and 100,000 unique ingredients. Each product has an average of 14.3 ingredients. 17,000 ingredients occur in more than 3 products, and only 1500 ingredients are present in 100+ products.
    - Each product is classified under 3 types of categories, in order of increasing specificity: aisle (16 choices), shelf (131 choices), and food category (1124 choices). For example, diet Pepsi falls under the food category "Soda -

[1] http://developer.foodessentials.com/

Diet Cola", the shelf "Soda", and the aisle "Drinks". Besides Chapter 3, all references to product categories refer to the 131 categories under "shelf".

- The ingredients for a product are listed in descending order of predominance by weight.
- Salt, water, and sugar are the three most popular ingredients, occurring in 51%, 37%, and 35% of all products, respectively. There are 13 ingredients that appear in more than 10% of all products.
- Given a product, we can determine its ingredient list. Given an ingredient, we can determine all products that contain it. We can do numerous types of analysis with this data. For example, Table 2.1 is a list of the top 10 food categories that are most likely to contain monosodium glutamate, or MSG.

2. Property-level information of the ingredients. We can represent each ingredient by the relationship hierarchy it forms. This data comes from the UMLS Metathesaurus[2]. For example, the hierarchy for monosodium glutamate (MSG) is as follows: `monosodium glutamate` → `glutamic acid` → `amino acid` → `carboxylic acid` → `organic compound`. We will transform this into a vector representation to describe each ingredient, allowing us to characterize unknown ingredients. We expand on the use of this information in Chapter 5.

    - There are 2.5 million entries in this database (and 10+ million relationships), but not all of them are necessarily relevant to ingredients that are used in food.

2.1.1 Preprocessing

We apply some preprocessing on the ingredient list to convert it from a string to a list of individual ingredients. To the best of our ability, we remove non-ingredients

[2]https://www.nlm.nih.gov/research/umls/knowledge_sources/metathesaurus/

Food Category	# products in category	% products w/MSG
Soup - Matzo Ball Soup	18	72%
Seasoning - Bouillon	180	71%
Stuffing Mixes	74	62%
Canned Soup - Cream Of	178	60%
Dip & Salsa - Ranch	60	58%
Salad Dressing - Ranch	368	57%
Canned Meals - Chili No Beans	62	52%
Dip & Salsa - Sour Cream Dip	47	51%
Snacks - Cheese Puffs	190	51%
Snacks - Onion Rings	22	50%

Table 2.1: 10 most common food categories containing MSG (by percentage).

(e.g. 'contains 2% or less of'), parenthesis (sub-ingredients of an ingredient), funny punctuations, capitalizations, and other irregularities. Nevertheless, we cannot catch all instances (e.g. 'salt' vs 'slat' vs 'less than 0.1% of salt'). We also convert all the ingredients to lower-case, singular form. Finally, we ignore extremely rare ingredients by limiting the total number of ingredients we are analyzing to the N most frequently-occurring ingredients. For the majority of this work, we used $N = 5000$. While there is no technical limitations to using a larger N, we chose $N = 5000$ as our upper limit for two reasons: 1) a larger N takes longer to train and 2) most ingredients after the top 5000 occur in less than 10 products (0.007%), which leads to an imbalanced data problem that we will describe in Section 3.3.1.

2.2 Known cases of adulterant-food product pairs

We found four sources that contain documented instances of adulterations:

- Rapid Alert System for Food and Feed (RASFF)[3]
- FDA Import Refusal Report (FDA)[4]
- U.S. Pharmacopeial Convention Food Fraud Database (USP)[5]

[3]http://ec.europa.eu/food/safety/rasff/index_en.htm
[4]http://www.accessdata.fda.gov/scripts/importrefusals/
[5]http://www.usp.org/food-ingredients/food-fraud-database

Name	Author	Entries	Unique Entries	Adulterants	Products	Categories	% adult. in other 3
RASFF	European Commission	7691	4403	425	3851	34	14.10%
FDA	US Food and Drug Administration	111183	245	30	14578	36	33.30%
USP	United States Pharmacopeia	2593	1775	1166	649	—	12.00%
EMA	Food Protection and Defense Institute	2189	2011	789	486	24	18.10%

Table 2.2: Comparison of four databases of adulterants. The *% adult. in other 3* column shows the percentage of adulterants that appear in at least one other database. We focus most of our analysis on the RASFF database, as we have qualitatively determined it to be the most comprehensive. Overall, there are 2221 unique adulterants, from which 167 of them appear in more than 1 source. "Melamine" is the only adulterant to appear in all four sources.

- Economically Motivated Adulteration Incidents Database (EMA)[6]

See Table 2.2 for a comparison of the four databases. To the best of our knowledge, there are no other English databases containing significant amounts of information regarding adulterants. For samples of the content found in these four sources, we refer readers to Appendix A. We chose to focus most of our analysis of adulterants using the RASFF database because it is the most comprehensive and realistic of the four. They also represent real import violations, something the EMA and USP sources do not have. On the other end of the spectrum, the FDA Import Refusals list is the least specific in terms of listing the exact adulterant found in each of the food products.

2.2.1 Rapid Alert System for Food and Feed (RASFF)

RASFF is a system developed by the European Commission that tracks various import violations of food-related products entering the European Union. We retrieved 7,691 entries of import alerts/rejections taken from 5 categories: food additives, heavy metals contaminant, industrial contaminant, composition, and chemical contaminant. Out of those there are 465 unique substances. 253 substances have been found in a food product that falls under one of the 131 product categories defined in the ingredients dataset.

Mercury, sulphite, and cadmium are the 3 most common adulterants, occurring in 14%, 11%, and 9% of the incidents, respectively.

[6]http://www.foodfraudresources.com/ema-incidents/

Out of the 465 substances, 178 (38%) of them appear in the database of ingredients in food products (Section 2.1). This means that the remaining 62% of the substances are not normally found in food products, and its presence is most likely due to some form of adulteration. When we narrow our search to only the 178 ingredients found in food products, we find that the majority of the RASFF entries contain ingredient/food category pairs that do not exist in food products. In other words, even though the ingredients are used in food products, the cases found in RASFF occur when these ingredients are found in food categories they are not designated for. For example, 'alcohol' is a valid ingredient, but 'alcohol' in 'soda' would be a case of adulteration.

We would like to point out that the RASFF database does not specify *why* the violation occurred, meaning that we do not know if the violation was accidental or intentional. For this study, we make the assumption that each pair (ingredient/product) is a positive instance of adulteration, whether or not it is intentional. We also do not keep track of the amount of the adulterant - we view any presence as an instance of adulteration, even though there are situations where it was simply a matter of there being too much of a substance. These are simplified assumptions, but this allows us to be methodical in our search and analysis despite the low volume of data.

2.3 Using chemical formulas

One idea we had is to make predictions about adulterations based on the standardized chemical names of the adulterants (e.g. the IUPAC name). The motivation comes from the fact that there will be many instances where we have not seen an adulterant during training, yet we must make some kind of predictions about this unknown adulterant. The name of a substance can reveal a lot of information about its properties, and we hope to leverage this fact to generate a representation of each unseen adulterant and use it as an input into our model to make predictions. Each substance can be uniquely identified by its standardized chemical name. For instance, 3-MCPD and 3-monochloropropane-1,2-diol both refer to 3-chloropropane-1,2-diol. There has been prior research in the natural language processing community in using the chem-

ical names to successfully predict chemical reactions. We hope we can apply similar models to this task.

Out of the 464 adulterants from RASFF, 134 (29%) contain a chemical formula in the UMLS Metathesaurus. We also tried using the CIRpy Python library to look up the chemical formulas of adulterants. CIRpy is an interface for the Chemical Identifier Resolver (CIR), a database developed by the CADD Group at the National Institute of Health. Out of 2221 total adulterants, we are able to find 437 adulterants that have easily-accessible chemical structures, for a recall of 20%. This is a lower bound, since some entries have no direct match but have an indirect match (e.g. 'rhodamine b' instead of 'colour rhodamine b'). Nevertheless, the low rate of recall is not sufficient for us to continue pursuing this direction.

We also tried extracting 10 million chemicals from the ChemNet database of reactions and cross-referenced the list of 2221 adulterants. Only 61 out of the 2221 adulterants (2.7%) appear in this database. For the RASFF European database, it is 29 out of the 425 adulterants (6.8%). Again, because of the low recall, we chose not to incorporate chemical formulas and reactions in our models.

Chapter 3

Characterization of Ingredients

In this chapter, we focus on the characterization of the ingredients in food products using techniques inspired from natural language processing. We lay the groundwork for models that can be extended to include adulterants.

3.1 Problem formulation

We have a set of N ingredients $S_I = \{x_1, x_2, \ldots, x_N\}$ and F food products $S_F = \{p_1, p_2, \ldots, p_F\}$. Each product j contains a subset of the ingredients: $p_j = \{x_{j,1}, x_{j,2}, \ldots\}$ and belongs to a particular category c_j. For example, we have a product named *Hunt's Tomato Paste* which consists of 3 ingredients: tomatoes, salt, and seasoning. It is categorized as "canned tomatoes - tomato paste".

Given a set of ingredients, one goal is to be able to predict the most likely category of this "product". Continuing the previous example, an input of "tomatoes, salt, and seasoning" should tell us that "canned tomatoes - tomato paste" would be a very likely category, whereas "milk additives - powdered milk" would not. A second goal is to be able to predict whether a set of ingredients is a valid combination of ingredients. We will describe this in deeper detail in the following sections.

In this chapter, we want to characterize the ingredients by using neural networks to accomplish the following four goals:

1. Generate a low-dimensional vector representation for each ingredient.

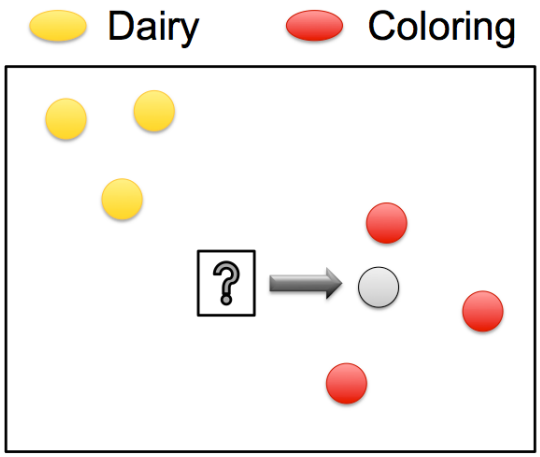

Figure 3-1: Characterizing unknown ingredients using 2-dimensional vector representations. In this case, the unknown ingredient is most likely a type food coloring.

2. Predict the food category given a list of ingredients.

3. Determine if a list of ingredients form a valid combination.

4. Predict characteristics given an unknown ingredient.

Figure 3-1 shows an illustration of Goal 1. If we are able to generate a low-dimensional vector representation for each ingredient, we can more easily characterize new ingredients based on this new representation.

3.2 Underlying model

For all tasks that we investigate, we use a multilayer perceptron (MLP) with a single hidden layer. We use a tanh activation function at the hidden layer. The model is implemented in Python on top of the *Theano* library. The code has been optimized to handle all the necessary computations in a reasonable amount of time (<1 hour per epoch). At each step, we calculate the gradient and perform stochastic gradient descent (SGD) on a batch. Data is split into a training set, a validation set, and a testing set. All results presented in this paper are generated from the validation or

testing set. We implemented our own grid search algorithm to choose hyperparameters such as the number of hidden nodes, the learning rate, number of epochs, and the regularization rate.

3.3 Skip-ingredient model

3.3.1 Approach

The skip-gram model, introduced by Mikolov et al. [9] for `word2vec`, learns to predict neighboring words given the current word. We modify this model to predict the other ingredients in the same product given a particular ingredient. We call this the *skip-ingredient model*. The input is a particular ingredient i, and the output is a probability distribution over all ingredients that i is likely to exist in the same product with (hereby referred to as the context). The advantage of this model is that the output size does not need to be fixed: we can generate an individual distribution of the likelihood for every ingredient in the output. If a product contains k ingredients, then we can produce k potential training points from that product.

We derive the cost function for the skip-ingredient model that is consistent with the derivation for the skip-gram model presented by Rong [12]. For each training point, we denote x^i as the the embedding representing the input ingredient, $w = \{w^h, w^o\}$ as the hidden and output layer weights, $x^o = \{x_1^o, x_2^o, \dots\}$ as the one-hot output vectors representing the context ingredients, z_j^o as the value of the output layer for ingredient j, O as the number of context ingredients, λ as the L_2 regularization constant. The final form is produced below:

$$J_s(x, w) = L_s(x, w) + \lambda \left(\sum (x^i)^2 + \sum w^2 \right), \tag{3.1}$$

where

$$L_s(x, w) = -\log p(x_1^o, x_2^o, \ldots | x^i) \tag{3.2}$$

$$= -\sum_{j \in context(x^i)} z_j^o + O \cdot \log \sum_j \exp(z_j^o) \tag{3.3}$$

At each iteration, we take the gradient of the cost function and update the weights (and also the input vector x^i) as follows:

$$w = w - \eta \nabla J_s(x, w), \tag{3.4}$$

where η is the learning rate.

When doing batch training, we add up the cost function for each sample in the batch and return the mean cost. Note that we convert the input from an N-dimensional one-hot vector to an embedding of a lower dimension ($x^i \in \mathbb{R}^d$), with the precise value depending on the total number of ingredients being evaluated (typically $d \in [10, 100]$). The embeddings are initialized randomly (between -1 and 1). After training, the inputs x^i will be the vector representation for each ingredient.

Input sampling

Rather than take every ingredient in the product to generate the context (which can be both computationally intensive and ineffective), we simply take the top k ingredients in that product (from the ingredient list). Recall that the ingredients are sorted in order of decreasing weight. We find that the top k ingredients significantly outperform random k ingredients, probably due to the fact that the ingredients in the beginning are much more indicative of the type of product. Using all ingredients also does not perform as well, again most likely due to the fact that ingredients not in the top k ingredients generates more noise than additional information about the product. Lastly, it turns out that randomly sampling m ingredients to be the input for each product (rather than rotate every ingredient) speeds up training time while

not signficantly reducing performance. k changes based on the number of ingredients N.

Selective dropout

One issue with the data is the imbalanced distribution of the ingredients. Using the input sampling method from above, popular ingredients such as salt and sugar will have significantly more training points than ingredients that appear in very few products. For example, using $N = 5000$ ingredients, the most popular 100 ingredients occur more often than the remaining 4900 ingredients combined. Therefore, during training, the model will tend to overfit the embeddings for the popular ingredients and underfit the embeddings for the remaining ingredients. Therefore, we developed a method called *selective dropout* to account for this imbalance. We resample the data at each iteration according to the distribution of the ingredients and drop selected training points. The basic idea is that we drop data points more frequently for more popular ingredients in order to create a more balanced distribution. The algorithm is described in Algorithm 1. We use the parameter min_count as an expected upper bound to the number of times an ingredient can occur at each iteration.

Algorithm 1 Selective dropout

$min\_count \leftarrow 500$
$S_I \leftarrow \{x_1, x_2, \ldots, x_N\}$
for i in S_I **do**
    $p_i \leftarrow \min(1, min\_count/count(i))$
end for
for each iteration **do**
    $T = \{\text{all training points}\}$
    $T' = \emptyset$
    for $t \in T$ **do**
        $i \leftarrow$ input ingredient of t
        if $p_i > random(0, 1)$ **then**
            $T' \leftarrow T' \cup \{t\}$
        end if
    end for
    $train\_model(T')$
end for

Scoring

Without annotated data, the only way for us to determine how "good" a set of embeddings are is to apply the k-nearest neighbors algorithm on the embeddings and manually inspect the results. This can become very tedious when determining an optimal set of parameters. Therefore, we annotated the top 100 ingredients (by frequency) with ingredients that are related to one another (called neighbors). For example, 'soybean oil' is related to ingredients such as canola oil, soybean, and vegetable oil. Closer matches produce a higher score (e.g. ascorbic acid is closer to vitamin C than vitamin B). We can then compare the nearest neighbors of any embedding (ranked by their cosine distances) with the annotated neighbors to produce a score. In addition, we manually labeled the top 1000 ingredients with their category (e.g. 'apple' is a fruit, 'red 40' is an artificial color). Using our annotations, here are the four scoring metrics that we look at for parameter selection and evaluation:

1. Frequency that at least one of the annotated ingredients occurs in the top 3 nearest neighbors.
2. Average best (lowest) rank of annotated neighbors.
3. Average rank of all annotated neighbors.
4. Average rank of ingredients in the same category (e.g. all vegetables).

3.3.2 Results

We trained the embeddings using $N = 120, 1000, 5000$ ingredients. The results are shown in Table 3.1 and the parameters used are shown in Table 3.2. After training the model, we fed the ingredients back into the model and looked at the output. The outputs overwhelmingly favor the popular ingredients such as 'salt', 'water', and 'sugar'. This makes sense because those are the ingredients that most commonly appear in the context. There are a few exceptions, such as 'cocoa butter' with 'milk chocolate'. We now turn our attention to the learned embeddings.

	N=120	N=1000	N=5000
% found in top 3	69% (17%)	41% (2%)	44% (.4%)
Avg best rank	6 (15)	37 (125)	117 (626)
Avg rank of neigh	20 (60)	133 (500)	422 (2500)
Avg rank of cat	35	287	1341

Table 3.1: Applying the scoring metric for training using different number of ingredients (N). The numbers in parenthesis show the score if the neighbors are randomly chosen. Note that a low score/rank does not necessarily imply a bad embedding: if there are 5000 ingredients, there may be better neighbors for some of the ingredients that can replace the original neighbors at the top of the nearest neighbors list. In addition, an embedding that optimizes for just the top 100 ingredients will appear better than an embedding that optimizes for all 5000 ingredients. Therefore, it is still important to occasionally check the embeddings to make sure they are reasonable.

	N=120	N=1000	N=5000
η	0.005-0.1	0.005-0.1	0.005-0.1
λ	0.0005	0.0005	0.0005
m	10	20	30
d	10	20	30
n_epochs	8	25	25
batch_size	200	100	200
max_output_len	4	10	12
min_count	None	100	500

Table 3.2: The model parameters used for different values of N. The parameters are chosen from a simple grid search.

	skip-ing	word2vec	random
% found in top 3	69%	56%	17%
Avg best rank	6	10	15
Avg rank of neigh	20	36	60

Table 3.3: Score metric for $N = 120$ on three type of embeddings: those generated by the skip-ingredient model, those obtained from `word2vec`, and random embeddings.

Embeddings

To analyze the embeddings, we took the nearest neighbors (by cosine distance) for each ingredient. A sample is shown in Table 3.10. Two ingredients that frequently occur in the same products do not necessarily share similar embeddings: a linear regression with the actual co-occurrence probability shows no significant correlations. From inspection, it appears that near neighbors refer to the same semantic concept, which is exactly what we want. *Ingredients have similar embeddings if they have similar context.*

Next, we map the embeddings to a 2-dimensional space using t-distributed stochastic neighbor embedding (t-SNE). The top 1000 ingredients are color-coded by their categories and plotted in Figure 3-2. Note that we left out ingredients that we are not able to properly categorize, as well as ingredients/chemicals considered as "additives", since we feel that the category is too broad (and difficult to label into subcategories). A clear pattern emerges: the model was able to cluster ingredients based on their categories under no supervision.

Comparison with `word2vec`

As a baseline, we compare the embeddings we generated using the skip-ingredient model with those from `word2vec`. We took all the ingredients that can be found in the set of pre-trained words and looked at their nearest neighbors. The results on the scoring metric is shown in in Table 3.3. The t-SNE plot for `word2vec` is also shown in Figure 3-2. We can conclude that the embeddings generated by our model

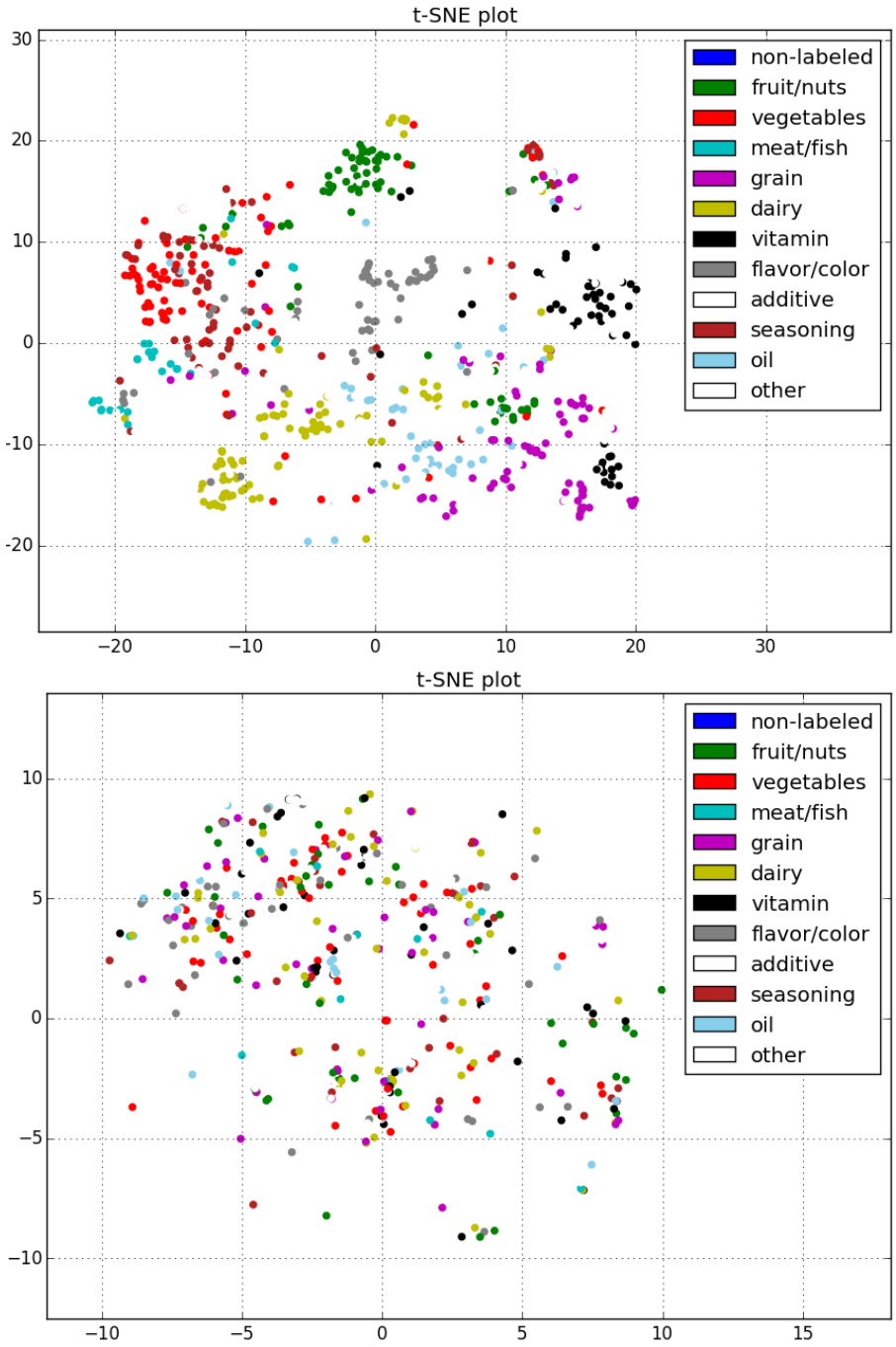

Figure 3-2: A t-distributed Stochastic Neighbor Embedding (t-SNE) plot of the skip-ingredient embeddings (top) and the `word2vec` embeddings (bottom). Note that the skip-ingredient embeddings have distinct clusters that belong to certain ingredient categories (e.g. vegetables and fruits). The multi-colored cluster in the top right corner is especially interesting: it turns out that every ingredient in that cluster is organic.

outperforms the pre-trained `word2vec` embeddings in terms of being able to cluster relevant ingredients.

Substitutions

We can use this model to perform a simple form of substitution, by replacing an ingredient with the nearest neighbor (e.g. replace 'milk' with 'skim milk'). Given a set of candidate substitution ingredients, we can also determine the ingredient that is closest to the target ingredient. While this may not work in all cases, it is a good start for determining substitutions.

Categorization of ingredients

Another task we can perform using these embeddings is to determine the category of an ingredient. For this task, we use the same list of 1000 ingredients that we annotated for the t-SNE visualization. For an unlabeled ingredient, we look at the k-nearest annotated ingredients and assign it the most frequent category. Using $k = 8$, we are able to obtain an accuracy of 70% on a test set of 100 ingredients. See Table 3.4 for an example of the predictions. If we take into account the category frequencies during labeling, we can likely further improve on this result. We now turn to the task of predicting the food category given the ingredients.

3.4 Predicting categories

3.4.1 Approach

Given a list of ingredients in a product, we want to predict its category. This can either be the aisle (16 choices), the shelf (128 choices), or the food category (1124 choices). We represent the input as a N-dimensional vector where index i is a one if ingredient i occurs in the input product, and zero otherwise. The output is a C-dimensional vector that denotes the probability distribution of the category, where C is the number of categories (e.g. 128 for shelf). This is a classification problem whose

Ingredient	*Predicted cat*	*Actual cat*
high fructose corn syrup	additive	additive
modified food starch	additive	additive
vitamin b2	vitamin	vitamin
enriched wheat flour	grain	grain
blue 1	flavor/color	flavor/color
nutmeg	seasoning	fruit/nuts
locust bean gum	dairy	additive
culture	dairy	dairy
turkey	meat/fish	meat/fish
dried cranberry	fruit/nuts	fruit/nuts
phytonadione	vitamin	additive

Table 3.4: Predicting category of unlabeled ingredients based on the nearest neighbors of the embeddings. Note that nutmeg can either be classified as a seasoning or as a fruit/nut. In addition, phytonadione (vitamin k) was wrong labeled as an additive, and correctly categorized by the model.

objective function can be defined below (as derived by Bishop [1]). The notation is similar to that used in Section 3.3. We introduce an indicator function y_c^o that is 1 when $x^o = c$ and 0 otherwise.

$$J_t(x, w) = L_t(x, w) + \lambda \left(\sum (w^h)^2 + \sum (w^o)^2 \right), \tag{3.5}$$

where

$$L_t(x, w) = \sum_{c=1}^{C} \big[- y_c \log(z_c^o) - (1 - y_c) \log(1 - z_c^o) \big] \tag{3.6}$$

We modify the neural network from the previous section to incorporate this new cost function. In addition, we apply a softmax in the output layer to generate a valid probability distribution for z^o.

3.4.2 Results

Mixture model

We first attempt to predict the categories using a baseline probabilistic model. In language topic modeling, we have a simple mixture model that predicts the most

likely topic z in a document of text:

$$\underset{z}{\text{argmax}} \prod_{i=1}^{N} p(w_i|z). \tag{3.7}$$

We can apply the same formula to our problem by replacing the words w_i with the ingredients x_i, and replacing the topic z with the food category c. N will be the number of ingredients in the product. After converting the problem to log probability, we obtain:

$$\underset{c}{\text{argmax}} \sum_{i=1}^{N} \log p(x_i|c). \tag{3.8}$$

This equation does very well on the training set, but does not generalize to unseen data. This is because of two problematic scenarios: 1) we encounter an ingredient not seen in training and 2) a seen ingredient has not occurred in category c previously. We solve the former issue by assigning all unseen ingredients a uniform prior distribution over the categories. The latter issue is dealt with by using *additive smoothing*, where each ingredient i is assigned the following probability:

$$p(x_i|c) = \frac{count(c,i) + \alpha}{count(i) + \alpha \cdot C}, \tag{3.9}$$

where $count(c,i)$ refers to the number of times ingredient i occurs in a product with category c, $count(i)$ is the total number of times ingredient i occurs, and C is the total number of categories.

Additive smoothing with a small α performs significantly better, as Figure 3-3 shows. Looking at the figure, we choose the optimal smoothing factor to be $\alpha = 1e{-}9$. Using this model for $N = 1000$, we are able to obtain an accuracy of 67.2%, 58.4%, and 43.0% for aisle, shelf, and food category, respectively.

Maximum entropy model

Next, we use the maximum entropy model (implemented as logistic regression in scikit-learn) to tackle this problem. For $N = 1000$, we obtain accuracies of 76.4%, 67.7%, and 45.5% for aisle, shelf, and food category, respectively (See Table 3.5).

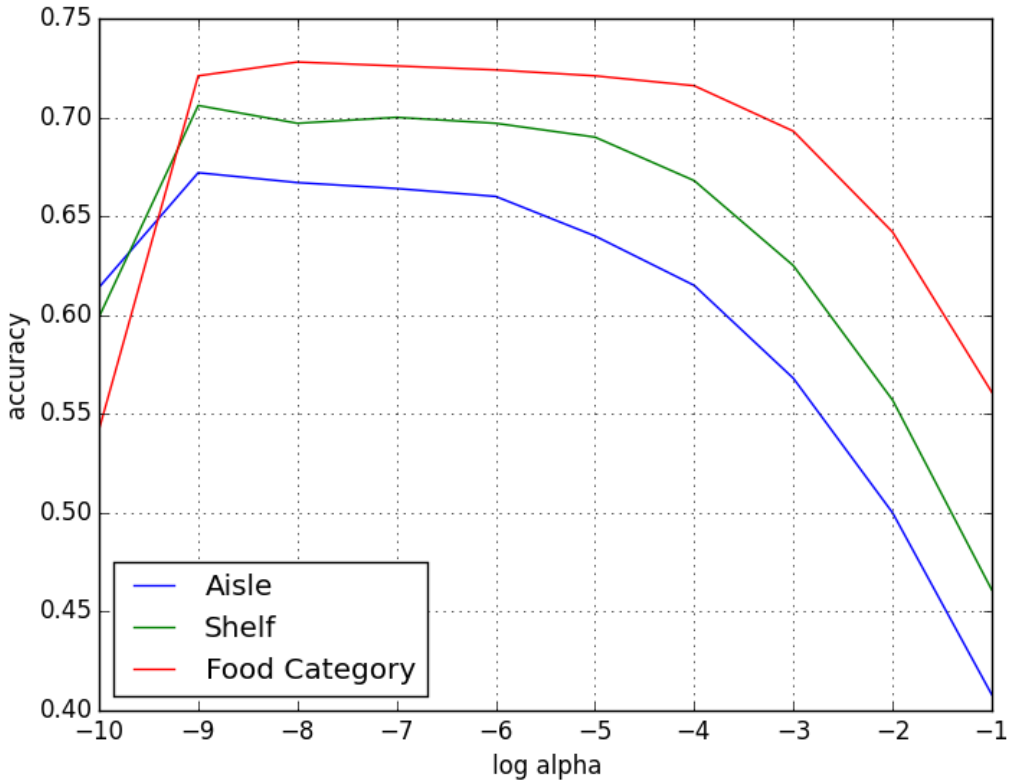

Figure 3-3: The effect of α on the accuracy of category predictions. All probabilities are the accuracies in predicting the aisle (e.g. using shelf to predict the aisle). The x-axis is in log scale. Note that the value for $x = -10$ actually represents no smoothing ($\alpha = 0$).

Neural network model

Lastly, we present the results of our neural network model. Again, using $N = 1000$, we get accuracies of 77.8%, 69.9%, and 50.3%. These results outperform both the maximum entropy model and the mixture model.

In addition, we can use the predictions for shelf to predict the aisle, and the predictions for food_category to predict the aisle and shelf (since all subcategories belong to the same super-category). The results is shown in Table 3.6.

model	aisle	shelf	food_category
mixture model	0.672	0.584	0.430
max entropy model	0.764	0.677	0.455
neural network	0.778	0.699	0.503
neural network (5k ings)	0.810	0.735	0.536

Table 3.5: Comparison of the various models used to predict the category given a set of ingredients. The first three models use $N = 1000$ ingredients, while the last model uses $N = 5000$ ingredients. Adding more ingredients to the neural network model improves performance.

	aisle	shelf	food_category
aisle	0.810	-	-
shelf	0.830	0.735	-
food_category	0.816	0.720	0.536

Table 3.6: We can also use subcategories (rows) to predict their parent categories (columns). However, this adds little additional predictive power, if any.

Generated embeddings

The embeddings generated by the categories are not as relevant as the embeddings generated by the skip-ingredient model, so we will not discuss it further here.

Using embeddings as input

Instead of using a length N one-hot vector to represent each ingredient, we want to try using the embeddings we generated from the skip-ingredient model. The result is presented in Table 3.7. The skip-ingredient embeddings perform slightly better than a randomized embedding for each ingredient. The embedding vector length is highly correlated with its performance, which makes sense: higher dimensions means more degrees of freedom for the model. It is interesting to note that random embedding performs better than the `word2vec` embeddings. A possible extension is to use the skip-ingredient model to generate embeddings of length 1000, and comparing that with the one-hot representation.

embedding type	d	accuracy
random	20	0.270
skip-ing	20	0.283
random	300	0.442
word2vec	300	0.373
random	1000	0.436
one-hot	1000	0.699

Table 3.7: In addition to the one-hot vector format (last row), we tried different other embeddings to represent the input ingredients used for predicting categories. The input dimension d is most correlated with prediction accuracy. However, given the same dimension d, using the skip-ingredient embeddings result in a higher accuracy than a random embedding, while the `word2vec` embeddings perform worse.

Predicting categories given a single ingredient

If we feed a single ingredient into this model, what category would the model predict this ingredient to fall in? We expected the model to output either the category with the most occurrences ($max(count(c_i))$) or the category with the highest percentage of occurrences ($max(\frac{count(c,i)}{N_c})$. But most of the time, this turned out to be not the case. For many cases, the model has learned what that singular ingredient represents: (milk $\rightarrow$ 'cream', yeast $\rightarrow$ 'baking additives & extracts', canola oil $\rightarrow$ 'vegetable & cooking oils', spices $\rightarrow$ 'herb & spices'). As a comparison, the categories with the most occurrences (and also highest percentage of occurrences) for yeast are "bread & buns' and 'pizza'.

3.5 Predicting valid combinations

3.5.1 Approach

To predict whether a list of ingredients form a valid combination, we use the same setup as Section 3.4.1. The output is 1 if the combination of ingredients is valid (i.e. can exist in a food product), and 0 if it is invalid. Since this is a classifier with two classes, we can use the same loss function as Equation 3.6.

We restrict the input to contain exactly the first k ingredients from the ingredient list. This is done to eliminate the need for normalization in the input space. In addition, since the ingredients are listed in order decreasing amount, we believe the first few ingredients possess the majority of the information about the product. In practice, we found that $k = 5$ works well. Increasing k leads to a higher accuracy, but less data (as there are fewer products with that many ingredients). For future work, we plan on removing this constraint.

3.5.2 Negative sampling

We now introduce our own version of negative sampling. Simply choosing k ingredients at random to create invalid combinations is insufficient: the simple maximum entropy model can generate an accuracy of 93%. This is because of the inbalanced data problem: the valid combinations contains ingredients that occur more frequently. Hence, "popular" ingredients are assigned a positive weights by the maximum entropy model. On the other hand, rare ingredients usually occur with invalid combinations, and are assigned negative weights. This accounts for the high accuracy in the maximum entropy model, but leads to little prediction power. Any combination of popular ingredients will result in a "valid" output by the model.

Therefore, we must generate invalid results differently. In addition to completely random combinations, we also generate invalid ingredients using the *same frequency distribution* as the valid ingredients. Therefore, if 'salt' appears in 10% of the valid combinations, it will also appear in roughly 10% of the invalid combinations. This forces our model to learn non-singular relationships in order to determine whether or not a combination is valid, since simply looking at an ingredient's popularity will not enable the model to differentiate between valid and invalid. We found that a 1:1 ratio of valid to invalid samples work well, with 95% of the invalid samples generated using this weighted methodology (the other 5% being random combinations of ingredients). Note that there is a trade-off in setting these parameters. For example, increasing the valid to invalid samples ratio will improve the valid accuracies at the expense

Model	N	V	I	I weighted
Max entropy	120	0.575	0.989	0.508
Max entropy	1000	0.612	0.976	0.441
Max entropy	5000	0.003	0.984	0.993
Neural network	120	0.942	0.914	0.844
Neural network	1000	0.861	0.968	0.950
Neural network	5000	0.877	0.956	0.936

Table 3.8: Comparison of the maximum entropy and neural network models on predicting valid/invalid combinations. The accuracy is broken down on three datasets: the valid (V) set, the invalid (I) set, and the weighted invalid (I weighted) set. Note that while maximum entropy can easily reject the invalid set, is unable to distinguish between the valid and weighted invalid sets. The neural network model performs well on all three datasets. Adding more ingredients does not seem hurt the accuracy.

of invalid accuracies. We chose the parameters such that the model outputs similar accuracies across all three datasets (valid, invalid, invalid weighted).

3.5.3 Results

The results are presented in Table 3.8. Similar to the previous section, we compare our neural network model with the maximum entropy model.

Maximum entropy model

After negative sampling is applied, the maximum entropy model performs similar to or worse than random for either the valid or the weighted invalid combinations. This does not change when we adjust the various parameters. We conclude that this model is unable to incorporate higher order relationships between the ingredients.

Neural network model

The neural network model performs significantly better across all datasets. Even though the ingredients are drawn from the same probability distribution, the model is able to differentiate between the valid and weighted invalid datasets relatively well. The exact mechanism behind the predictions has yet to be analyzed.

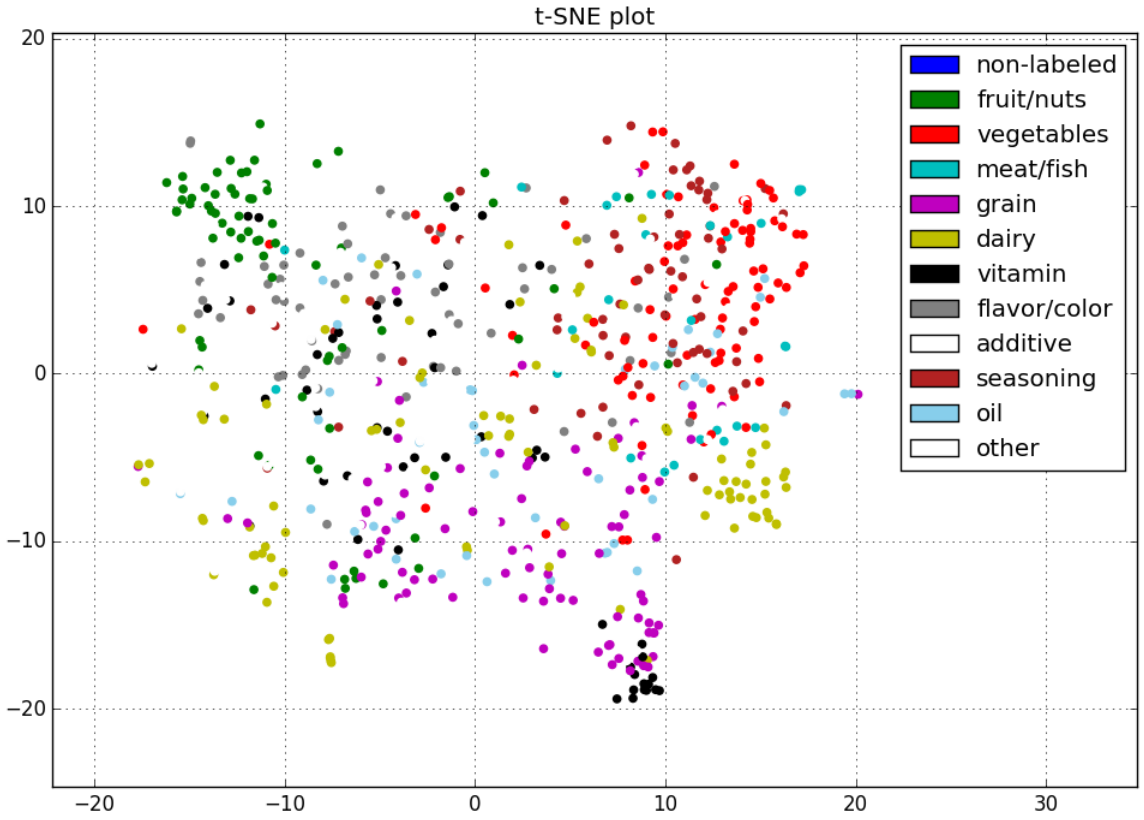

Figure 3-4: t-SNE visualization of the embeddings trained using the valid/invalid model. While the clusters are not as tight as those in the skip-ingredient model, they are still easily distinguishable.

Generated embeddings

The ingredient embeddings generated by the neural network, represented as the weights from the input to the hidden layer (w^h), are quite reasonable, as shown by their nearest neighbors in Table 3.11. Using our scoring function, the embeddings perform almost as well as those generated by the skip-ingredient model. The t-SNE visualization is shown in Figure 3-4. The fact that the model is able to cluster similar ingredients is quite interesting, since at no point during training did we isolate a particular ingredient (contrary to the skip-ingredient model).

embedding type	d	V	I	I weighted
random	20	0.640	0.609	0.654
skip-ing	20	0.720	0.821	0.842
random	300	0.836	0.857	0.808
word2vec	300	0.845	0.851	0.759
random	1000	0.848	0.794	0.850
one-hot	1000	0.861	0.968	0.950

Table 3.9: In addition to the one-hot vector format (last row), we tried different other embeddings to represent the input ingredients for the valid/invalid model. The input dimension d is correlated with prediction accuracy. However, given the same dimension d, using the skip-ingredient embeddings result in a higher accuracy than a random embedding, while a similar improvement is not present for the `word2vec` embeddings.

Using embeddings as input

We try using the embeddings we generated from this model as input, in a similar manner as Section 3.4.2. The result is presented in Table 3.9, and mirrors the result from Section 3.4.2.

Substitutions

We can take valid combinations of k ingredients and substitute $k' \in (1, 2, \ldots, k)$ ingredients. As k' increases to k (more substitutions), the model shifts from outputting overwhelmingly valid to overwhelmingly invalid. When we substitute invalid combinations, the model continues to output overwhelmingly invalid. This result makes sense intuitively. We cannot currently check the precise accuracy of these substitutions due to the lack of annotated data. This area will be further explored in future work.

Additions/removals

In addition to substitutions, we tried adding and removing ingredients from the k-ingredient combinations for $k = 5$. When adding ingredients, we add random ingredients. When removing ingredients, we randomly remove ingredients currently in the combination. The percentage of inputs predicted as valid is shown in Figure

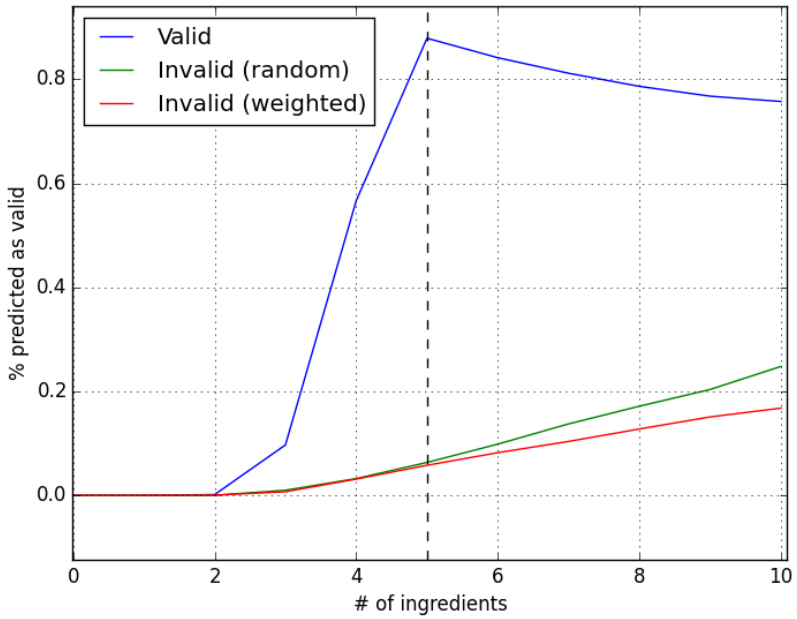

Figure 3-5: Starting with 5 ingredients, we add and remove up to 5 ingredients and plot the percentage of combinations the neural network model labeled as 'valid'. The graph makes intuitive sense: for valid combinations, adding additional ingredients will decrease probability of being marked 'valid', while the opposite is true for invalid combinations. As we remove ingredients, the model quickly labels all combinations as 'invalid', most likely due to the fact that the inputs are not normalized, and hence using a subset of ingredients as the input will never trigger the threshold.

3-5. Note that the model outputs all 1-ingredient and 2-ingredient combinations as 'invalid', which could be correct or incorrect depending on the definition of a valid combination. But as with substitutions, it is difficult to determine the validity of the results without annotated data. However, we can certainly improve the model in future work by incorporating training data consisting of different lengths.

3.6 Predicting unseen ingredients

There will be cases where we are given an ingredient that has not been seen in the training data. This is especially relevant in the cases of adulterants, which are

(obviously) not present on the ingredients list. Using the UMLS database described in Section 2, we can look up the properties of ingredients not seen during training.

3.6.1 Mapping unseen ingredients to embeddings

We learn a mapping between the property-level representation and the ingredient embeddings. That idea is that we look for ingredients in the training data that have similar properties as the unseen ingredients. As shown by Herbelot and Vecchi [3], one way we can learn the mapping is to apply a partial least squares (PLS) regression. We decided to use a k-nearest neighbor approach, as this approach performs similarly to PLS in the dataset used by Herbelot and Vecchi.

Now that we have a mapping, we can map any unknown ingredient to an embedding by first generating a property-level hierarchy representation and then applying the mapping. We generated the property-based representations for 1000 ingredients using this data. We then took 1000 unseen ingredients and found the nearest neighbors in the seen ingredients based on the cosine similarities of the property-level representations. The results are shown in Table 3.12. We describe the hierarchies in greater detail in Chapter 5.

The models we introduced were able to successfully generate useful embeddings, predict the food category of a list of ingredients, and determine if a combination of ingredients is valid. This will help us in future work in characterizing illegal substances and predicting which food products they are likely to occur in. In addition, these results can have various implications in other fields whose datasets can be translated in a similar manner to fit these models (e.g. generating list of chemicals in a valid reaction). We next turn our attention to an alternative model to predicting ingredient/product category pairs: collaborative filtering.

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
cream	cultured skim milk	skim milk	sour cream
modified corn starch	autolyzed yeast	modified cornstarch	monosodium glutamate
garlic powder	spice	onion powder	dehydrated onion
sodium phosphate	smoke flavor	smoke flavoring	sodium diacetate
vegetable oil	canola	oil	cottonseed
iron	niacin	thiamine	ferrous sulfate
baking soda	sodium bicarbonate	monocalcium phosphate	ammonium bicarbonate
preservative	polysorbate 60	preservatives	sodium propionate
beef	pork	mechanically separated chicken	sodium nitrite
yellow 6	yellow 5	red 3	yellow 5 lake

Table 3.10: Selection of ingredients and their nearest neighbors based on the cosine distances of the embeddings trained using the skip-ingredient model. The model is able to cluster the broader semantic meaning of each ingredient (e.g. beef is a meat, yellow 6 is a color).

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
cream	skim milk	milk	milkfat
modified corn starch	food starch-modified	modified food starch	confectioners glaze
garlic powder	spices	garlic	pepper
sodium phosphate	beef broth	part skim mozzarella cheese	pork
vegetable oil	corn oil	brown rice	canola oil
iron	ferrous sulfate	vitamin b1	riboflavin
baking soda	tapioca flour	organic dried cane syrup	granola
preservative	citric acid	potassium phosphate	sucralose
beef	mechanically separated chicken	pork	mechnically separated turkey
yellow 6	pistachio	red #40	ester gum

Table 3.11: Selection of ingredients and their nearest neighbors based on the cosine distances of the embeddings generated by the valid/invalid model. Note that even though we never trained each ingredient individually, the model was able to cluster the broader semantic meaning of each ingredient. Its performance seems to be on par with the embeddings generated by the skip-ingredient model.

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
vanilla flavor	organic vanilla extract	vanilla	organic vanilla
raisin paste	organic tomato paste	tomato paste	potato flake
dill	herb	organic basil	mustard
cheese sauce	sauce	worcestershire sauce	tomato sauce
green	red	artificial color	color
bleached flour	enriched bleached flour	corn flour	partially defatted peanut flour
sausage	pepperoni	ham	bacon
cane syrup	organic dried cane syrup	dried cane syrup	glucose-fructose syrup
organic almond	almond	tree nut	hazelnut
light tuna	fish	anchovy	sardines

Table 3.12: Given previously unseen ingredients, we can use the UMLS database to find the nearest neighbors in the seen ingredients. We can then use this new representation to make various predictions (such as the ones presented in this paper).

Chapter 4

Collaborative Filtering

Our task is to be able to identify likely adulterant, food product pairs. One way to do this is through a technique called collaborative filtering. Collaborative filtering is most popularly applied to the area of recommendation system, in particular for movies. Given a list of ratings that users have given for certain movies, the goal is to predict the ratings for movies the users have not seen. The movies with the highest predicted ratings are then recommended to the users. We will apply the similar logic for predicting adulterations. In this case, food products are the users and the adulterants are the movies. Instead of predicting movies that the user will like, this collaborative filtering method will predict adulterants that are likely to adulterate the food product.

To accomplish collaborative filtering, we implement a method called weighted alternating least squares (ALS), as presented in [17]. We refer readers to the paper for the exact algorithm details. To summarize, we try to find a low-rank matrix factorization of M into U and T that gives us:

$$M \approx Y = U \cdot V^T, \tag{4.1}$$

where M is the matrix where the i, j entry refers to the possibility of adulterant i occurring in product j. A value of $+1$ means that that this combination is definitely possible, while a value of -1 means that it is definitely not possible. Since we do

not know all the values of M, we use Y to approximate it. The goal is that for a given adulterant/product pair that does not have an entry in M, we can use Y to determine how likely this pair is.

4.1 Results

Using this approach, we are able to come up with a prediction for each product/adulterant pair. We used the RASFF dataset described in 2.2.1. Below are the steps we followed to set up the model:

1. From the 3,852 products and 464 adulterants, we filtered out all adulterants and products that only occur once. We are left with 185 products and 74 adulterants. There are 511 data points, which we denote as positive instances. This corresponds to a $\frac{511}{185 \cdot 74} = 3.7\%$ sparsity.

2. We filled out the matrix M (185×74) with those 511 data points labeled $+1$.

3. Because we do not have direct access to negative labels (pairs that cannot occur), we labeled 511 random points in M as -1. We assume those combinations to be negative instances.

4. We use the ALS algorithm iteratively to find a low-rank factorization of M into U $(185 \times r)$ and V $(74 \times r)$ (Equation 4.1). r is the rank. We found that $r = 5$ works best.

5. We use the entries in $Y = UV^T$ to approximate M.

A visualization of the matrices M and Y can be seen in Figure 4.1. We used a 60/20/20 training/validation/test split. The cross-validated results on the test set gives us a prediction accuracy of 57.7% on the positive $(+1)$ cases. We can apply one more "trick": we only make a prediction if the model is confident about the

Product	Substance	Actual	Predict
apricot jam	E210 - benzoic acid	True	True
canned vegetables	sulphite	True	True
herbal supplement	arsenic	True	True
grill spices	colour Para Red	True	False
pickles	sulphite	True	False
tuna	mercury	True	No guess
raisins	aliphatic hydrocarbons	True	No guess
pickled vegetable	colour E110 - Sunset Yellow FCF	False	True
tomatoes	colour Sudan 1	Unknown	True
spinach	nitrate	Unknown	True

Table 4.1: Sample predictions by the collaborative filtering model compared to the actual results.

prediction. This involves simply applying a threshold α over the matrix Y:

$$Y'[i,j] = \begin{cases} 1, & \text{if } Y[i,j] > \alpha \\ -1, & \text{if } Y[i,j] < -\alpha \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

Using an optimum threshold improves our accuracy to up to 70% for the values where the model outputs a prediction ($Y' \neq 0$). See Figure 4-1 to see the effects of a changing threshold. Table 4.1 shows a list of sample predictions.

So far, we have made our predictions based solely on the limited set of food/adulterant pairs of instances found in the RASFF database, in the absence any knowledge of what the products or chemicals are. For our next steps, we plan to improve our current method by adding features to the food products and chemicals. Similar to how information such as the genre, director, and lead actors of movies can be used to make better predictions, we can utilize various properties of food products and chemicals to make more accurate adulteration predictions. We explore that in the next chapter.

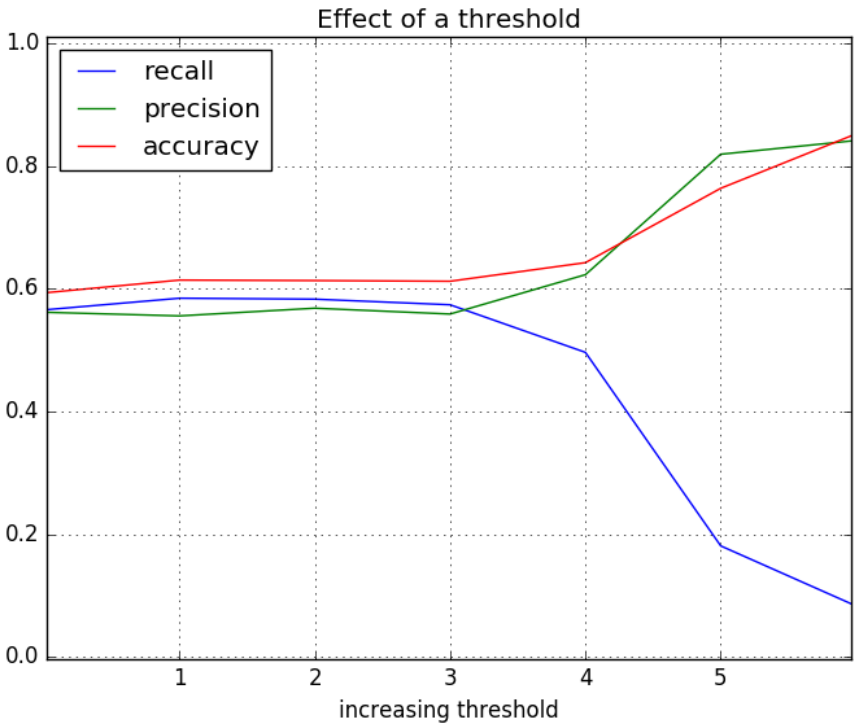

Figure 4-1: Effect of a threshold on evaluation metrics. As we increase the threshold, the accuracy and precision increases, at the expense of recall. Recall is the percentage of correctly identified positive cases. Precision is the percentage of positive predictions that are correct. Accuracy is the percentage of correctly identified positive cases where a prediction exist.

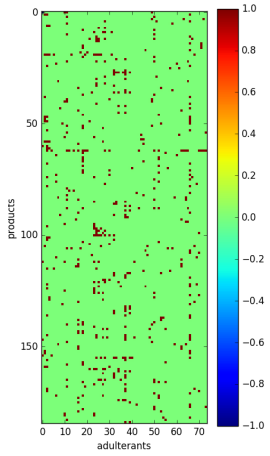

Figure 4-2: Training data showing only the positive (+1) instances of adulterant x with product y.

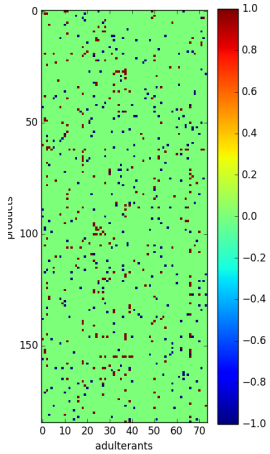

Figure 4-3: Matrix M showing positive (+1) & negative (-1) instances of adulterant x with product y.

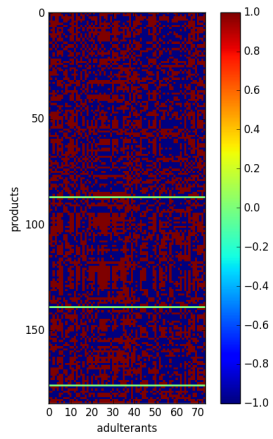

Figure 4-4: Matrix Y showing positive (+1) & negative (-1) predictions of adulterant x with product y.

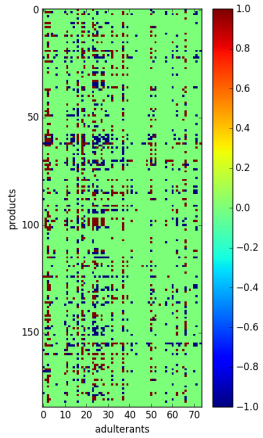

Figure 4-5: Matrix Y' showing positive (+1) & negative (-1) predictions after applying a threshold.

Chapter 5

Ingredient Hierarchies

For the remainder of this paper, we will focus on the following problem. Given an ingredient, our goal is to be able to predict the most likely product categories that this ingredient can occur in. Continuing from the previous example from Chapter 3, an input of "tomatoes" should tell us that "canned tomatoes" would be a very likely category, whereas "milk additives" would not be.

Up to this point, the model has no understanding of what any of the ingredients mean. We can label the ingredients numerically from 1 to N, and the model can generate the exact same results. Obviously, there are additional information about each ingredient that we can gather from the name. For example, we know that "milk" is a dairy product, "FD&C red #40" is a red dye, and "tartrazine" is a yellow dye. A human would be able to identify that "red #40" and "tartrazine" are more similar to each other than milk. Can we teach that to our model? In other words, we want to see if we can gather chemical/physical properties of the individual ingredients. We turn to the Unified Medical Language System (UMLS) Metathesaurus to extract the hierarchy relationship for each ingredient (Section 2.1).

The UMLS Metathesaurus contains hierarchy relationships aggregated from multiple sources. We limit our search of the relationships to four sources: Systematized Nomenclature of Medicine - Clinical Terms (SNOMEDCT), National Cancer Institute (NCI), National Drug File - Reference Terminology (NDFRT), and Management Sciences for Health (MSH).

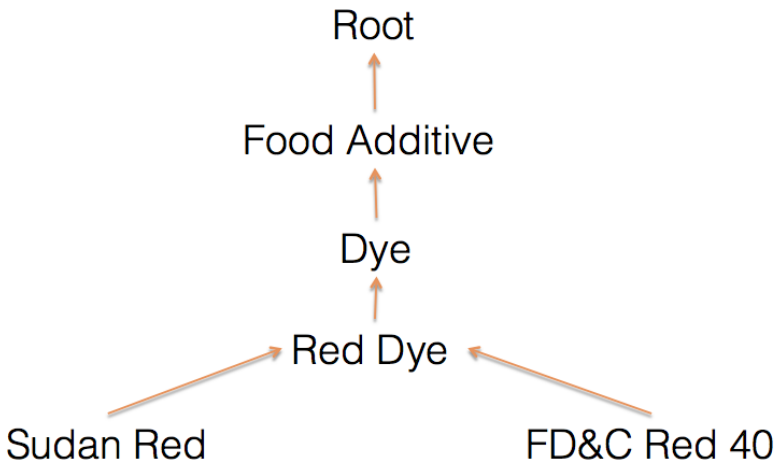

Figure 5-1: Sample hierarchy for red dyes. From the hierarchy, we can tell that Sudan Red and FD&C Red 40 are more similar than Sudan Red and say, a yellow dye.

For example, here is the hierarchy extracted for the ingredient "wheat flour":

`wheat flour` → `wheat product` → `wheat` → `grain` → `plant structure` → `plant material` → `organic natural material` → `natural material`

See Figure 5-1 for a visualization of a sample hierarchy.

The property-level representations of substances generated using the UMLS Metathesaurus are helpful in mapping a substance name to a vector representation that encodes the substance's chemical and physical properties. We want to leverage the fact that *similar ingredients tend to have similar hierarchies, which in turn means they tend to appear in similar food products.*

To generate a hierarchy representation for each ingredient, we used the following steps for each ingredient:

1. Assume that there are H unique nodes in the hierarchies for the entire set of ingredients. Each node is assigned an index from 0 to $H-1$. We initialize every ingredient to a zero vector of dimension H. We call this vector v. $H = 3751$ in our study.

2. We looked up the ingredient in the ULMS Metathesaurus. If the ingredient did not exist because it is too specific (e.g. extra virgin olive oil), we iteratively truncate the ingredient name until we are able to find a match (e.g. olive oil). If no match exists, we return the zero vector.

3. If a match exists, we extract the hierarchy for the ingredient. Note that multiple hierarchies can exist for the same ingredient. Assume there are k unique nodes. Let $I = i_1, \ldots, i_k$ be the indices that these k nodes correspond to. We assign $v[i] = 1, \forall i \in I$.

4. We normalize v to sum to 1.

5.1 Nearest neighbor algorithm

We try a simple nearest neighbors algorithm to predict the likely product categories for each adulterant.

We first generated the vector representations for the 5000 most popular ingredients from the FoodEssentials ingredient/food products database. Out of 424 adulterants from the RASFF database, we are able to generate a vector representation for 358 substances (84%). 249 (54%) have a direct match (no need to truncate the ingredient). For each of the 358 adulterants with a valid hierarchy representation, we found the 10 nearest neighbors (via the smallest cosine distances) from among the hierarchy representations of the 5000 ingredients. We looked at the most common food categories that these nearest neighbors occur in. From this we are able to create a probabilistic model that outputs the most likely categories for each adulterant, and the most likely adulterants for each category.

Table 3.12 in Chapter 3 shows the nearest neighbors for unseen ingredients calculated using the hierarchy representations. The high-level idea is that we want to find ingredients that are similar to the adulterant. Since we have more information about ingredients than adulterants, we can leverage information to predict the likely food categories for an adulterant (and vice versa).

Using this model, we obtained the top 5 categories for the possible adulterant nickel: meal replacement supplements, breakfast drinks, weight control, snack, energy & granola bars, and energy, protein & muscle recovery drinks. We can also predict adulterant from the category. Here are the top 5 possible adulterants for the product category "liquid water enhancer": sucralose, acesulfame k, succinic acid, citrate, and ethylcarbamate.

5.2 Neural network approach

We now try a more sophisticated model for making predictions: neural networks. Our neural network takes as input the vector representation of an ingredient/substance, as described earlier. The output is a softmax layer for the 131 possible food categories, with a score assigned to each of 131 categories. A higher score indicates a stronger relation between the input (the substance) and the output (the category). We can think of the score as the probability that given an ingredient, it is found in that product category.

For training, we use the empirical probability distribution of the product categories as the true output. For example, if an ingredient appeared 30 times in category A and 70 times in category B, and nowhere else, we take the true distribution to be 0.3 for category A and 0.7 for category B. Therefore, if the model were able to predict 0.3 for category A, 0.7 for category B, and 0 for all other categories, it would have a loss of 0. We use the cross-entropy loss function, similar to the loss introduced in Section 3.6.

Note that since we used a softmax layer over the food categories, the sum of the probabilities for each of the food categories must add to 1. Rather than giving an independent probability for each ingredient / product category pair, we take into account the entire distribution of food categories. If a particular ingredient / product category pair received a low probability, it could mean two things: 1) this is not a likely pairing or 2) there are not many products in this category that contain this ingredient, relative to all the other product categories. We believe this is a good

ingredient	category 1	category 2	category 3
wheat flour	bread & buns	cookies & biscuits	frozen appetizers
soybean oil	bread & buns	frozen appetizers	salad dressing
enzymes	cheese	bread & buns	frozen appetizers
milk	cheese	ice cream	yogurt
natural flavors	frozen appetizers	ice cream	candy
maltodextrin	energy drinks	snack bars	chips & snacks
high fructose corn syrup	ice cream	soda	cookies & biscuits

Table 5.1: Top 3 outputs of trained model on sample training data (ingredients)

ingredient	category 1	category 2	category 3
sorbitan tristearate	cookies & biscuits	snack bars	ice cream
tartrazine	candy	pickles	ice cream
dioxins	energy drinks	snack bars	soda
allura red ac	candy	soda	ice cream
aluminum sodium sulphate	frozen appetizers	snack bars	frozen dinners
brilliant blue fcf	candy	ice cream	cookies
ethyl-p-hydroxybenzoate	soda	pickles	soda

Table 5.2: Top 3 outputs of trained model on sample test data (adulterants)

representation since we want to bias our model to predict pairs that are both likely and have a large number of products.

The FoodEssentials ingredient/food product database has over a million training data points of ingredient / product category pairs. We convert each input to its property vector representation and assign each food category to a numerical index. We use a 60/40 training/validation split from this dataset. Finally, to test the model, we pass in the property vector representation of the adulterants from the RASFF database as the input. A sample of the results are shown in Table 5.1 and Table 5.2.

5.3 Evaluation

Recall that for each input ingredient, the model outputs a probability for each product category. We rank the product categories in descending probabilities. Of course, the category with the highest probability (rank=1) will be the most likely category for

this ingredient. We use two standard information retrieval (IR) metrics to evaluate this ranking: *mean average precision* (MAP) and *precision at N* (P@N). The mean average precision compares the predicted rankings with the true rankings (ordered by frequency of occurrence), and is calculated as follows:

$$MAP = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} \frac{j}{pred\_rank(c_{i,j})} \tag{5.1}$$

where i is the current ingredient, N is the number of ingredients in our dataset, j is the true rank, k is the total number of product categories that ingredient i has been found in, and $c_{i,j}$ is the product category corresponding to the true rank j for ingredient i. If the predicted rankings is the same as the true rankings, then it is easy to verify that the MAP will be 1.

Precision at N measures the percentage of the top N predicted products that have actually occurred in the true dataset, where N is the total number of food categories that the ingredient has appeared in. So for example, if an ingredient occurred in only categories A and B, and the model predicted A and C as the top 2 categories, then the P@N metric would be 0.5. If the model had predicted B and A, it would be 1.0. Mathematically, it is calculated as follows:

$$P@N = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{k} \sum_{j=1}^{k} \mathbb{1}(pred\_rank(c_{i,j}) \leq k), \tag{5.2}$$

where the variables are the same as those in Equation 5.1, and $\mathbb{1}$ denotes the indicator function. Note that in the above formula, N is the total number of ingredients. Rather, we use k to denote the N in P@N in order to maintain the same variables from Equation 5.1, which makes this a slight misnomer. Again, if the predicted rankings is the same as the true rankings, or any permutation of it, the P@N will be 1.

Table 5.3 shows the evaluation metrics of our trained model. The baseline model uses the mean occurrence distribution of the food categories as the prediction rankings for every input. The inputs to the MLP were scrambled for the "MLP (random)"

Model	Training set	Validation set	Adulterants
Random	0.162 / 0.132	0.172 / 0.142	0.045 / 0.011
Baseline	0.347 / 0.317	0.357 / 0.327	0.042 / 0.007
MLP (random)	0.350 / 0.318	0.351 / 0.323	0.031 / 0.001
MLP (dim=100)	0.536 / 0.479	0.478 / 0.429	0.103 / 0.047
MLP (dim=5321)	0.666 / 0.597	0.497 / 0.446	0.078 / 0.019

Table 5.3: Comparison of the various models used to predict the category given an ingredient. The two metrics shown are the mean average precision (MAP) and precision at N (P@N), respectively.

model. We also used principal component analysis (PCA) to reduce the dimensionality of the hierarchy representations from 5321 to 100 in order to show that one does not need a high-dimensional input vector to perform similarly well. The MLP with full dimensions performs the best on the validation and adulterants datasets. There are several reasons why the scores for adulterants might be low:

- There are 131 categories and a much smaller fraction of them appear in the data compared to the ingredients. Therefore, this makes it harder for the model to make accurate predictions.

- The data is not completely representative of our goal: just because an adulterant/food category pair does not appear in our data does not mean this pairing cannot occur. It could be that this pairing is legal and thus not considered an adulterant. Therefore, a prediction made by the model that does not appear in the data does not necessarily mean that the prediction is infeasible.

- Unlike with the training data, we do not account for the prevalence of each data pair. In other words, we weigh each pair the same, regardless of how frequently or rare this pair is.

- There is less information available about adulterants in general.

To summarize, we have created a model that is able to predict the most likely food categories for an unseen substance using hierarchical properties of the substance. This model performs well as long as we are able to find the substance in the ULMS database. What if we are unable to find a corresponding entry? The next chapter discusses a possible solution using Wikipedia.

Chapter 6

Leveraging Text Data using Recurrent Neural Networks

Recurrent neural networks (RNN) were first developed for use in natural language processing tasks such as document similarity and machine translation. Their key strength is the ability to retain memory for a specific task, something traditional neural networks cannot do. This is done by connecting the output of the previous layer to the input of the current layer. This allows the RNN to process an arbitrary sequence of inputs, making it ideal for applications in language, images, and speech. Due to the inherit "deep" architecture of the RNN from having many layers, traditional backpropagation techniques do not work in this setting. This is because propagating the gradients often results in an exploding or vanishing gradient. There has been a lot of research done in recent years on how to combat this problem, the most famous of which is the long short term memory (LSTM) by Hochreiter & Schmidhuber [4]. We leave the details of the workings of a RNN and LSTM for another time. For this task, we can think of it as a black box that takes as input a piece of text and outputs a vector representation of that text.

One way to make predictions is to leverage text corpora to learn a representation for each ingredient using natural language processing techniques. We chose the Wikipedia corpus for this task, as an article exists for most ingredients in our dataset. Wikipedia contains an abundance of information and knowledge with regards to prac-

tically anything. Given a substance, such as an ingredient or adulterant, one can learn a lot about the substance by reading the associated Wikipedia article. One will also be likely to make somewhat informed predictions on which food products this substance can occur in. Note that any other source of text corpus could be similarly applied, and future work could focus on leveraging text sources outside of Wikipedia.

Rather than extract the entire article (which can be quite long), we only extract the summary text of the corresponding Wikipedia article, which is the top part of the article before the main content. On a high level, we use a recurrent neural network (RNN) as an encoder to map this summary text of the article into a vector representation. We can then use this vector representation to make predictions on the likely product categories. RNNs have been widely used in the natural language processing community in recent years to process text for tasks ranging from machine translation to topic modeling.

Machines do not inherently understand text. Therefore, we must first convert each word into a vector representation, or word embedding. The idea is that similar words will have similar representations, as determined by some distance metric (i.e. cosine distance). These word embeddings must be pre-trained using a large text corpus. For our work ,we use the GloVe pre-trained vectors (Common Crawl, 42 billion tokens, 300-dimensional) [11]. We also tried the Google News word2vec vectors [9], but they did not perform as well.

Figure 6-1 for a schematics of the RNN model. We use a modified version of the recurrent convolutional neural network model from [7]. While we do not focus on the details of the RNN model for this study, it is important to understand the inputs and outputs, as well as the overall process, which we present below:

1. We first tokenize the summary text.

2. We then map each token to its corresponding GloVe word vector. If the token is not found, we assign it a vector zeros.

3. We now have a vector of word vectors for each ingredient. This is the input to the RNN.

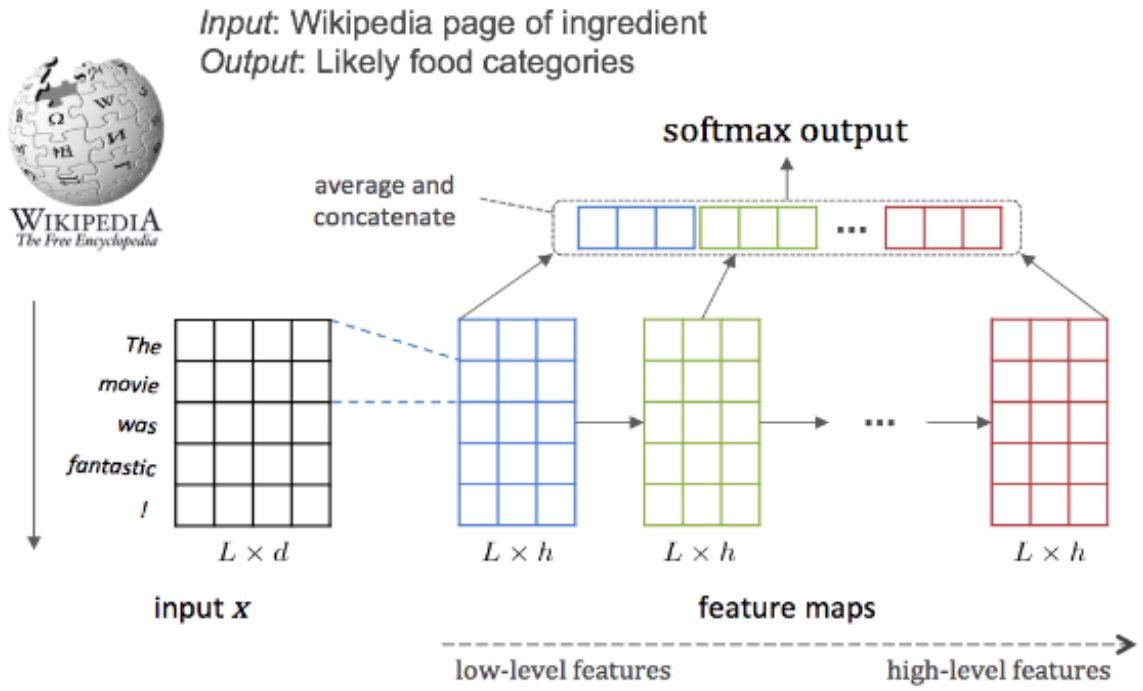

Figure 6-1: Illustration of the recurrent neural network model (RNN), modified slightly from [6] (used with permission). It takes as input the summary text of the Wikipedia article corresponding to an ingredient, and outputs a vector representation of this ingredient. In the intermediate layers of the model, several feature map layers map the input into different levels of feature representations. The stacked features are averaged within each layer and then concatenated. For this work, we can view the model as a black box.

4. The RNN performs mini-batch training. It groups ingredients with the same length (number of tokens) together in a batch. For each batch, the RNN takes in one word vector at a time and ultimately outputs a hidden representation for each layer.

5. All the hidden representations are concatenated to generate a final representation for each ingredient.

6. We also compute the representations for each product category. Similar to the ingredients, each product category has an associated Wikipedia text. We follow the same process as above.

7. We now have two matrices containing representations for each ingredient $(n \times d)$ and for each ingredient $(p \times d)$, where n is the number of ingredients, p is the number of products, and d is the hidden dimension.

8. We take the dot product of the two matrices to get a matrix of $n \times p$. Taking the softmax of this matrix gives us a prediction over the product categories for each ingredient. Section 6.1 will go over this in greater detail.

9. The model computes the cross-entropy loss and performs backpropagation. The process is repeated for the remaining batches. This completes one epoch of training.

6.1 Problem formulation

Let θ be the parameters of the RNN. We denote $v_\theta(x) \in \mathbb{R}$ as the RNN mapping of any text x into a vector representation, where d is the dimension of the hidden representation specified in θ. Let x_i and z_p be the Wikipedia pages for ingredient $i \in I$ and product category $p \in P$, respectively. We use the same parameters θ to generate the representations for both the ingredients and product categories. Therefore, $v_\theta(x_i)$ is the vector representation for ingredient i and $v_\theta(z_p)$ is the vector representation for product category p for an RNN model with parameters θ.

We can train the RNN model to predict the product categories given a particular ingredient:

$$P(p|i,\theta) = \frac{1}{Z_\theta} \exp\{v_\theta(z_p) \cdot v_\theta(x_i)\}, \tag{6.1}$$

$$\text{where } Z_\theta = \sum_{p' \in P} \exp\{v_\theta(z_{p'}) \cdot v_\theta(x_i)\} \tag{6.2}$$

Note that this is a similar formulation to that of a collaborative filtering model with user/item feature vectors produced by the RNN. The traditional CF model does not properly apply in this case, as we are predicting a probability distribution over all the product categories rather than a binary prediction for each category.

We define our loss function to be the cross-entropy loss from a multi-class prediction problem:

$$L(i,\theta) = -\sum_{i \in I} \sum_{p \in P} \hat{q}(p|i) \log P(p|i,\theta), \tag{6.3}$$

where $\hat{q}$ is the observed probability of the product category. Our objective is to minimize this loss.

6.2 Finding Wikipedia articles

We tried using the MediaWiki API to automate the process of generating the appropriate Wikipedia article for each ingredient by taking the top search result for each ingredient. This generates the incorrect article for many of the ingredients. For example "black-eyed peas" were redirected to the article about the band starring Will.I.Am et al.

Hence, we turned to Mechanical Turk and asked Turkers to find us the appropriate Wikipedia article for each ingredient. This gave us much better results. Over half the articles were different from our previous attempt using the API. Once we have the associated Wikipedia articles, we use the MediaWiki API to download the summary text.

The median word length of a Wikipedia summary text is around 50 words. For ingredients without a corresponding Wikipedia entry (or the summary text is under 5 words), we do not use it for training.

6.3 Experimental setup

6.3.1 Training, validation, and test sets

We split the ingredients dataset of 5000 ingredients into training and validation sets (2:1 split). We then use the adulterants dataset as the test set. Because the two datasets are not directly comparable, an extension would be to split the ingredients dataset into training, validation, and test sets, and then perform cross-validation.

6.3.2 Use of GPU

The RNNs are implemented on top of Theano, which means that we are able to leverage GPUs to aid us in our computation. Each epoch takes around 15 minutes to run on the GPU. If we run the model without the product vectors (Section 6.4.1), we can improve the speed by 15x. This is because the addition of the product vectors require us to compute the representation for each of the 131 products at *every* iteration.

6.3.3 Evaluation

Recall from Section 5.3 that we use the mean average precision (MAP) and precision at N (P@N) as our evaluation metrics. We will use the same two metrics to evaluate our RNN model.

While these two metrics have their strengths, they collectively lack one thing: a qualitative assessment. We hope to leverage the crowdsourcing capabilities of Amazon Mechanical Turk to develop our qualitative evaluation assessment in the future. For each ingredient/chemical, we take the top k ingredient/product pairs predicted by the model and ask "Turkers" to mark whether this is A) a reasonable pair that would likely be found together (e.g. 1% milk/yogurt) or B) not a reasonable pair (e.g. 1%

Parameter	Value
regularization (λ)	0.000001
learning rate	0.001
hidden dimension (d)	50
layers	1
batch size	5
n-gram	2
activation function	tanh
dropout	0.01
epochs	100

Table 6.1: Optimal hyperparameters for the RNN model.

milk/soda). We will then weigh this result with the rank of the pair so that more confident predictions will be rewarded (or penalized) appropriately.

For more sophisticated substances/chemicals, this task becomes more difficult. Our preliminary analysis shows that Turkers have difficulty identifying likely adulterant / product category pairs, as they can often appear in non-obvious scenarios. In this situation, we can enlist the knowledge of domain experts rather than Turkers, but the process would be identical. Combined with the quantitative evaluation metrics, this will provide us with a more comprehensive view of the performance of our model.

6.3.4 Hyperparameters

We use SGD with a mini-batch size of 5. For the RNN model, we used Adam [5] as the optimization method with the default setting suggested by the authors. We used a hidden dimension of $d = 50$ (while a higher d tend to lead to improved results, training time is much slower). For regularization, we used the L2 norm of the weights.

We performed an extensive search over the space of the hyperparameters. Table 6.1 shows the best setting we have found.

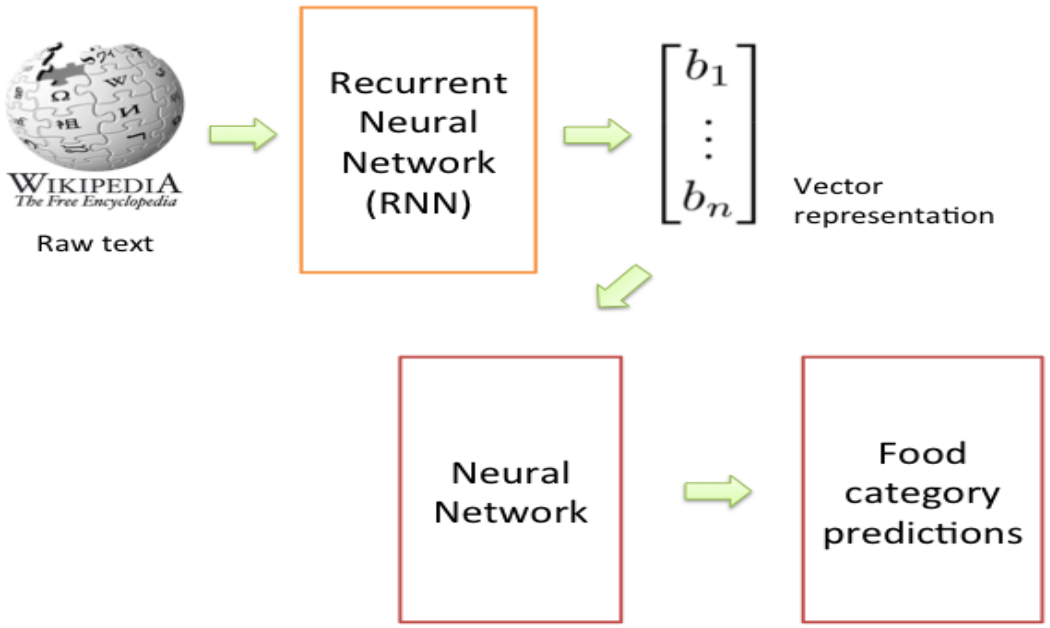

Figure 6-2: The RNN model process without incorporating product information.

6.4 Alternate versions

We also explored two other model architectures to tackle this problem. They are minor alterations to the existing model.

6.4.1 No product vectors

The model we described above assumes that we have also generated a vector representation of the products, $v_\theta(z_p)$. How will the model perform if we were to not use this representation in our prediction? In that case, we will just have a matrix of a vector of dimension $n \times d$ after Step 6 of the overall process from the beginning of the chapter, where n is the number of ingredients and d is the hidden dimension. This vector representation will then be passed to another layer of the neural network, which maps the $n \times d$ matrix to a $n \times p$ matrix. A softmax is finally applied to predict the likely food categories. This is a similar architecture as the one we used to make predictions in Section 5.2. This version can be summarized in Figure 6-2.

As we will show in the upcoming Results section, this version of the model does slightly better than the model we introduced above. This is likely because the final neural network layer contains additional parameters that can be tuned by the model, adding additional degrees of freedom compared to the previous model, where a dot product was used instead to map the hidden representation into a probability distribution. The benefit of the previous model was that we can use the ingredient and product representations to generate a sequential model, which we will present in the next chapter.

6.4.2 Augmenting with hierarchy representation

For each ingredient, at the final layer right before the softmax, we added the hierarchy representation from Chapter 5. We want to see if we can combine the Wikipedia article and the hierarchy representation to obtain a better overall representation of an ingredient, and hence lead to a better predictive model. Because of the high dimensionality of this representation (compared to the RNN representation), we used principal component analysis (PCA) to reduce the dimensionality to be equal to the dimension of the RNN representation. We then fed the concatenation of the two representations into a final softmax layer, similar to the above version with no product info.

As we will show in the next section, we have yet to tune this version of the model to outperform the "normal" RNN model.

6.5 Results

Table 6.2 shows evaluation metrics of the RNN model. The metrics are described in Section 6.3.3. The random model generates a random ranking of food categories for each ingredient. The baseline model uses the mean occurrence distribution of the food categories as the prediction rankings for every input. The MLP model is the model from Section 5.2. The current RNN model outperforms all other baselines in the validation and test sets. See Table 6.3 for a list of sample predictions.

Model	Training set	Validation set	Adulterants
Random	0.162 / 0.132	0.172 / 0.142	0.045 / 0.011
Baseline	0.347 / 0.317	0.357 / 0.327	0.042 / 0.007
MLP	0.666 / 0.597	0.497 / 0.446	0.078 / 0.019
RNN (no products)	0.617 / 0.552	0.539 / 0.481	0.168 / 0.074
RNN (hierarchy)	0.647 / 0.578	0.522 / 0.468	0.100 / 0.017
RNN (d=200)	0.638 / 0.573	0.539 / 0.483	0.157 / 0.051
RNN	0.610 / 0.545	0.528 / 0.475	0.161 / 0.060

Table 6.2: Comparison of the various models used to predict the category given an ingredient. The two metrics shown are the mean average precision (MAP) and precision at N (P@N), respectively. We compare the RNN model with the two alternate versions described in 6.4, as well as a version with $d = 200$. Using a higher dimension performs better at the expense of training time.

Figure 6-3 is a visualization of the probability distributions of food categories for four ingredient inputs. Different ingredients correspond to different food categories of high interest.

We have showed that our deep neural network model performs significantly better than both a random prediction and a baseline prediction based on the occurrence frequency of the categories. We note that that the model is able to make these predictions based only on the Wikipedia entry for each ingredient. Looking at the sample predictions, the model does quite well on ingredients with a thorough Wikipedia entry. This is a promising result that opens the door for new research directions. Our next step is to incorporate more information into the model, such as combining the hierarchical information used in the MLP model. We expect the performance of this model to increase.

Ingredient	*Wikipedia article*	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
oatmeal	Oatmeal	cereal (0.564)	snack, energy & granola bars (0.196)	breads & buns (0.039)
watermelon juice	Watermelon	fruit & vegetable juice (0.352)	ice cream & frozen yogurt (0.205)	yogurt (0.064)
jasmine rice	Jasmine rice	flavored rice dishes (0.294)	rice (0.237)	herbs & spices (0.062)
shrimp extract	Shrimp (food)	fish & seafood (0.491)	frozen dinners (0.128)	frozen appetizers (0.113)
meatball	Meatball	pizza (0.180)	breakfast sandwiches (0.128)	frozen dinners (0.120)
polysorbate 80	Polysorbate 80	chewing gum & mints (0.531)	candy (0.092)	baking decorations (0.049)
ketchup	Ketchup	ketchup (0.461)	salad dressing & mayonnaise (0.049)	other cooking sauces (0.044)
benzoic acid	Benzoic acid	powdered drinks (0.062)	fruit & vegetable juice (0.051)	candy (0.045)
sibutramine	Sibutramine	energy drinks (0.193)	specialty formula supplements (0.149)	herbal supplements (0.092)
nitrite	Nitrite	sausages, hotdogs & brats (0.310)	pepperoni, salami & cold cuts (0.257)	bacon, sausages & ribs (0.057)
tadalafil	Tadalafil	energy drinks (0.522)	soda (0.141)	formula supplements (0.064)

Table 6.3: Sample predictions generated by the model on eight unseen ingredients and three adulterants. The number in parenthesis represents the probability provided by the model.

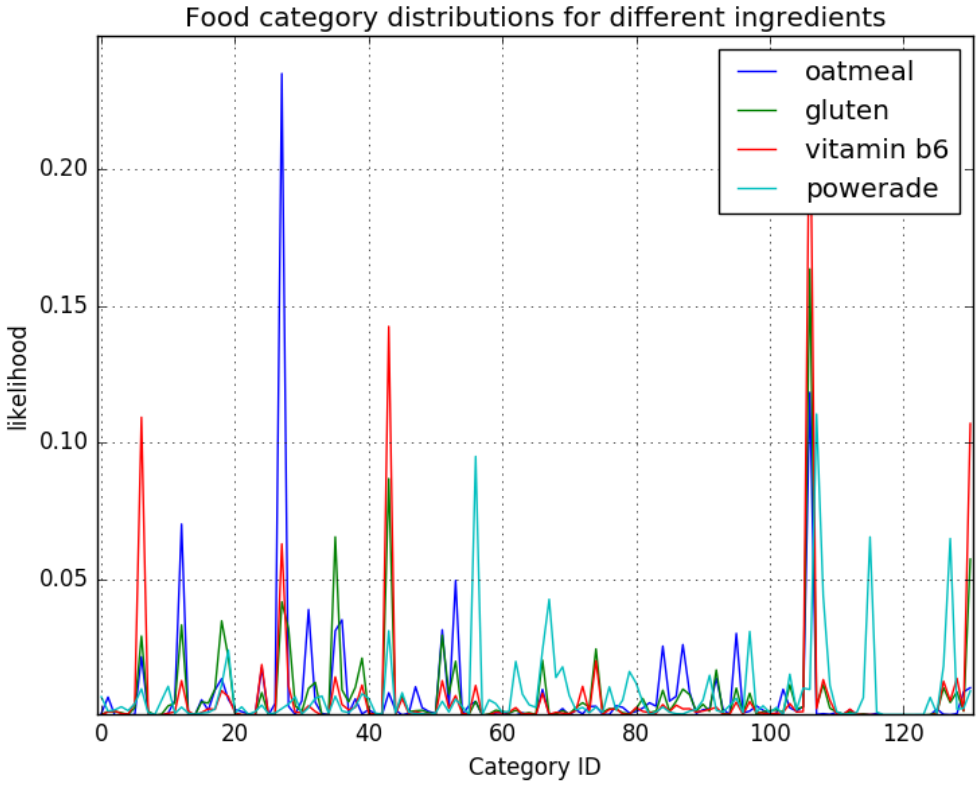

Figure 6-3: Visualization of the probability distributions of food product categories for four ingredient inputs.

Chapter 7

Sequential Refinement

The RNN model, once trained, would be able to predict a probability distribution over the product categories for any new or existing ingredient as long as a text representation exists. However, this method does not provide an obvious way of adjusting predictions after observing only a few instances. We therefore explore the scenario where we are given a new ingredient and a few instances of their occurrence. Can we update the model in a sequential or online fashion such that we can continually improve its performance with each successive data point?

We introduce parameters $w = [w_1, \ldots, w_d], w_j \in \mathbb{R}^+$ that will be adjusted in an axis parallel manner:

$$P(p|i, \theta) = \frac{1}{Z_{\theta,w}} \exp\{v_\theta(z_p)^T \text{diag}(w) v_\theta(x_i)\}, \tag{7.1}$$

$$\text{where } Z_{\theta,w} = \sum_{p' \in P} \exp\{v_\theta(z_{p'})^T \text{diag}(w) v_\theta(x_i)\}, \tag{7.2}$$

and $\text{diag}(w)$ is a diagonal matrix with the entries specified by w. We assume the RNN parameters θ and the vector representations $v_\theta(z_p)$ and $v_\theta(x_i)$ to be fixed at this stage. We will update the feature weights w in an online fashion in response to each observation for a new ingredient.

Because we imagine it is possible for each new ingredient to have only a few observations, it is important to properly regularize the weights as to avoid overfitting.

Ideally, we want to keep each parameter w_j as close to 1 as possible. A reasonable choice of regularizer is a Gamma distribution that is the conjugate prior to the inverse variance:

$$reg(w) = \lambda \sum_{j=1}^{d} [\log(w_j) - w_j + 1] \tag{7.3}$$

where λ is the overall regularization parameter[1]. We now incorporate the regularizer in our loss function:

$$L(i, w) = -\sum_{k=1}^{n} \log P(p_k|i, w) - reg(w), \tag{7.4}$$

where n is the total number of observed categories up to this point: $p_1, \ldots, p_n$. It can be verified that this loss function is convex, and thus allows us to use a variety of powerful tools and techniques as described in lecture. In our study, we run stochastic gradient descent (both online and batch) to minimize the loss. Note that each new ingredient must be trained independently and assigned its own feature weights.

For each ingredient, we wish to minimize the loss function specified in 7.4 by adjusting w. We will apply stochastic gradient descent (SGD) to update w:

$$w := w - \eta \nabla L_k(i, w), \tag{7.5}$$

where η is the learning rate and $L_k(i, w)$ refers to the loss given by the last k observed product categories. True SGD is performed when $k = 1$, normal gradient descent is performed when $k = n$, and mini-batch is performed when $1 < k < n$.

w is initialized as a vector of ones. We continuously adjust w through the above process until we have completely processed all of the data for a particular ingredient, in which case we start over for a new ingredient.

[1]Another possible regularizer is to use the L2 norm: $\lambda \sum_j (w_j - 1)^2$.

Parameter	Meaning	Value
λ	regularization	0.01
η	learning rate	0.1
α	naive multiplier	0.02
z	mini-batch size	10

Table 7.1: Optimal hyperparameters for the online learning model.

7.1 Experimental Setup

We use the RNN model in Chapter 6 to generate our vector representations $v_\theta(x_i)$ and $v_\theta(z_p)$. Refer to Section 6.3 for the experimental setup of the RNN model.

7.1.1 Evaluation

We evaluate our various models using the loss function from Equation 7.4, without the regularization term. The loss is updated at each iteration of the algorithm using the w at that point in time.

7.1.2 Hyperparameters

Since we used a hidden dimension of $d = 50$, this also corresponds to $|w|$ for the online feature weights. Though $d = 200$ performs better, we show all of our results for $d = 50$ because the faster computational time allows us better iterate through various parameterizations.

There are several parameters we need to tune for online learning, and they are presented in Table 7.1. Similar to Homework 1, we use an adaptive learning rate: $\eta = \frac{\eta_0}{\sqrt{t}/100}$. To avoid numerical issues during training, we limit the feature weights to lie in $w_j \in [10^{-8}, 2]$.

Algorithm	Loss
True	1.18
Uniform	4.88
Baseline	3.67
Naive	3.64
Batch all	3.57
Mini-batch	3.11
Online	3.13

Table 7.2: The mean loss over all the ingredients in the validation set (1666 ingredients) for the various methods. Mini-batch with $z = 10$ and online updates ($z = 1$) performed the best, on average.

7.1.3 Simulating online scenario

For each ingredient, we have a sequence of product categories where this ingredient is observed. In the online setting, pass the sequence one at a time to the model. There are two ways we generate this sequence of product categories:

Use true counts of product categories: In this scenario, for each ingredient, we used the true counts of the product categories it has appeared in. In other words, if 'sugar' appeared in 'soda' 200 times, 'Soda' would appear in the sequence 200 times. We randomly permute the sequence to generate an iid sequence based on the true distribution. This method more accurately mirrors the real-world scenario.

Generate counts according to true distribution: The previous method of sequence generation results in an uneven distribution of the number of product categories for each ingredient. A popular ingredient such as 'salt' would have a sequence of over 20,000 observations, while an ingredient such as 'half & half' would only have 20. By generating a fixed length sequence of n observations based on the category occurrence distribution, we can eliminate any issues in training associated with varying-length sequences. While all of the results presented assume the previous scenario, we found that the mean loss is lower in this scenario with $n = 1000$, most likely due to the fact that the median sequence length is around 50.

Algorithm	*Ingredient*	*Loss*	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
True	oatmeal	1.75	cookies & biscuits (0.373)	chili & stew (0.294)	breads & buns (0.108)
Baseline	oatmeal	5.39	cereal (0.564)	granola bars (0.196)	breads & buns (0.039)
Naive	oatmeal	5.35	cereal (0.561)	granola bars (0.197)	breads & buns (0.039)
Batch all	oatmeal	5.61	granola bars (0.390)	frozen appetizers (0.093)	pasta (0.076)
Mini-batch	oatmeal	4.55	cereal (0.784)	breads & buns (0.091)	granola bars (0.037)
Online	oatmeal	3.84	cookies & biscuits (0.292)	cereal (0.252)	granola bars (0.148)

Table 7.3: Sample predictions generated by the model on the ingredient "oatmeal" at the conclusion of the sequence (102 observations). The number in parenthesis represents the probability provided by the model. In this example, the online method significantly outperformed all the other methods. Note that the online method tended to "nudge" the predictions towards the true predictions. The effect will likely be more pronounced with an increased dimension size d and less regularization.

7.2 Results

We compared the loss of our online algorithm to several other baseline metrics:

- True - We use the true distribution for each prediction.
- Uniform - We use a uniform distribution for each prediction.
- Baseline - We use the (same) probability distribution from the RNN, $P(p|i, \theta)$, for each prediction.
- Naive - Starting with the baseline distribution, at each step, we multiply the probability of the observed category by $(1 + \alpha)$.
- Batch all - At each step, we use the entire observed sequence to perform gradient descent and update w.
- Mini-batch - At each step, we use a mini-batch size of z to perform gradient descent and update w.
- Online - At each step, we use only the current observation to perform stochastic gradient descent ($z = 1$).

See Table 7.2 for a comparison of the different algorithms in the online setting. For sample predictions for the ingredient "oatmeal", see Table 7.3. We tuned our

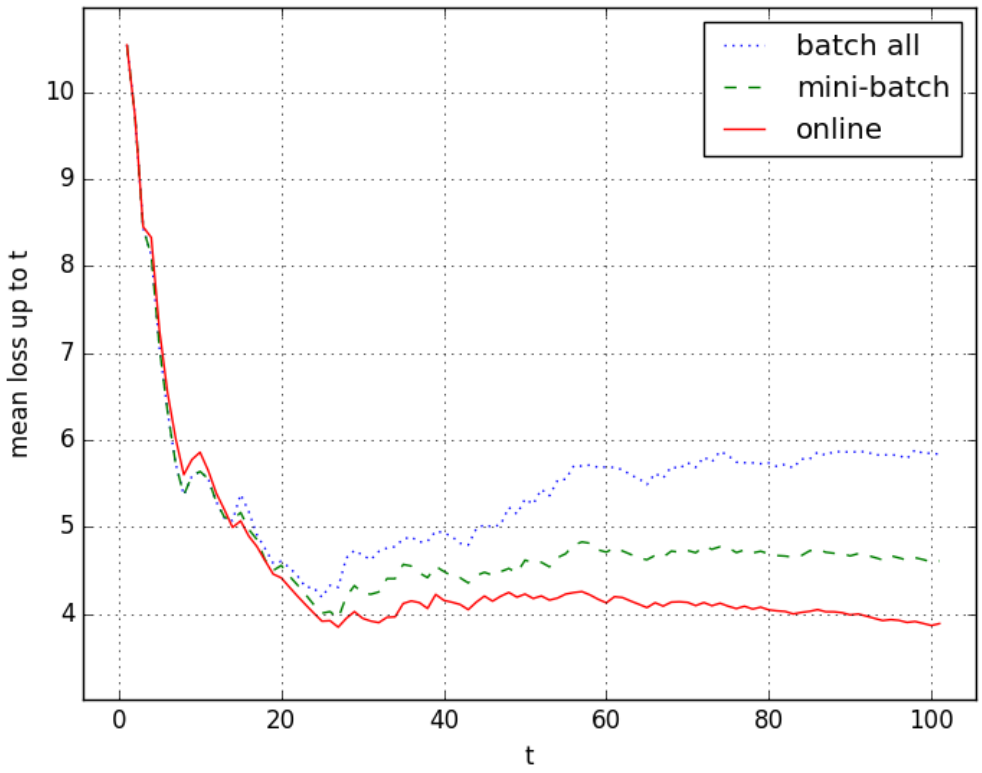

Figure 7-1: The running mean of the loss over time using the online algorithm for ingredient "oatmeal". The online method performs the best in this particular instance.

parameters on the training set and reported our results on the validation set. Running gradient descent on the entire set of seen observations result has decent performance when the step size is small. It is by far the slowest, especially as the sequence size increases. SGD with just the latest observation can become rather unstable at times, but performs fairly well on average and is very fast. Updates with a fixed mini-batch size seem to offer the best of both worlds.

We show the loss over time for the ingredient 'oatmeal' in Figure 7-1. We also show the effect of the sequence length on the improvement in loss (over the baseline). See Figure 7-2 for a scatter plot of this effect. Figure 7-3 shows the correlation between the true loss and the improvement in loss. It seems counter-intuitive that the larger the true loss, the harder it becomes for an online algorithm to improve it.

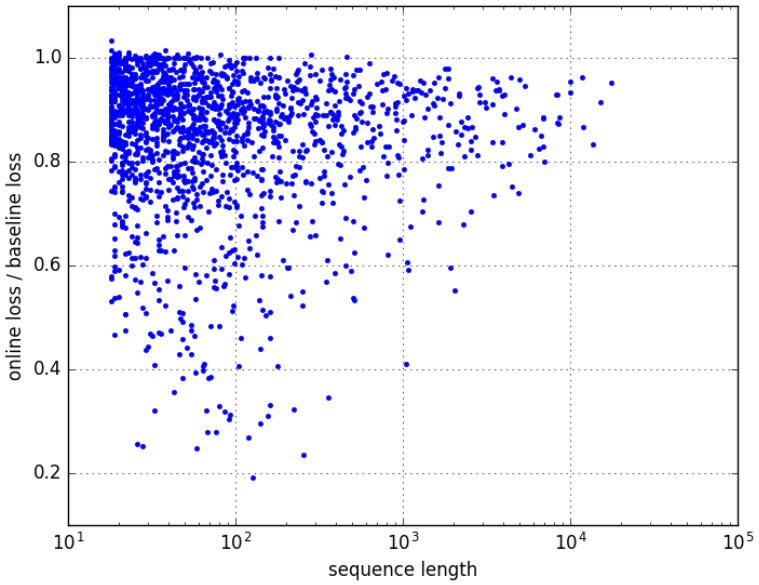

Figure 7-2: Plot showing the relationship between the sequence length and the improvement in loss (online loss / baseline loss). We use a log scale on the x-axis for the sequence length. An interesting pattern appears, though there seems to be no strong linear correlation.

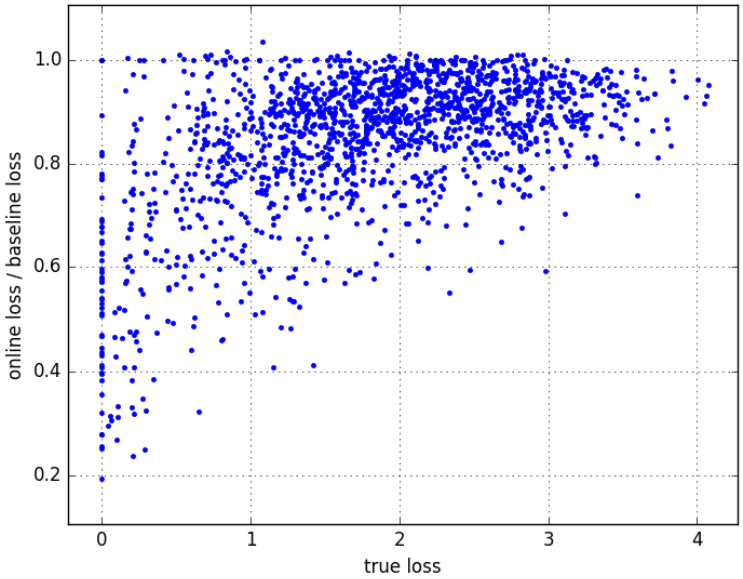

Figure 7-3: Plot showing the relationship between the true loss (using the true distribution for predictions) and the improvement in loss (online loss / baseline loss). It seems that as the true loss increases, it is also more difficult for the online algorithm to improve upon the true loss.

Chapter 8

Conclusions

We successfully demonstrate a new application for deep neural networks by building an automated model to predict likely food product categories from a given ingredient. We are able to show the effectiveness of neural networks when supplied with different input representations of ingredients (hierarchical properties and text data). As far as we are aware, this is the first study of its kind on learning representations for ingredients and applying them for various prediction tasks. All models are also extended to the case of adulterants.

8.1 Contributions

Below, we show the main contribution of each chapter in this work:

2. Collected comprehensive datasets on a) properties of ingredients and food products and b) known cases of adulterant-food product pairs.

3. Generated meaningful representations of ingredients based only on neighboring ingredients in the same products (the skip-ingredient model).

4. Demonstrated the use of collaborative filtering techniques to predict the likelihood of a given adulterant and food product.

5. Incorporated the hierarchical properties of ingredients and adulterants to predict likely food product categories using neural networks.

6. Introduced the use of recurrent neural networks to generate meaningful vector representations of ingredients and adulterants from their Wikipedia pages; used these representations to predict likely food product categories.

7. Showed the use of sequential learning techniques to incrementally improve the model with each additional observation.

8.2 Future work

While our primary goal is to be able to make predictions about adulterants, a lot of the work done in this paper is based on standard food ingredients. As mentioned before, this is primary due to the fact that we have significantly more data for food ingredients and in what food products they occur in that for adulterants. We have yet to find a comprehensive and centralized database with information about adulterants. It would be extremely helpful for the field of food safety to develop a centralized resource with knowledge about various adulterants and where they were found. This would also help us better train and evaluate our existing models. With that said, we are confident that the techniques we have developed in this work can be smoothly applied to any data about adulterants.

We want to stress that everything presented in this project is a "first steps" attempt. We are nowhere close to exhausting the tricks we can apply to improve our predictions. Because this is a novel application, our task for this work was not to squeeze every ounce of performance out of our models, but rather show promising directions for which future researchers can pursue. It is entirely possible for us to sit down, tweak a few additional parameters (such as increasing the hidden dimension size), and get a 2% increase in the mean average precision overnight. Just as how Rome was not built in a day, we also do not plan to completely establish the boundaries of our models. We do hope, however, that we have laid the foundation for future researchers interested in applying machine learning models to problems in food safety. We hope our work will provide a source of inspiration to them and many others.

Another future goal is to be able to convert our research into a format that is more readily accessible by the general public. Right now, the only way to replicate the results presented in this work is to download the raw code and run it yourselves. Of course, you would also have to manually obtain the data as well. We hope to be able to create a web interface that allows anyone to be able to enter ingredients and adulterants and see the product categories predictions. Or in the case of our work on RNNs and Wikipedia, one can enter a block of text and ask the model to make predictions based on the text. Increasing the accessibility of this work will be a major step in publicizing this new field of research, and we hope to be able to accomplish this in the near future.

To conclude, we have taken the first steps towards generating an automated system for the detection of adulterants in food. Ensuring the safety of food being sold commercially is crucial to a stable and thriving society, and we hope to see this field expand in the coming years as techniques mature and incidents becomes more sophisticated.

Appendix A

A.1 Supplementary Information about the Datasets

field	data
upc	857602004038
aisle	Diet & Nutrition
shelf	Snack, Energy & Granola Bars
food_category	Bars - Nutrition & Energy Bars
brand	Inbalance Health Corp
manufacturer	Inbalance Health Corp
product_description	Bar
product_name	Inbalance Health Corp Bar
product_size	2 oz
serving_size	56
serving_size_uom	g
servings_per_container	1
ingredients	organic sunflower seed butter, organic brown rice protein, organic crispy brown rice, organic rice bran, organic blue amber agave, organic pea fiber, organic chocolate liquor, organic cherry, organic cherry concentrate, monk fruit extract

Table A.1: Sample product entry from the FoodEssentials database.

Ingredient	# of products
salt	72933
water	52749
sugar	50055
citric acid	33448
riboflavin	21440
folic acid	21051
thiamine mononitrate	18981
niacin	18136
natural flavor	17516
soy lecithin	16916

Table A.2: Most common ingredients from FoodEssentials.

adulterant	instances
mercury	1069
sulphite	876
cadmium	683
colour Sudan 1	569
benzo(a)pyrene	385
E 210 - benzoic acid	254
colour Sudan 4	180
arsenic	161
lead	161
iodine	129

Table A.3: Most common adulterants (RASFF).

product	adulterant	instances
swordfish	mercury	270
dried apricot	sulphite	173
palm oil	colour Sudan 4	98
swordfish	cadmium	84
soy sauce	3-monochlor-1,2-propanediol	75
olive- residue oil	benzo(a)pyrene	61
dried seaweed	iodine	50
food supplement	1,3 dimethylamylamine	45
carbonated soft drink	E 210 - benzoic acid	42
spices	colour Sudan 1	37

Table A.4: Most common product-adulterant combinations (RASFF).

adulterant	category	instances
mercury	Fish & Seafood	1028
cadmium	Fish & Seafood	588
sulphite	Fruits, Vegetables & Produce	394
colour Sudan 1	Herbs & Spices	394
sulphite	Fish & Seafood	352
benzo(a)pyrene	Fish & Seafood	173
colour Sudan 4	Vegetable & Cooking Oils	152
E 210 - benzoic acid	Soda	147
benzo(a)pyrene	Vegetable & Cooking Oils	146
iodine	Fruits, Vegetables & Produce	115

Table A.5: Most common adulterant/category pairs (RASFF).

category	count
Fishery/Seafood Products	25884
Vegetables and Vegetable Products	22044
Fruit and Fruit Products	12125
Spices, Flavors, and Salts	7882
Candy without Chocolate, Candy Specialties, and Chewing Gum	6137
Bakery Products, Doughs, Bakery Mixes, and Icings	4687
Multiple Food Dinners, Gravies, Sauces, and Specialties (Total Diet)	3906
Vitamins, Minerals, Proteins, & Unconventional Dietary Specialties	3397
Cheese and Cheese Products	3301
Whole Grains, Milled Grain Products, and Starch	3127

Table A.6: The most common categories from the FDA Import Refusals list. There are 326,927 entries from 1/1/02 to 12/31/15. When filtering the entries to only consider adulterations of food products, 111,183 entries remain (34%).

charge code	category	instances
FILTHY-249	Fishery/Seafood Products	10109
PESTICIDE-241	Vegetables and Vegetable Products	7429
SALMONELLA-9	Fishery/Seafood Products	6438
SALMONELLA-9	Spices, Flavors, and Salts	4637
NO PROCESS-83	Vegetables and Vegetable Products	4512
UNSAFE COL-11	Candy	3820
FILTHY-249	Vegetables and Vegetable Products	3419
FILTHY-249	Fruit and Fruit Products	3147
UNSAFE COL-11	Bakery Products, Doughs, Bakery Mixes, and Icings	2198
NEEDS FCE-62	Vegetables and Vegetable Products	2150

Table A.7: Most common entries by refusal code and food category (FDA).

adulterant	category	instances
pesticide	Vegetables and Vegetable Products	8406
salmonella	Fishery/Seafood Products	6438
salmonella	Spices, Flavors, and Salts	4637
color additive	Candy	3822
pesticide	Fruit and Fruit Products	2254
color additive	Bakery Products, Doughs, Bakery Mixes, and Icings	2201
animal drug	Fishery/Seafood Products	1569
color additive	Fruit and Fruit Products	1339
color additive	Soft Drinks and Waters	1305
color additive	Snack Food Items (Flour, Meal, or Vegetable Base)	1107

Table A.8: Most common adulterant/category pairs (FDA).

adulterant	count
Melamine	70
Hazelnut oil	47
Sunflower oil	47
Oil (non-authentic origin)	43
Soybean oil	43
Milk (bovine)	41
Corn oil	31
Water	31
Starch	22
Urea	22

ingredient	count
Milk (fluid)	271
Olive oil	158
Honey	110
Olive oil (extra virgin)	103
Saffron (Crocus sativus L.)	56
Saffron	54
Orange juice	47
Coffee	41
Milk (powder)	35
Grapefruit seed extract	34

Table A.9: Most common adulterants and ingredients (USP).

ingredient	adulterant	count
Olive oil	Hazelnut oil	28
Milk (fluid)	Water	22
Milk (fluid)	Melamine	22
Olive oil	Sunflower oil	21
Honey (botanical origin specific)	Honey of non-authentic botanical origin	17
Olive oil	Soybean oil	17
Olive oil (extra virgin)	Hazelnut oil	16
Olive oil (extra virgin)	Sunflower oil	15
Milk (fluid, caprine)	Milk (bovine)	14
Milk (fluid, ovine)	Milk (bovine)	12

Table A.10: Most common ingredient / adulterant pairs (USP).

adulterant	count
water	41
non-organic wheat	34
pork	33
urea	23
Beef	22
Chinese honey	18
non-organic millet	17
non-organic alfalfa	17
non-organic barley	17
non-organic corn	17

product	count
milk	110
beef	36
ghee	30
organic wheat	30
grouper	29
mawa	27
lamb	27
honey	26
black pepper	23
extra virgin olive oil	21

Table A.11: Most common adulterants and products (EMA).

adulterant	product	category	count
water	milk	Dairy Products	23
urea	milk	Dairy Products	6
hydrogen peroxide	milk	Dairy Products	6
Beef	lamb	Meat Products	5
palm oil	ghee	Dairy Products	4
caustic soda	milk	Dairy Products	4
methanol	vodka	Alcoholic Beverages	4
non-organic wheat	organic wheat	Grains and Grain Products	4
vegetable oil	ghee	Dairy Products	4
pork	beef	Meat Products	3

Table A.12: Most common ingredient / adulterant pairs (EMA).

Bibliography

[1] C. M. Bishop. *Pattern recognition and machine learning*. Springer Science, 2006.

[2] K. Everstine, J. Spink, and S. Kennedy. Economically motivated adulteration (EMA) of food: common characteristics of EMA incidents. *Journal of Food Protection*, 76(4):723–735, 2013.

[3] A. Herbelot and E.M. Vecchi. Building a shared world: Mapping distributional to model-theoretic semantic spaces. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2015)*, 2015.

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

[5] Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representation (ICLR 2015)*, 2015.

[6] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Molding cnns for text: non-linear, non-consecutive convolutions. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2015)*, 2015.

[7] Tao Lei, Hrishikesh Joshi, Regina Barzilay, Tommi Jaakkola, Katerina Tymoshenko, Alessandro Moschitti, and Lluis Marquez. Semi-supervised question retrieval with recurrent convolutions. *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL 2016)*, 2016.

[8] Minh-Thang Luong, Quoc V. Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. *Proceedings of the International Conference on Learning Representation (ICLR 2016)*, 2016.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, 2013.

[10] Deepak Narayanan. Building and processing a dataset containing articles related to food adulteration. Master's thesis, Massachusetts Institute of Technology, 2015.

[11] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 2014.

[12] Xin Rong. Word2vec parameter learning explained. *CoRR, abs/1411.2738*, 2014.

[13] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[14] Yangqiu Song and Dan Roth. Unsupervised sparse vector densification for short text similarity. *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL 2015)*, 2015.

[15] Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural network. *Proceedings of the International Conference on Machine Learning (ICML 2011)*, 2011.

[16] C.Y. Teng, Y.R. Lin, and L.A. Adamic. Recipe recommendation using ingredient networks. *ACM Web Science (WebSci 2012)*, 2012.

[17] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize, 2008.