Learning to Interpret and Generate Instructional Recipes

Chloé Kiddon

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Yejin Choi, Chair

Luke Zettlemoyer, Chair

Hannaneh Hajishirzi

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Learning to Interpret and Generate Instructional Recipes

Chloé Kiddon

Co-Chairs of the Supervisory Committee:
Assistant Professor Yejin Choi
Computer Science & Engineering

Associate Professor Luke Zettlemoyer
Computer Science & Engineering

Enabling computers to interpret and generate instructional language has become increasingly important to our everyday lives: we ask our smartphones to set reminders and send messages; we rely on navigation systems to direct us to our destinations. We define instructional recipes as a special case of instructional language, where completion of the instructions results in a goal object. Some examples include cooking recipes, craft projects, and assembly instructions. Developing systems that automatically analyze and generate instructional recipes requires finding solutions to many semantic challenges, such as identifying implicit arguments (e.g., given the sentence "Bake for 15 min," identifying *what* is being baked and *where* the baking occurs) and learning physical attributes of entities (e.g., which ingredients are considered "dry"). Amassing this information has previously relied upon high-cost annotation efforts. We present a pair of models that can interpret and generate instructional recipes, respectively, and are trained on large corpora with minimal supervision – only identification of the goal (e.g., dish to make), list of materials (e.g., ingredients to use), and recipe text. Our interpretation model is a probabilistic model that (1) identifies the sequence of actions described by the text of an instructional recipe and (2) how the provided materials (e.g., ingredients) and entities generated by actions (e.g., the mixture created by "Combine

flour and sugar") are used. Our generation model is a novel neural language model that (1) generates an instructional recipe for a specified goal (e.g., dish to make), while (2) using all the required materials provided (e.g., list of ingredients to use). We also present an adaptation of our generation model that can jointly generate recipe text and its underlying structure. Experiments show that our models can successfully be trained to interpret and generate instructional recipes from unannotated text, while at the same time learning interpretable domain knowledge.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First, I would like thank my advisors Yejin Choi and Luke Zettlemoyer and express my sincere appreciation for all of their guidance and support. My success in my academic career is due in no small part to their advice and willingness to take on a lost seventh year Ph.D. student. I have been incredibly fortunate to have the opportunity to work with both of them.

I also want to thank other mentors I have had along the way: my undergraduate research advisor Chris Manning, my undergraduate advisor Jerry Cain, my first advisor at UW Dan Weld, and my Google internship mentor Kevin Murphy. I would like to additionally thank Chris Manning for not reading over the applications for his summer research interns in 2009 when I was a freshman and choosing randomly instead; joining the Stanford NLP group changed the course of my life.

I have been very lucky to be supported and inspired along this journey by many friends and educators at the University of Washington, Stanford University, and elsewhere, and I extend them my thanks (in somewhat chronological order): Dan Jurafsky, Anna Rafferty, Bill MacCartney, Jenny Finkle, Marie-Catherine de Marneffe, Nathanael Chambers, Dan Ramage, David Hall, Jason Prado, David Pfau, Kayur Patel, Kathleen Tuite, Abe Friesen, Janara Christensen, Tony Fader, Morgan Dixon, Lydia Chilton, Raphael Hoffmann, Alan Ritter, Jesse Davis, Vibhav Gogate, Daniel Lowd, Hoifung Poon, Aniruddh Nath, Mathias Niepert, Adrian Sampson, Ray Mooney, Andrew McCallum, Cynthia Matuszek, Juri Ganitkevitch, Katerena Kuksenok, Yonatan Bisk, Bob West, Jonathan Malmaud, Karl Pichotta, Mark Yatskar, Eunsol Choi, Jesse Dodge, Luheng He, Nicholas FitzGerald, Kenton Lee, Victoria Lin, Tom Kwiatkowski, Yannis Konstas, Mike Lewis, Sameer Singh, Sasha Rush, Antoine Bosselut, and Srinivasan Iyer.

I also wish to give my deepest thanks to Linda Shapiro, Anna Karlin, and Lindsay Michimoto for their help in improving my academic career.

I wish to thank my mother for her endless support and love, for pushing me to stay in the graduate program until the end, and for being there through the good times and the hard times. Thank you also to my husband Alex, who has supported me through the final years of my degree. I am excited to move forward in life with you.

My graduate career was supported in part by an NSF Graduate Research Fellowship and funding from DARPA and the Intel Science and Technology Center for Pervasive Computing.

DEDICATION

to my husband, Alex

Chapter 1

INTRODUCTION

The interpretation and generation of instructional language by computers is becoming increasingly prevalent in our everyday lives, helping us to achieve a variety of goals. Our phones interpret commands to set our alarms and send our messages. Navigation systems within our vehicles generate turn-by-turn instructions to direct us to our destinations – and update those instructions if we drive off course. Enabling computers to handle these types of tasks requires systems that can both (1) interpret and execute instructions given by human users and (2) generate human-understandable text instructions for users to follow.

However, the automatic interpretation and generation of instructional language that helps us create new artifacts has been far less common due to the complexities of understanding and generating the plans that serve as the backbones of these texts. These plans represent the sequence of actions, with any associated arguments, that need to take place in order for the instructional language to be completed successfully. We define this subclass of instructional language, which we call *instructional recipes*, as language where successful completion of the instructions results in the creation of a specified goal object. Examples of instructional recipes include cooking recipes, craft project instructions, furniture assembly instructions, and wet lab procedures. Instructional recipes have a *title*, which is a sequence of tokens that define the specified goal object, and a set of *materials* that are solely and completely used in the creation of the goal object. Figure 1.1 shows an example of an instructional recipe in the baking domain. In this case, the goal object is a batch of blueberry muffins and the set of materials is the ingredient list. The recipe directions define the sequence of actions required to transform the items in the ingredient list into blueberry muffins.

The dual abilities to interpret and generate instructional recipes are important for many

Title: Blueberry Muffins I

Ingredients:
1 cup milk
1 egg
1/3 cup vegetable oil
2 cups all-purpose flour
2 teaspoons baking powder
1/2 cup white sugar
1/2 cup fresh blueberries

Directions:
1. Preheat oven to 400 degrees F (205 degrees C). Line a 12-cup muffin tin with paper liners.
2. In a large bowl, stir together milk, egg, and oil. Add flour, baking powder, sugar, and blueberries; gently mix the batter with only a few strokes. Spoon batter into cups.
3. Bake for 20 minutes. Serve hot.

Figure 1.1: An example of a recipe for blueberry muffins. Recipe from http://allrecipes.com/Recipe/Blueberry-Muffins-I/.

real-world applications. Wiegand et al. (2012) described a plausible scenario for knowledge acquisition within the food domain, which is predominantly represented by recipes: a virtual customer advice system that can recommend corrective revisions of recipes, identify ingredient substitutions, or plan a menu. Automatic understanding of recipe text would complement and extend such work, allowing for not only advice for cooks but also the explicit revision or generation of recipes. Laroche et al. (2013) proposed Cooking Coach, a spoken dialogue system to help a user search for recipes and prepare the recipe. A similar, if not more futuristic, application is the construction of robotic cooking assistants (Beetz et al., 2011; Bollini et al., 2013).

1.1 Thesis Statement

The vast majority of previous approaches to training systems for the interpretation and generation of instructional recipes collect recipe data annotated with plan structures and then use that annotated data to train models (Maeta et al., 2015; Mori et al., 2014a; Tasse and Smith, 2008; Pinel et al., 2015). These annotations identify the sequence of actions of the recipe and how those actions connect. For example, if a recipe contains the sentence

"Pour the batter in the pan."

then the annotations should identify that (1) there is a POUR action, (2) the action acts upon "the batter," (3) the action occurs in "the pan," (4) "the batter" is a food entity that was created by a previous action, and (5) which of the previous actions created "the batter." However, large-scale annotation efforts to generate these recipe structure annotations is time- and cost-intensive: available data sets each contain only a few hundred examples (Mori et al., 2014b; Tasse and Smith, 2008). The plan annotation process is a significant bottleneck to exploiting the numerous and large sources of instructional recipes that exist on the Internet (e.g., *allrecipes.com* for cooking recipes, *instructables.com* for craft and home projects). The thesis of my research is that instructional recipe understanding, in particular the abilities to interpret recipes as actionable plans and to generate novel recipes for given dishes, can be achieved by developing models that can learn internal representations of recipes without direct supervision by training on large unannotated recipe corpora.

1.2 Interpreting Recipes as Actionable Plans

An instructional recipe is a text representation of an actionable plan that an autonomous entity can follow. This plan is composed of a sequence of actions, or events, that each act upon a set of entities. These actions act on entities from the set of materials as well as entities that are generated from previous actions. Each action consumes the entities it acts

on and generates a new entity or, in certain cases, a set of new entities.[1] By the end of the sequence of actions, only one entity remains: the specified goal object.

A system for interpreting instructional recipes has to handle many complex semantic challenges. As an example, take the following simple recipe:

Salsa

Ingredients: 2 ripe red tomatoes, 1 red onion, 1 lime, and salt to taste

Steps:

1. Chop the tomatoes.
2. Dice the onion.
3. Combine the tomatoes and onion.
4. Squeeze half a lime over the salsa.
5. Sprinkle with salt.
6. Serve salsa with chips.

We will now discuss three semantic challenges that are unique to processing procedural language using the above salsa recipe.

Imperative Sentence Processing One semantic challenge encountered is identifying each action and its predicate-argument structure. The vast majority of sentences in a recipe are imperative sentences. Parsers have trouble with imperative sentences since the sentence structure tends to be different than declarative sentences, which make up the majority of parser training data (Hara et al., 2011). Domains such as cooking recipes tend to use vocabulary that cause additional trouble for parsers trained on out-of-domain data: they can contain verbs that are frequently used as nouns in other domains, e.g., "dice," "squeeze." If these semantic challenges are addressed, a system will correctly segment the above example's

[1]For example, in a baking recipe, the instructions might specify to separate the egg yolks from the egg whites. This "separate" action will create two entities: the egg yolks and the egg whites.

steps into the actions' predicates and their respective arguments:

1. Chop (PREDICATE) the tomatoes (ARGUMENT) .
2. Dice (PREDICATE) the onion (ARGUMENT) .
3. Combine (PREDICATE) the tomatoes (ARGUMENT) and (CONJUNCTION) the onion (ARGUMENT) .
4. Squeeze (PREDICATE) half a lime (ARGUMENT) over (PREPOSITION) the salsa (ARGUMENT) .
5. Sprinkle (PREDICATE) with (PREPOSITION) salt (ARGUMENT) .
6. Serve (PREDICATE) salsa (ARGUMENT) with (PREPOSITION) chips (ARGUMENT) .

Argument Identification and Entity Resolution Once the actions are identified, a recipe interpretation system must determine which entity is being referred to by each argument string. In step 1, "the tomatoes" refers to "ripe red tomatoes" in the ingredient list; similarly, "the onion" in step 2 refers to the ingredient "red onion." In step 6, the "chips" refers to a new entity that is neither from the ingredient list or generated by a previous action. The system must be able to identify these entities as extraneous instead of forcing all entities to be from the ingredient list or generated by an action.[2] Resolving entities that are generated by previous actions can be a more complex task. In step 3, the chopped tomatoes and diced onions are combined to create one single, new entity that is a mixture of both; this entity is referred to as "the salsa" in step 4. The system would have to know that a combination of chopped tomato and onion could be referred to as a "salsa." Writers of recipes also tend to use physical properties of entities when referencing them, such as "wet ingredients" or "hot mixture." Interpreting recipes as plans thus requires a large amount of domain-specific knowledge.

However, just as often, the writers of recipes will omit arguments of verbs that can be inferred from context. For example, in step 5 of the example, the SPRINKLE action has

[2]This phenomenon occurs frequently in cooking recipes as mentions to optional ingredients (e.g., "If you want, you can garnish with chopped cilantro.") or advice on what dishes to serve with the dish (e.g., the last sentence of a curry recipe may be "Serve over rice.").

two arguments: the salt and an implicit argument that corresponds to the output of the SQUEEZE action (i.e., the combination of tomato, onion, and lime). Detection and resolution of implicit arguments such as these is an instance of zero anaphora detection and resolution (Tetreault, 2002; Whittemore et al., 1991; Palmer et al., 1986; Silberer and Frank, 2012).

Handling Dependencies among Inter-Action Connections In the salsa example, once the implicit argument is identified and sourced, the full set of inter-action connections (i.e., entities and their associated referring expressions) are as follows:

output of the CHOP action $\longrightarrow$ "the tomatoes" argument of the COMBINE action,
output of the DICE action $\longrightarrow$ "the onion" argument of the COMBINE action,
output of the COMBINE action $\longrightarrow$ "the salsa" argument of the SQUEEZE action,
output of the SQUEEZE action $\longrightarrow$ IMPLICIT argument of the SPRINKLE action,
output of the SPRINKLE action $\longrightarrow$ "the salsa" argument of the SERVE action.

These inter-action connections are not independent of each other. For example, "the salsa" in step 6 is the output of the SPRINKLE action. However, salt on its own cannot be referred to as "salsa." Only after identifying that "the salt" is being sprinkled on the output of the SQUEEZE action does the connection between the output of the SPRINKLE action and "the salsa" in step 6 make sense. The possibility of long-range dependencies among inter-action connections complicates the task of selecting the best action structure for a recipe as potential structures must be evaluated at a global level.

1.3 Generating Recipes using Neural Sequence Generation Models

While recipe interpretation requires domain-specific knowledge for entity resolution and identifying when implicit arguments exist, generating a new instructional recipe requires not only this knowledge but additionally a deep understanding of how objects can be produced. This includes general knowledge about families of recipes: for example, the text of a muffin recipe

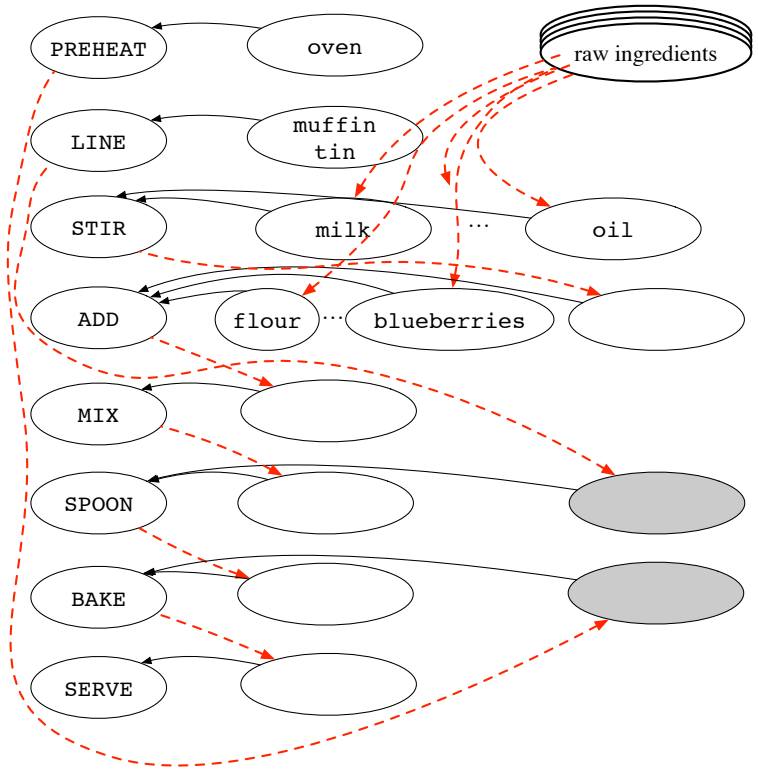


Figure 1.2: The plan of actions for the blueberry muffin recipe in Figure 1.1

must have certain actions (e.g., combining ingredients, baking) and properties (e.g., correctly using a muffin tin). Additionally, the task requires knowledge of how to generate specific recipe instances: for instance, a blueberry muffin recipe should have blueberries introduced during an action that combines other ingredients, rather than incorrectly as the argument of an action that will only sprinkle the blueberries on top of the muffins at the end.

The generation of instructional recipes, mainly cooking recipes, has been studied since the 1980s when it was used as a domain for early planning (Hammond, 1986) and referring expression (Dale, 1988) research. However, recipe generation requires knowledge of the plan of the recipe to generate, and, in most previous cases, proposed systems have not had mechanisms to create novel plans and, thus, recipes. In the few systems that have had this ability, they have either been restricted to extremely simple types of recipes (Morris

et al., 2012) or have required significant manual engineering (Pinel et al., 2015). Under the paradigm of previous text generation research, two tasks must be tackled: the first task resolves the conceptual or strategic decisions about *what to say*, while the second task focuses on the linguistic or tactical tasks of *how to say it* (Thompson, 1977). For example, using this paradigm to generate a blueberry muffin recipe such as that in Figure 1.1, the first step (i.e., determining *what to say*) would be to generate a plan of actions that represents the sequence of things that must happen in order to generate blueberry muffins. Figure 1.2 depicts the plan for the blueberry muffin recipe from Figure 1.1. This plan identifies the eight actions required to create blueberry muffins (i.e., PREHEAT, ..., SERVE). In this diagram, the black arrows identify the entity arguments for each action; greyed ovals are location entities. The red dashed arrows identify where these entities came from: for example, the STIR action has several arguments from the ingredient list, and the ADD action has several different arguments from the ingredient list as well as the output of the STIR event.

The next step would be to generate natural language text expressions for each of the actions in the plan, i.e., figure out *what to say*. This includes determining the correct sentence structure and figuring out the best referring expressions for each non-implicit argument. For example, the MIX action can be transformed into any of the following sentences (among others):

1. Mix.

2. Mix the batter.

3. Mix the blueberry mixture.

4. Mix the flour mixture.

In sentence 1, the direct object argument is left implicit. In sentences 2-4, different referring expressions are used to describe the output of the STIR event. There is also a variety in how

to describe ingredients: “2 cups all-purpose flour” can be referred to as “all-purpose flour” or, more usually, “flour”.

However, recent natural language generation work has introduced the possibility that it is no longer necessary to explicitly generate a plan first and from that generate output text in the way described above. Approaches based on recurrent neural network (RNN) architectures can generate accurate and readable text token-by-token (or character-by-character) from a representation of the generation goal without first constructing an intermediate plan representation (Wen et al., 2015; Sutskever et al., 2014; Karpathy and Li, 2014; Zhang and Lapata, 2014). The existing RNN architectures, though, have some flaws that must be overcome in order to apply them to the generation of longer texts, such as instructional recipes. RNNs rely on a low-dimensional hidden vector to maintain both the current semantic and syntactic state of the generation process; this can cause issues when the range of possible outputs is large or the lengths of possible outputs is long. There is also no standard way of incorporating additional information, such as a list of materials to use, into the RNN architecture, although there have been proposals (Jia et al., 2015). Developing a model for generating instructional recipes with RNNs frees us from the complexities of defining how recipe plans can be generalized, such as identifying the specifications of different recipe types (e.g, muffins) and recipe attributes (e.g., **blueberry** muffins). Instead, it requires the introduction of much more structure into the RNN framework than is generally used (e.g., requiring a material to not be used in the generated recipe more than once).

1.4 Thesis Overview

This thesis describes a series of models for recipe understanding that can each be trained from a corpus of unannotated recipe text. Chapter 2 discusses previous approaches to recipe interpretation and generation, as well as work related to the methods we use to approach the tasks. In Chapter 3, we present a probabilistic model that learns to interpret recipes as actionable plans. We train our model using unsupervised learning to avoid the need to annotate plans, and, as a byproduct of learning, we obtain domain-specific knowledge.

In Chapter 4, we present a model for generating novel recipe text given just the title and material list. We introduce a novel neural network architecture, the *neural checklist model*, that solves problems that exist in previous RNN architectures when applied to the task of recipe generation. These two chapters show that there is enough signal that exists in recipe corpora to train recipe interpretation and generation systems. In Chapter 5, we consider an approach for combining these two efforts that modifies the neural checklist model to jointly generate recipe text and underlying structure that identifies the origins (e.g., ingredient list or previous action) of entities. We propose directions for future work in the area of recipe understanding in Chapter 6, and we conclude in Chapter 7.

Chapter 2

RELATED WORK

While procedural language has been relatively less studied in prior literature, there has been increasing research interest in recent years. There are many practical impacts of models that can handle procedural language: such technology is integral to navigation systems and automated help lines (e.g., flight booking systems, informational systems). Our research focuses on instructional recipes, a subclass of procedural language. Previous research has examined many aspects of instructional recipes – more specifically within the domain of cooking recipes – as well as other types of procedural language.

In this chapter, we first review related research on procedural language interpretation (Section 2.1) and script learning (Section 2.2) and then we discuss previous approaches to recipe interpretation in particular in more depth (Section 2.3). Then to motivate our recipe generation model, we briefly highlight prior literature on natural language generation (Section 2.4) and provide an overview of how neural networks have been used for generation as well as for other natural language tasks (Section 2.5). Following those sections, we discuss previous recipe generation systems (Section 2.6). We conclude this chapter with an examination of related work within the cooking recipe domain (Section 2.7).

2.1 Prior Work on Procedural Language

There is a substantive body of research in transforming human language instructions into actionable plans. Many of these efforts focused on navigation instructions evaluated within a virtual environment developed in MacMahon et al. (2006). Navigation instructions, like instructional recipes, are goal-oriented texts, but entities in navigation instructions are used as landmarks and do not change, whereas entities in instructional recipes are used, altered,

and combined in order to generate the goal. Chen and Mooney (2011) presented a method for learning an instruction semantic parser by observing humans following navigation instructions; in this work, like ours, the underlying plans in the training data were unobserved. However, by following the humans' actions, the navigation instruction learning system has more signal to train from than just the text of the instructions alone. Artzi and Zettlemoyer (2013) presented a joint probabilistic model of text's meaning and context (i.e., grounded information about the current location) in order to interpret and execute navigate instructions in the same domain. The authors used an online learning algorithm and weakly supervised training to learn their model; in our work, we learn a joint probabilistic model over the text and graph of actions in a completely unsupervised manner. Andreas and Klein (2015) used a conditional random field to align navigation instructions and executable actions. More recently, Mei et al. (2016a) presented strong results on the same navigation task from MacMahon et al. (2006) by training a neural encoder-aligner-decoder model that learns to directly generate actions from instructional text. Our approach to recipe generation uses a similar encoder-decoder approach but with an mechanism for alignments between tokens in the output and tokens in the input list of materials, which is concealed using a standard neural network.

The work of Branavan et al. (2009) used a simulation system to learn to interpret Windows troubleshooting guides and game tutorials. In the experiments, the set of possible actions was provided; our task of learning the semantics of instructional recipes from their text alone requires us to also learn what the possible actions are. Another difference between the instructional language in this body of prior work and our task is that while both include instructions to be performed in sequence, our instructions may require the use of entities that were generated much earlier in the recipe. Branavan et al. (2011) similarly described a simulation system to learn to play the game Civilization II. The authors use a neural network to learn value function approximations for different states and actions, and these approximations are used in a Monte-Carlo search algorithm to try to win the game. In contrast to our task, following the exact instructions in Civilization II (i.e., the game manual)

will not complete the goal, in this case, of winning the game. These instructions are only used to learn values for states and actions that can be used then to develop a winning strategy.

The work of Lau et al. (2009) interpreted how-to instructions using a variety of models, but their system used labeled training data and did not find connections amongst different instructions. Fujiki et al. (2003) learned script knowledge from a large text corpus by estimating pairs of actions. There is also related work on the reverse task: generating natural language instructions from planning information (Ansari and Hirst, 1998).

2.2 Script Learning

Broadly, cooking knowledge can be considered as a type of script knowledge (Schank and Abelson, 1977). A script is defined as "a standardized sequence of events that describes some stereotypical human activity such as going to a restaurant or visiting a doctor" (Barr and Feigenbaum, 1981). Most prior work on learning script knowledge considered domains such as newswire and storybooks that are not procedural (Fujiki et al., 2003; Chambers and Jurafsky, 2009; Pichotta and Mooney, 2014; Balasubramanian et al., 2013), or crowdsourced the knowledge instead of learning it directly from data (Regneri et al., 2010, 2011).

Rudinger et al. (2015) used unsupervised script induction methods applied to a dataset of restaurant narratives to learn Schank and Abelson's restaurant script (Schank and Abelson, 1977). Through their experiments, the authors are able to conclude that algorithms for learning narrative chains can be used on domain-specific corpora to learn specialized scripts. While food-related, the restaurant script does not include food preparation steps. However, Rudinger et al. (2015) suggested that these types of methods can be applied to other domains – such as recipes.

Regneri et al. (2010, 2011) investigated how to learn scripts and their participants. Unlike the narrative schema learning of Chambers and Jurafsky (2009), this work tried to learn scripts that can include knowledge that is not usually elaborated on in the usual text corpora – such as shopping or eating in a fast-food restaurant. In order to get training data for these

scripts, the authors had non-experts on Amazon Mechanical Turk[1] describe "event sequences for given scenarios" in the form of bullet point sequences of events (Regneri et al., 2010).

Our recipe segmentation model learns to extract the sequence action verbs from a recipe, which requires knowledge of what kinds of actions occur in recipe text (see Section 3.4). However, we do not go so far as learning an explicit "script" for different recipe types (e.g., muffins, pot roast, etc.). Our generation model, though, implicitly learns this kind of "script" information so that it can generate the correct recipe text for a given title (see Chapter 4).

2.3 Prior Work on Recipe Interpretation

Researchers in the area of procedural language have turned to the domain of cooking recipes with increased frequency as in-domain data is easily obtained. There have been several previous studies of the task of interpreting cooking recipes. Some researchers have approached this task from the perspective of teaching robots how to cook. Bollini et al. (2013) presented BakeBot, a robot that could bake cookies; the focus of this work was on the robotic movements, while the overarching plan of how to bake the cookies was provided. Beetz et al. (2011) demonstrated that a pair of robots could take and follow a particular recipe from the Internet for pancakes; it's applicability to recipes other than that pancake recipe was not investigated.

Karlin (1988b) presented SEAFACT (Semantic Analysis For the Animation of Cooking Tasks) a natural language interface to a system that would animate a CGI robot performing the cooking actions. SEAFACT parsed recipe text using a complex manually-generated domain-specific knowledge base about both entities and linguistics terms as well as a rule-based grammar. Karlin (1988b,a) gave a detailed analysis of linguistic phenomena that occur in cooking recipes and how they translate into physical actions for a robotic entity to execute. For example, the authors examined the variation of how different plural objects can affect the action structure of a recipe: "chop the nuts" represents one action since nuts are

[1] http://www.mturk.com/

small and can be chopped all at once, but "chop the tomatoes" requires multiple chopping actions for each tomato. Under the constrained environment that SEAFACT worked under, these semantics could be handled, but training a model using a data-driven machine learning approach to handle these semantic aspects at a large scale is an intriguing area for future work (see Section 6.1).

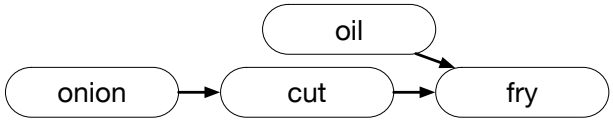
The SOUR CREAM system of Tasse and Smith (2008) used a instruction language for cooking tasks (MILK) and released a data set of 300 annotated recipes. However, they did not make full connections between arguments and their originating actions. MILK can not differentiate among different argument strings, so origins for each cannot be identified distinctly. In Chapter 3, we propose a new graph representation for recipes that is more fine-grained.

Jermsurawong and Habash (2015) presented a dependency-tree-based representation called SIMMR (Simplified Ingredient Merging Map in Recipes) for cooking recipes and an associated parser for parsing recipes into this form. Similar to our approach to parsing recipes, this system first identifies the references to ingredients in the recipe and then links the actions of the recipe. In contrast to our approach, the methods presented in this work require supervised training, in their case evaluating using the small data set from Tasse and Smith (2008). Also, SIMMR can only represent recipes that have a tree-structure (e.g., SIMMR can't handle verbs such as "separate" that generate more than one entity).

2.3.1 Recipe Texts as Work Flow Graphs

The system presented in Mori et al. (2012) converts recipe texts to work flows using an NLP pipeline of Japanese word segmentation, named entity resolution, and then predicate-argument extraction via syntactic analysis. Different portions of the pipeline use partially annotated data: for example, the NER tagger classifies words as foods, quantities, tools, etc., and is trained from a corpus of annotated data. In follow-up work, Yamakata et al. (2013) looked at how to generalize recipes returned from a query into a canonical recipe. Each recipe was represented as a tree structure with ingredients as leaf nodes and cooking

actions – from a set of 11 categories – as intermediate nodes. For example, a sentence "Cut an onion and fry in oil" would create the following tree:



While recipes can be directed acyclic non-tree graphs, the authors limited their recipes to tree-structured recipes so that they could use tree-mapping algorithms. Given a set of recipe trees for a particular query, the trees were mapped to each other node-to-node and the closest two trees based on an edit distance were integrated using heuristics until one canonical tree remained. More recently, Mori et al. (2014b) described a Japanese dataset of cooking recipes manually annotated with *flow graphs*. Flow graphs have vertices for all actions and entities in a recipe but do not structurally differentiate between the two types. They also do not include syntactic information such as which argument an entity is a part of, which we have found useful for learning verb selectional preferences.

2.4 Concept-to-Text Natural Language Generation

A standard approach to generating natural language text is first to determine the non-linguistic concepts to be conveyed by the output text (i.e., *what to say*) and then to realize those concepts in natural language text (i.e., *how to say it*) (Thompson, 1977; Reiter and Dale, 2000). Typically these steps are performed using a pipeline approach (Barzilay and Lapata, 2005; Liang et al., 2009). However, they also can be trained and executed jointly (Angeli et al., 2010; Kim and Mooney, 2010; Konstas and Lapata, 2013). In these prior works, content selection required selecting the correct information at the correct granularity from a large database or overgenerated world state. Recipe generation, on the other hand, requires content *planning* because the correct non-linguistic specification for a requested recipe will most likely not already exist in some database. Mei et al. (2016b) presented a joint model of content selection and linguistic realization using a recurrent neural network with a coarse-to-fine aligner. Our generation model is not given and does not first, or jointly,

select an explicit semantic world state of which to generate text from. Instead, any semantic state for generation is encoded implicitly in our recurrent neural network.

2.5 Neural Networks for Natural Language Tasks

Our approach to recipe generation builds upon a strong body of previous work that using neural networks for natural language generation and other natural language processing tasks.

2.5.1 Neural Networks for Natural Language Generation

Neural networks have recently become a prominent technique for tasks that involve generating natural language text, such as machine translation and dialogue generation. The accuracy of these models improves as the amount of training data increases; therefore, the growing availability of big data resources has allowed use of these models to flourish. Recent work showed that recurrent neural networks can be used as language models – crucial to generating coherent syntactically-correct text – and can be superior to ngram-based methods (Mikolov et al., 2010, 2011).

Neural networks are not completely new to natural language. Kukich (1987) used neural networks to create ANA, a phrasal sememe-to-morpheme translator (i.e., a model that generates a text phrase as a set of morphemes given a list of semantic statements) for stock market data. The training set was only 113 possible output phrases (e.g., "posted a sharp loss", "staged a modest rally") hand-coded with their respective sememes, but evaluations showed the promise of using this type of connectionist network for generation tasks.

Much more recently, neural networks trained on larger amounts of data have begun to be applied successfully to the task of dialogue system generation. In this task, a model must generate a piece of natural language text to represents some given semantics. For example, an informational system about restaurants might need to communicate the semantics

$$\mathsf{inform(name{=}Piperade;good\_for\_meal{=}dinner;food{=}basque)}$$

and will generate the text

Piperade is a basque restaurant that is good for dinner.

Wen et al. (2015) and Wen et al. (2016) present a neural architecture for generating natural language dialogue system responses by keeping track of what information still needs to be said. Their work focuses on generating short (1-2 sentence) texts from a few pieces of information. The model we propose in Chapter 4 works under those conditions, and we also target longer texts, such as recipe composition, with possibly many agenda items (e.g., ingredients). Their architecture is not immediately applicable to domains with higher semantic variability, such as recipe composition, where the possible space of agenda items (i.e., ingredients) is significantly larger compared to the possible space of dialogue acts in data considered in Wen et al. (2015). Additionally, our model can learn token-based similarities among agenda items (e.g., learning that "ripe tomatoes" and "red tomatoes" can be used similarly) instead of treating each as completely unique. Their architecture uses a vector the size of all possible dialogue acts, which would not scale for recipe generation where there are many possible ingredients. For the dialogue system output generation task, the number of distinct dialogue acts was 248; applying this system to cooking recipes where the number of possible ingredients is extremely large is infeasible.

2.5.2 Maintaining Coherence in Neural Networks

Maintaining coherence and avoiding duplicates have been recurring challenges when generating text using RNNs. When generation longer texts, such as cooking recipes, the importance of overcoming these challenges becomes more acute: for example, a cooking recipe would not be accurate if the introduction of an ingredient occurs more than once. The model of Jia et al. (2015) uses an extra input representing a global semantic state to RNNs to guide the generation of image captions. In their model, the maintenance of the agenda is outside the neural architecture. We also use extra inputs to an RNN to guide generation (i.e., the goal and agenda), but unlike the semantic state in Jia et al. (2015), our agenda is continually updated as the text is generated.

2.5.3 Attention Models for Natural Language Tasks

We take advantage of attention mechanisms which have been used for many NLP tasks such as machine translation (Balasubramanian et al., 2013), abstractive sentence summarization (Rush et al., 2015), machine reading (Cheng et al., 2016), image caption generation (Xu et al., 2015), and textual entailment (Rocktäschel et al., 2016). Attention models generate a learned soft alignment between a target vector and a list of comparison vectors. This alignment is a probabilistic distribution representing how close the target vector is to each comparison vector. Our model uses attention models to record what has been said and to select new ingredients to be referenced. Recent work in machine translation models used previous attention to inform future time steps, but it did not explicitly accumulate attention over time (Luong et al., 2015). Xu et al. (2015) presented a neural caption generation model that adds a constraint to the training objective that makes sure each image part is attended to in the caption; however, this constraint is only used during training, where our model accumulates attention in our checklist during generation as well. The model of Tu et al. (2016) similarly keeps track of what has been covered already by an attention model for a neural machine translation task; this model uses the attention history as an input at each time step. A crucial difference of our model compared to this and previous attention models is that our model only uses the computed attention alignment when generating tokens that are references to ingredients instead of using the computation at every time step.

2.6 Previous Approaches to Recipe Generation

The first reported recipe generation system was EPICURE built by Robert Dale in the late 1980s (Dale, 1988). While producing an end-to-end system, Dale's research focused mostly on developing a sophisticated knowledge representation and a procedure for generating appropriate referring expressions within a connected discourse. More recent research on developing systems for generating recipes has focused on "computational creativity," creating systems that generate artifacts that would be deemed novel and appropriate by knowledgeable human

judges. We discuss these systems and other recipe generation systems in turn next.

2.6.1 EPICURE: Generating Complex Discourse

Dale (1988) focused on generating referring expressions for objects and processes within a connected discourse. His particular case study supporting his research was generating natural language instructions for cooking recipes. Given a particular goal dish, EPICURE first produces a hierarchical plan consisting of the operations at varying levels of detail (e.g., "Prepare the onions." might be decomposed into "Peel the onions." and "Chop the onions."). The level of detail used for each instruction depends on the assumed knowledge of the hearer of the recipe. The requisite ingredients and a top-level goal and decomposition of that goal into a hierarchical plan is assumed to be known by EPICURE. The development of a planning system to automatically generate that information was beyond the scope of this research.

Dale's approach uses a complex ontology to represent objects and events. The knowledge base uses a generalized notion of an object to represent both singular and non-singular objects (e.g., *a carrot*, *three carrots*, or *three pounds of carrots*) and can easily model changes in a dynamic environment (e.g., changing from *three pounds of carrots* to *three pounds of grated carrots*) (Dale, 1989). Entities in the knowledge base corresponding to events identify the exact change in properties that occur to the incoming object(s) when the event occurs. When generating text, EPICURE considers what the semantic content should be recoverable from the utterance by the hearer, and how to generate adequate and efficient referring expressions, using pronominalization, *one*-anaphora, or ellipsis as allowed.

Referring expression generation is vital to the task of recipe generation. Unlike Dale's approach that is centered around ontological and rule-based methods, our models learn to identify good referring expressions and generate new referring expressions using data-driven machine learning methods.

2.6.2 Recipes as Computational Creativity

Recently, there has been increasing interest in the field of computational creativity, a subfield of artificial intelligence. People working within this field develop systems that exhibit behaviors or produce artifacts that can be considered acts of creativity.[2] In particular interest to our work, there is a task within the area of computational creativity of creating *culinary creativity* systems. The 8th Computer Cooking Contest, where the evaluation of computer generated recipes involves a taste test, was held in Germany in September 2015.[3]

The PIERRE (Pseudo-Intelligent Evolutional Real-time Recipe Engine) system from Morris et al. (2012) generates recipes by applying a genetic algorithm to the ingredient lists from a dataset of recipes. This research focused solely on crockpot recipes (e.g., soups, stews, chilis), which all have the same set of instructions: "Combine ingredients and bring to a boil. Reduce heat and simmer until done, stirring occasionally. Serve piping hot and enjoy." The research focus of this work was limited to the – quite complex – problem of how to select and pair ingredients for a tasty dish.

Most notable of these systems is IBM's Chef Watson, a recipe generation system with an online app[4] and its very own cookbook.[5] Pinel et al. (2015) described many of the technical details behind IBM's approach to recipe generation. The system takes as input domain knowledge and (1) proposes novel combinations of ingredients (including the proportions of each), (2) generates possible recipe plans for that ingredient list (including recipe step duration), and (3) outputs textual instructions for the plan. The system relies on a manually engineered domain-specific knowledge database that contains not only known recipes but also qualitative properties of different foods, such as odor descriptors, nutritional content, and pleasantness evaluations. Such a large-scale engineering effort is beyond the scope of our

---

[2]Determining a precise definition or metric for creativity, however, is also under debate.

[3]http://ccc2015.loria.fr/

[4]https://www.ibmchefwatson.com/

[5]IBM and the Institute of Culinary Education. *Cognitive Cooking with Chef Watson: Recipes for Innovation from IBM & the Institute of Culinary Education.* Sourcebooks. 2015.

work. In contrast, we are interested in how well we can generate recipes without any domain knowledge provided. Our system may not be able to know that a "sausage and banana cake" will taste horrible, but we want to be able to generate a recipe for it at a user's request. The algorithms that generate ingredient proportions and recipe step duration are particularly helpful for generating viable recipes. We consider this type of greater semantic coverage in future work (see Section 6.1).

To generate a new recipe plan, for each ingredient in the ingredient list, the IBM system creates plausible subgraphs of actions for the intended dish. If no subgraph is found for an ingredient-dish pairing, the ontology looks for similar ingredients for the dish (e.g., considering meat instead of fish). The subgraphs are clustered, and the clusters are scored. Then, the highest scoring ingredient clusters are merged into a full recipe plan using a variation of a minimum common supergraph computation (Bunke et al., 2000); lower-performing subgraphs are reconsidered if the merging is unsuccessful. In contrast, we train our neural network architecture to generate the appropriate text for a given recipe title, without learning explicit, interpretable information about how to generate recipes for different object types.

2.6.3 FlowGraph2Text

Recent research has also examined recipe generation from out from under the guise of computational creativity. Mori et al. (2014a) presented a method, FlowGraph2Text, for generating a cooking recipe given a *flow graph* representation. The authors automatically collected a set of sentence templates, which they call *sentence skeletons*, from real recipe texts. Given a particular flow graph, each graph segment that represents an action and its arguments is matched with the most frequent sentence template seen with the correct type signature. Then the action and argument words from the flow graph are linguistically realized within the template to generate the text. No method for generating new flow graphs for specified objects is provided; under the text generation paradigm of Thompson (1977), FlowGraph2Text focuses solely on the task of *how to say it*, rather than *what to say*. The model we present

in Chapter 4 can generate recipe text given only the title and ingredient list.

2.6.4 CHEF and Case-based Planning

Generating new recipes by making use of a knowledge base of previously seen recipes is highly related to research in the area of case-based planning. Case-based planning systems utilize past experiences when developing new plans rather than systems of rules. Case-based planning is useful when trying to generate complex plans that might be more easily created by adapting similar previously-generated plans rather than re-planning for every new set of goals. Planning out Szechwan cooking recipes was the domain used to demonstrate the potential of the first case-based planner, CHEF (Hammond, 1986). In the research presented in Hammond (1986), recipe plans are stored along with what goals they satisfy (e.g., a dish that is a stir-fry, a dish that contains meat) and which goals they failed to satisfy. When presented with a new goal, the planner finds a similar plan and modifies that plan in order to satisfy the specifications of the new goal (e.g., swapping out one vegetable for another in a recipe). If the original plan failed in some way (e.g., the vegetables were soggy), the plan is modified further. However, the modification module in this system was manually engineered.

2.7 Other Tasks within the Cooking Domain

Due to the abundance of cooking recipes corpora available online, cooking recipes have been used as a domain for many other natural language understanding tasks. Recipes have been used alongside their online reviews in order to learn actionable refinements (e.g., swap "honey" for "sugar") and align them to the appropriate recipe steps (Druck and Pang, 2012). Druck (2013) also used features of recipes and these reviews to identify recipe attributes (e.g., "refreshing" or "creamy"). Greene (2015) trained a CRF model to extract structured information (e.g., amounts) from ingredient lists.

There has also been work developing a probabilistic model using PropBank to identify the proper frames for verbs and learn multimodal cooking semantics grounded on videos of

people following recipes (Malmaud et al., 2014, 2015). Cooking recipes have also been used as a domain for extracting domain-specific knowledge and ontologies (Gaillard et al., 2012; Nanba et al., 2014). Nedović (2013) learned generative models in ingredient distributions to map ingredients to different world cuisines.

Abend et al. (2015) presented work investigating unsupervised learning of lexical event ordering and evaluating within the domain of cooking recipes. Given a set of predicate-argument event pairs, the task is to determine the proper order of all the events. Relying on the generality that most recipes are short, the authors looked at all possible orderings of the events to find the optimal one; they represented the problem as an Integer Linear Program (ILP) and solved using ILP optimization techniques. The only aim of lexical event ordering is to find the correct order of events. While lexical features of the arguments helped order the events (e.g., a feature identifying if an event has argument "oil" and is followed by an event with argument "onions"), individual arguments were not linked with the events that created them. This functionality is important for generating a coherent plan for discourse, such as is required for an instructional recipe. The algorithm in Abend et al. (2015) assumes no textual ordering is known, so that it can be applied to more general texts. However, since the event ordering in recipe text is sequential a majority of the time, textual ordering is a very useful clue when analyzing recipe text.

Chapter 3

INTERPRETATION

Despite the fact that instructional language is crucially important to our everyday lives, there has been relatively little effort to design algorithms that can automatically convert it into an actionable form. Existing methods typically assume labeled training data (Lau et al., 2009; Maeta et al., 2015) or access to a physical simulator that can be used to test understanding of the instructions (Branavan et al., 2009; Chen and Mooney, 2011; Bollini et al., 2013; Beetz et al., 2011). In Kiddon et al. (2015), we presented the first approach for learning to construct action graphs from natural language text alone, with no other direct supervision, applied to problem of interpreting cooking recipes. We will describe the details of our method and experiments in this chapter.

Given a recipe, our task is to segment it into text spans that describe individual actions and construct an *action graph* whose nodes represent actions and edges represent the flow of arguments across actions, for example as seen in Figure 3.1. We apply the approach to understanding cooking recipes, where the action graph models how ingredients combine and flow through the actions to produce a finished dish. In a construction domain, a graph might model how the raw materials are constructed to create a desired piece of furniture. This task poses unique challenges for semantic analysis. First, null arguments and ellipses are extremely common (Zwicky, 1988). For example, sentences such as "Bake for 50 minutes" do not explicitly mention what to bake or where. Second, we must reason about how properties of the physical objects are changed by the described actions, for example to correctly resolve what the phrase "the wet mixture" refers to in a baking recipe. While building a dollhouse or baking a cake, for example, several entities split and merge, or otherwise change their states drastically (e.g., paint a dollhouse, whip egg whites into a foam), which create unique

challenges for tracking entities and reference resolution. Although linguistic context is important to resolving both of these challenges, more crucial is common sense knowledge about how the world works, including what types of things are typically baked or what ingredients could be referred to as "wet."[1]

These challenges seemingly present a chicken and egg problem — if we had a high quality semantic analyzer for instructions we could learn common sense knowledge simply by reading large bodies of text. However, correctly understanding instructions requires reasoning with exactly this desired knowledge. We show that this conflict can be resolved with an unsupervised learning approach, where we design models to learn various aspects of procedural knowledge and then fit them to unannotated instructional text. Cooking recipes are an ideal domain to study these two challenges simultaneously, as vast amounts of recipes are available online today, with significant redundancy in their coverage that can help bootstrap the overall learning process. For example, there are over 400 variations on "macaroni and cheese" recipes on allrecipes.com, from "chipotle macaroni and cheese," to "cheesy salsa mac."

We present two models that are learned with hard EM algorithms: (1) a segmentation model that extracts the actions from the recipe text, and (2) a graph model that defines a distribution over the connections between the extracted actions. The common sense knowledge is encoded in the second model which can, for example, prefer graphs that model implicit verb arguments when they better match the learned selectional preferences. The final action graph is constructed with a local search algorithm, that allows for global reasoning about ingredients as they flow through the recipe.

3.1 Task Definition

Procedural text such as a recipe defines a set of actions, i.e. *predicates*, applied to a set of objects, i.e. *arguments*. A unique challenge in procedural text understanding is to recover

[1]The goal of representing common sense world knowledge about actions and objects also drives theories of frame semantics (Fillmore, 1982) and script knowledge (Schank and Abelson, 1977). However, our focus is on inducing this style of knowledge automatically from procedural texts.

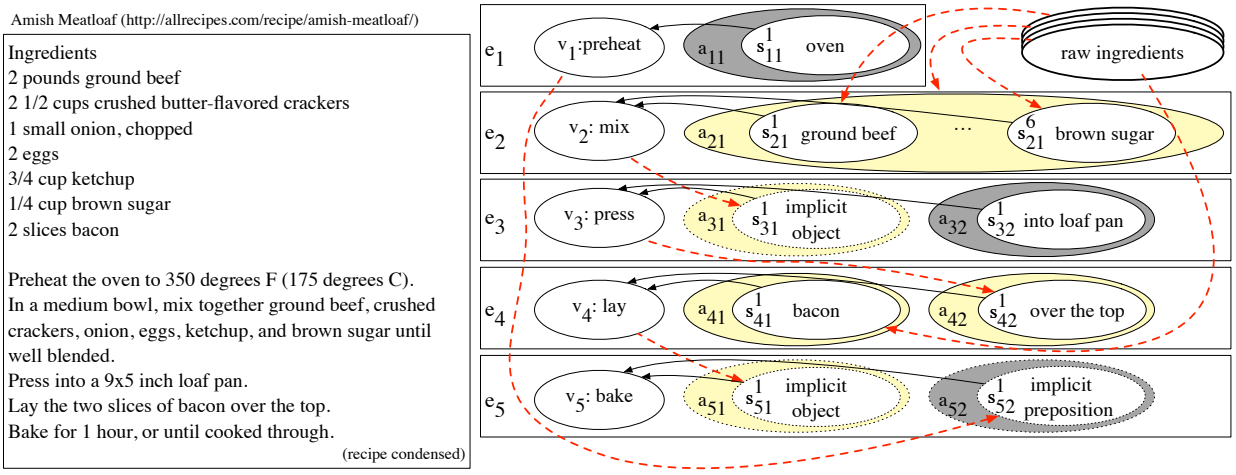


Figure 3.1: An input recipe (left) and a partial corresponding output action graph (right). Each rectangle (e_i) represents an action. The leftmost oval (v_i) in each action is the action's verb and the following ovals (a_{ij}) represents the verb's arguments. The yellow ovals represent foods; the grey ovals represent locations. Argument ovals with dotted boundaries are implicit, i.e., not present in text. The inner white ovals (s_{ij}^k) are string spans. The red dashed lines represent connections to string spans from their originating verb or raw ingredient. The string spans also connect to their associated verb in the action diagram to model the flow of ingredients. For example, there is a directed path from each raw ingredient to the implicit object of bake, representing that the object being baked is composed of all of the raw ingredients.

how different arguments flow through a chain of actions; the results of intermediate actions (e.g., "Boil the pasta until al dente.") provide the inputs for future actions (e.g., "Drain."). We represent these correspondences with an *action graph*.

Our task is to identify the events and entities and to generate the flow and ordering of the events that are required to complete the recipe. That is, given a recipe, we want to be able to answer the following questions:

1. What are the **actions** taken in this recipe?

2. For each action, what are its **arguments**?

3. Given each argument of an action, is it (1) a raw ingredient entity (e.g., flour, butter),

(2) an intermediate entity form from applying actions to one or more raw ingredients and other intermediate forms (e.g., batter, dry mixture), (3) a location (e.g., pan, oven), or (4) something else (e.g., action duration, required utensil).

4. If an argument is an intermediate entity, what is the **originating event** that created that entity?

For example, in the sentence "Pour the batter in the pan," we want to identify (1) there is a pouring action with the verb "pour" and arguments "the batter" and "in the pan," (2) "the pan" is a location, (3) "the batter" is an intermediate entity created by a previous action, and (4) which previous action created the batter. After inferring all of this information for an entire recipe, we will be able to diagram how the recipe should be executed by a human explicitly.

In this section, we first describe our structured representation of recipe text, then we define how components of the recipe connect. Finally, we will show how given a recipe and a set of connections we can construct an action graph that models the flow of ingredients through the recipe. Figure 3.1 provides a detailed running example for the section.

3.1.1 Recipe R

A recipe R is a piece of text that describes a list of instructions and a (possibly-empty) set of raw ingredients that are required to perform the instructions. The first step is to segment the text into a list of verb-argument tuples, called *actions*, $E_R = \{e_1 = (v_1, \mathbf{a}_1), \ldots, e_n = (v_n, \mathbf{a}_n)\}$. Section 3.4 will describe an unsupervised approach for learning to segment recipes. Each action e_i pairs a verb v_i with a list of arguments $\mathbf{a}_i$, where a_{ij} is the j^{th} argument of verb v_i. In Figure 3.1, each row contains an action with a verb in the white oval and its arguments in the yellow and gray ovals.

Each argument is a tuple $a_{ij} = (t_{ij}^{syn}, t_{ij}^{sem}, S_{ij})$ with a syntactic type $t^{syn}(a) \in T^{syn} = \{DOBJ, PP\}$, a semantic type $t^{sem}(a) \in T^{sem} = \{food, location, other\}$, and a list of text string spans $S_{ij} = \{s_{ij}^1, \ldots, s_{ij}^{|S_{ij}|}\}$, where s_{ij}^k is the k^{th} span in the j^{th} argument of verb

v_i. In Figure 3.1, the spans of each argument are represented by the white ovals inside of the argument ovals. For example, a_{21} contains a span for each raw ingredient being mixed in the second action (e.g., s^1_{21} ="ground beef," s^6_{21} ="brown sugar"). The syntactic type determines whether the argument is the direct object or a prepositional phrase argument of the verb in the recipe text. All other syntactic constructs are ignored and left for future work. The semantic types include food, location, and other. In Figure 3.1, the food arguments are shown as yellow ovals and the location arguments are shown as gray ovals. Arguments of other semantic types are marked as *other* if they are neither foods nor the location of the action (e.g., "Mash **using a fork**").

Implicit Arguments

It is common in recipes and other instructional texts for authors to omit arguments that can be inferred from context. Take the following recipe snippet:

Preheat the oven.
Place the dough in the bread pan.
Bake for 45 minutes.

The predicate "bake" does not specify explicitly what is to be baked and in what location that entity is to be baked. However, from the context it is easy to see that the dough in the pan should be baked in the preheated oven. A system for interpreting recipes must be able to elicit when a predicate is missing explicit arguments and what those arguments should be. In our labeled test set, we found that 33.2% of the arguments were implicit.

We augment the set of arguments for each verb to include a set of *implicit* arguments with empty string spans. This allows making connections to arguments that the author does not mention explicitly (e.g., the elided direct object of "bake" in e_5). Every verb is assigned one implicit prepositional phrase argument, and, if a verb has no argument with syntactic type $DOBJ$, an implicit direct object. These arguments have indeterminate semantic types,

which are to be determined based on how they connect to other actions. For example, in Figure 3.1, when the implicit object of "bake" is connected to the output of the "lay" action, it is inferred to be of type $food$ since that is what is created by the "lay" action. However, when the implicit PP argument of "bake" is connected to the output of the "preheat" action, it is inferred to be a location since "preheat" does not generate a food.

3.1.2 Connections C

Segmenting a recipe into actions does not alone provide a full recipe interpretation. An outline of how these actions interact with each other is required. In Figure 3.1, the object of "bake" in e_5 is unknown until connected to the "lay" event. Our interpretation of a recipe needs to be supplemented by connections from the outputs of events to the arguments of later events.

Given a segmented recipe, we can build graph connections. A *connection* identifies the origin of a given string span as either the output of a previous action or as a new ingredient or entity being introduced into the recipe.

A connection is a six-tuple $(o, i, j, k, t^{syn}, t^{sem})$ indicating that there is a connection from the output of v_o to the argument span s_{ij}^k with syntactic type $t^{syn} \in T^{syn}$ and semantic type $t^{sem} \in T^{sem}$. o, i, j, k are integers and t^{syn} and t^{sem} are discrete types from T^{syn} and T^{sem} respectively.

We call o the *origin index* and i the *destination index*. For example, in Figure 3.1, the connection from the output of the "press" verb (e_3) to "over the top" (s_{42}^1) would be $(3, 4, 2, 1, PP, food)$. If a span introduces raw ingredient or new location into the recipe, then $o = 0$; in Figure 3.1, this occurs for each of the spans that represent raw ingredients as well as "oven" and "into loaf pan."

If there is a connection to an implicit argument where $o = 0$, the implicit argument is assumed to not exist in the interpretation of the recipe; in Figure 3.1, the unused implicit prepositional arguments for e_1 through e_4 have been omitted from the diagram.

Given a recipe R, a set of connections C is valid for R if there is a one-to-one correspon-

dence between spans in R and connections in C, and if the origin indexes of connections in C are 0 or valid verb indexes in R, $\forall(o, i, j, k, t^{syn}, t^{sem}) \in C,\ o \in \{\mathbb{Z} \mid 0 \leq o \leq |E_R|\}$.

3.1.3 Action graph G

A recipe R and a set of connections C define an *action graph*, which is a directed graph $G = (V, E)$. Each raw ingredient, verb, and argument span is represented by a vertex in V. Each argument span vertex is connected to its associated verb vertex, and each connection $c = (o, i, j, k, t^{syn}, t^{sem})$ adds a corresponding edge to E. Edges from connections with semantic type $food$ propagate ingredients through the recipe; edges from connections with semantic type *location* propagate a location. Figure 3.1 shows an action graph. By following the edges, we can tell that the implicit food entity that is being baked in the final action has been formed from the set of ingredients in the mixing action and the bacon from e_4 and that the baking action occurs inside the oven preheated in e_1.

3.2 Probabilistic Connection Model

Our goal is, given a segmented recipe R, to determine the most likely set of connections, and thus the most likely action graph. As defined in Section 3.1.1, segmenting a recipe identifies the sequence of actions in the text as a list of verb-argument tuples. When describing our probabilistic connection model, we assume that the recipe has already been segmented; Section 3.4 describes our unsupervised segmentation model.

We model (1) a prior probability over C, $P(C)$ (Section 3.2.1), and (2) the probability of seeing a segmented recipe R given a set of connections C, $P(R|C)$ (Section 3.2.2). The most likely set of connections will maximize the joint probability: $P(R|C)P(C)$. A summary of this model is presented in Figure 3.2, and the details are described in the this section.

3.2.1 Connections Prior Model

The probability of a set of connections C depends on features of the incoming set of connections for each action. Let a *destination subset* $\mathbf{d}_i \subseteq C$ be the subset of C that contains all

- **Input:** A set of connections C and a recipe R segmented (**Section 3.4**) into its actions $\{e_1 = (v_1, \mathbf{a}_1), \ldots, e_n = (v_n, \mathbf{a}_n)\}$
- The joint probability of C and R is $P(C, R) = P(C)P(R|C)$, each defined below:

1. **Connections Prior (Section 3.2.1)**: $P(C) = \prod_i P(\mathbf{d}_i|\mathbf{d}_1, \ldots, \mathbf{d}_{i-1})$
   Define $\mathbf{d}_i$ as the list of connections with destination index i. Let $c_p = (o, i, j, k, t^{syn}, t^{sem}) \in \mathbf{d}_i$. Then,
   - $P(\mathbf{d}_i|\mathbf{d}_1, \ldots, \mathbf{d}_{i-1}) = P(vs(\mathbf{d_i})) \prod_{c_p \in \mathbf{d_i}} P(\mathbb{1}(o \to s_{ij}^k)|vs(\mathbf{d_i}), \mathbf{d}_1, \ldots, \mathbf{d}_{i-1}, c_1, \ldots, c_{p-1})$
     (a) $P(vs(\mathbf{d_i}))$: multinomial verb signature model (**Section 3.2.1**)
     (b) $P(\mathbb{1}(o \to s_{ij}^k)|vs(\mathbf{d_i}), \mathbf{d}_1, \ldots, \mathbf{d}_{i-1}, c_1, \ldots, c_{p-1})$: multinomial connection origin model, conditioned on the verb signature of $\mathbf{d}_i$ and all previous connections (**Section 3.2.1**)
2. **Recipe Model (Section 3.2.2)**: $P(R|C) = \prod_i P(e_i|C, e_1, \ldots, e_{i-1})$
   For brevity, define $\mathbf{h}_i = (e_1, \ldots, e_{i-1})$.
   - $P(e_i|C, \mathbf{h}_i) = P(v_i|C, \mathbf{h}_i) \prod_j P(a_{ij}|C, \mathbf{h}_i)$ (**Section 3.2.2**)
     Define argument a_{ij} by its types and spans, $a_{ij} = (t_{ij}^{syn}, t_{ij}^{sem}, S_{ij})$.
     (a) $P(v_i|C, \mathbf{h}_i) = P(v_i|g_i)$: multinomial verb distribution conditioned on verb signature (**Section 3.2.2**)
     (b) $P(a_{ij}|C, \mathbf{h}_i) = P(t_{ij}^{syn}, t_{ij}^{sem}|C, \mathbf{h}_i) \prod_{s_{ij}^k \in S_{ij}} P(s_{ij}^k|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i)$
        i. $P(t_{ij}^{syn}, t_{ij}^{sem}|C, \mathbf{h}_i)$: deterministic argument types model given connections (**Section 3.2.2**)
        ii. $P(s_{ij}^k|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i)$: string span model computed by case (**Section 3.2.2**):
           A. $t_{ij}^{sem} = food$ and $origin(s_{ij}^k) \neq 0$: IBM Model 1 generating composites (**Part-composite model**)
           B. $t_{ij}^{sem} = food$ and $origin(s_{ij}^k) = 0$: naïve Bayes model generating raw food references (**Raw food model**)
           C. $t_{ij}^{sem} = location$: model for generating location referring expressions (**Location model**)

Figure 3.2: Summary of the joint probabilistic model $P(C, R)$ over connection set C and recipe R

connections that have i as the destination index. In Figure 3.1, $\mathbf{d}_3$ contains the connection from v_2 to the implicit object as well as a connection to "into loaf pan" with an origin index of 0. Using the chain rule, the probability of C is equal to the product of the probability of each of the destination subsets:

$$P(C) \quad = \quad \prod_i P(\mathbf{d}_i|\mathbf{d}_1, \ldots, \mathbf{d}_{i-1}).$$

The probability of each destination subset decomposes into two distributions, a verb signature model and a connection origin model:

$$P(\mathbf{d}_i|\mathbf{d}_1, \ldots, \mathbf{d}_{i-1}) = P(vs(\mathbf{d_i})) \times \prod_{c_p \in \mathbf{d_i}} P(\mathbb{1}(o \rightarrow s_{ij}^k)|vs(\mathbf{d_i}), \mathbf{d}_1^{i-1}, c_1^{p-1}).$$

We define each of these distributions below.

Verb Signature Model

A destination subset $\mathbf{d}_i$ deterministically defines a *verb signature* g_i for verb v_i based on the syntactic and semantic types of the connections in $\mathbf{d}_i$ as well as whether or not each connection has a non-zero origin index. If the origin index is 0 for all connections in $\mathbf{d}_i$, we call v_i a *leaf*. (In Fig, 3.1, v_1 (preheat) and v_2 (mix) are leafs.) Formally, the *verb signature* g_i for a verb v_i given a destination set $\mathbf{d}_i$ consists of two parts:

1. **type:** $\{t^{syn} \mid \exists(o, i, j, k, t^{syn}, food) \in \mathbf{d}_i\}$

2. **leaf:** true iff $(o, i, j, k, t^{syn}, t^{sem}) \in \mathbf{d}_i \Rightarrow o = 0$

For example, in Figure 3.1, the signature for the "mix" action is $g_2 = (\{DOBJ\}, true)$ and the signature for the "lay" action is $g_4 = (\{DOBJ, PP\}, false)$. Given that there are two syntactic types (i.e., $DOBJ$ and PP) and each verb signature can either be labeled as a leaf or not, there are eight possible verb signatures.

We define a deterministic function that returns the verb signature of a destination subset: $vs(\mathbf{d}_i) = g_i$. $P(vs(\mathbf{d}_i))$ is a multinomial distribution over the possible verb signatures.

Connection Origin Model

We define $\mathbb{1}(o \rightarrow s_{ij}^k)$ as an indicator function that is 1 if there is a connection from the action with index o to the span s_{ij}^k. The probability that a string span has a particular origin depends on (1) the verb signature of the span's corresponding verb, and (2) the previous connections. If, for example, g_i has **leaf**$= true$, then the origin of s_{ij}^k must be 0. If an origin has been used in a previous connection, it is much less likely to be used again.[2]

We assume that a destination subset is a list of connections: if $c_p \in \mathbf{d}_i$, we define c_1^{p-1} as the connections that are prior to c_p in the list. Similarly, $\mathbf{d}_1^{i-1}$ is the set of destination sets $(\mathbf{d}_1, \ldots, \mathbf{d}_{i-1})$. The connection origin model is a multinomial distribution that defines the probability of an origin for a span conditioned on the verb signature and all previous connections:

$$P(\mathbb{1}(o \rightarrow s_{ij}^k) | vs(\mathbf{d_i}), \mathbf{d}_1^{i-1}, c_1^{p-1}),$$

where $c_p = (o, i, j, k, t^{syn}, t^{sem})$.

3.2.2 Recipe Model

Given a set of connections C for a recipe R, we can determine how the actions of the recipe interact and we can calculate the probability of generating a set of recipe actions $E_R = \{e_1 = (v_1, \mathbf{a}_1), \ldots, e_n = (v_n, \mathbf{a}_n)\}$. Intuitively, R is more likely given C if the destinations of the connections are good text representations of the origins. For example, a string span "oven" is much more likely to refer to the output of the action "Preheat the oven" than "Mix flour and sugar."

We define the history $\mathbf{h}_i$ of an action to be the set of all previous actions: $\mathbf{h}_i =$

[2] A counterexample in the cooking domain is separating egg yolks from egg whites to be used in separate components, only to be incorporated again in a later action.

$(e_1, \ldots, e_{i-1})$. The probability of a recipe R given a set of connections C can be factored by the chain rule:

$$P(R|C) = \prod_i P(e_i|C, \mathbf{h}_i).$$

Given C and a history $\mathbf{h}_i$, we assume the verb and arguments of an action are independent:

$$P(e_i|C, \mathbf{h}_i) = P(v_i|C, \mathbf{h}_i) \prod_j P(a_{ij}|C, \mathbf{h}_i).$$

Since the set of connections deterministically defines a verb signature g_i for a verb v_i, we can simplify $P(v_i|C, \mathbf{h}_i)$ to the multinomial distribution $P(v_i|g_i)$. For example, if g_i defines the verb to have an ingredient direct object, then the probability of "preheat" given that signature will be lower than the probability of "mix."

The probability of an argument $a_{ij} = (t_{ij}^{syn}, t_{ij}^{sem}, S_{ij})$ given the connections and history decomposes as follows:

$$\begin{aligned} P(a_{ij}|C, \mathbf{h}_i) &= P(t_{ij}^{syn}, t_{ij}^{sem}|C, \mathbf{h}_i) \\ &\times P(S_{ij}|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i). \end{aligned}$$

Argument Types Model

The first distribution, $P(t_{ij}^{syn}, t_{ij}^{sem}|C, \mathbf{h}_i)$, ensures that the syntactic and semantic types of the argument match the syntactic and semantic type of the incoming connections to spans of that argument. The probability is 1 if all the types match, 0 otherwise. For example, in Figure 3.1, this distribution would prevent a connection from the "preheat" action to the food argument a_{42}, i.e., "over the top," since the semantic types would not match.

String Span Models

The second distribution, $P(S_{ij}|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i)$, models how likely it is to generate a particular string span given the types of its encompassing argument, the connections, and history.

We assume the probability of each span is independent:

$$P(S_{ij}|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i) = \prod_{s_{ij}^k \in S_{ij}} P(s_{ij}^k|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h}_i).$$

We break this distribution into three cases. To help describe the separate cases we define the function $origin(s, C)$ to determine the origin index of the connection in C to the span s. That is, $origin(s_{ij}^k, C) = o \Leftrightarrow \exists(o, i, j, k, t^{syn}, t^{sem}) \in C$.

Part-composite Model When the encompassing argument is a food and the origin is a previous verb (i.e., $P(s_{ij}^k|t_{ij}^{syn}, t_{ij}^{sem} = food, origin(s_{ij}^k) \neq 0, C, \mathbf{h}_i)$), then the probability of the span depends on the ingredients that the span represents given the connections in C. In Figure 3.1, "the top" span in argument a_{42} represents a food mixture that contains all the ingredients from a_{21}. Certain strings are more likely to be composed of certain ingredients. For example, "dressing" is more likely given ingredients "oil" and "vinegar" than given "chicken" and "noodles". We use IBM Model 1 (Brown et al., 1993) to model the probability of a composite destination phrase given a set of origin food tokens. Let $food(s_{ij}^k, C)$ be the set of spans in food arguments such that there is a directed path from those arguments to s_{ij}^k. IBM Model 1 defines the probability of a span given the propagated food spans, $P(s_{ij}^k|food(s_{ij}^k, C))$.[3]

Raw Food Model When the encompassing argument is a food but the origin index is 0 (i.e., $P(s_{ij}^k|t_{ij}^{syn}, t_{ij}^{sem} = food, origin(s_{ij}^k) = 0, C, \mathbf{h}_i)$), then there is no flow of ingredients into the span. A span that represents a newly introduced raw ingredient (e.g., "bacon" in e_4 of Figure 3.1) should have a high probability. Additionally, sometimes authors mention ingredients that do not occur in the ingredients list, either by mistake or because the

[3]IBM Model 1 cannot handle implicit arguments. In this case, we model the probability of having an implicit food argument given the length of the connection (i.e., implicit food arguments nearly deterministically connect to the previous action). The probability of non-empty string spans is scaled accordingly to ensure a valid probability distribution.

ingredients are optional, and the probability that these spans have an origin index of 0 should also be high. Spans that denote the output of actions (e.g, "batter," "banana mixture") should have low probability. We use a naïve Bayes model over the tokens in the span $P(s|\text{is raw}) = \prod_\ell P(w_\ell|\text{is raw})$ where w_ℓ is the ℓ^{th} token in s (e.g., "mixture" would have a very low probability but "flour" would be likely).

Location Model When the encompassing argument is a location (i.e., $t_{ij}^{sem} = location$), $P(S_{ij}|t_{ij}^{syn}, t_{ij}^{sem}, C, \mathbf{h})$ models the appropriateness of the origin action's location for the destination. If the string span is not implicit, the model deterministically relies on string match between the span and the location argument of the verb at the origin index. For example, the probability of "the preheated oven" conditioned on an origin with location "oven" is 1, but 0 for an origin with location "bowl." If the span s_{ij}^k is empty, we use a multinomial model $P(loc(origin(s_{ij}^k, C))|v_i)$ that determines how likely it is that an action v_i occurs in the location of the origin verb. For example, baking generally happens in an oven and grilling on a grill, but not vice versa. For example, in Figure 3.1, the probability of the location span of "bake" is determined by $P(\text{"oven"} \mid \text{"bake"})$.

3.3 Local Search

Connections among actions and arguments identify which ingredients are being used by which action. For example, in Figure 3.1, we know that we are baking something that contains all the ingredients introduced in e_2 and e_4 because there is a path of connections from the introduction of the raw ingredients to the implicit object of "bake". We cannot make decisions about the origins of arguments independently; the likelihood of each edge depends on the other edges. Identifying the most likely set of connections is, therefore, intractable.

We adopt a local search approach to infer the best set of connections.[4] We initialize the set of connections using a sequential algorithm that connects the output of each event to

[4]Similar local search methods have been shown to work well for other NLP tasks, including recent work on dependency parsing (Zhang et al., 2014).

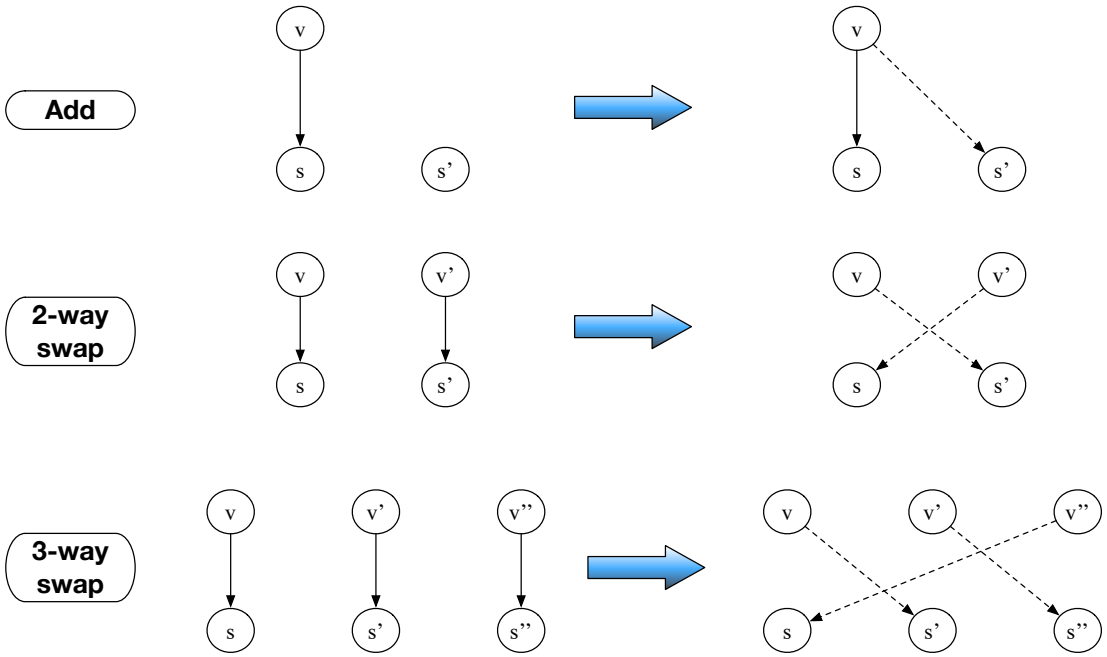


Figure 3.3: The three types of local search operators. For swaps, one of the origins can be 0.

an argument of the following event, which is a strong baseline as shown in Section 3.6. We score potential local search operators that can be applied to the current set of connections C and make a greedy selection that most improves the joint probability of the recipe and connections $P(C, R)$ until no search operator can improve the probability. We constrain the search so all verbs have a direct object (i.e., implicit direct objects connect to a previous action).

We employ three types of search operators (see Figure 3.3 for details):

Op_Add changes the origin index of a connection in C from 0 to the index of an event.

Op_2Swap swaps the origin indexes of two connections. This works even if one of the origin indexes is 0.

Op_3Swap rotates the origin indexes of three connections. This works even if one of the

origin indexes is 0. For efficiency reasons, we only allow 3-way swaps with destination indexes within 4 events of each other.

3.4 Segmentation

Our inference and learning algorithms assume as input a recipe segmented into a set of events $E_R = \{(v_1, \mathbf{a}_1), \ldots, (v_n, \mathbf{a}_n)\}$. We designed a segmentation system that could be trained on our un-annotated data set of mostly imperative sentences. Our system achieves an F1 score of 95.6% on the task of identifying the correct verbs in the test set.[5]

Segmentation Model We define a generative model for recipes as:

$$P(R) = P(n) \prod_{i}^{n} P(v_i) P(m \mid v_i) \prod_{j=1}^{m} P(a_{ij}).$$

We first select a number of verbs n in the recipe from a geometric distribution. Given the number of verbs, we select a set of verbs $\mathbf{V} = \{v_1, \ldots, v_n\}$ using a multinomial distribution. For each verb v_i, we select a number of arguments m from a separate multinomial distribution that has the probability of 0, 1, 2, or 3+ arguments given the verb, $P(m \mid v_i)$. For each argument, we generate a string using a bigram model,

$$P(a_{ij}) = \prod_{\ell} P(w_\ell | w_{\ell-1}),$$

where w_ℓ is the ℓ^{th} word of a_{ij}.

Inference Given tokenized sentence $T = (w_1, \ldots, w_k)$, we enumerate all possible segmentations and choose the one with the highest probability. To keep this efficient, we use a closed set of possible verbs (i.e., the set of words that appear first in a sentence and can be a

[5]Early efforts using a state-of-the-art parser could only achieve an F1 score of 73.6% for identifying verbs, likely due to a lack of imperative sentences in the training data. This result motivated us to develop our segmentation system.

verb) and assume a closed set of words (e.g., prepositions, adverbs) can only follow the start token in the argument bigram model. Thus, annotating the verbs in a sentence determines a unique set of argument strings. Despite scoring the segmentations for all possible sets of verbs, we found the process to be very efficient in practice.

Learning For unsupervised learning, we again employ a hard EM approach. We initialize our models, segment all of the training data, re-estimate the parameters, and iterate these steps until performance on a development set converges.

We estimate the initial verb multinomial model using counts from the first word of each sentence in the dataset, which are normally verbs in imperative sentences, and filter out any words that have no verb synsets in WordNet (Miller, 1995). All other models are initialized to be uniform.

3.5 Experimental Setup

3.5.1 Data Set

To evaluate, we use a recipe corpus from *allrecipes.com*, which has been previously used in other natural language system evaluations (Malmaud et al., 2014; Druck and Pang, 2012). Allrecipes.com has over 750,000 cooking recipes, and the recipes all use the same clean structure for separating ingredient lists from recipe steps from other non-essential information.

We collected 2456 recipes (with over 23,000 sentences) from allrecipes.com by searching for 20 dish names (e.g., including "banana muffins", and "deviled eggs"). We randomly sampled, removed, and hand labeled 33 recipes for a development set and 100 recipes for test. All models were trained on the unannotated recipes; the dev set was used to determine the stopping point for training. Each recipe in the test set has 13 actions on average.

3.5.2 Recipe Pre-processing

To pre-process each recipe, we first use the segmentation system described in Section 3.4. Then, we use a string classification model to determine the semantic type (e.g., $food$,

location, or *other*) of an argument based on its spans. In future iterations, we may allow the local search to swap the semantic types of the arguments; however, for now they are set as evidence. We identify spans as raw ingredients based on string match heuristics (e.g., in Fig. 3.1, the span "crushed crackers" represents the ingredients "crushed butter-flavored crackers"). We stem all words and ignore non-content words (e.g., determiners).

3.5.3 Sequential Baseline

Because most connections are sequential – i.e., argument spans are most often connected to the output of the previous verb – sequential connections make a strong baseline; we connect the output of each predicate to the first available argument span of the following predicate. If no argument exists, an implicit argument is created. We run this baseline with and without first identifying raw ingredients in the recipe; if raw ingredient spans are identified, the baseline will not connect the previous event to those spans. Performance suffers significantly if the raw ingredients are not identified beforehand.

3.5.4 Evaluation Metrics

We report F-measure by comparing the predicted connections from actions to spans (i.e., where the origin index > 0) against gold standard annotations. We don't evaluate connections to raw ingredients as we create those connections during pre-processing (see Section 3.5.2).

3.5.5 Model Initialization

The following section describes our experimental set up for the initialization of different probability distributions. The verb signature model (Section 3.2.2) is initialized by first identifying $food$ arguments using string overlap with the ingredient list. All other arguments' types are considered unknown, and partial counts were awarded to all verb signatures consistent with the partial information. The first verb in each recipe was assumed to be the

only leaf. The string classification model for the pre-processing step was initialized by using the initialized verb signature model to identify the types of $DOBJ$ arguments. The string classification model was estimated using the argument tokens given the types. We initialized the part-composite model (Section 3.2.2) so that exact string matches between ingredients and spans are given high probabilities and those without are given low probabilities. Given the initialized string classification model, the raw food model (Section 3.2.2) is initialized counting whether or not tokens in food arguments occur in the ingredient list. The probability of an implicit location (Section 3.2.2) is initialized to a hand-tuned value using the dev set.

3.6 Results

3.6.1 Quantitative Results

We trained our model for four iterations of hard EM until performance converged on the development set. Table 3.1 presents our results on the test set. We compare our model to the sequential baselines using both the output of our segmentation system and oracle segmentations. We perform significantly better than the sequential baselines, with an increase in F1 of 8 points over the more competitive baseline using our segmentation system and an increase of 8 points using the oracle segmentations.

3.6.2 Qualitative Results

We find that the learned models demonstrate interpretable cooking knowledge. Table 3.3 shows the top composite tokens for different ingredients as learned by the part-composite model (Section 3.2.2). The composite tokens show parts of the ingredient (e.g., after "eggs" can be split into "whites" or "yolks") or composites that are likely to contain an ingredient (e.g., "flour" is generally found in "batter" and "dough"). Unsurprisingly, the word "mixture" is one of the top words to describe a combination of ingredients, regardless of the ingredient. The model also learns modifiers that describe key properties of ingredients (e.g.,

Algorithm	**Prec**	**Rec**	**F1**
Automatic segmentations			
Sequential baseline	55.7	52.7	54.1
Sequential baseline w/ ingredients	60.4	57.2	58.8
Our model before EM	65.8	62.7	64.2
Our model after EM	**68.7**	**65.0**	**66.8**
Oracle segmentations			
Sequential baseline	67.8	65.2	66.5
Sequential baseline w/ ingredients	73.5	70.7	72.0
Our model before EM	77.1	74.8	75.9
Our model after EM	**81.6**	**78.5**	**80.0**

Table 3.1: Performance of our algorithm against the sequential baselines

Verb	Top location tokens	
bake	oven - 55.4%	min - 0.7%
mix	bowl - 32.6%	hand - 0.9%
press	pan - 24.7%	dish - 6.5%
stir	bowl - 5.5%	skillet - 2.0%
fry	heat - 11.9%	skillet - 10.2%
cool	rack - 10.5%	pan - 3.8%
boil	water - 15.5%	saucepan - 5.2%

Table 3.2: The top scoring location token for example verbs. The percentage is the percent of times the verb has that as a visible location token.

Ingredient	Top composite tokens
eggs	egg, yolk, mixture, noodles, whites, cook, top, potato, cold, fill
beef	beef, mixture, grease, meat, excess, cook, top, loaf, sauce, ground
flour	flour, mixture, dough, batter, top, crust, ingredients, sauce, dry, pie
noodles	noodles, cook, mixture, egg, sauce, top, meat, drain, pasta, layer
chicken	chicken, mixture, salad, cook, dressing, pasta, soup, breast, vegetables, noodles
pumpkin	pumpkin, mixture, pie, filling, temperature, seeds, mash, oven, crust, dough
bananas	banana, mixture, batter, muffin, bread, egg, wet, cup, ingredients, slice

Table 3.3: Examples of ingredients with their top inferred composite words

Verb	Top verb signature	(%)
add	$\{DOBJ, PP\}$	58%
	$\{DOBJ\}$	27%
combine	$\{DOBJ\}$:leaf	68%
	$\{DOBJ\}$	17%
bake	$\{DOBJ\}$	95%
grease	$\{\}$:leaf	75%
pour	$\{DOBJ, PP\}$	68%
	$\{DOBJ\}$	27%
reduce	$\{PP\}$	90%
	$\{DOBJ\}$	8%

Table 3.4: The top verb signatures for example verbs. The syntactic types identify which arguments of the verb are foods and "leaf" means no arguments of the verb connect to previous actions.

flour is "dry" but bananas are "wet") which is important when evaluating connections for sentences such as "Fold the wet mixture into the dry ingredients."

Table 3.2 shows the location preferences of verbs learned by the location model (Section 3.2.2). Some verbs show strong preferences on locations (e.g., "bake" occurs in an oven, "mix" in a bowl). The top location for a "boil" action is in "water," but in other recipes "water" is an ingredient.

Table 3.4 shows learned verb signatures. For example, "add" tends to be a non-leaf action, and can take one or two food arguments (e.g., one food argument: "Heat the pan. Add onions." vs. two food arguments: "Add the wet mixture to the dry mixture.") We learn that the most likely verb signature for "add" has two food arguments; since over 74% of the occurrences of "add" in the data set only have one visible argument, the segmentation alone is not enough to determine the signature.

Figure 3.4 gives the ingredient list and steps of a recipe for "banana buttermilk breakfast muffins topped with Nutella" from our development set. Figures 3.5 and 3.6 show the generated action graphs using the automatic recipe segmentation and the gold-standard segmentation, respectively. With both segmentations, the system gets the correct set of

actions and connections among them. The exception is that with the sentence "Let cool in a pan for 10 minutes," the automatic system could not distinguish that "cool" was part of the verb action rather than an entity to use; in both cases, the LET/LET COOL event has the output of the BAKE event as an argument. However the action graph from the automatic segmentation thinks it is an implicit prepositional phrase and the true segmentation has the output of BAKE as the direct object of "let cool". The only other differences between the segmentations occur in some other arguments, e.g., the true argument "to cool completely" is split up into "to cool" and "completely" by the automatic segmentation system. Some additional examples of output action graphs on recipes from our development set are shown in Appendix A.

3.6.3 Error Analysis

Finally, we performed an error analysis of our full model on the development set. 24% of the errors were due to missing or incorrect actions caused by segmentation errors. Among the actions that were segmented correctly, 82% of the outgoing connections were sequential. Of those, our system missed 17.6% of the sequential connections and 18.3% of the non-sequential connections.

The best F1 possible using our segmentation system is 88.0%, which is significantly higher than our results using that segmentation system. We believe the reason behind this is that our system attempts to make connections given the recipe provided. If actions are missing from the recipe parse, the parser may try to do something intelligent with what it was given but inadvertently propagate more connection errors.

Banana Buttermilk Breakfast Muffins Topped with Nutella
(http://allrecipes.com/recipe/banana-buttermilk-breakfast-muffins-topped-with-nutella)

Ingredients
1 cup all-purpose flour
1 cup whole wheat flour
1/2 cup quick-cooking rolled oats
2 teaspoons baking powder
1 teaspoon baking soda
4 over-ripe bananas, mashed
2 eggs
1/2 cup granulated sugar
1/2 cup unsweetened applesauce
1/2 cup buttermilk
1 teaspoon vanilla or maple extract
3/4 cup NUTELLA®

Directions
1. Preheat oven to 350 degrees F (180 degrees C). Spray non-stick muffin pan with cooking spray.
2. In a large bowl, combine flours, oats, baking powder and baking soda.
3. In another bowl, whisk together bananas, eggs, sugar, applesauce, buttermilk and vanilla. Pour over dry ingredients and stir until just combined. Spoon into prepared muffin pan.
4. Bake for 20 to 25 minutes or until a tester inserted into the center of a muffin comes out clean. Let cool in pan for 10 minutes. Transfer to rack to cool completely.
5. Spread each muffin with 1 tablespoon (15 mL) of NUTELLA®.

Figure 3.4: Ingredients and recipe text for "Banana buttermilk breakfast muffins topped with Nutella" from the development set

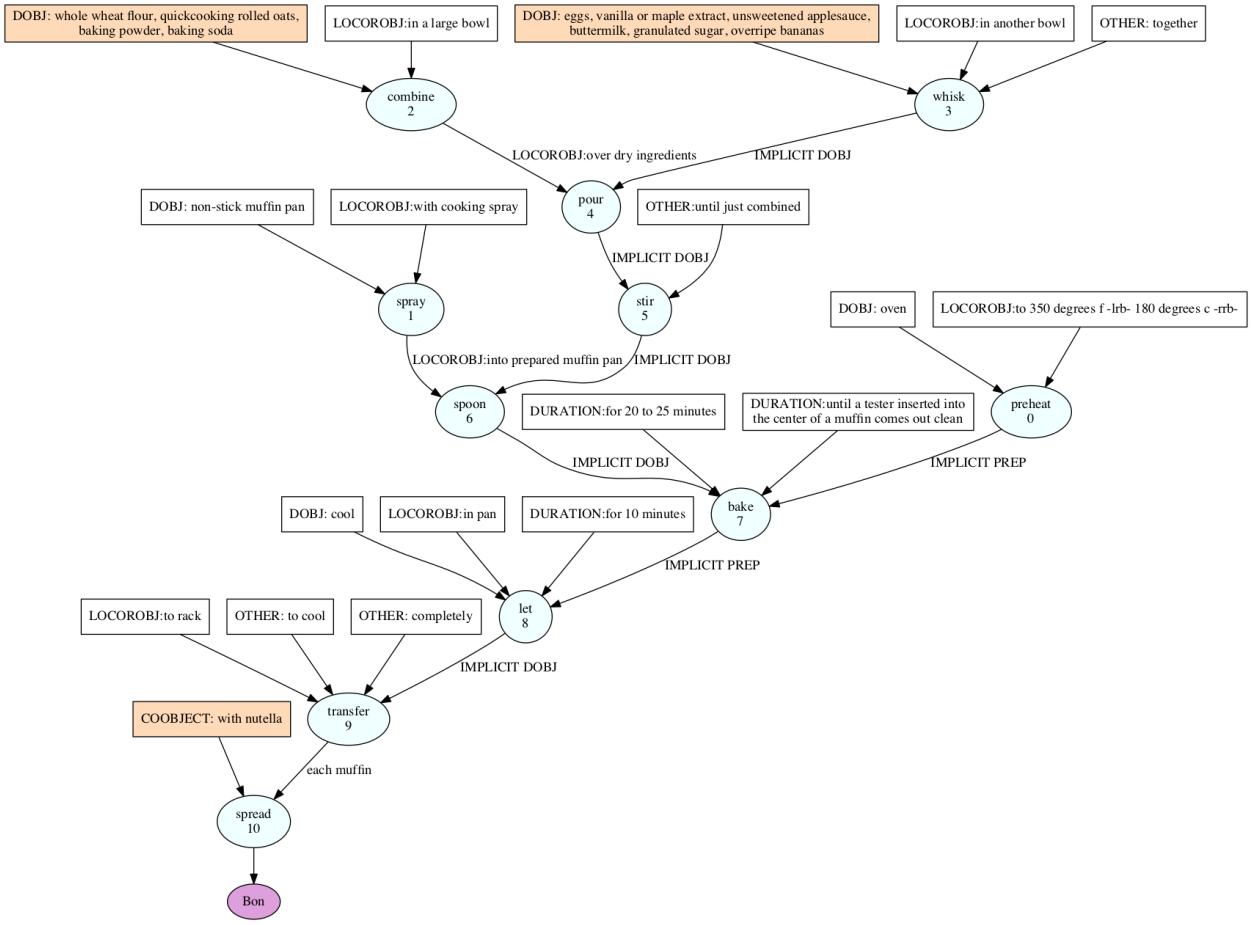


Figure 3.5: Generated action graph for "Banana buttermilk breakfast muffins topped with Nutella" using the automatically generated segmentation. Light blue ovals are the verbs of actions. Arrows into verb ovals are their arguments with the associated argument string if given. Orange boxes represent ingredients.

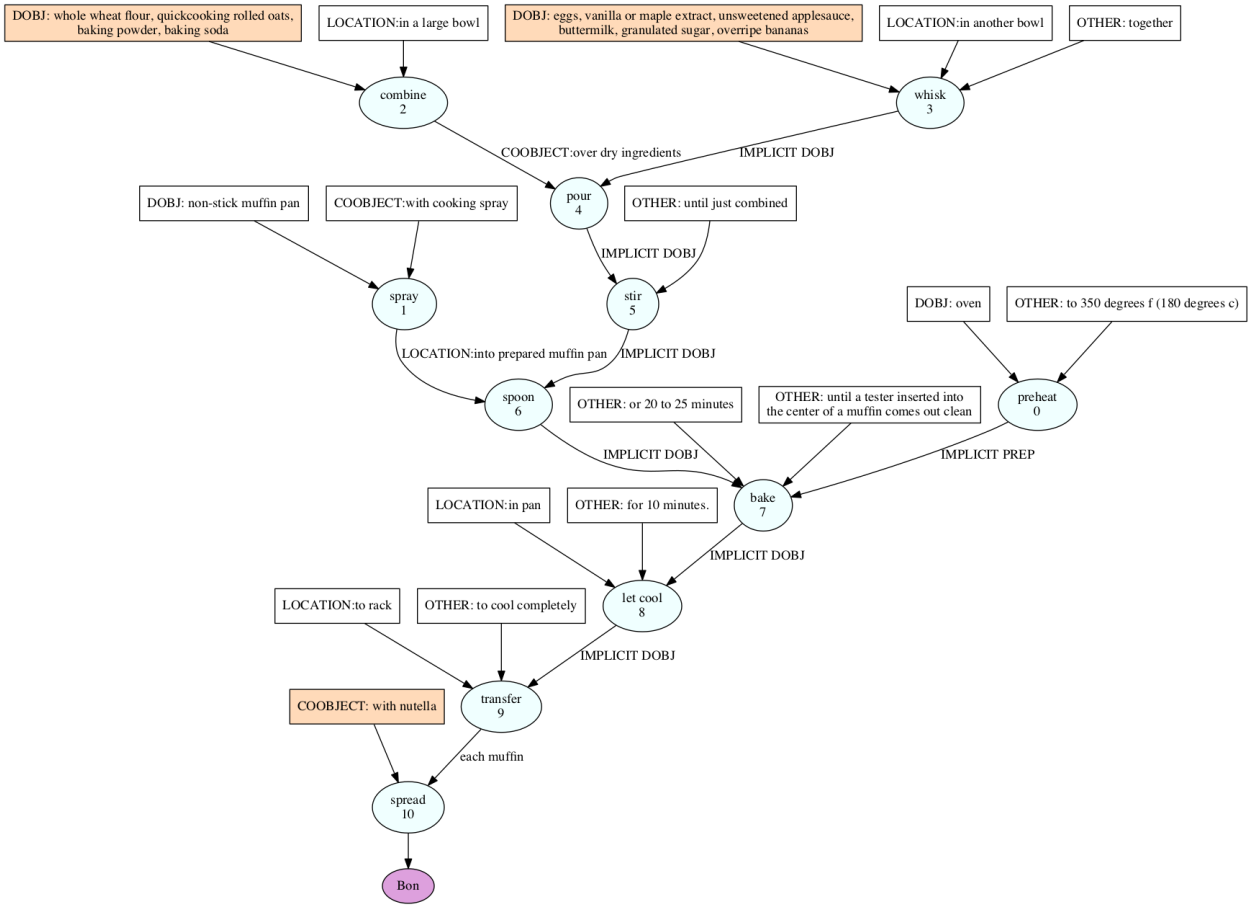


Figure 3.6: Generated action graph for "Banana buttermilk breakfast muffins topped with Nutella" using the gold-standard segmentation. Light blue ovals are the verbs of actions. Arrows into verb ovals are their arguments with the associated argument string if given. Orange boxes represent ingredients.

Chapter 4

GENERATION

In Chapter 3, we presented a method for learning how to interpret the text of a given recipe as an actionable plan. Our work shows that it is possible to train a model for recipe interpretation from a corpus of unannotated recipes. This model can correctly identify the predicate-argument structure of recipe steps and connect these steps as a unified plan structure. The interpretation model we developed also learns domain-specific knowledge as a by-product. In short, there is enough signal in these unannotated recipe corpora to learn how to understand a given recipe. However, a yet unanswered question is if the same unannotated corpora can also be used to learn how to generate *novel* recipes of the same domain.

Generating recipes from a given recipe plan has been explored since the 1980s when Robert Dale used it as a domain for his referring expression research. However, short of restricting the variation of recipes (Morris et al., 2012) or compiling vast amounts of manually-engineered knowledge (Pinel et al., 2015), there has been no previous system with the ability to generate novel recipe plans. A planner that could generate such plans would be highly complex and would require much domain-specific knowledge. Plans for object types (e.g., muffins) would require understanding of which elements are necessary to certify that the end result will be an object of that type (e.g., the use of a muffin tin). Additionally, different attributes may require those plans to add certain raw materials (e.g., *blueberry* muffins), restrict the set of raw materials allowed (e.g., *vegan* muffins), or even restrict the types of actions allowed in the plan (e.g., *no-bake* muffins). Recently, research has shown that recurrent neural network (RNN) architectures are effective for natural language generation tasks (Sordoni et al., 2015; Xu et al., 2015; Wen et al., 2015; Mei et al., 2016b). Using such mod-

els erases the need for an explicit planner since these models generate output text directly without first generating an intermediate plan representation.

One natural language task that has especially benefited from the advent of neural networks is machine translation. Machine translation converts text in one language to text in another language while retaining the same meaning. At a high level, the task of recipe generation can also be seen as a translation task: a recipe title must be translated into recipe text. However, for recipe generation, there is an additional input constraint from the materials list. Most machine translation models generate some kind of alignment between the words of the output text and the words of the input text. Since recipe generation has two inputs – the title and the material set – the words of the text can be semantically important to generate the correct goal object *or* semantically important to using the materials list correctly. Our approach for recipe generation draws from the literature for neural network machine translation while adapting for these dual inputs. A recipe generation model must be both goal-oriented as well as agenda-driven.

An issue with recurrent neural network architectures for generation is that they typically generate locally coherent language that is on topic but overall can miss pieces of information that should have been introduced in the output text. For example, when generating a cooking recipe, an RNN may lose track of which ingredients have already been mentioned. The basic internal state representation of an RNN (namely, hidden state embeddings) must contain information on both how to generate coherent natural language and how to integrate all the given semantic elements; compressing all this information into the low-dimensional embedding space may not always be feasible. Recent work has focused on adapting these architectures to improve coverage with application to generating customer service responses, such as hotel information, where a single sentence describes a few key points (Wen et al., 2015). Our focus is instead on producing longer texts with many agenda items, as required, for example, to generate complete cooking recipes.

The model for recipe generation we present in this chapter is designed to translate the combination of a recipe title and a material list into recipe text and is structured such that

it can more easily maintain a global topic (e.g., a goal object) when generating long output texts. More specifically, our *neural checklist model* generates a natural language description for achieving a goal, such as generating a recipe for a particular dish, while using a new checklist mechanism to keep track of an agenda of items that should be mentioned, such as a list of ingredients (see Figure 4.1) (Kiddon et al., 2016). The checklist model complements a general neural language model by tracking which specific agenda items have been mentioned in the discourse so far; the model learns to interpolate among three components at each time step: (1) an encoder-decoder language model that represents the overall goal of the text, (2) an attention model that tracks remaining agenda items that need to be introduced, and (3) an attention model that tracks the used, or checked, agenda items. Together, these components allow the model to learn representations that best predict which words should be included in the text and when references to agenda items should be checked off the list (see check marks in Figure 4.1). This allows for more accurate agenda-specific generation as the text unfolds.

For example, Figure 4.1 depicts a checklist model generating a cooking recipe: when generating an ingredient reference, a reference to the most probable ingredient is generated, taking into account ingredient availability from the checklist. Depending on context, the model will choose either a new ingredient to refer to or a previously-referenced ingredient. When generating the token after "Dice the", the model chooses among the unused ingredients, but when generating the reference to the entity the onion should be added to, the model chooses between the two used ingredients (i.e., "tomatoes" and "onion"). The entire model, including the check-list mechanism, uses soft decisions that can be trained jointly with backpropagation.

We evaluate our approach on two tasks: a new cooking recipe generation task and the dialogue act generation from Wen et al. (2015). In both cases, the model must correctly describe a list of agenda items: an ingredient list or a set of facts, respectively. Generating recipes additionally tests the ability to maintain coherence in long procedural texts. Experiments in dialogue generation demonstrate that the approach outperforms previous work for

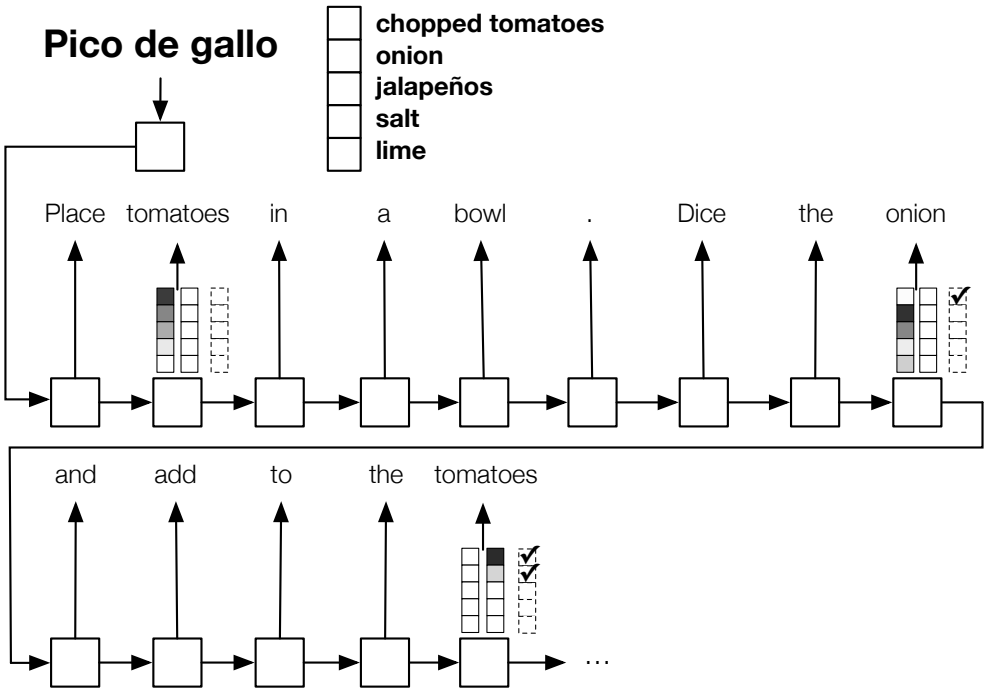


Figure 4.1: Example neural checklist model recipe generation. A checklist (right dashed column) tracks which agenda items (top boxes; "salt," "lime," etc.) have already been used (checked boxes). The model is jointly trained to interpolate an RNN (e.g., encode "pico de gallo" and decode a recipe) with attention models over new (left column) and used (middle column) items that identify likely items for each time step (shaded boxes; "tomatoes," etc.).

speech acts, with up to a 4 point BLEU improvement. Our model also scales to cooking recipes, where both automated and manual evaluations demonstrate that it maintains the strong local coherence of baseline RNN techniques while significantly improving the global coverage by effectively integrating the agenda items.

4.1 Task

Given a goal $\mathbf{g}$ and an agenda $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{|E|}\}$, our task is to generate a sequence of tokens $\mathbf{x}$ for achieving the goal by making use of each item on the agenda. For example, for the task of cooking recipe generation, the goal is the recipe title (e.g., "pico de gallo"

in Figure 4.1), and the agenda is the ingredient list (e.g., "lime," "salt"). For the task of dialogue system text generation, the goal is the dialogue type (e.g., inform or query) and the agenda contains information to be mentioned in the generated text (e.g., a hotel or restaurant's name and address). For example, if $\mathbf{g}$ ="inform" and E = {name(Hotel Stratford), has_internet(no)}, an appropriate output text sequence might be $\mathbf{x}$ ="Hotel Stratford does not have internet."

4.1.1 Relation to Machine Translation

We considered our task as a variation on machine translation when designing our model. Machine translation is the task of translating a text in one language to semantically-equivalent text in a second language. Our task does not involve translation between languages; however, we are trying to translate a given text string that represents the goal into text instructions that represent the steps required to generate that goal. The difference between our task and that of machine translation is that we have two inputs, the goal and the agenda, which affects how the output should be generated. With machine translation, almost all of the output tokens can be aligned to words in the input text. With our task, the output tokens can either be structure related to generating the goal or references to agenda items: output tokens either align to the goal or to the agenda. In Figure 4.1, the output token "tomatoes" aligns to the input ingredient string "chopped tomatoes," and the output token "onion" aligns to the input ingredient string "onion." The other output tokens identify the actions required to generate the goal "pico de gallo." The output tokens do not have to align equally to the two sources. For example in cooking recipes, only a minority – approximately 6-10% – of the output text aligns to the input ingredient list, mainly the references to ingredients. In the next section, we will describe how our model is uniquely capable of generating text while seamlessly switching focus between the goal and agenda.

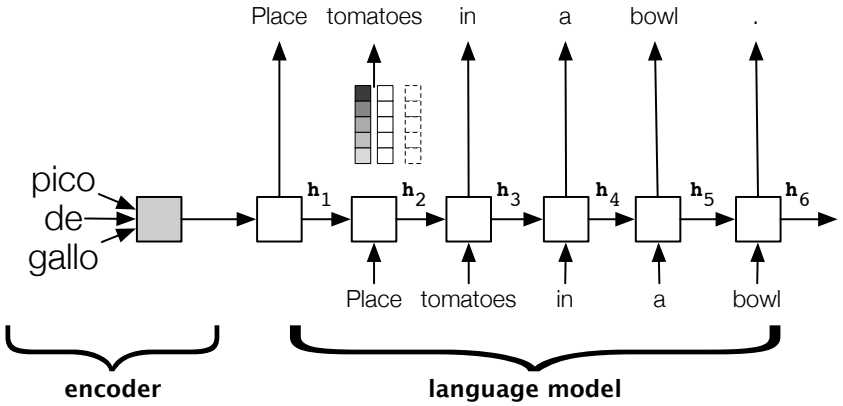


Figure 4.2: Diagram depicting how the neural checklist model works. An encoder generates a representation of the title. Then, that representation initializes a language model to decode the recipe token by token.

4.2 Model

At a high level, our neural checklist model uses a language model – initialized with the goal – to generate output text token by token. Figure 4.2 depicts more precisely how the neural checklist model will generate the pico de gallo example from Figure 4.1. The language model is initialized using a representation of the goal "pico de gallo," and at each time step i the model computes the next word using stored information from the previous time step, $\mathbf{h}_{i-1}$, as well as the previously generated word. The encoder generates a bag-of-words representation of the goal text. The language model then generates the output text token by token. At each step, the language model computes a value that represents whether it wants to generate a reference to a new agenda item, a reference to a previously-referenced agenda item, or neither. If the language model is not generating a reference to an agenda item, the language model generates the output token directly; in the pico de gallo example above, this happens when generating the tokens "Place," "in," "a," "bowl," and the period at the sentence's end. When the language model generates a reference to a new agenda item, the most likely agenda item is identified and used in the computation that generates its own reference.

In Figure 4.2, a probability distribution at the second time step over the available ingre-

dients (which includes all ingredients), represented by the shaded squares, identifies the first ingredient, "chopped tomatoes" as identified in Figure 4.1, as the most likely ingredient. A good reference token to this ingredient, "tomatoes," is then generated.

Our model stores a vector that acts as a soft checklist of what agenda items have been used so far during generation. In the pico de gallo example from Figure 4.1, when the language model wants to generate a reference to a new ingredient after generating "Dice the," the probability distribution does not give any probability mass to the "chopped tomatoes" ingredient as it has already been used.

This checklist is updated every time an agenda item reference is generated and is used to compute the available agenda items at each time step. The available items are used as an input to the language model and to constrain which agenda items can still be referenced during generation.

To allow the model to be trained properly, the three-way decision among generating a non-agenda-item token or a new/used agenda item reference is a soft probabilistic computation. Similarly, the checklist model accumulates the likelihood of having referenced each agenda item at each step, rather than making a hard decision about which agenda items have been used. Figure 4.3 shows a graphical representation of our model. The rest of this section will provide the details of our model, referring back to this diagram.

4.2.1 Basic Neural Network Framework

Our model uses a recurrent neural network (RNN) as its language model. RNNs store a vector that represents the current state of the network and sequentially apply a transition function to each symbol of an input sequence and the current state vector to generate the next state vector. The sequence of state vectors can be used to generate an output sequence. RNNs recursively apply a transition function to each symbol $\mathbf{x}_t$ of an input sequence and a hidden state $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

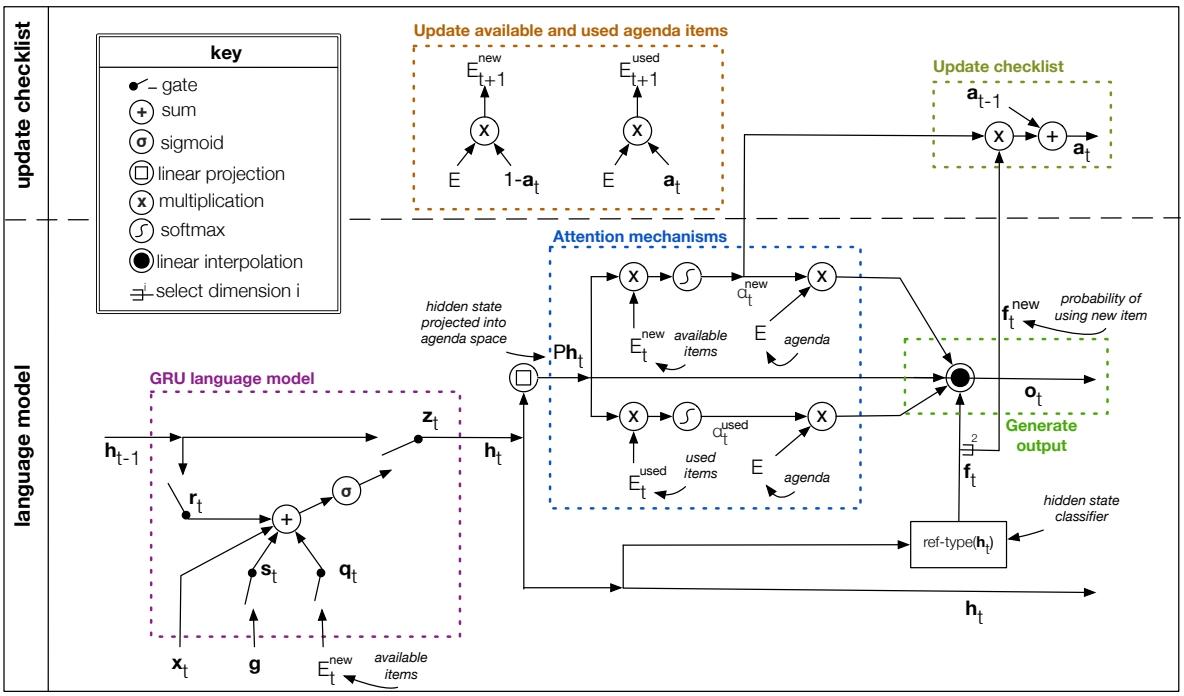


Figure 4.3: A diagram of the neural checklist model. The bottom portion depicts how the model generates the output embedding $\mathbf{o}_t$. The top portion shows how the checklist and available/used agenda item matrices are updated. The box labeled "GRU language model" is a Gated Recurrent Unit adapted to take in extra inputs: the goal $\mathbf{g}$ and the set of available agenda items E_t^{new} (Section 4.2.6). The "Attention mechanisms" box shows the attention mechanisms to select the most likely agenda items (Section 4.2.4 and 4.2.5). The output embedding $\mathbf{o}_t$ is computed via a linear interpolation among the agenda items and GRU hidden state $\mathbf{h}_t$ weighted by the hidden state classifier $ref\text{-}type()$ (Section 4.2.3).

We encode the goal as a bag-of-words and pass it to a decoder RNN that outputs a text sequence. This framework is related to the *RNN Encoder-Decoder* framework, proposed by Cho et al. (2014) and Sutskever et al. (2014), that uses one RNN to generate an embedding of an input and a second RNN to decode that embedding as a text sequence.

4.2.2 Input Variable Definitions

We assume the goal $\mathbf{g}$ and agenda items E (see Section 4.1) are each defined by a set of tokens. Goal tokens come from a fixed vocabulary $\mathcal{V}_{goal}$, the item tokens come from a fixed vocabulary $\mathcal{V}_{agenda}$, and the tokens of the text $\mathbf{x}_t$ come from a fixed vocabulary $\mathcal{V}_{text}$. In an abuse of notation, we represent each goal $\mathbf{g}$, agenda item $\mathbf{e}_i$, and text token $\mathbf{x}_t$ as a k-dimensional word embedding vector. We compute these embeddings by creating an indicator vector of the vocabulary token (or set of tokens for a goal or agenda item) in the space $\{0,1\}^{|\mathcal{V}_z|}$ and then embedding it using a $k \times |\mathcal{V}_z|$ embedding matrix, where $z \in \{goal, agenda, text\}$ depending whether we are generating a goal, agenda item, or text token. These embedding matrices are not given, and we train their parameters jointly with the training of the model through back propagation. However, for simplicity of notation, in the following sections assume that the inputs are already embedded into this low-dimensional space.

Given a goal embedding $\mathbf{g} \in \mathbb{R}^k$, a matrix of L agenda items $E \in \mathbb{R}^{L \times k}$, a checklist soft record of what items have been used $\mathbf{a}_{t-1} \in \mathbb{R}^L$, a previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^k$, and the current input word embedding $\mathbf{x}_t \in \mathbb{R}^k$, our architecture computes the next hidden state $\mathbf{h}_t$, an embedding used to generate the output word $\mathbf{o}_t$, and the updated checklist $\mathbf{a}_t$.

4.2.3 Generating Output Token Probabilities

To generate the output token probability distribution (see "Generate output" box in Figure 4.3), $\mathbf{w}_t \in \mathbb{R}^{|V_{text}|}$, we project the *output hidden state* $\mathbf{o}_t$ into the vocabulary space and apply a softmax:

$$\mathbf{w}_t = \text{softmax}(W_o \mathbf{o}_t),$$

where $W_o \in \mathbb{R}^{|V| \times k}$ is a trained projection matrix. The output hidden state is the linear interpolation of (1) content $\mathbf{c}_t^{gru}$ from a Gated Recurrent Unit (GRU) language model, (2) an encoding $\mathbf{c}_t^{new}$ generated from the new agenda item reference model (Section 4.2.4), and

(3) and an encoding $\mathbf{c}_t^{used}$ generated from a previously used item model (Section 4.2.5):

$$\mathbf{o}_t = f_t^{gru}\mathbf{c}_t^{gru} + f_t^{new}\mathbf{c}_t^{new} + f_t^{used}\mathbf{c}_t^{used}.$$

The interpolation weights, f_t^{gru}, f_t^{new}, and f_t^{used}, are probabilities representing how much the output token should reflect the current state of the language model or a chosen agenda item. f_t^{gru} is the probability of a non-agenda-item token, f_t^{new} is the probability of an new item reference token, and f_t^{used} is the probability of a used item reference. In the Figure 4.1 example, f_t^{new} is high in the first row when new ingredient references "tomatoes" and "onion" are generated; f_t^{used} is high when the reference back to "tomatoes" is made in the second row, and f_t^{gru} is high the rest of the time.

To generate these weights, our model uses a three-way probabilistic classifier, $ref\text{-}type(\mathbf{h}_t)$, to determine whether the hidden state of the GRU $\mathbf{h}_t$ will generate non-agenda tokens, new agenda item references, or used item references. $ref\text{-}type(\mathbf{h}_t)$ generates a probability distribution $\mathbf{f}_t \in \mathbb{R}^3$ as

$$\mathbf{f}_t = ref\text{-}type(\mathbf{h}_t) = softmax(\beta S\mathbf{h}_t),$$

where $S \in \mathbb{R}^{3\times k}$ is a trained projection matrix and β is a temperature hyper-parameter. $f_t^{gru} = \mathbf{f}_t^1$, $f_t^{new} = \mathbf{f}_t^2$, and $f_t^{used} = \mathbf{f}_t^3$. $ref\text{-}type()$ does not use the agenda, only the hidden state $\mathbf{h}_t$: $\mathbf{h}_t$ must encode when to use the agenda, and $ref\text{-}type()$ is trained to identify that in $\mathbf{h}_t$.

4.2.4 New Agenda Item Reference Model

The two key features of our model are that it (1) predicts which agenda item is being referred to, if any, at each time step and (2) stores those predictions for use during the generation process. These components allow for improved output texts that are more likely to mention agenda items while also avoiding repetition and references to irrelevant items not in the agenda.

These features are enabled by a *checklist vector* $\mathbf{a}_t \in \mathbb{R}^L$ that represents the probability each agenda item has been introduced into the text. The checklist vector is initialized to all zeros at $t = 1$, representing that all items have yet to be introduced. The checklist vector is a soft record with each value $\mathbf{a}_{t,i}$ in the range $[0, 1]$.[1]

We introduce the remaining items as a matrix $E_t^{new} \in \mathbb{R}^{L \times k}$, where each row is an agenda item embedding weighted by how likely it is to still need to be referenced. For example, in Figure 4.1, after the first "tomatoes" is generated, the row representing "chopped tomatoes" in the agenda will be weighted close to 0. We calculate E_t^{new} using the checklist vector (see "Update [...] items" box in Figure 4.3):

$$E_t^{new} = ((\mathbf{1}_L - \mathbf{a}_{t-1}) \otimes \mathbf{1}_k) \circ E,$$

where $\mathbf{1}_L = \{1\}^L$, $\mathbf{1}_k = \{1\}^k$, and the outer product $\otimes$ replicates $\mathbf{1}_L - \mathbf{a}_{t-1}$ for each dimension of the embedding space. $\circ$ is the Hadamard product (i.e., element-wise multiplication) of two matrices with the same dimensions.

The model predicts when an agenda item will be generated using $ref\text{-}type()$ (see Section 4.2.3 for details). When it does, the encoding $\mathbf{c}_t^{new}$ approximates which agenda item is most likely. $\mathbf{c}_t^{new}$ is computed using an attention model that generates a learned soft alignment $\boldsymbol{\alpha}_t^{new} \in \mathbb{R}^L$ between the hidden state $\mathbf{h_t}$ and the rows of E_t^{new} (i.e., available items). The alignment is a probability distribution representing how close $\mathbf{h}_t$ is to each item:

$$\boldsymbol{\alpha}_t^{new} \propto \exp(\gamma E_t^{new} P \mathbf{h}_t),$$

where $P \in \mathbb{R}^{k \times k}$ is a learned projection matrix and γ is a temperature hyper-parameter. In Figure 4.1, the shaded squares in the top line (i.e., the first "tomatoes" and the onion references) represent this alignment. The attention encoding $\mathbf{c}_t^{new}$ is then the attention-

[1] By definition, $\mathbf{a}_t$ is non-negative. We truncate any values greater than 1 using a hard tanh function.

weighted sum of the agenda items:

$$\mathbf{c}_t^{new} = E^T \boldsymbol{\alpha}_t^{new}.$$

At each step, the model updates the checklist vector based on the probability of generating a new agenda item reference, $\mathbf{f}_t^{new}$, and the attention alignment $\boldsymbol{\alpha}_t^{new}$. We calculate the new checklist $\mathbf{a}_t$ as $\mathbf{a}_t = \mathbf{a}_{t-1} + (\mathbf{f}_t^{new} \cdot \boldsymbol{\alpha}_t^{new})$.

4.2.5 Previously Used Item Reference Model

We also allow references to be generated for previously used agenda items through the previously used item encoding $\mathbf{c}_t^{used}$. This is useful in longer texts – when agenda items can be referred to more than once – so that the agenda is always responsible for generating its own referring expressions. The example in Figure 4.1 refers back to tomatoes when generating to what to add the diced onion.

At each time step t, we use a second attention model to compare $\mathbf{h}_t$ to a used items matrix $E_t^{used} \in \mathbb{R}^{L \times k}$. Like the remaining agenda item matrix E_t^{new}, E_t^{used} is calculated using the checklist vector generated at the previous time step:

$$E_t^{used} = (\mathbf{a}_{t-1} \otimes \mathbf{1}_k) \circ E.$$

The attention over the used items, $\boldsymbol{\alpha}_t^{used} \in \mathbb{R}^L$, and the used attention encoding $\mathbf{c}_t^{used}$ are calculated in the same way as those over the available items (see Section 4.2.4 for comparison):

$$\boldsymbol{\alpha}_t^{used} \propto \exp(\gamma E_t^{used} P \mathbf{h}_t),$$
$$\mathbf{c}_t^{used} = E^T \boldsymbol{\alpha}_t^{used}.$$

4.2.6 GRU Language Model

Our decoder RNN adapts a Gated Recurrent Unit (GRU) (Cho et al., 2014). Given an input $\mathbf{x}_t \in \mathbb{R}^k$ at time step t and the previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^k$, a GRU computes the next hidden state $\mathbf{h}_t$ as

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t)\mathbf{h}_{t-1} + \mathbf{z}_t\tilde{\mathbf{h}}_t.$$

The *update gate*, $\mathbf{z}_t$, interpolates between $\mathbf{h}_{t-1}$ and new content, $\tilde{\mathbf{h}}_t$, defined respectively as

$$\mathbf{z}_t = \sigma(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1}),$$
$$\tilde{\mathbf{h}}_t = \tanh(W\mathbf{x}_t + \mathbf{r}_t \odot U\mathbf{h}_{t-1}).$$

$\odot$ is an element-wise multiplication, and the *reset gate*, $\mathbf{r}_t$, is calculated as

$$\mathbf{r}_t = \sigma(W_r\mathbf{x}_t + U_r\mathbf{h}_{t-1}).$$

W_z, U_z, W, U, W_r, $U_r \in \mathbb{R}^{k\times k}$ are trained projection matrices.

We adapted a GRU to allow extra inputs, namely the goal $\mathbf{g}$ and the available agenda items E_t^{new} (see "GRU language model" box in Figure 4.3). These extra inputs help guide the language model stay on topic. Our adapted GRU has a change to the computation of the new content $\tilde{\mathbf{h}}_t$ as follows:

$$\begin{aligned}\tilde{\mathbf{h}}_t \;=\; & \tanh(W_h x_t + \mathbf{r}_t \odot U_h\mathbf{h}_{t-1} \\ & + \mathbf{s}_t \odot Y\mathbf{g} + \mathbf{q}_t \odot (\mathbf{1}_L^T Z E_t^{new})^T,\end{aligned}$$

where $\mathbf{s}_t$ is a *goal select* gate and $\mathbf{q}_t$ is a *item select* gate, respectively defined as

$$\mathbf{s}_t = \sigma(W_s\mathbf{x}_t + U_s\mathbf{h}_{t-1}),$$
$$\mathbf{q}_t = \sigma(W_q\mathbf{x}_t + U_q\mathbf{h}_{t-1}).$$

$\mathbf{1}_L$ sums the rows of the available item matrix E_t^{new}. Y, Z, W_s, U_s, W_q, $U_q \in \mathbb{R}^{k \times k}$ are trained projection matrices. The goal select gate controls when the goal should be taken into account during generation: for example, the recipe title may be used to decide what the imperative verb for a new step should be. The item select gate controls when the available agenda items should be taken into account (e.g., when generating a list of ingredients to combine). The GRU hidden state is initialized with a projection of the goal: $\mathbf{h}_0 = U_g\mathbf{g}$, where $U_g \in \mathbb{R}^{k \times k}$.

4.2.7 Training

Given a training set of (goal, agenda, output text) triples $\{(\mathbf{g}^{(1)}, E^{(1)}, \mathbf{x}^{(1)}), (\mathbf{g}^{(2)}, E^{(2)}, \mathbf{x}^{(2)}), \ldots, (\mathbf{g}^{(J)}, E^{(J)}, \mathbf{x}^{(J)})\}$, we train model parameters by minimizing negative log-likelihood:

$$NLL(\theta) = -\sum_{j=1}^{J}\sum_{i=2}^{N_j} \log p(\mathbf{x}_i^{(j)} | \mathbf{x}_1^{(j)}, \ldots, \mathbf{x}_{i-1}^{(j)}, \mathbf{g}^{(j)}, E^{(j)}; \theta),$$

where $\mathbf{x}_1^{(j)}$ is the start symbol. We use mini-batch stochastic gradient descent, and back-propagation all the way through the embeddings of goals, agenda items, and text tokens, so that the embedding matrices are learned jointly.

It is sometimes the case that weak heuristic supervision on latent variables can be easily gathered to improve training. For example, for recipe generation, we can approximate the linear interpolation weights $\mathbf{f}_t$ and the attention updates $\mathbf{a}_t^{new}$ and $\mathbf{a}_t^{used}$ using string match heuristics comparing tokens in the text to tokens in the ingredient list.[2] When this extra signal is available, we add mean squared loss terms to $NLL(\theta)$ to encourage the latent variables to take those values; for example, if $\mathbf{f}_t^*$ is the true value and $\mathbf{f}_t$ is the predicted value, a loss term $-\text{MSE}(\mathbf{f}_t^*, \mathbf{f}_t)$ is added, where MSE is the mean squared error, $\text{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$.

When this signal is not available, as is the case with our dialogue generation task, we

[2]Similar to $\mathbf{a}_t^{new}$, $\mathbf{a}_t^{used} = \mathbf{f}_t^{used} \cdot \boldsymbol{\alpha}_t^{used}$.

instead introduce a mean squared loss term that encourages the final checklist $\mathbf{a}_{N_j}^{(j)}$ to be a vector of 1s (i.e., every agenda item is accounted for). This is similar to the standard neural machine translation approach to using attention models where a loss is included in the objective function that ensures the accumulated attention from each time step adds up to 1 for each word of the input, i.e., all input words have been accounted for in the output. The difference is that the amount of attention that is accumulated at each time step is modulated by the probability of using the attention model at that step; that is, if the probability of generating an agenda item, $\mathbf{f}_t^{new}$, is low, then the attention from the new agenda item reference model at that step will only minimally add to the final accumulated attention.

4.3 Experimental Setup

Our model was implemented and trained using the Torch scientific computing framework for Lua.[3]

4.3.1 Experiments

We evaluated neural checklist models on two natural language generation tasks. The first evaluation was a cooking recipe generation task. Given a recipe title (i.e., the name of the dish) as the goal and the list of ingredients as the agenda, the task is to generate the correct recipe text. Recipe generation is an ideal task on which to evaluate our model, as the outputs tend to be longer than most generation tasks and maintaining coherence is difficult. Our second evaluation is a task from Wen et al. (2015) of generating dialogue output for hotel and restaurant information systems. The task is to generate a natural language text output for a system given a query type (e.g., informing or querying) and a list of facts to convey (e.g., a hotel's name and address).

[3]http://torch.ch/

4.3.2 Parameters

We constrain the gradient norm to 5.0 and initialize parameters uniformly on $[-0.35, 0.35]$. We used a beam of size 10 for generation. Based on dev set performance, a learning rate of 0.1 was chosen, and the temperature hyper-parameters (β, γ) were $(2, 1)$ for the recipe task and $(1, 10)$ for the dialogue task. The recipe task had an hidden state size of $k = 200$; the dialogue task had $k = 80$ to compare to previous models. For the recipe task, we use a batch size of 30 to balance training speed and amount of training data used; for the dialogue task, the batch size is 10.

4.3.3 Recipe Data and Pre-processing

We use the Now You're Cooking! (NYC) software's recipe dataset. NYC is an Windows recipe management program[4]. The NYC website has data files that include over 158,000 recipes in easy-to-download compressed files[5]. The recipes in these text files are in a special Meal-Master$^{\text{TM}}$ format for storing recipes. These files have less standardized structure than other resources, but their size and the fact that they come pre-compiled (i.e., instead of requiring web scraping to obtain) makes the NYC corpus ideal to work with. Abend et al. (2015) used this data set for evaluation, which also promotes this as a new standard for recipe data sets.

For each recipe in the corpus, we extracted the title, ingredient list, and text. We heuristically removed sentences that were not recipe steps from the text (e.g., author notes, nutritional information, publication information). To make ingredient references more likely to be single tokens, we ran the word2phrase tool bundled with word2vec[6] on the training data ingredient lists to find common two-token ingredient phrases (e.g., "powdered_sugar") and then collapsed tokens in the recipe text based on discovered phrases. Titles and ingredients

[4]http://www.ffts.com/

[5]Recipes and format at http://www.ffts.com/recipes.htm

[6]https://code.google.com/p/word2vec/

were cleaned of non-word tokens. Ingredients additionally were stripped of amounts (e.g., "1 tsp"). 82,740 recipes were used for training, and 1,000 each for development and testing. As mentioned in Sec. 4.2.7, we approximate true values for the interpolation weights and attention updates for recipes using heuristics that match tokens in the text to tokens in the ingredient list. The first ingredient reference in a sentence cannot be the first token or after a comma (e.g., the bold tokens cannot be ingredients in "**oil** the pan" and "in a large bowl, **mix** [...]").

Data Statistics

Automatic recipe generation is difficult due to the length of recipes, the size of the vocabulary, and the variety of possible dishes. Figure 4.4 shows a graph of length of recipe to the number of recipes of that length in our training data set. The average recipe length is 78 tokens, and the longest recipe has 814 tokens.[7]

The vocabulary of the recipe text from the training data (i.e., the text of the recipe not including the title or ingredient list) has 14286 unique tokens. Figure 4.5 shows the count of each of the vocabulary tokens in the training data; tokens are sorted by count. About 31% of tokens (4462 tokens) in the recipe vocabulary occur at least 100 times in the training data; 8.6% of the tokens (1241 tokens) occur at least 1000 times. Any token that occurred under 10 times was not included in the vocabulary and marked as an unknown token.

The training data also represents a wide variety of recipe types, defined by the recipe titles. Figure 4.6 shows the count of each of the 3706 unique title vocabulary tokens; tokens are sorted by count. Only 20% of the title tokens (723 tokens) in the title vocabulary occur at least 100 times in the training data, which demonstrates the large variability in the titles. After the token "with", the most seen title token is "chicken" with 9505 mentions.

---

[7]For all recipe data statistics, the tokens count is performed after collapsing tokens from phrase2vec (See Section 4.3.3).

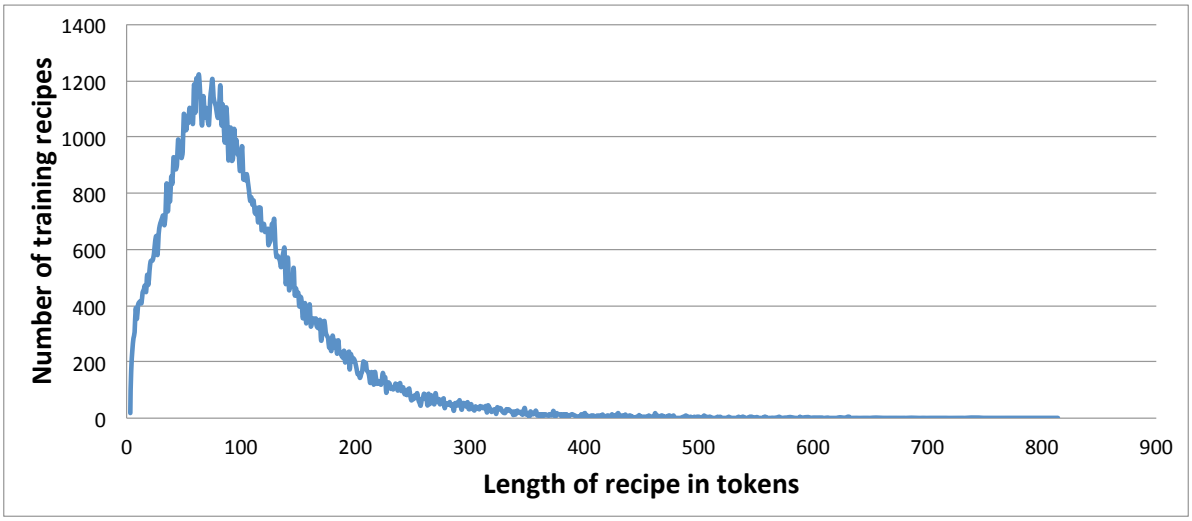


Figure 4.4: A graph of the recipe length in collapsed tokens in the Now You're Cooking training data set (e.g., "powdered_sugar" is counted as one token)

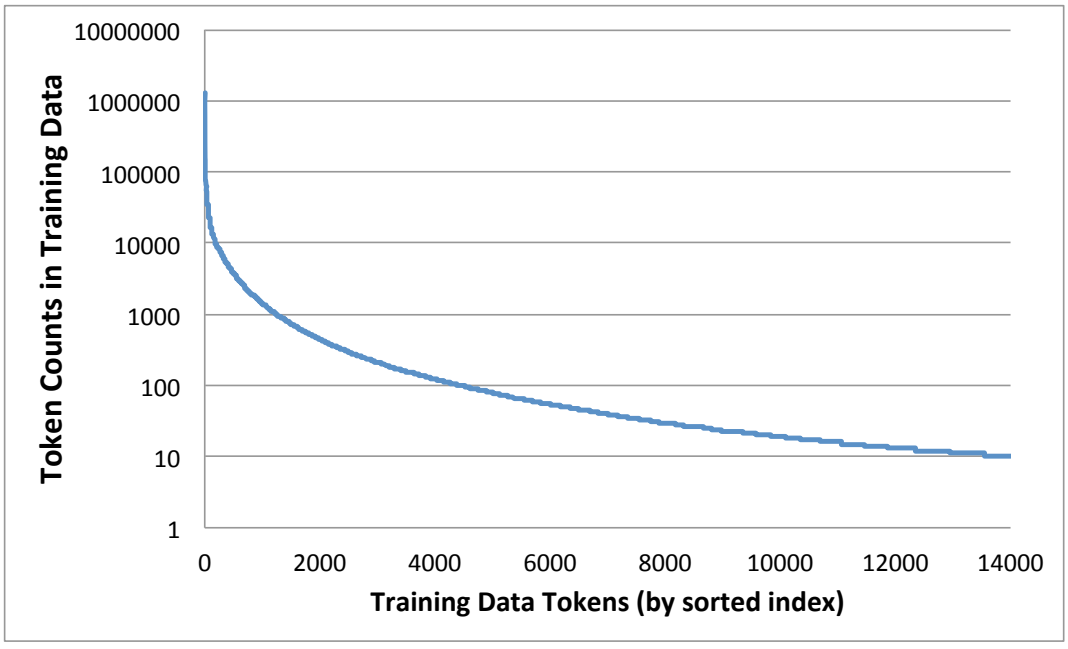


Figure 4.5: A graph of the counts of recipe text vocabulary tokens in the training data set. Tokens are sorted by count.

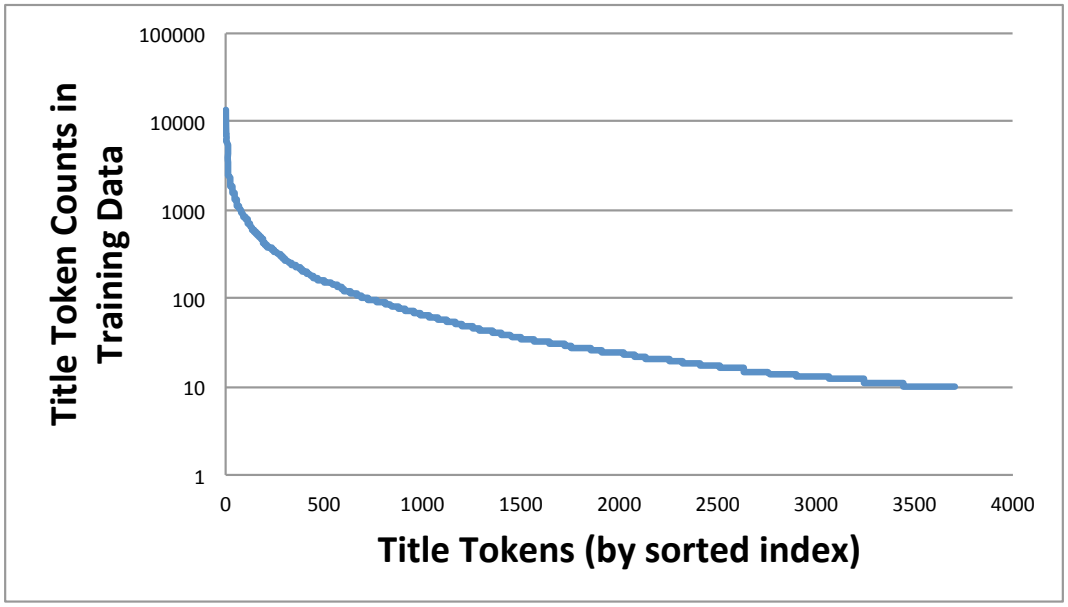


Figure 4.6: A graph of the counts of title vocabulary tokens in the training data set. Tokens are sorted by count.

4.3.4 Dialogue System Data and Processing

We used the hotel and restaurant dialogue system corpus from Wen et al. (2015) and the same train-development-test split used in that paper. We used the same pre-processing, sets of reference samples, and baseline output, and we were given model output to compare against.[8] For training, slot values (e.g., "Red Door Cafe") were replaced by generic tokens (e.g., "NAME_TOKEN"). After generation, generic tokens were swapped back to specific slot values. Minor post-processing included removing duplicate determiners from the re-lexicalization and merging plural "-s" tokens onto their respective words.

[8]We thank the authors for sharing their system outputs.

4.3.5 Models

Our main baseline *EncDec* is a model using the RNN Encoder-Decoder framework proposed by Cho et al. (2014) and Sutskever et al. (2014). The model encodes the goal and then each agenda item in sequence and then decodes the text using GRUs. The encoder has two sets of parameters: one for the goal and the other for the agenda items. For the dialogue task, we also compare against the *SC-LSTM* system from Wen et al. (2015) and the handcrafted rule-based generator described in that paper. For the recipe task, we also compare against a basic attention model, *Attention*, that generates an attention encoding from comparing the hidden state $\mathbf{h}_t$ to the agenda. That encoding is added to the hidden state, and a nonlinear transformation is applied to the result before projecting into the output space.

Our model is labeled *Checklist*. The *Checklist+* is the Checklist model with an altered decoding process to correct for missed agenda items: if the generated text does not use every agenda item, embeddings corresponding to missing items are multiplied by increasing weights. If a newly generated text has the same or fewer items generated, the process ends and the previously text is selected.

We also consider two ablations of our checklist model on the recipe task. First, we evaluate the adapted GRU language model: the ablated model gets the goal and agenda information at each step, but the agenda never updates and the GRU hidden state $\mathbf{h}_t$ is used as the output, $\mathbf{o}_t = \mathbf{h}_t$. The second ablation updates the checklist and available agenda E_T^{new} but only uses $\mathbf{h}_t$ as the output.

4.3.6 Metrics

We compare on BLEU-4 using the Moses system,[9] and the METEOR evaluation metric (Denkowski and Lavie, 2014). For recipes, we also compute the percentage of the ingredients from the agenda that are referred to, using a string-match heuristic that matches ingredient tokens to recipe text tokens, as well as TF-IDF cosine similarity. The ingredient percentage

[9]http://www.statmt.org/moses/

Model	BLEU-4	METEOR	% ingredients	Cosine Sim
EncDec	3.51	9.5	23.2%	0.728
Attention	3.74	9.83	30.0%	0.724
Checklist	4.93	12.0	59.5%	0.768
- only GRU	4.63	10.7	33.1%	0.748
- $\mathbf{o}_t = \mathbf{h}_t$	**5.54**	11.7	36.4%	0.756
Checklist+	5.24	**12.8**	**74.6%**	**0.769**

Table 4.1: Quantitative results on the recipe task. The line with $\mathbf{o}_t = \mathbf{h}_t$ has the results for the non-interpolation ablation.

Model	Syntax	Ingredient use	Follows goal
EncDec	4.512	3.492	3.321
Checklist+	4.118	3.906	3.657
Truth	4.313	4.266	4.189

Table 4.2: Human evaluation results on the generated and true recipes. Scores range in $[1, 5]$.

is a useful metric, but approximate due to noise in the ingredient list, recipes that refer to multiple ingredients at once (e.g., "all ingredients"), and references without exact token matches (e.g., "meat" to refer to "pork"). The tf-idf weights were computed based on the test set recipes.

4.4 Recipe Generation Results

Fig. 4.1 shows metrics comparing the different systems for recipe generation. Our checklist model performs better than both baselines in all metrics. With the exception of the no interpolation ablation on BLEU, our model also improve over its ablations. All BLEU and METEOR scores are low, which is expected when using these metrics on long texts; we also report TF-IDF cosine similarity scores that increase over the baseline and compare the generations at the document level. Our model significantly outperforms the baselines and ablations in terms of ingredient list coverage, with Checklist doubling and Checklist+ nearly

tripling the number of ingredients identified compared to the EncDec baseline.

4.4.1 Human Evaluation

We used Amazon Mechanical Turk to have humans evaluate the generated recipes on (1) the grammaticality of the recipe, (2) how well the recipe matches the ingredient list, and (3) how well the generated recipe fit the title. We selected 100 recipes at random from the test set and rated the true recipe, EncDec baseline, and Checklist+ model; five turkers evaluated each recipe-model combination. For each question we used a Likert scale with five options translating to a $[1, 5]$ scale.

Table 4.2 shows the averaged scores over the responses. The Checklist+ greatly outperforms the baseline in terms of using the ingredient list correctly and generating a recipe that fits the title, which shows our model's ability to generate coherent goal-oriented text while successfully covering more of the ingredient list. Perhaps surprisingly, the baseline beats both the true recipes and the recipes generated by the Checklist+ model in terms of having better syntax. This is partly due to noise in the parsing of the true recipes and partly because the baseline tends to output shorter and simpler but less goal-specific text, which achieve higher syntax scores but worse on other metrics. An error analysis on the dev set shows that the baseline generates recipes that use the phrases "all ingredients" or "the ingredients" over a quarter of the time, whereas only 8.9% of true recipes use that construction. This simplifies the recipe text, but using all ingredients in one recipe step is unlikely to generate most dishes correctly.

4.4.2 Qualitative Analysis

Fig. 4.7 shows three development set recipes with generations from the the EncDec and Checklist+ models. The EncDec model is much more likely to both use incorrect ingredients and to introduce ingredients more than once (e.g., "Blend [...] baking soda, salt, and baking soda"). In the "Almond-raspberry thumbprint cookies" example, the Checklist+ model refers to both almond and raspberry fillings as "filling"; generating the most precise necessary

referring expressions is future work. The Checklist+ model is much better at properly using the ingredient list than the baseline method. There is still a way to go to learn and generate a correct ordering of actions for the recipe title (i.e., goal), but we are very encouraged by the many similarities of our outputs to the recipe titles.

Text generated with neural networks tends to be simple and bland (Li et al., 2016). We analyzed the vocabulary of the true development set recipes compared to the vocabulary used by our Checklist+ model as well as the EncDec baseline. Figure 4.8 shows a graph of the counts of the different tokens in the development set recipe vocabulary. In the development set recipes, there are 5,141 unique tokens used with 120 tokens used at least 100 times and 1,034 tokens used at least 10 times. Figure 4.9 shows the same graph but using the token counts from the development set recipes generated by the Checklist+ model. We find that the vocabulary used by the Checklist+ model is about a third as large as the true vocabulary, which is consistent with expectations about neural network generation models. In terms of highly used tokens, the Checklist+ vocabulary is actually more similar to the true vocabulary: 110 tokens are used at least 100 times and 530 tokens, about half as many as the true vocabulary, are used at least 10 times. Figure 4.10 shows the graph for the token counts from the development set recipes generated by the EncDec baseline. The vocabulary here is more limited: about 19% of the size of the true vocabulary. 101 tokens are used at least 100 times and only 440 tokens, 90 fewer than the Checklist+ vocabulary, are used at least 10 times. Our Checklist+ model uses a much more varied vocabulary than the baseline, but there is still work to do to generate a vocabulary as large as used in the true recipes. Training the model to use a larger vocabulary, perhaps by changing the objective function as in Li et al. (2016), is left to future work.

4.5 Dialogue System Results

Figure 4.3 shows our results on the hotel and restaurant dialogue system generation tasks. HDC is the rule-based baseline from Wen et al. (2015). For both domains, the checklist model achieved the highest BLEU-4 and METEOR scores, but both neural systems, including the

Model	Hotel		Restaurant	
	BLEU	**METEOR**	**BLEU**	**METEOR**
HDC	55.52	48.10	44.39	43.42
EncDec	86.88	59.34	77.27	52.81
SC-LSTM	86.53	60.84	74.49	54.31
Checklist	**90.61**	**62.10**	**77.82**	**54.42**

Table 4.3: Quantitative evaluation of the top generations in the hotel and restaurant domains

EncDec baseline, performed very well. In particular, the neural checklist model gets a jump in improvement in BLEU over SC-LSTM, whereas the METEOR scores are much more similar. We attribute this to the neural checklist model's computation of the output hidden state: if the model is likely generating an agenda item reference, the output hidden state vector will reflect the embedding of that particular agenda item (see Section 4.2.3 for details). The model is trained to use the agenda item embeddings when generating agenda item references; if an item is not on the agenda, it is unlikely that words that reference it will appear in the output text. This improves the precision of our output since output text relating to items not on the agenda is much less likely to be generated; since BLEU is a precision-based metric, our model should, and does, show an improvement in BLEU. METEOR is a recall-based metric; the similarity in scores show that both the neural checklist model and SC-LSTM are able to generate text that uses the whole of the given agenda. The power of our model is in generating longer texts, but this experiment on dialogue system responses shows that our model can generalize well to other tasks with different kinds of agenda items and goals.

Title: Almond-raspberry thumbprint cookies
Ingredients: 1 cup butter or margarine softened, 1 cup sugar, 1 can solo almond filling, 2 egg yolks, 1 tsp almond extract, 2 1/2 cups all-purpose flour, 1/2 tsp baking powder, 1/2 tsp salt, 1 can solo raspberry filling

Truth	**EncDec**	**Checklist+**
Beat butter and sugar in medium bowl with electric mixer until light and fluffy. Add almond filling, egg yolks and almond extract; beat until blended. Stir in flour, baking powder and salt with wooden spoon to make soft dough. Cover; refrigerate at least 3 hours or overnight. Shape dough into 1 " balls. Place on ungreased baking sheets, about 11/2 " apart. Press thumb into center of each ball to make indentation. Spoon 1/2 teaspoon raspberry filling into each indentation. Bake 11-13 minutes or until edges of cookies are golden brown. Cool on baking sheets 1 minute. Remove from baking sheets; cool completely on wire racks.	In a large bowl, beat butter and sugar until light and fluffy. Add **eggs**, one at a time, beating well after each addition. Blend in flour, baking powder, **baking soda**, salt and **baking soda**. Add to creamed mixture and mix well. Stir in **nuts** and **vanilla**. Drop dough by rounded tablespoonfuls 2 " apart onto ungreased cookie sheets. Bake 8 to 10 minutes or until edges are lightly browned. Cool on wire racks.	In a mixing bowl, cream butter and sugar until light and fluffy. Gradually beat in filling. Stir in flour, baking powder, salt and egg yolks. Mix in filling. Drop by rounded teaspoonfuls 2 inches apart onto ungreased cookie sheet. Bake at 350 for 12 to 15 minutes or until lightly browned. Cool on wire rack. Store in tightly covered container.

Title: Skillet chicken rice
Ingredients: 2 lb chicken pieces skinned, 3 cups mushrooms fresh sliced, 4 carrots peeled sliced 1/2", 3/4 cups long grain rice, 1/2 c onion chopped, 1 tsp poultry seasoning, 1 tsp bouillon chicken granules, 1/4 tsp salt

Truth	**EncDec**	**Checklist+**
Spray a 12-inch skillet with non-stick spray coating. Brown chicken pieces on all sides over medium heat about 15 minutes. Remove chicken. Drain fat from skillet, if neccessary. Add mushrooms, carrots, rice, onion, bouillon, poultry seasoning, 2 cups water, salt. Place chicken atop rice mixture. Cover; simmer 30 minutes or till chicken and rice are done.	In a large skillet, brown chicken in **oil**. Add **chicken** and rice; cook, stirring, until chicken is no longer pink. Add **rice** and **broth**; bring to a boil. Reduce heat to low; cover and cook 5 minutes or until rice is tender and liquid is absorbed. Stir in **rice**. Heat to boiling; reduce heat. Cover and simmer until rice is tender, about 20 minutes. Stir in **rice**. Heat to boiling; reduce heat. Cover and simmer 20 minutes or until rice is tender and liquid is absorbed and rice is tender.	In large skillet, heat rice and onion. Add carrots and mushrooms; cook, stirring constantly, until hot and bubbly. Stir in seasonings. Cook and stir over medium heat until hot and bubbly. Stir in chicken and salt. Cover and cook 5 minutes or until chicken is no longer pink in center. Serve over **rice**.

Title: Chocolate covered potato chips
Ingredients: 8 oz sweet milk chocolate, 24 thick sliced ridged potato chips

Truth	**EncDec**	**Checklist+**
Melt 6 ounces of chocolate. Remove from heat. While chocolate is still hot, mix the 2 ounces of chocolate into the melted chocolate, in 3 additions, stirring until each addition is completely melted before adding the next. Dip the potato chips, 1 at a time, in the chocolate. Coat completely and lift with a small fork. Shake off excess chocolate by rapping the fork on the edge of the bowl lightly. Remove any drips from the bottom by running the fork across the edge of the bowl. Slide the chips onto a cookie sheet lined with parchment or wax paper. Allow to cool until solid. Let chips sit at room temperature or in the refrigerator.	Preheat oven to 350 degrees. Grease and flour a 9 x 13 inch pan. Bake at 350 degrees for 30 minutes.	Bake in a 400 degree f oven for 15 minutes or until lightly browned. Drain on paper towels. Melt chocolate in top of double boiler over simmering water. Remove from heat and cool to room temp. Add potato mixture to the potato mixture; mix well. Cover tightly with plastic wrap. Refrigerate until firm enough to handle. Shape into croquettes. Fry in **hot oil** until golden brown. Drain on paper towels. Serve hot.

Figure 4.7: Example development set generated recipes. Tokenization, newlines, and capitalization changed for space and readability. Bolded ingredient references are either ingredients not in the list and/or duplicated initial ingredient references.

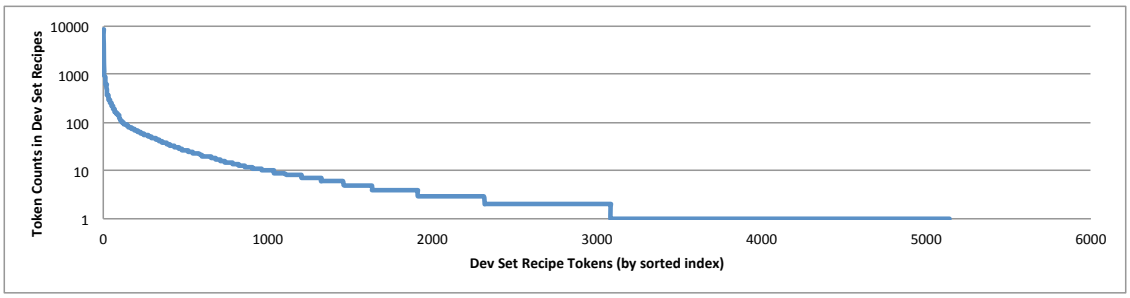


Figure 4.8: Development set recipe token counts

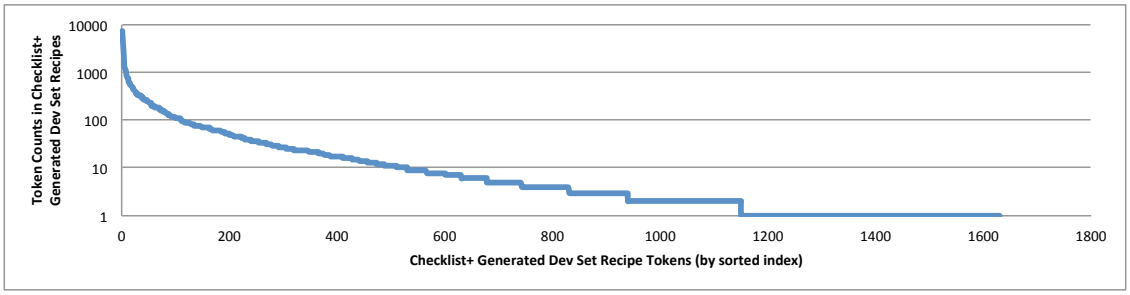


Figure 4.9: Development set recipe token counts from Checklist+ generated recipes

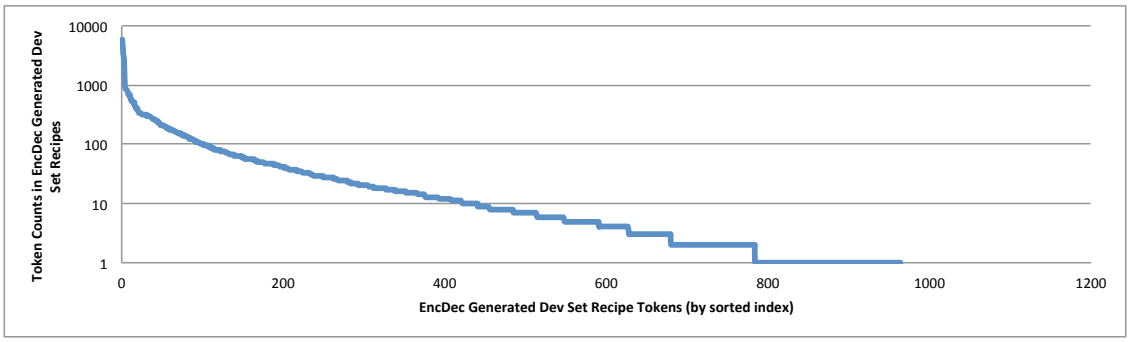


Figure 4.10: Development set recipe token counts from EncDec generated recipes

Chapter 5

JOINT GENERATION OF RECIPE TEXTS AND ACTION GRAPHS

As seen in the previous chapter, by using a neural checklist model, we can generate the text of new recipes specified by the goal object and the list of materials to use (see Chapter 4 for details). The neural checklist model is a general architecture for natural language generation tasks that have a specified goal and agenda. Instructional recipe generation is one task that fits that framework nicely, and, as shown by the experimental results in Section 4.5, the neural checklist model can also be applied to dialogue system output generation. However, instructional recipes have certain unique properties (e.g., actions generate entities) that are not explicitly modeled by a neural checklist model. In this chapter, we present an adaptation of the neural checklist model, *the neural recipe model*, that is specifically designed to generate instructional recipes.

One disadvantage neural checklist models have for generating valid instructional recipes is that they do not explicitly keep track of intermediate entities, which are a unique phenomenon to recipes. Intermediate entities are entities generated by actions in the recipe text that are used as the arguments of subsequent actions in order to generate the goal object. For example, a recipe for a cake will first combine the ingredients into a batter entity, and then the batter is baked to become the cake. In the neural checklist model, keeping track of intermediate entities is left to the language model: the hidden state of the language model should encode the current state of the recipe and, hopefully, what intermediate entities have been created and not used yet. Not only is this additional information that the hidden state of the language model must encode in order to generate a valid recipe, but by keeping intermediate entity information implicit in the model, it is more difficult to identify which recipe

tokens are references to intermediate entities.

The ability to identify intermediate entity references is important for applications in which we want to generate an action graph for the generated recipe (e.g., having a robot execute a generated recipe). The neural checklist model is semi-interpretable: we can extract structured information from the values of the variables of the generation model to identify when ingredients are introduced into the output text. Yet this is still far less structure than is required for the automatic execution of a generated instructional recipe. We could extract an action graph for the recipe using the methods outlined in Chapter 3 as part of a pipeline approach that first generates text and then the associated action graph. However, this would introduce a second source of errors, and it is possible that the extracted action graph could have a different structure than the implicit structure being used by the neural checklist model. A better solution is one that jointly generates the recipe text and more of the underlying action graph structure.

The neural recipe model is trained not only to generate text that is goal oriented and that uses on ingredient list but also to generate text that is structured like a recipe. Every time the text mentions a particular action, an intermediate entity is generated that must be used by a later action (or be the goal entity). Every time the text references an intermediate entity – even implicitly – it must select from the set of available intermediate entities. As an example of how the neural recipe model may improve recipe generation, the following is the start of the recipe generated by the Checklist+ model for "Chocolate covered potato chips" from Figure 4.7:

> *Bake in a 400 degree f oven for 15 minutes or until lightly browned. Drain on paper towels. Melt chocolate in top of double boiler over simmering water. Remove from heat and cool to room temp. Add potato mixture to the potato mixture; mix well.*

This recipe contains certain errors that may have been less likely if the generation process had

been jointly generating recipe structure. For example, the action in the first sentence, "*Bake in a 400 degree f oven [...],*" is a BAKE event with an implicit direct object argument. As a general rule, implicit arguments in recipe text refer to previously generated entities. However, as the first sentence of this recipe, there are no previous actions that could have generated this implicit entity. If we could design a recipe text generation model that would force the model to choose a previously generated entity if it wanted to generate an implicit entity reference, then this first sentence would be extremely unlikely to be generated. Additionally, the last sentence in this snippet, "*Add potato mixture to the potato mixture [...],*" is an ADD action with two arguments both referred to as "the potato mixture". Unfortunately, there are no potato mixtures generated in prior steps for either of these arguments to refer to. The first "potato" token is introducing the "24 thick sliced ridged potato chips" ingredient, but this token is part of the larger phrase "potato mixture" that is a poor referring expression for the ingredient entity.

The problem of these phantom intermediate entities can be solved in a similar way to how the neural checklist model handles ingredient references during recipe generation. Before generating each word of the output text, the neural checklist model decides whether or not it will generate a reference to one of the ingredients and, if so, which ingredient it will reference. If generating an ingredient reference, the model uses the embeddings of the ingredients weighted by likelihood to determine what output token to be generated. In this way, the model is unlikely to generate tokens that reference ingredients that are not on the list or have already been used. We can also explicitly identify which ingredient is being referenced.

This same method can be used to constrain and identify references to intermediate entities generated by previous actions. With this additional information, we can identify the origins of all entities in the recipe text. This would be only a small step away from a complete action graph representation.[1] One additional complexity, however, is that we need to handle implicit

[1]For a complete action graph, we need to identify the action verbs and align arguments to the verbs. One approach would be to use the automatic segmentation system from Section 3.4 with the identification of definitive argument tokens as additional evidence. We discuss a second approach to complete action graph generation in Section 5.6.1 that involves generating syntactic tags for each output token.

references to intermediate entities. For example, in the "Chocolate covered potato chips" recipe, the text "Remove from heat" makes an implicit reference to the output of the previous sentence (i.e., "Melt chocolate in top of double boiler over simmering water."). Without an explicit token reference to the intermediate entity generated by the previous sentence, we will not be able to identify if, and if so where, that intermediate entity is referenced. A simple solution is to add a dummy token to indicate an implicit entity reference. The previous output text would instead be generated as "Remove IMP from heat," where the token "IMP" would refer to the intermediate entity generated by the previous sentence. This solution is described in more detail in Section 5.4.

In this chapter, we present the neural recipe model, an extension of the neural checklist model that generates recipe text while maintaining the origins of all entity references, not just ingredient references. Collectively, this information can be used to generate a rough sketch of an action graph. Preliminary experiments show that this adapted model improves the generation of referring expressions over the standard neural checklist model for non-ingredient entities. We also discuss directions for future improvements of this joint generation model based on a qualitative analysis of the recipes generated.

5.1 Recipe Generation Task

The neural recipe model is designed for a recipe generation task that is based on the same task which the neural checklist model is designed for (see Section 4.1 for details): given a goal $\mathbf{g}$ and an agenda $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{|E|}\}$, generate a sequence of tokens $\mathbf{x}$ for achieving this goal by making use of each item on the agenda. For recipe generation, we assume that this sequence of tokens represents a sequence of actions, and that each action creates an intermediate entity (or the goal object). Unlike the task of the neural checklist model, the recipe generation task has an additional constraint that ensures every intermediate entity created by the sequence of actions is referred to in the generated recipe (or is the goal entity).

We define a Boolean variable $b_i \in \{0, 1\}$ such that b_i is true if and only if the output token x_i represents the end of an action that has created an entity. For example, the period token

at the end of the sentence "Chop the tomatoes ." identifies the end of the text that generates chopped tomatoes. For an output sequence of tokens $\mathbf{x}$, we have an output sequence of Boolean values $\mathbf{b}$. The number of entities created during the course of the recipe, then, is $||\mathbf{b}||_1$. Let $\mathbf{b}^{ind} \in \mathbb{N}^{||\mathbf{b}||_1}$ be the vector of indices of where entities are created (i.e., $\mathbf{b}_i = 1$ if and only if there exists exactly one j such that $\mathbf{b}^{ind}_j = i$); we assume the vector is ordered such that $\mathbf{b}^{ind}_j < \mathbf{b}^{ind}_k$ if $j < k$. Let $D_{|\mathbf{x}|} = \{\mathbf{d}_1, \ldots, \mathbf{d}_{||\mathbf{b}||_1}\}$ be the set of entities created during the course of generating the sequence $\mathbf{x}$, such that $\mathbf{d}_j$ is the entity created by the action that ends with token $\mathbf{b}^{ind}_j$.

With these definitions, the recipe generation task becomes: given a goal $\mathbf{g}$ and an agenda $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{|E|}\}$, generate a sequence of tokens $\mathbf{x}$ for achieving this goal by making use of each item on the agenda *such that* if $D_{|\mathbf{x}|}$ is the set of intermediate entities created during the course of generating $\mathbf{x}$ then every entity in $D_{|\mathbf{x}|}$ must have been referred to at some point in $\mathbf{x}$.

5.2 Neural Recipe Model

The neural recipe model is an adaptation of the neural checklist model and therefore generates text in a similar way. At each step the recurrent neural language model computes its next hidden state as well as a probability distribution $\mathbf{f}_t$ that represents what kind of output token to generate: a new ingredient reference, a used ingredient reference, a non-reference token, or – unique to the neural recipe model – a reference to an intermediate entity. The embedding used to generate the output token, $\mathbf{o}_t$, is computed via a linear interpolation of four content vectors weighted by $\mathbf{f}_t$. The neural recipe model, like the neural checklist model, stores a vector that represents a soft checklist of which agenda items have been referenced so far in the generated text. Uniquely, the neural recipe model stores a second checklist that represents a soft checklist of which intermediate entities have been referenced so far in the generated text. The set of intermediate entities increases as the text is generated: after each sentence, a new entity embedding is computed and added to a list. When a new intermediate entity is created, it is initialized to be available based on the intermediate entity checklist.

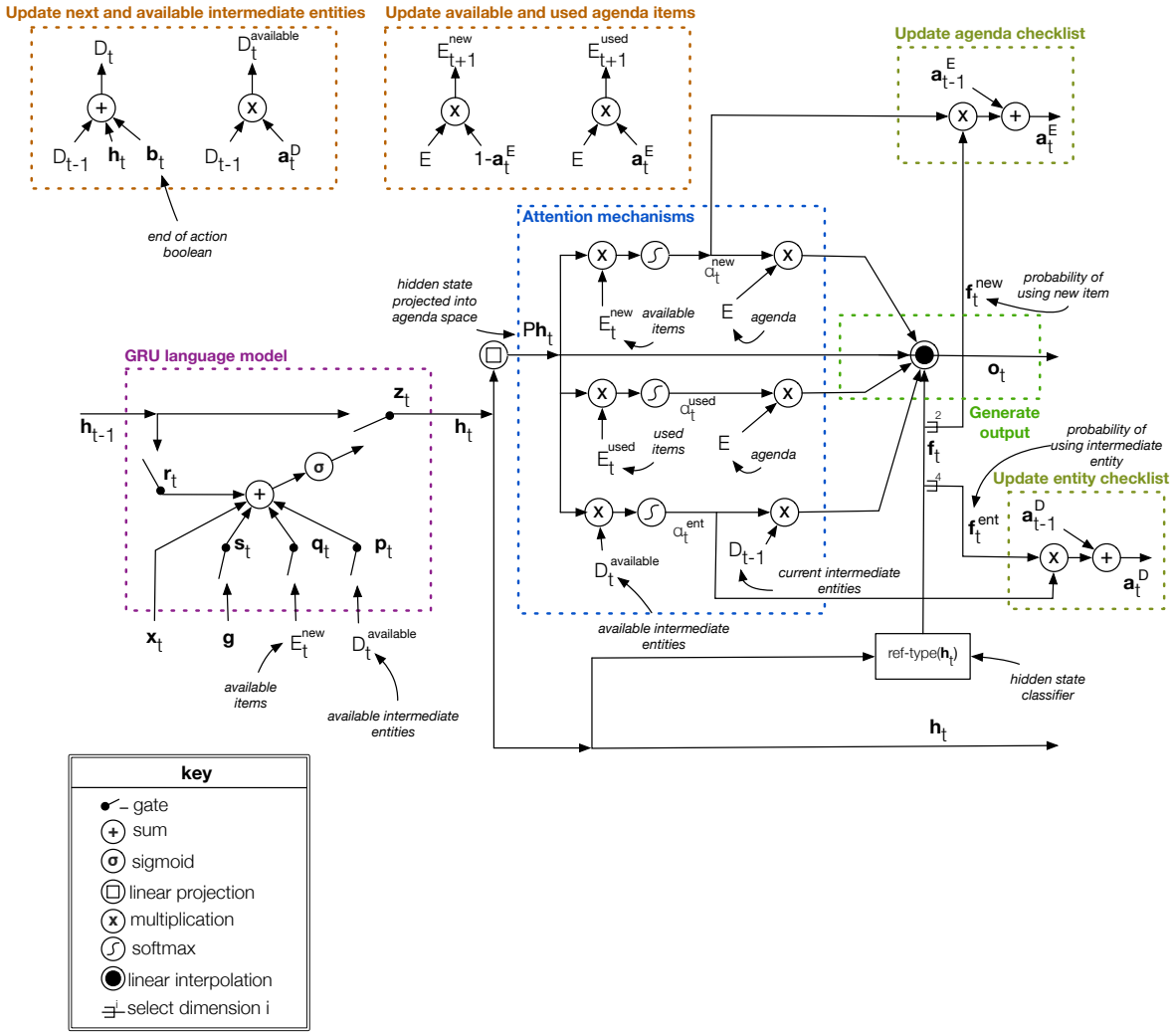


Figure 5.1: A diagram of the recipe neural checklist model. To generate the output embedding $\mathbf{o}_t$, the language model (box labeled "GRU language model", Section 5.2.3) computes the next hidden state. This hidden state is compared against new/used agenda items and the available intermediate entities (box labeled "Attention mechanisms", Section 4.2.4 for new agenda items, Section 4.2.5 for used agenda items, and Section 5.2.2 for available intermediate entities) to generate four possible content vectors. The output embedding $\mathbf{o}_t$ is computed via a linear interpolation of those content vectors (box labeled "Generate output", Section 5.2.1) weighted by the hidden state classifier *ref-type*(). The three boxes along the top of the diagram show how the next/available intermediate entity matrices, the available/used agenda item matrices, and the agenda checklist are updated. On the right side of the diagram, the "Update entity checklist" box depicts how the checklist of available entities is updated.

The neural checklist model is trained such that at the end of the recipe generation process only one intermediate entity should remain: the goal object.

Given a goal embedding $\mathbf{g} \in \mathbb{R}^k$, a matrix of L agenda items $E \in \mathbb{R}^{L \times k}$, a checklist soft record of what items have been used $\mathbf{a}^E_{t-1} \in \mathbb{R}^L$, a matrix of entities generated so far $D_{t-1} \in \mathbb{R}^{M \times k}$, a checklist soft record of what entities have been used $\mathbf{a}^D_{t-1} \in \mathbb{R}^M$, a previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^k$, and the current input word embedding $\mathbf{x}_t \in \mathbb{R}^k$, our architecture computes the next hidden state $\mathbf{h}_t$, an embedding used to generate the output word $\mathbf{o}_t$, the updated agenda and entity checklists $\mathbf{a}^E_t$ and $\mathbf{a}^D_t$, and the possibly-updated matrix of entities D_t.[2] For simplicity, we will assume that a new entity is generated if and only if a newline token is generated. This is a high-precision assumption, but it does miss out on cases when a sentence has more than one action. For example, the sentence "Chop the tomatoes, and dice the onions." generates two entities. We discuss how we might remove this assumption in the future in Section 5.6.2.

In the rest of this section, we discuss the specific adaptations we make to the neural checklist model in order to handle these intermediate entities. These adaptations include (1) updating the computation of the output hidden state $\mathbf{o}_t$ to include the possibility of generating a reference to an intermediate entity (Section 5.2.1), (2) how to identify the likely intermediate entity using a third attention model (Section 5.2.2), and (3) updating the GRU language model to take in the currently available intermediate entities as an additional input (Section 5.2.3). Figure 5.1 gives a graphical representation of the neural recipe model, to refer to throughout the section.

5.2.1 Generating Output Token Probabilities

The first change we must make is in how we determine the output hidden state vector $\mathbf{o}_t$, which is used to generate the probability distribution over vocabulary tokens. In particular, we must take into account the possibility of generating references to intermediate entities

[2] $\mathbf{a}^E_t$ refers to the same variable as $\mathbf{a}_t$ in Chapter 4. We add the superscript in this chapter to differentiate the two checklists in this model.

(see "Generate output" box in Figure 5.1).

Section 4.2.3 describes the linear interpolation used to compute the output hidden state $\mathbf{o}_t$:

$$\mathbf{o}_t = f_t^{gru}\mathbf{c}_t^{gru} + f_t^{new}\mathbf{c}_t^{new} + f_t^{used}\mathbf{c}_t^{used}.$$

The interpolation weights in this equation, f_t^{gru}, f_t^{new}, and f_t^{used}, are probabilities representing how much the output token should reflect the (1) current state of the language model, (2) a new agenda item reference, or (3) a used agenda item reference. To include the possibility of generating a reference to an intermediate entity, we add a fourth term to this linear interpolation:

$$\mathbf{o}_t = f_t^{gru}\mathbf{c}_t^{gru} + f_t^{new}\mathbf{c}_t^{new} + f_t^{used}\mathbf{c}_t^{used} + f_t^{ent}\mathbf{c}_t^{ent}.$$

f_t^{ent} is the probability of an intermediate entity reference token and $\mathbf{c}_t^{ent}$ is an encoding generated from the entity reference model (Section 5.2.2).

To compute f_t^{ent} we extend the $ref\text{-}type(\mathbf{h}_t)$ probabilistic classifier from a three-way classifier to a four-way classifier. The new $ref\text{-}type(\mathbf{h}_t)$ classifier generates a probability distribution $\mathbf{f}_t \in \mathbb{R}^4$ in the same way as in the original neural checklist model,

$$\mathbf{f}_t = ref\text{-}type(\mathbf{h}_t) = softmax(\beta S\mathbf{h}_t),$$

with the exception of the dimensions of the trained projection matrix $S \in \mathbb{R}^{4\times k}$. The four dimensions of $\mathbf{f}_t$ are assigned to interpolation weights as follows: $f_t^{gru} = \mathbf{f}_t^1$, $f_t^{new} = \mathbf{f}_t^2$, $f_t^{used} = \mathbf{f}_t^3$, and $f_t^{ent} = \mathbf{f}_t^4$.

5.2.2 Entity Reference Model

To allow references to intermediate entities, we incorporate a third attention model into the neural checklist model (see "Attention mechanisms" box in Figure 5.1 for depictions of the three attention models). At each time step t, we will compare $\mathbf{h}_t$ to the available intermediate entity matrix $D_t^{available}$. This matrix is computed using the entity checklist vector generated

at the previous time step:

$$D_t^{available} = ((\mathbf{1}^M - \mathbf{a}_{t-1}^D) \otimes \mathbf{1}_k) \circ D_{t-1},$$

where $\mathbf{1}_M = \{1\}^M$, $\mathbf{1}_k = \{1\}^k$, and the outer product $\otimes$ replicates $\mathbf{1}_M - \mathbf{a}_{t-1}$ for each dimension of the embedding space. $\circ$ is the Hadamard product (i.e., element-wise multiplication) of two matrices with the same dimensions.

The encoding $\mathbf{c}_t^{ent}$ is computed using the attention-weighted sum of the intermediate entities:

$$\mathbf{c}_t^{ent} = D_{t-1}^T \boldsymbol{\alpha}_t^{ent},$$

where $\boldsymbol{\alpha}_t^{ent}$ is soft attention alignment that represents how close $\mathbf{h}_t$ is to each available intermediate entity:

$$\boldsymbol{\alpha}_t^{ent} \propto \exp(\gamma D_t^{available} P \mathbf{h}_t).$$

$P \in \mathbb{R}^{k \times k}$ is the same learned projection matrix and γ is the same temperature hyperparameter as used in the new- and used-agenda item attention models (See Sections 4.2.4 and 4.2.5).

At each step, the model updates the entity checklist vector based on the probability of generating an intermediate entity reference, $\mathbf{f}_t^{ent}$, and the attention alignment $\boldsymbol{\alpha}_t^{ent}$. We calculate the update to the entity checklist as $\mathbf{a}_t^D = \mathbf{a}_{t-1}^D + (\mathbf{f}_t^{ent} \cdot \boldsymbol{\alpha}_t^{ent})$.

Updating the Entity Matrix

During the generation of a recipe, the ingredient list stays static: what changes is whether or not each ingredient has been used so far. However, the list of intermediate entities grows as a recipe is generated and each entity becomes unavailable once it is referred to later on (see "Update next and available intermediate entities" box in Figure 5.1).[3] At time step t,

[3]Under this paradigm, each action only generates a single entity. Therefore, the output graph structure will be a tree, i.e., each action only generates one entity. As discussed in Chapter 3, the sequence of actions in a recipe can not always be represented by a tree (e.g., "Separate the yolks from the egg whites.").

an entity is only added to the intermediate entity matrix D_t if the Boolean action-ending variable $\mathbf{b}_t$ is true, otherwise the matrix remains the same:

$$D_t = \begin{cases} \begin{pmatrix} D_{t-1} \\ (Q\mathbf{h}_t)^T \end{pmatrix}, & \text{if } \mathbf{b}_t = 1 \\ D_{t-1}, & \text{otherwise,} \end{cases}$$

where $Q \in \mathbb{R}^{k \times k}$ is a trained projection matrix that projects the language model hidden state into an intermediate entity embedding space.

5.2.3 Updated GRU Language Model

At each time step, the language model in the neural checklist model takes as input the available ingredients (see Section 4.2.6). In this updated version, the language model also takes as input the available intermediate entities (see "GRU language model" box in Figure 5.1). The computation of the new content $\tilde{\mathbf{h}}_t$ is computed as follows:

$$\begin{aligned} \tilde{\mathbf{h}}_t \;=\; & \tanh(W_h x_t + \mathbf{r}_t \odot U_h \mathbf{h}_{t-1} \\ & + \mathbf{s}_t \odot Y\mathbf{g} + \mathbf{q}_t \odot (\mathbf{1}_L^T Z^{new} E_t^{new})^T \\ & + \mathbf{p}_t \odot (\mathbf{1}_M^T Z^{ent} D_t^{available})^T, \end{aligned}$$

where $\mathbf{p}_t$ is an *entity select* gate, defined as

$$\mathbf{p}_t = \sigma(W_p \mathbf{x}_t + U_p \mathbf{h}_{t-1}).$$

However, we will define our updated neural checklist model under this assumption for now for simplicity. Extending the possible recipe structures is left for future work.

$\mathbf{1}_M$ sums the rows of the available intermediate entity matrix $D_t^{available}$. Z^{ent}, W_p, $U_p \in \mathbb{R}^{k \times k}$ are trained projection matrices and the rest of the parameters are as defined in Section 4.2.6.[4] The entity select gate controls when the available intermediate entities should be taken into account (e.g., when deciding what the next action should be based on what entities have previously been generated).

5.3 Training

Our training procedure is the same as that of the neural checklist model (see Section 4.2.7 for details), with the exception that we add an additional mean squared error loss term that encourages references to all intermediate entities in the output sequence. The loss term is computed using the value of the entity checklist at the end of generating the output sequence, which represents which intermediate entities were referred to in the sequence. Given a training set of (goal, agenda, output text) triples $\{(\mathbf{g}^{(1)}, E^{(1)}, \mathbf{x}^{(1)}), \ldots, (\mathbf{g}^{(J)}, E^{(J)}, \mathbf{x}^{(J)})\}$, we train model parameters by minimizing negative log-likelihood with that additional mean squared error loss term:

$$NLL(\theta) \quad = \quad -\sum_{j=1}^{J} \sum_{i=2}^{N_j} \log p(\mathbf{x}_i^{(j)} | \mathbf{x}_1^{(j)}, \ldots, \mathbf{x}_{i-1}^{(j)}, \mathbf{g}^{(j)}, E^{(j)}; \theta) - \eta \text{MSE}(\mathbf{a}_{N_j}^D, \mathbf{1})$$

where $\mathbf{x}_1^{(j)}$ is the start symbol, η is a hyperparameter that increases the importance of the loss term, and MSE is the mean squared error, $\text{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$.

5.4 Handling Implicit Arguments

Many times in recipe text, entities generated by previous actions are represented implicitly by the text. For example, in the following recipe snippet

Pour the batter in the pan.

---

[4] Z^{new} is the same trained parameter matrix as Z in the neural checklist model. The superscript was added to differentiate with the new parameter matrix.

Bake for 15 minutes.

the direct object of "bake" is implicit and is the output of the POUR event. Recipe generation using the neural checklist model from Chapter 4 can not account for this implicit argument as it does not exist as a token to be generated. In our recipe generation experiments on our neural recipe model we added a special "IMP" token to represent implicit entities. For example, the previous recipe snippet would now look like

Pour the batter in the pan.
Bake IMP for 15 minutes.

Instances of this implicit argument token were added heuristically to the training data. The token was added after verbs where the direct object was missing or if a verb did not have an prepositional phrase argument that contains ingredients. This overpopulates the training data with implicit tokens, but we hypothesize that the model will train to use the implicit token only when there is a previous entity available to use. Making the training data more precise in its use of generating implicit tokens is an area for future work.

5.5 Preliminary Evaluation

We performed a preliminary evaluation of the model proposed in this chapter on generating recipes for baked goods. We used subsets of the Now You're Cooking recipe train, dev, and test sets from Section 4.3.3 that contains only baking recipes: recipes that contained the token "cookies," "cake," "brownies," "biscuits," "muffins," "blondies," or "pie" in the title.

As with the recipe generation task from Chapter 4, we used heuristics to label the text of the recipes with approximate information about what type of reference each token in the text is. However, unlike the prior chapter, we only have partial information for this experiment: some tokens have labeled data while others do not or only have partial labels. For example, the first instance of a token that occurs in the ingredient list is labeled as a reference to the associated ingredient, and every instance of "IMP" is labeled as a reference to some unknown intermediate entity.

Model	BLEU-4	METEOR	% ingredients
Checklist	5.36	11.2	33.1%
Checklist+	**6.91**	13.0	53.3%
Recipe	5.84	12.7	32.7%
Recipe+	6.37	**14.1**	**56.3%**

Table 5.1: Quantitative results on the baking recipe task.

Parameters We constrained the gradient norm to 5.0 and initialized parameters uniformly on $[-0.1, 0.1]$.[5] We used a learning rate of 0.1, and the temperature hyper-parameters (β, γ, η) were assigned to $(5, 2, 50)$. We used a hidden state size of $k = 128$. Using a mini-batch size of 50 with recipes batched by token length, we had a training data set size of 11550, a development data set size of 130, and a test data set size of 137. For generation, we used a beam of size 10.

Models We compared recipes generated using the neural recipe model (Recipe) to recipes generated using the neural checklist model (Checklist) from Chapter 4. For both models, we additionally generated recipes using the altered decoding process described in Section 4.3.5 that re-generates recipes where ingredients are missing; these models are labeled Checklist+ and Recipe+ in the evaluation.

5.5.1 Preliminary Results

Figure 5.1 compares recipes generated by the neural recipe model described in this chapter to recipes generated by the standard neural checklist model described by Chapter 4.

Ingredient Use Results Using a heuristic to count the ingredient mentions, the Checklist and Recipe model on average generate similar amounts of the ingredient list; the Checklist+ and Recipe+ models are also similar in this regard. This shows that the neural recipe

[5]The parameter initialization was changed from those from Section 4.3.2 due to overflow errors.

Model	mixture	batter	dough	crust	cake	pie	cookies
True Recipes	93	43	37	27	63	58	7
Checklist	61	19	2	32	29	34	0
Checklist+	**72**	19	3	30	32	31	0
Recipe	177	57	**45**	36	**77**	**53**	**5**
Recipe+	195	**54**	46	**27**	102	52	**5**

Table 5.2: Counts from the development set baking recipes of common referring expressions in the baking domain. The bold numbers in the model counts are the closest to the numbers (in terms of absolute counts) from the true recipes in the top line.

model still generates text that is driven by the agenda (i.e., ingredient list), just as the neural checklist model. Compared to the results in Chapter 4, the percentage of ingredients used by the models is lower in this experiment. This perhaps is due to the fact that in baking recipes, large sets of ingredients are commonly introduced together, e.g., "Sift the dry ingredients," which the token-match heuristics do not register as ingredients being used.

BLEU Results In terms of precision-based BLEU-4, Checklist+ slightly outperforms Recipe+ and Recipe slightly outperforms Checklist. However, the numbers are very similar in both cases. Both model types are generating similar recipes for given baked goods. One possible explanation for why the neural recipe model is not much better is that when heuristically generating additional labels for training, there are increased ambiguities for reference types. We train the neural checklist model and neural recipe model with heuristic annotations on the output tokens that identify the type of reference (e.g., ingredient reference, intermediate entity reference). When generating data to train the neural checklist model, if an ingredient token occurs more than once, subsequent tokens are labeled as used ingredient references. However, with the neural recipe model, there are two key differences. First, subsequent ingredient tokens cannot be labeled as used ingredient references as a rule because some of them will be action-generated entity references. For example, in the text "Pat the chicken dry. Put **the chicken** into the pan.," the second reference to "chicken"

represents the output of the PAT event. Second, for tokens like these subsequent ingredient tokens, there is no evidence during training. Additional evidence is only provided for (1) initial ingredient references and (2) the implicit token "IMP," and (3) ingredient references that occur in phrases that are labeled as *other* by the segmentation system (see Section 3.4 for details).

METEOR Results In terms of recall-based METEOR, the neural recipe model outperforms the neural checklist model. The neural recipe model is better at generating referring expressions to action-generated entities than the neural checklist model is. Figure 5.2 shows the counts of certain common entity tokens that occur in baking recipes in the true development set baking recipes and the recipes generated by the systems. The Checklist and Checklist+ models generate far fewer of these terms than the Recipe and Recipe+ models, and the counts in the true recipes are much closer to the counts from the recipes generated from the neural recipe models (with the exception of the token "mixture" which the neural recipe model over generates). This shows that the adaptation to the neural checklist model that forces the model to identify intermediate entities helps generate better recipes that are closer to the true recipes.

Qualitative Evaluation Figure 5.2 shows three example generations using Checklist+ and Recipe+ for development set recipes. In bold are phrases that should refer to intermediate entities. The Recipe+ generations have more intermediate entity reference phrases (e.g, "the dough," "the butter mixture") than the Checklist+ generations, which use almost all implicit entity references. Not all of these are correct phrases for the recipes to contain: for example, in the Recipe+ generation for "Quick & easy brownies," the recipe has "Pour the chocolate mixture into the prepared pan [...]" where there is no previous entity that has chocolate in it.

Underlined tokens in the recipes generated by Recipe+ are tokens where the model was likely generating an intermediate entity reference (i.e., $\mathbf{f}_t^{ent} \geq 0.50$). Interestingly, many

times the determiner "the" is the token that the model gives a high likelihood of being an intermediate entity reference. However, in these cases, the subsequent token is correct for the situation (e.g., "the dough," "the muffins"). The model is learning to generate correct entity references, but not in the intended way. Additional experimentation is needed to develop models that properly use definite determiners to refer to previously mentioned entities.

Additionally, although not pictured in Figure 5.2, the neural recipe model is only generating sequential recipes: that is, every time a sentence generates an intermediate entity, it is referred to in the next sentence. One possible explanation is that this is due to the inherent linear nature of recurrent neural network text generation. An avenue of future work is training the neural recipe model using labeled data from the output of the interpretation system from Chapter 3.

5.6 Future Improvements for Complete Action Graph Generation

The model presented in this chapter is able to identify which tokens in a generated output text are references to entities and can resolve which entities those are. Given that information, it is possible to infer an action graph by identifying the verbs in the text and connecting the argument entities accordingly. However, there are certain improvements we could make to the generation model in this chapter that would lead to better action graphs and better generated recipes overall. We discuss two potential improvements in the rest of this section.

5.6.1 Generating Syntactic Tags

The current formulation of the model identifies tokens as being references to ingredients, references to action-generated entities, or non-reference tokens. One vital missing piece of structured recipe information is the sequence of actions; the neural checklist model identifies the argument entities, but not the verbs. Using a segmentation model, such as that described in Section 3.4, we can extract the verbs from the recipe text. However, a different approach would be to generate syntactic tags on each of the generated tokens. Figure 5.3 illustrates a potential adaptation of the neural checklist model that generates the next token and then its

Title: Plum pie
Ingredients: 1 cup sugar, 1/2 cup all-purpose flour, 1/2 teaspoon ground cinnamon, 1 tablespoon margarine or butter, 1 package for 9-inch two crust pie, 6 cups fresh purple plum slices

Truth	**Checklist+**	**Recipe+**
Heat oven to 425 degrees. Prepare pastry. Mix sugar , flour and cinnamon in large bowl. Stir in plums. Turn into pastry-lined pie plate. Dot with margarine. Cover with top crust that has slits cut in it; seal and flute. Cover edge with 2- to 3-inch strip of aluminum foil to prevent excessive browning. Remove foil during last 15 minutes of baking. Bake 35 to 45 minutes or until crust is brown and juice begins to bubble through slits in crust.	Preheat oven to 350 f. In a large bowl, mix butter, sugar, spices, and butter. Cook over medium heat, stirring constantly, until mixture thickens. Remove from heat; stir in flour. Pour into prepared crust. Bake at 350 degrees for 20 minutes.	Preheat oven to 350 degrees. In large mixer bowl, cream butter and sugar until light and fluffy. Beat **the butter** in a large bowl until light and fluffy. Gradually add the egg, one at a time, beating well after each addition. On a lightly floured surface, roll dough out dough on a floured board and knead **the dough** into a ball. Roll **dough** out on floured surface, and fit a 9" pie pan. Bake **the crust** for about 10 minutes. Reduce heat to 350 and bake for 10 minutes longer or until golden brown. Serve warm or cold.

Title: Banana and oat bran muffins
Ingredients: 1 cup flour, 1 egg white, 1 cup oat bran, 1 cup buttermilk, 1 teaspoon baking powder, 1/4 cup brown sugar, 1 teaspoon baking soda, 1 banana; large and mashed, 1/2 teaspoon cinnamon, 2 tablespoons canola oil, 1/4 teaspoon ground ginger, 1 tablespoon honey, 1/8 teaspoon salt, 1/2 teaspoon orange zest; grated

Truth	**Checklist+**	**Recipe+**
Preheat the oven to 400. Coat 12 muffin cups with Pam. In a medium bowl, combine the flour, oat bran, baking powder, baking soda, cinnamon, ginger and salt. Mix well. Add the egg white, buttermilk, sugar, banana, oil, honey and orange zest. Stir just until **the dry ingredients** are moistened; divide **the batter** evenly among **the muffin cups**. Bake on the center oven rack for 14 to 17 minutes, or until the tops are light brown. Loosen **the muffins** and remove to a wire rack. Cool for 10 minutes before serving.	Preheat oven to 400 degrees. In a large bowl combine flour, baking powder, flour, baking soda, salt and cinnamon; mix well. Combine cake and next in a large bowl with wire whisk. Add chocolate mixture and mix well and stir just enough to moisten. Fill greased muffin tins 2/3 full. Bake at 375 degrees for 20 to 25 minutes or until golden brown. Serve hot with butter and homemade jam or jelly.	Preheat oven to 350 degrees f. sift together salt and buttermilk together in a large bowl. In a large bowl, combine flour, 1 cup sugar, cheese, honey, baking soda, baking soda and cinnamon. Add **the flour mixture** to **the butter mixture**, stirring just until moistened. Fill muffin tins 2/3 full. Bake **the muffins** for 20 to 25 minutes or until golden brown. Serve warm or at room temperature.

Title: Quick & easy brownies
Ingredients: 3 cups Brownie mix–see recipe, For brownie mix 1/2 cup chopped pecans, 3 eggs beaten, 1 1/2 teaspoons vanilla extract

Truth	**Checklist+**	**Recipe+**
Combine all ingredients, stirring until blended. Spoon into a greased and floured 8-inch square pan. Bake at 350 degrees for 35 to 40 minutes. Cool and cut into squares.	Preheat oven to 350 degrees. Grease an 8 inch square baking pan. In a large mixer bowl, beat eggs until light and fluffy. Add eggs one at a time, beating well after each addition. Stir in vanilla. Pour into **greased 8 inch square cake pan**. Bake at 350 degrees for 30 to 35 minutes or until done.	Preheat oven to 350 degrees. Grease a 9 x 13 x 9 x 2-inch baking pan. Mix pecans, pecans, eggs, vanilla and vanilla in a large bowl. Add the eggs, one at a time, stirring well after each addition until smooth. Pour **the chocolate mixture** into **the prepared pan** and bake **the brownies** for 35 to 40 minutes, or until a toothpick inserted in the center comes out clean. Let cool for 10 minutes before removing from the pan.

Figure 5.2: Example generated recipes from the baking development set. Tokenization, newlines, and capitalization changed for space and readability. Bold tokens are referring expressions for intermediate entities. Underlined tokens are intermediate entity references based on the *ref-type*() hidden state classifier.

associated tag. These tags completely segment the recipe text and using similar heuristics to those used by the segmentation system from Section 3.4, we can identify all of the actions. The information from the tags, along with the identification of the origins of each entity would provide a complete action graph representation.

Our available training data is not annotated with these tags, so the true tag values will not be able to be used for training. However, we can approximate these values using the output from our segmentation system (see Section 3.4 for details).

5.6.2 Identifying Correct Action Boundaries

The prior section describes a way of jointly generating the recipe text and its segmentation so that all the information required to compose an action graph is generated concurrently with the recipe text. The second, and perhaps more crucial, improvement to make to our joint recipe text and action graph generation model is a better way of identifying when actions have occurred. In Section 5.2 we stated our simplifying assumption that during generation a new entity is created every time and only when a newline token is generated. For sentences that contain more than one action, this is an inaccurate assumption. To avoid using this assumption, we need the model to identify when the text for a particular action "ends." Using the tagging scheme from the previous section, a simple extension would be to have a tag that labels the end of an action. Then, when that tag appears an entity can be added to the list. The segmentation system to generate approximate tags does not have a mechanism to identify the tokens that represent the ends of actions. However, we could use simple heuristics to mark these tokens, such as conjunction words between actions (e.g., "Chop the tomatoes **and** dice the onion") and endline tokens.

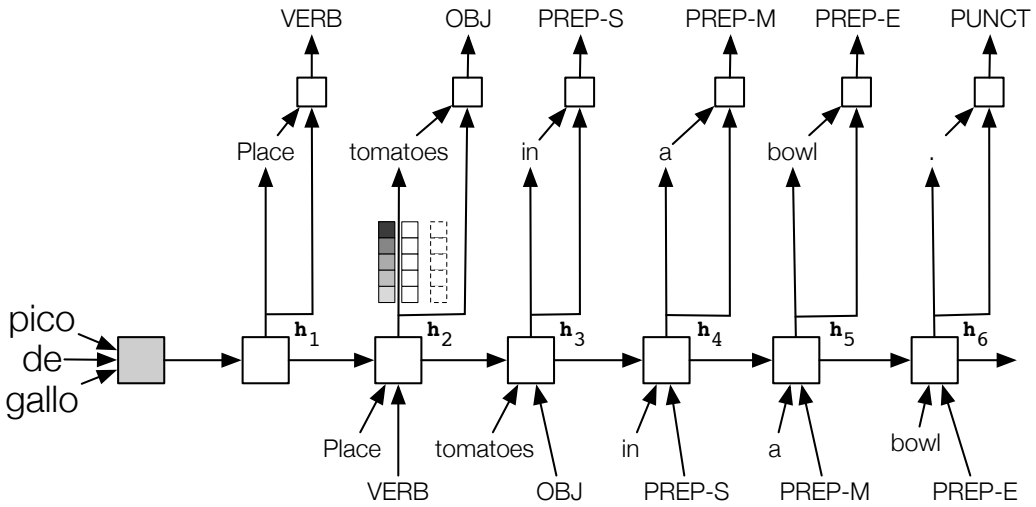


Figure 5.3: A diagram of a potential neural checklist model architecture adaptation that generates tokens and tags. At each time step the probability distribution over the vocabulary is generated first. Then that embedding and the new hidden state embedding are used to generate the appropriate tag. "-S" means the start of a phrase, "-M" is the middle of a phrase, and "-E" is the end. The tags would also be passed into the language model cell as an additional input.

Chapter 6

FUTURE DIRECTIONS FOR RECIPE UNDERSTANDING

In the previous chapters, we have shown that models to both interpret and generate instructional recipes can be trained on corpora of unannotated recipes. In the case of interpreting instructional recipes (Chapter 3), we were also able to show that domain-specific knowledge can be learned in an unsupervised way. However, there are still many avenues for future work in the area of recipe understanding. These include broadening semantic coverage for recipe interpretations (Section 6.1), improving generation of referring expressions (Section 6.2), extending the types of entities that can be generated (Section 6.3), expanding our evaluations to other domains (Section 6.4), and advancing our generation methods for greater novelty (Section 6.5), all of which we will discuss in this chapter.

6.1 Improvements through Greater Semantic Coverage

One improvement we foresee as being crucial in future work is expanding the semantic coverage of our interpretation model. This can happen on a couple different fronts including improving the types of information extracted from recipes and extending the power of the representation of actions by identifying repetitive structures.

Better Handling of Tools, Durations, and Locations The action graph interpreter of Kiddon et al. (2015) only allows the semantic types of action arguments to be foods, locations, or other. This ignores other information that is important to recipe handling, such as tools and durations of actions.

Also, connections are currently only allowed in action graphs between the outputs of actions and later arguments: this means that only foods and *highlighted* locations (e.g., an

oven that was preheated, a pan that was greased) are traced through the recipe. Better handling and tracing of locations and other information through the actions of the recipe will be useful not only for generating useful recipes for users, but also this information may improve action graph interpretation. For example, if the recipe states "Add the green beans to a new bowl," then the reference to a *new* bowl should help the algorithm decide that it is unlikely in the action *add* that the green beans are being added to any food that was referenced earlier in the recipe.

Parsing Repetitive Structures Action graphs can not have loops: the output of an action can not be used by a prior action. However, recipes can have repeated structures that should be represented. For example, the recipe for "Twisted chicken salad with tostadas" from our interpretation evaluation's development set is:

1. Heat oil in a large heavy skillet over medium-high heat.
2. One at a time, slip a tortilla into the hot oil.
3. If the tortilla starts to puff up, press it down with a fork.
4. When crisp and brown, remove to paper towels.
5. Repeat with remaining tortillas.
6. Place the chicken in a bowl, and separate with a fork.
7. Stir in the mayonnaise.
8. Dice the jalapenos (reserving the liquid), and stir them into chicken salad.
9. Pour in 1 tablespoon jalapeno juice, season with salt and pepper, and stir well.
10. To serve, spread the chicken salad on the tostadas.[1]

The first half of the recipe involves frying tortillas one at a time. Unlike the actions graphs we current can accurately represent, step 5, "Repeat with remaining tortillas," is not a single

[1]Recipe text from http://allrecipes.com/recipe/twisted-chicken- salad-with-tostadas

action: it directs the user to repeat the first four steps using each of the remaining tortillas from the ingredient list.

Repetition can occur in recipes even without a explicit "repeat" verb. The number of repetitions can be specified as a cardinal count adverbial (e.g., "baste twice"), as a frequency adverbial (e.g, "stir frequently"), or in a plural object (e.g., "chop the tomatoes" means each tomato should be chopped in turn) Karlin (1988b).

It is unclear if an unsupervised training method can learn this repetition structure. Additionally, if we do properly identify the repeated actions, then "the tostadas" in step 10 will be collectively the output of each repeated REMOVE (to paper towels) event. This will require an update to the action graph structure, or at least extra labels to mark this structure. An investigation into trying to learn repetitive structures using data-driven machine learning is left as future work.

6.2 Improving Generated Referring Expressions

As currently formulated, when generating agenda item references, the neural checklist model uses attention models to identify which item from the agenda should be referenced (see Section 4.2 for more details). Attention models determine a distribution over the set of input embeddings (i.e., in our case, the agenda item embeddings). One issue with this approach is that it does not easily allow for more than one item to be selected with high probability at a time. For example, a vegetable soup recipe might call on the chef to "chop all the vegetables". In this case, it is likely that the ingredient list will include more than one vegetable. Unfortunately, a distribution over the set of ingredients can either only select one ingredient with a high probability or select a set of ingredients with approximately equal but low probability, depending on the size of that set (e.g., if there are two vegetables on the list, the best the distribution can do is 50% probability for each vegetable). In instructional recipes, there are many cases where multiple or all of the items from the set of materials can be referenced at the same time. An important avenue for future work is revising the attention models to allow sets of items to be selected together. A simplified strategy that

would allow for either one or all of the agenda items (e.g, "combine **all ingredients**" to be selected at once is to add a dummy agenda item that corresponds to all of the agenda items. If selected, all other agenda items would be "checked off" the list. If an agenda item is ever referred to, the dummy item must be removed from consideration.

6.3 Generating Partial Ingredient and Multiple Intermediate Entities

The current training of our a neural checklist model for generating recipes uses the assumption that each ingredient (or output of an action) will be referenced exactly one time. For ingredients this is not always the case: for example, in a baking recipe flour can be divided and added into the recipe at different times, and when stir-frying, the fry oil may be added a little at a time as each different stir-fry vegetable or meat is cooked separately. The one-reference-per-ingredient assumption is not required by the neural checklist model: in Section 4.5, we saw that the model could be used for dialogue system responses where the output text for each agenda item was often multiple tokens in length. For training, we used heuristics to approximate the values for the hidden state classifier and the attention values such that the model trained to use each ingredient completely as it was mentioned. In order to train the model to allow ingredients to be used more than once, we would need to alter the heuristic labels and lower the temperature parameter that makes the hidden state classifier probability distribution more peaked. However, it is unclear if the generation model would learn to use ingredients partially, as that situation is more unlikely to begin with.

The neural checklist model adapted to take into account references to intermediate entities makes the assumption that each action only generates a single entity. Every time an action ending token is generated (in the case of our experiments, the endline token), a single entity embedding is added to the matrix of available entity embeddings. Howevever, actions can generate more than one entity; action graphs are graphs, not trees. Allowing varying numbers of entities to be generated by each action would require adding a generative mechanism to the neural checklist model that first selects the number of entities to generate and then generates each entity in turn. A neural model with such a mechanism is unlikely to train well, even

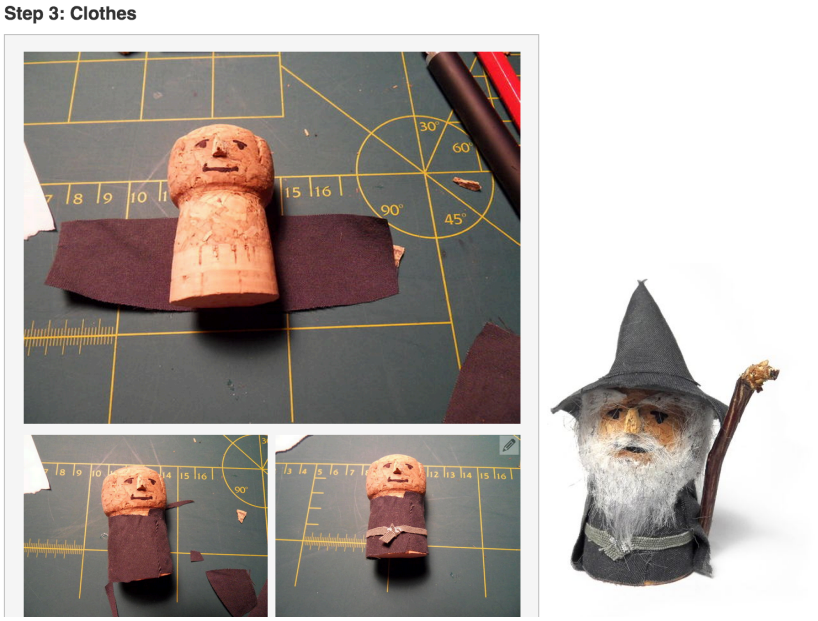


Figure 6.1: An instruction from the tutorial on Instructables.com for creating a cork figure of Gandalf. The images on the left are those associated with the particular instruction. The image on the right is of the completed craft.

with additional labeled information identifying how many entities are being created for each action. An investigation into extending the neural checklist model to generate recipes with non-tree structures are an interesting avenue for the future.

6.4 Evaluating on Other Domains

Cooking recipes are an ideal domain to evaluate our recipe generation algorithms, as vast amounts of these recipes are available online today, with significant redundancy in their coverage that will enable the learning process. However, our methods should extend beyond the domain of cooking recipes. We mention two possible domains for future experiments here.

6.4.1 Craft Instructions

Craft instructions, while less bountiful than cooking recipes, are also widely shared online. *Instructables.com* is a popular DIY site with over 100,000 projects and a whole top-level section devoted to craft projects. The difficulty with using craft instructions as a domain for our evaluation is that many instructions are dependent on accompanying visual aids. For example, if you want to use the instructions on Instructables to transform a champagne cork into a stunning rendition of Gandalf from The Lord of the Rings[2], the text instructions tend to be less specific than required to actually complete the task, and rely on the user using the associated photographs to complete the project accurately. To create Gandalf's robe, see Figure 6.1, the instructions tell the user to "cut a strip of grey fabric just long enough to go around the cork. Glue it on with UHU glue." The exact placement of the *gluing* event is ambiguous in the text, but unambiguous when guided by the photography. However, an action graph for the recipe can still be generated, and so we can still use the domain for evaluation. The action graphs may just be too underspecified to generate a recipe that users can follow; integrating images into recipe learning and generation is an area for future work.

6.4.2 Wet Lab Protocols

Another possible domain for future evaluations are wet lab protocols. These are instructions for various lab procedures that involve chemicals and biological matter. Examples include DNA precipitation or protein isolation. Wet labl protocol corpora do exist online (e.g., OpenWetWare http://openwetware.org/wiki/Protocols, or Protocol Online http://www.protocol-online.org/prot/). However, the size of these corpora is far smaller than the sizes of cooking recipes or craft project instructions. This could pose a problem for training our generation methods, as neural networks are very training-data-intensive.

---

[2]http://www.instructables.com/id/Champagne-Cork-Gandalf/

6.5 Creative Recipe Generation

We formulated the recipe generation task of Chapter 4 such that the recipe text is generated given the title and set of ingredients. Our evaluations show that our neural checklist model can generate goal-oriented and agenda-driven text as is necessary for recipe generation. However, a clear and important avenue for future research is to develop a broader, more creative system for recipe generation. As mentioned in Chapter 1, a major application for cooking recipe understanding would be virtual cooking helpers and planners. A user of such a system might ask what recipes were possible given the ingredients they have on hand; another use case might be a user desiring a particular food and having the system generate a correct set of ingredients and then the steps of the associated recipe. This kind of cooking helper system would exhibit computational creativity and be able to develop novel recipes from scratch. Realizing such an extension to our current model of recipe generation requires two parts: (1) a way to generate the titles of novel dishes either from scratch or from a provided set of possible ingredients and (2) a way to generate a correct set of ingredients with proper amount information.

6.5.1 Generating Novel Titles

A general recipe generation system should have the ability to generate completely new recipe title names either from scratch or given a set of possible ingredients. We could achieve both using neural network models. As we discussed in Chapter 4, recurrent neural networks are very effective at generating locally coherent texts. To generate reasonable recipe titles from scratch, we would only need to train a recurrent neural network on a corpus of recipe titles. If we want instead to generate recipe titles given a set of possible ingredients, one proposal is to use a encoder-decoder-style neural network that encodes a set of input ingredient embeddings as one combined embedding and the decodes a title token by token.

6.5.2 Generating Quantified Ingredients

A generated ingredient list would not be complete without designated amount information attached to each ingredient. For example, in baking, ingredient amounts have to be very precise in order for the recipe to work. For simplicity, the interpretation and generation systems we present in this paper ignored material amounts. Pinel et al. (2015) examined into how amounts can be generated for new recipes by using a notion of balancing ingredients. Ingredient proportions for a new recipe should be selected such that the proportions of nutrients (e.g., fat, protein) and ingredient type (e.g., fruit, vegetable) are similar to those of previous recipes for the same dish. The authors propose a cost function for nutrient balancing in recipes. The system presented in Pinel et al. (2015) used manually-engineered knowledge of the properties of different ingredients; in contrast, to maintain our system's domain flexibility and generality, we seek to limit the amount of domain-specific knowledge that must be given as an input to our systems.

Morris et al. (2012) used a simple ingredient hierarchy and a genetic algorithm to create a model that generates novel soup and stew recipes – where the recipes all had the same steps but different ingredient lists. The model trained on only 4,748 recipes and could generate complete ingredient lists with amounts. Using this work as inspiration, we could try a similar algorithm for generating new ingredient lists. Instead of using a fixed ingredient hierarchy, we could use word embeddings to identify similar ingredients. Recent work has shown that it is possible to even determine hierarchies of entities from word embeddings, which would also improve our ingredient list generation process (Fu et al., 2014).

Chapter 7

DISCUSSION AND CONCLUSIONS

An obstacle to interpreting and generating more complex types of instructional language has always been the design and generation of the plans that underlie the instructions. Determining the plans for instructional recipes is particularly difficult as they feature a set of entities that evolve and merge together through the course of the instruction set. Generating new instructional recipes requires keeping track of the current state of the set of entities as well as ensuring that the output text is an accurate textual representation of instructions to generate the goal object. Without large-scale and costly annotation and knowledge base engineering efforts, the overarching goal of instructional recipe understanding has remained elusive.

Fortunately, unannotated instructional recipe data is bountiful and easily obtainable. Many websites exist that are devoted to categorizing and storing cooking recipes, craft projects, and other types of instructions recipes. Machine learning methods can process large amounts of data in a meaningful way in order to gain a deeper understanding of a domain. My thesis is that the combination of machine learning and the large amount of available recipe data can further instructional recipe understanding, gaining us both the ability to interpret given recipes as actionable plans as well as generating recipes given a goal and set of materials to use.

In Chapter 3, we presented unsupervised methods for segmenting and identifying the latent connections among actions in recipe text. Our output *action graph* representation models the flow of materials through the sequence of actions that result in the goal object. Action graphs are directed acyclic graphs where each leaf represents a raw material or tool (e.g., an oven) and one sink, representing the final action of the recipe (e.g., "*Serve* the

dish hot"). The outgoing edges from raw materials and arguments point to the actions that use them; outgoing edges from an action point to arguments of later actions that represent the intermediate entity created by that action. We developed a joint probabilistic model over action graphs and recipe text that we trained in an unsupervised manner using hard Expectation Maximization. Experiments with cooking recipes demonstrated the ability to recover high quality action graphs, outperforming a strong sequential baseline by 8 points in F1. As a byproduct of its structure, our model learns a variety of domain-specific knowledge, such as verb signatures and the probable ingredient components for different composites. Our evaluation shows that our model is able to learn how to interpret recipe text from unannotated text alone.

Next in Chapter 4, we described our efforts on developing a model that could learn how to generate recipe text after training on unannotated text. We presented the *neural checklist model* that generates globally coherent text by explicitly keeping track of what has been said and still needs to be said from a provided agenda. Recurrent neural networks can generate locally coherent text but often have difficulties representing more globally what has already been generated and what still needs to be said – especially when constructing long texts. These models rely on a low-dimensional embedding to represent all syntactic and semantic information required for the generation task; for longer, more complicated generations, this embedding can become overloaded. Also adding additional inputs to a recurrent neural network, such as an agenda to drive generation, is not standardized. As recipes tend to be longer texts than what most neural network generation systems have been used for, solving these inherent problems was a crucial task. The neural checklist model generates output by dynamically adjusting the interpolation among a language model and a pair of attention models that encourage references to agenda items. In doing so, the model allows attention to the agenda to be switched on and off depending on when the output text wants to refer back to it. Additionally, since the model explicitly keeps track of which agenda items have yet to be used, the model is more likely to stay focused on both the goal and the remaining parts of the agenda. Evaluations on cooking recipes and dialogue system responses demonstrate

high coherence with greatly improved semantic coverage of the agenda.

The research presented in this thesis shows it is possible to take advantage of the abundance of recipe data and apply machine learning methods for recipe understanding tasks. We can identify the underlying plans of recipe texts, generate new recipe texts, and discover domain-specific knowledge without costly annotation efforts. We also presented preliminary work in Chapter 5 that shows we can jointly generate recipe text and its underlying structure in a way that additionally improves the quality and the interpretability of the generation process. The methods we put forward in this document advance what has previously been possible for interpreting and generating instructional recipes and make steps towards greater recipe understanding. In Chapter 6, we discussed many areas of future work in this area that will deeper this understanding further.

Progress on the automatic handling instructional recipes expands the range of activities that automated assistants can help us with. It allows virtual assistance to move more easily into spaces of our lives where there once was none. Artificially intelligent cooking assistants can help us while we are in our kitchens or grocery stores, determining what ingredients we may need and what dishes it is possible for us to create together. Laboratories take one step closer to automating protocols that were once exclusively the domain of humans. Not only can progress in this area help systems become smarter, but it will also help imbue systems with computational creativity, the ability to generate novel artifacts. Through our research efforts and the prior and contemporary research being performed in this field, we are ever closer to a richer and more collaborative connection with the smart and assistive devices that pervade life today.

BIBLIOGRAPHY

Abend, O., Cohen, S. B., and Steedman, M. (2015). Lexical event ordering with an edge-factored model. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics – Human Language Technologies (NAACL HLT 2015)*, pages 1161–1171.

Andreas, J. and Klein, D. (2015). Alignment-based compositional semantics for instruction following. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1165–1174.

Angeli, G., Liang, P., and Klein, D. (2010). A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512.

Ansari, D. and Hirst, G. (1998). Generating warning instructions by planning accidents and injuries. In *the 9th International Workshop on Natural Language Generation*, pages 118–127.

Artzi, Y. and Zettlemoyer, L. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1(1):49–62.

Balasubramanian, N., Soderland, S., Mausam, and Etzioni, O. (2013). Generating coherent event schemas at scale. In *Proceedings of the 2013 Conference on Empirical Methods on Natural Language Processing*, pages 1721–1731.

Barr, A. and Feigenbaum, E. (1981). *The Handbook of Artificial Intelligence, Volume 1.* William Kaufman Inc., Los Altos, CA.

Barzilay, R. and Lapata, M. (2005). Collective content selection for concept-to-text generation. In *Proceedings of the 2005 Conference on Empirical Methods in Natural Language Processing*, pages 331–338.

Beetz, M., Klank, U., Kresse, I., Maldonado, A., Mosenlechner, L., Pangercic, D., Ruhr, T., and Tenorth, M. (2011). Robotic roommates making pancakes. In *Proceedings of the 11th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 529–536.

Bollini, M., Tellex, S., Thompson, T., Roy, N., and Rus, D. (2013). Interpreting and executing recipes with a cooking robot. *Experimental Robotics*, 88:481–495.

Branavan, S., Chen, H., Zettlemoyer, L., and Barzilay, R. (2009). Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, pages 82–90.

Branavan, S., Silver, D., and Barzilay, R. (2011). Non-linear monte-carlo search in Civilization II. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pages 2404–2410.

Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics*, 19:263–311.

Bunke, H., Jiang, X., and Kandel, A. (2000). On the minimum common supergraph of two graphs. *Computing*, 65(1):13–25.

Chambers, N. and Jurafsky, D. (2009). Unsupervised learning of narrative schemas and their participants. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages 602–610.

Chen, D. L. and Mooney, R. J. (2011). Learning to interpret natural language navigation instructions from observations. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-2011)*, pages 859–865.

Cheng, J., Dong, L., and Lapata, M. (2016). Long short-term memory-networks for machine reading. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734.

Dale, R. (1988). *Generating Referring Expressions in a Domain of Objects and Processes*. PhD thesis, Centre for Cognitive Science, University of Edinburgh.

Dale, R. (1989). Cooking up referring expressions. In *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*, pages 68–75.

Denkowski, M. and Lavie, A. (2014). Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the EACL 2014 Workshop on Statistical Machine Translation*, pages 376–380.

Druck, G. (2013). Recipe attribute prediction using review text as supervision. In *Proceedings of the 2nd Workshop on Cooking with Computers (CwC)*, pages 25–33.

Druck, G. and Pang, B. (2012). Spice it up? Mining refinements to online instructions from user generated content. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 545–553.

Fillmore, C. J. (1982). *Frame semantics*, pages 111–137. Hanshin Publishing Co., Seoul, South Korea.

Fu, R., Guo, J., Qin, B., Che, W., Wang, H., and Liu, T. (2014). Learning semantic hierarchies via word embeddings. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics*, pages 1199–1209.

Fujiki, T., Nanba, H., and Okumura, M. (2003). Automatic acquisition of script knowledge from a text collection. In *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 2*, pages 91–94.

Gaillard, E., Nauer, E., Lefevre, M., and Cordier, A. (2012). Extracting generic cooking adaptation knowledge for the TAAABLE case-based reasoning system. In *Proceedings of the 1st Workshop on Cooking with Computers (CwC)*.

Greene, E. (2015). Extracting structured data from recipes using conditional random fields. The New York Times Open Blog.

Hammond, K. J. (1986). CHEF: A model of case-based planning. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 267–271.

Hara, T., Matsuzaki, T., Miyao, Y., and Tsujii, J. (2011). Exploring difficulties in parsing imperatives and questions. In *Proceedings of the 5th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages 749–757.

Jermsurawong, J. and Habash, N. (2015). Predicting the structure of cooking recipes. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 781–786.

Jia, X., Gavves, E., Fernando, B., and Tuytelaars, T. (2015). Guiding long-short term memory for image caption generation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2407–2415.

Karlin, R. F. (1988a). Defining the semantics of verbal modifiers in the domain of cooking tasks. In *Proceedings of the 26th Annual Meeting on Association for Computational Linguistics*, pages 61–67.

Karlin, R. F. (1988b). SEAFACT: Semantic analysis for animation of cooking tasks. Technical report, University of Pennsylvania, Department of Computer and Information Science.

Karpathy, A. and Li, F. (2014). Deep visual-semantic alignments for generating image descriptions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3128–3137.

Kiddon, C., Ponnuraj, G. T., Zettlemoyer, L., and Choi, Y. (2015). *Mise en Place*: Unsupervised interpretation of instructional recipes. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 982–992.

Kiddon, C., Zettlemoyer, L., and Choi, Y. (2016). Globally coherent text generation with neural checklist models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Kim, J. and Mooney, R. J. (2010). Generative alignment and semantic parsing for learning from ambiguous supervision. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING 2010)*, pages 543–551.

Konstas, I. and Lapata, M. (2013). A global model for concept-to-text generation. *Journal of Artificial Intelligence Research (JAIR)*, 48:305–346.

Kukich, K. (1987). *Where do Phrases Come from: Some Preliminary Experiments in Connectionist Phrase Generation*, pages 405–421. Springer Netherlands, Dordrecht.

Laroche, R., Dziekan, J., Roussarie, L., and Baczyk, P. (2013). Cooking coach spoken/multimodal dialogue systems. In *Proceedings of the 2nd Workshop on Cooking with Computers (CwC)*, pages 34–35.

Lau, T., Drews, C., and Nichols, J. (2009). Interpreting written how-to instructions. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, pages 1433–1438.

Li, J., Galley, M., Brockett, C., Gao, J., and Dolan, B. (2016). A diversity-promoting objective function for neural conversation models. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics Human Language Technologies (NAACL-HLT 2016)*, pages 110–119.

Liang, P., Jordan, M. I., and Klein, D. (2009). Learning semantic correspondences with less supervision. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, pages 91–99.

Luong, M., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

MacMahon, M., Stankiewicz, B., and Kuipers, B. (2006). Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, pages 1475–1482.

Maeta, H., Sasada, T., and Mori, S. (2015). A framework for procedural text understanding. In *Proceedings of the 14th International Conference on Parsing Technologies*, pages 50–60.

Malmaud, J., Huang, J., Rathod, V., Johnston, N., Rabinovich, A., and Murphy, K. (2015). What's cookin'? Interpreting cooking videos using text, speech and vision. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 143–152.

Malmaud, J., Wagner, E. J., Chang, N., and Murphy, K. (2014). Cooking with semantics. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 33–38.

Mei, H., Bansal, M., and Walter, M. R. (2016a). Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the Thirtieth National Conference on Artificial Intelligence (AAAI-16)*, pages 2772–2778.

Mei, H., Bansal, M., and Walter, M. R. (2016b). What to talk about and how? Selective generation using lstms with coarse-to-fine alignment. In *The 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 720–730.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Proceedings of INTERSPEECH 2010, the 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048.

Mikolov, T., Kombrink, S., Burget, L., Cernocký, J., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP 2011)*, pages 5528–5531.

Miller, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41.

Mori, S., Maeta, H., Sasada, T., Yoshino, K., Hashimoto, A., Funatomi, T., and Yamakata, Y. (2014a). FlowGraph2Text: Automatic sentence skeleton compilation for procedural text generation. In *Proceedings of the 8th International Natural Language Generation Conference*, pages 118–122.

Mori, S., Maeta, H., Yamakata, Y., and Sasada, T. (2014b). Flow graph corpus from recipe texts. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 26–31.

Mori, S., Sasada, T., Yamakata, Y., and Yoshino, K. (2012). A machine learning approach to recipe text processing. In *Proceedings of the 1st Workshop on Cooking with Computers (CwC)*.

Morris, R. G., Burton, S. H., Bodily, P. M., and Ventura, D. (2012). Soup over bean of pure joy: Culinary ruminations of an artificial chef. In *Proceedings of the International Conference on Computational Creativity (ICCC 2012)*, pages 119–125.

Nanba, H., Doi, Y., Tsujita, M., Takezawa, T., and Sumiya, K. (2014). Construction of a cooking ontology from cooking recipes and patents. In *Proceedings the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 507–516.

Nedović, V. (2013). Learning recipe ingredient space using generative probabilistic models. In *Proceedings of the 2nd Workshop on Cooking with Computers (CwC)*, pages 13–18.

Palmer, M. S., Dahl, D. A., Schiffman, R. J., Hirschman, L., Linebarger, M., and Dowding, J. (1986). Recovering implicit information. In *Proceedings of the 24th Annual Meeting on Association for Computational Linguistics*, pages 10–19.

Pichotta, K. and Mooney, R. (2014). Statistical script learning with multi-argument events. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 220–229.

Pinel, F., Varshney, L. R., and Bhattacharjya, D. (2015). A culinary computational creativity system. In Besold, T., editor, *Computational Creativity Research: Towards Creative Machines*, chapter 16. Atlantis Press.

Regneri, M., Koller, A., and Pinkal, M. (2010). Learning script knowledge with web experiments. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 979–988.

Regneri, M., Koller, A., Ruppenhofer, J., and Pinkal, M. (2011). Learning script participants from unlabeled data. In *Proceedings of the Conference on Recent Advances in Natural Language Processing*, pages 463–470.

Reiter, E. and Dale, R. (2000). *Building Natural Language Generation Systems*. Cambridge University Press, New York, NY, USA.

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kociský, T., and Blunsom, P. (2016). Reasoning about entailment with neural attention. In *Proceedings of the 2016 International Conference on Learning Representations*.

Rudinger, R., Demberg, V., Durme, B. V., and Pinkal, M. (2015). Learning to predict script events from domain-specific text. In *Proceedings of *SEM 2015: The Fourth Joint Conference on Lexical and Computational Semantics*, pages 205–210.

Rush, A. M., Chopra, S., and Weston, J. (2015). A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 379–389.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, plans, goals and understanding : an inquiry into human knowledge structures*. The Artificial intelligence series. L. Erlbaum, Hillsdale, N.J.

Silberer, C. and Frank, A. (2012). Casting implicit role linking as an anaphora resolution task. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics - Volume 1: Proceedings of the Main Conference and the Shared Task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation*, pages 1–10.

Sordoni, A., Galley, M., Auli, M., Brockett, C., Ji, Y., Mitchell, M., Nie, J.-Y., Gao, J., and Dolan, B. (2015). A neural network approach to context-sensitive generation of conversational responses. In *Conference of the North American Chapter of the Association for Computational Linguistics Human Language Technologies (NAACL-HLT)*, pages 196–205.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc.

Tasse, D. and Smith, N. A. (2008). SOUR CREAM: Toward semantic processing of recipes. Technical Report CMU-LTI-08-005, Carnegie Mellon University, Pittsburgh, PA.

Tetreault, J. R. (2002). Implicit role reference. In *Proceedings of the International Symposium on Reference Resolution for Natural Language Processing*, pages 109–115.

Thompson, H. S. (1977). Strategy and tactics: a model for language production. In *Papers from the Thirteenth Regional Meeting of the Chicago Linguistics Society*, pages 89–95. Chicago Linguistics Society.

Tu, Z., Lu, Z., Liu, Y., Liu, X., and Li, H. (2016). Modeling coverage for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 545–553.

Wen, T., Gasic, M., Mrksic, N., Rojas-Barahona, L. M., Su, P., Vandyke, D., and Young, S. J. (2016). Multi-domain neural network language generation for spoken dialogue systems. In *Proceedings of the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 120–129.

Wen, T., Gasic, M., Mrksic, N., Su, P., Vandyke, D., and Young, S. J. (2015). Semantically conditioned LSTM-based natural language generation for spoken dialogue systems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1711–1721.

Whittemore, G., Macpherson, M., and Carlson, G. (1991). Event-building through role-filling and anaphora resolution. In *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics*, pages 17–24.

Wiegand, M., Roth, B., and Klakow, D. (2012). Knowledge acquisition with natural language processing in the food domain: Potential and challenges. In *Proceedings of the 1st Workshop on Cooking with Computers (CwC)*.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A. C., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. *Proceedings of the 32nd International Conference on Machine Learning*, pages 2048–2057.

Yamakata, Y., Imahori, S., Sugiyama, Y., Mori, S., and Tanaka, K. (2013). Feature extraction and summarization of recipes using flow graph. In *Proceedings of the 5th International Conference on Social Informatics*, pages 241–254.

Zhang, X. and Lapata, M. (2014). Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 670–680.

Zhang, Y., Lei, T., Barzilay, R., and Jaakkola, T. (2014). Greed is good if randomized: New inference for dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1013–1024.

Zwicky, A. M. (1988). On the subject of bare imperatives in english. In Duncan-Rose, C. and Vennemann, T., editors, *On Language: Rhetorica, Phonologica, Syntactica - A Festschrift for Robert P. Stockwell from His Friends and Colleagues*, pages 437–450. Routledge, London.

Appendix A

EXAMPLE VISUALIZATIONS OF ACTION GRAPHS

A.1 Easy whole-wheat banana muffins

Steps

1. Preheat oven to 350 degrees F (175 degrees C).

2. Mix bananas, salad dressing, and sugar in a large bowl until smooth. Stir flour, baking soda, and salt into banana mixture until batter is just moistened. Divide batter evenly into 24 muffin cups.

3. Bake in the preheated oven until a toothpick inserted into the center comes out clean, about 17 minutes.

Ingredients

- 1 cup mashed bananas
- 1 cup creamy salad dressing (such as Miracle Whip)
- 3/4 cup white sugar
- 2 cups whole wheat flour
- 2 teaspoons baking soda
- 1/2 teaspoon salt

Data Parsed from this [URL](http://allrecipes.com/recipe/easy-whole-wheat-banana-muffins)

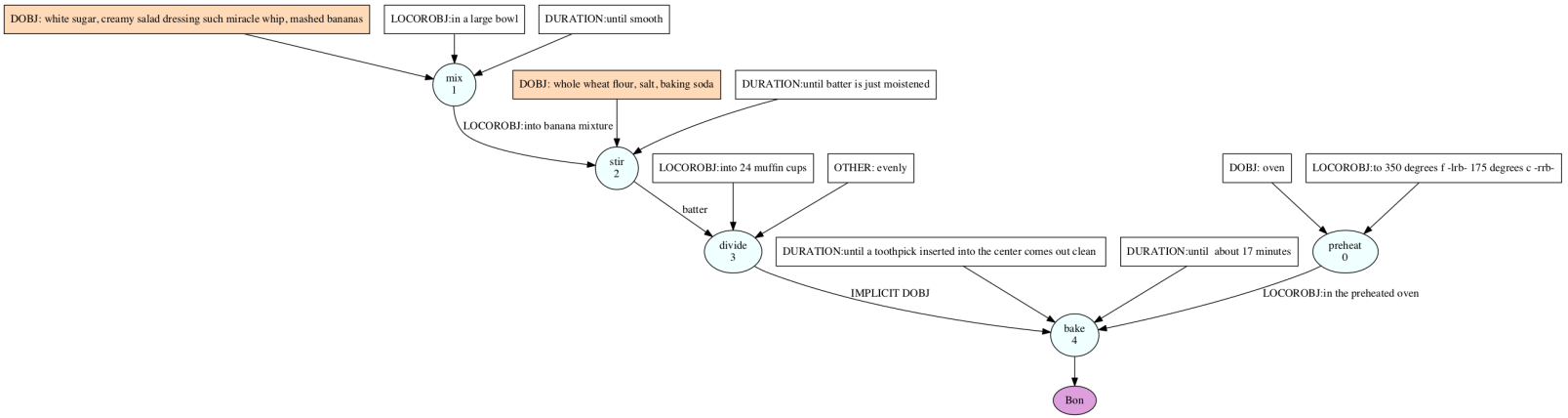


Figure A.1: Generated action graph for “Easy whole-wheat banana muffins” using the automatically generated segmentation

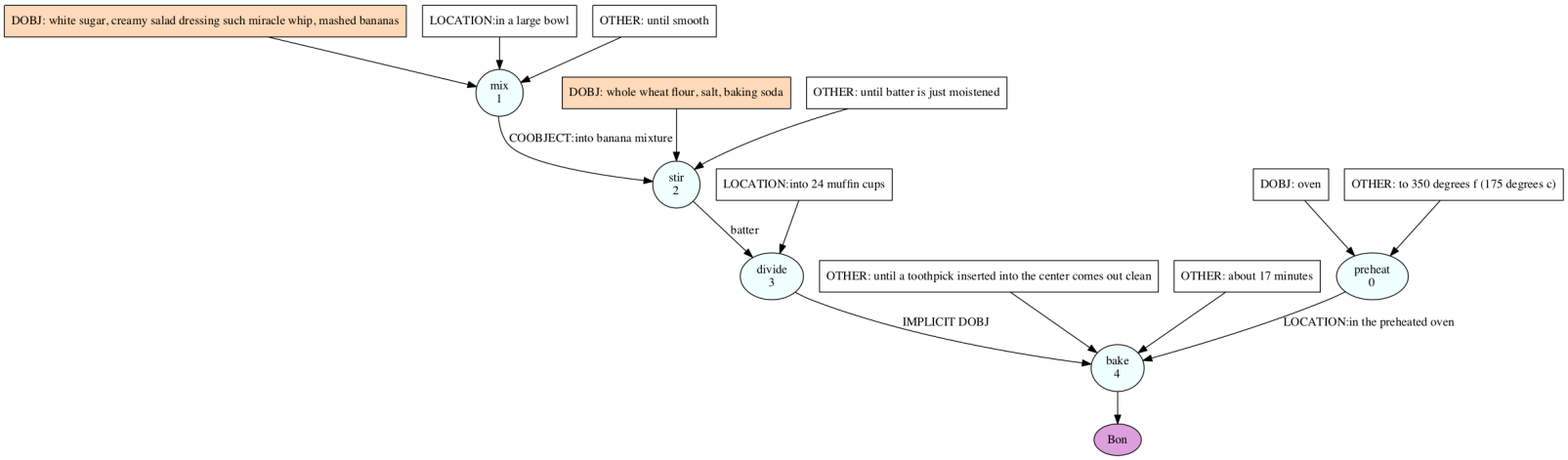


Figure A.2: Generated action graph for "Easy whole-wheat banana muffins" using the gold-standard segmentation

A.2 Pecan butterscotch pie

Steps

1. In the top of a double boiler, combine brown sugar, cornstarch, and 1/4 teaspoon. salt. Stir in milk and corn syrup. Cook over boiling water, stirring constantly, 20 minutes or until thickened.

2. In a medium bowl, beat egg yolks until thick and lemon colored. Gradually stir 1/2 cup of hot mixture into yolks. Pour back into remaining milk mixture, stirring constantly. Cook 5 minutes over boiling water, stirring frequently. Remove from heat; stir in butter, vanilla and 3/4 cup pecans. Pour into pastry shell. Preheat oven to 350 degrees F (175 degrees C.)

3. Beat egg whites cream of tartar, and pinch of salt until foamy. Gradually add sugar, beating until stiff peaks form. Spread meringue over filling, sealing to edge of pastry. Sprinkle with remaining 1/4 cup pecans.

4. Bake in the preheated oven for 12 minutes, or until golden brown. Cool to room temperature. Chill thoroughly.

Ingredients

- 1/2 cup packed dark brown sugar
- 1/4 cup cornstarch
- 1/4 teaspoon salt
- 2 cups milk
- 1/4 cup light corn syrup

- 3 egg yolks, beaten
- 3 tablespoons butter
- 1 teaspoon vanilla extract
- 3/4 cup chopped pecans
- 1 (9 inch) pie shell, baked
- 3 egg whites
- 1/4 teaspoon cream of tartar
- 1 pinch salt
- 3/8 cup white sugar
- 1/4 cup chopped pecans

Data Parsed from this [URL](http://allrecipes.com/recipe/pecan-butterscotch-pie)

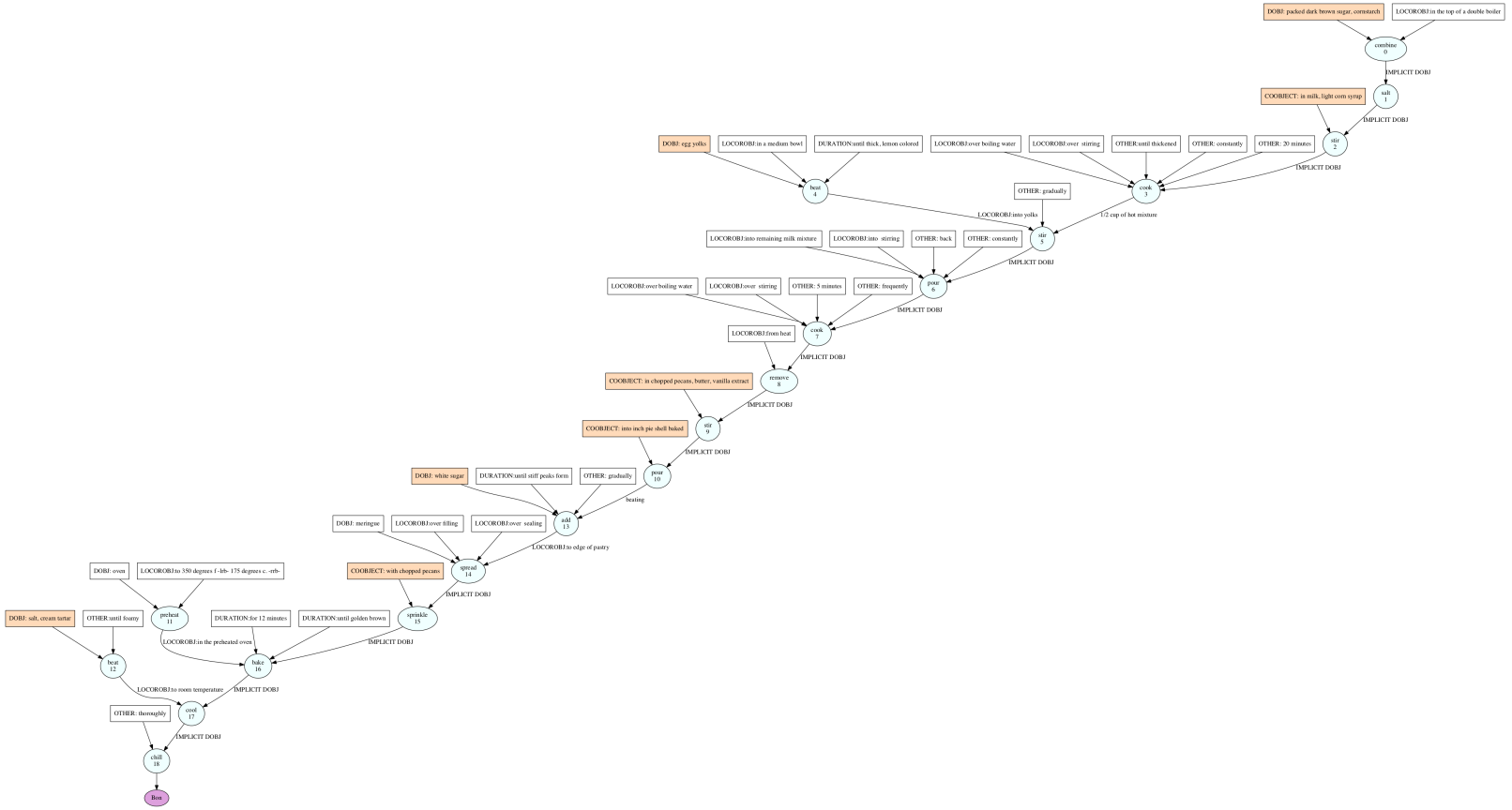


Figure A.3: Generated action graph for “Pecan butterscotch pie” using the automatically generated segmentation

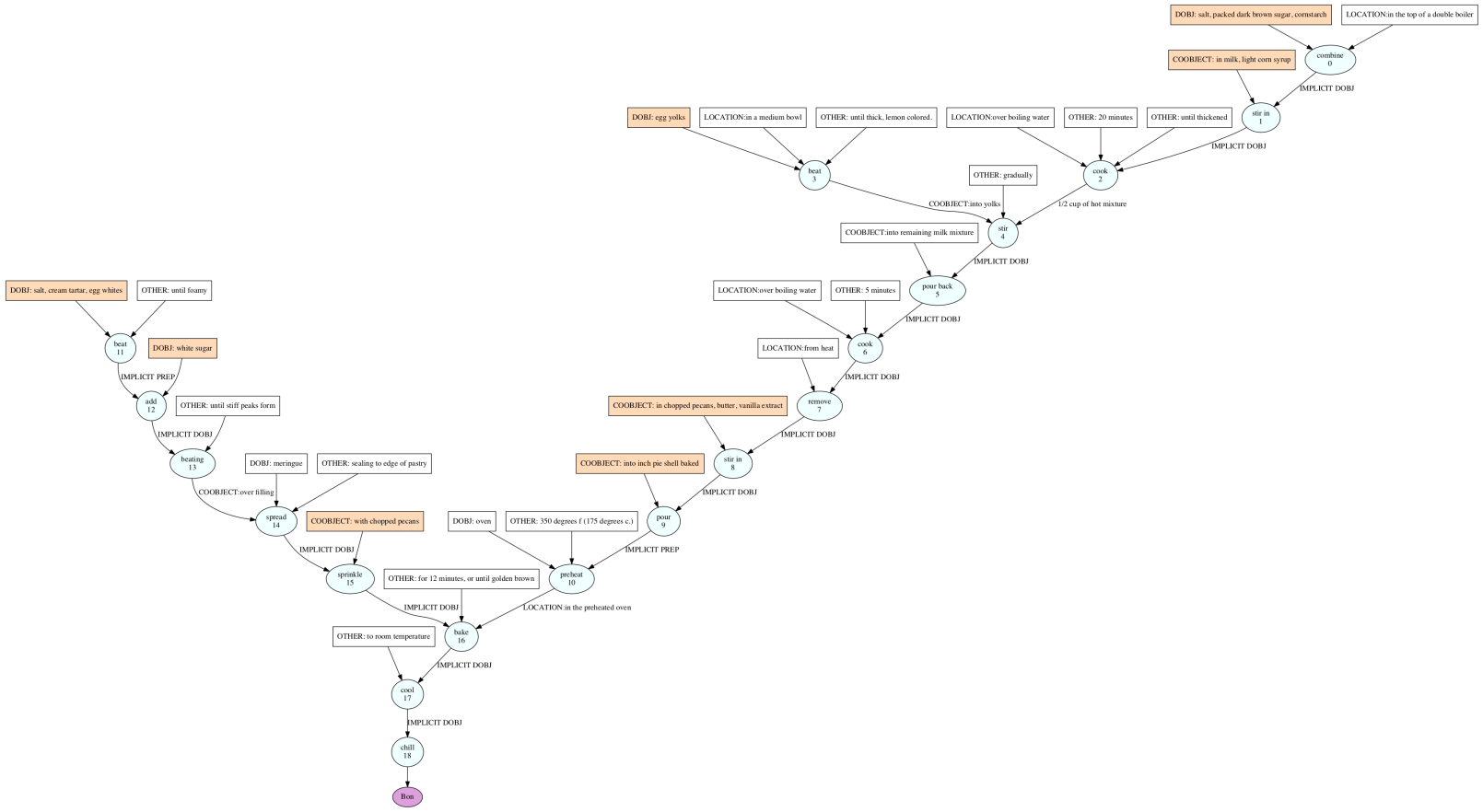


Figure A.4: Generated action graph for “Pecan butterscotch pie” using the gold-standard segmentation

A.3 Teriyaki chicken salad

Steps

1. Marinate chicken in the orange juice, soy sauce and lemon-lime carbonated beverage for several hours or overnight.

2. Preheat grill to medium-high heat. Remove chicken from marinade and drain. Place chicken on hot grill and cook for 6 to 8 minutes on each side, or until juices run clear. Remove, cool, and cut into strips.

3. Whisk together the lemon juice, vegetable oil, salt, pepper and garlic cloves. Allow garlic cloves to sit in dressing for a few hours and remove before pouring on the salad.

4. In a salad bowl, combine the lettuce, tomatoes, mozzarella, Parmesan and marinated chicken strips. Pour dressing over salad; toss and serve.

Ingredients

- 4 skinless, boneless chicken breast halves
- 1 cup orange juice
- 1 cup soy sauce
- 1 (12 fluid ounce) can or bottle lemon-lime flavored carbonated beverage
- 1 lemon, juiced
- 3/4 cup vegetable oil
- 1 teaspoon salt

- 1/2 teaspoon ground black pepper
- 4 cloves garlic
- 2 heads romaine lettuce
- 2 tomatoes, chopped
- 1/3 cup mozzarella cheese
- 1/4 cup grated Parmesan cheese

Data Parsed from this [URL](http://allrecipes.com/recipe/teriyaki-chicken-salad)

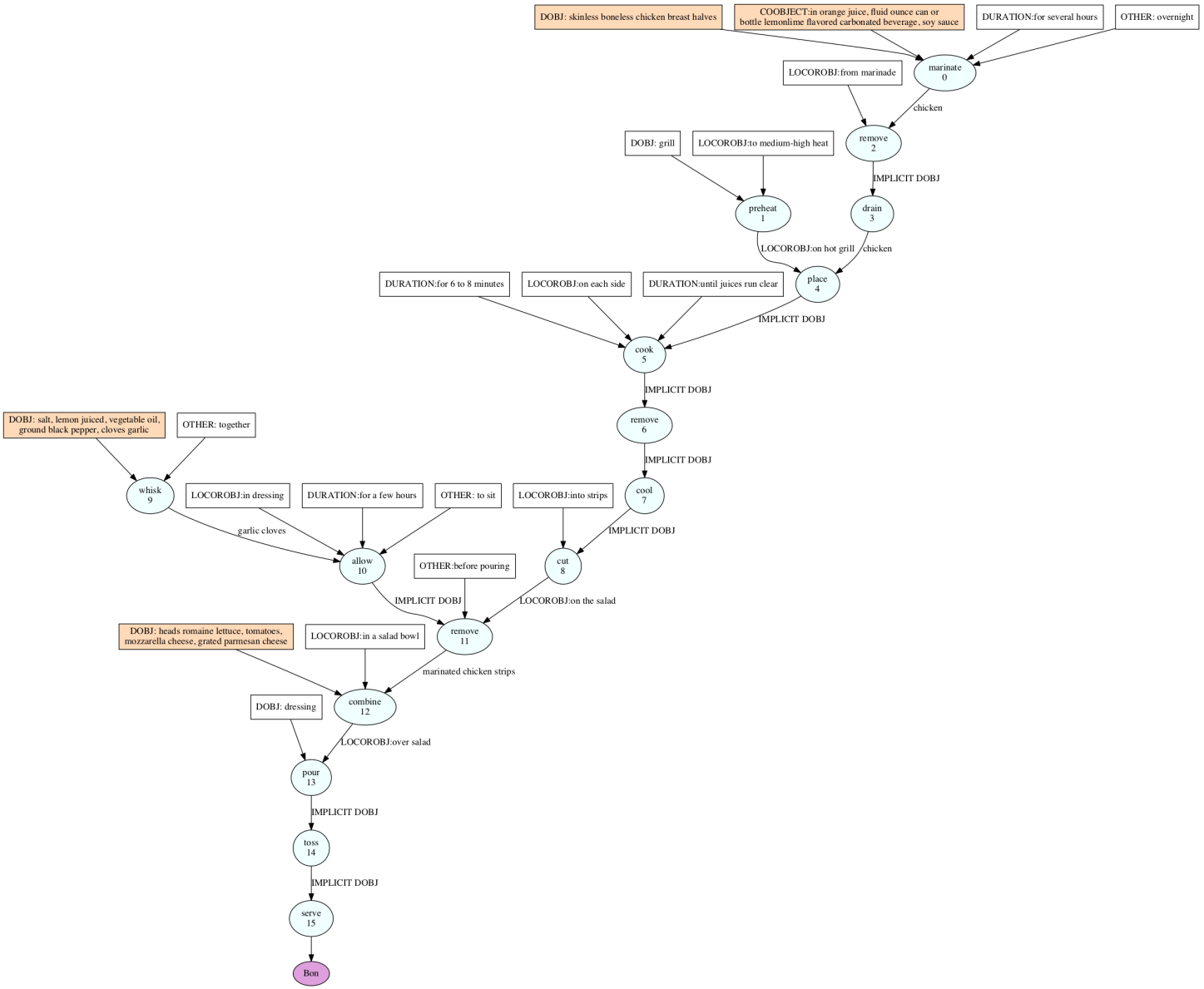


Figure A.5: Generated action graph for “Teriyaki chicken salad” using the automatically generated segmentation

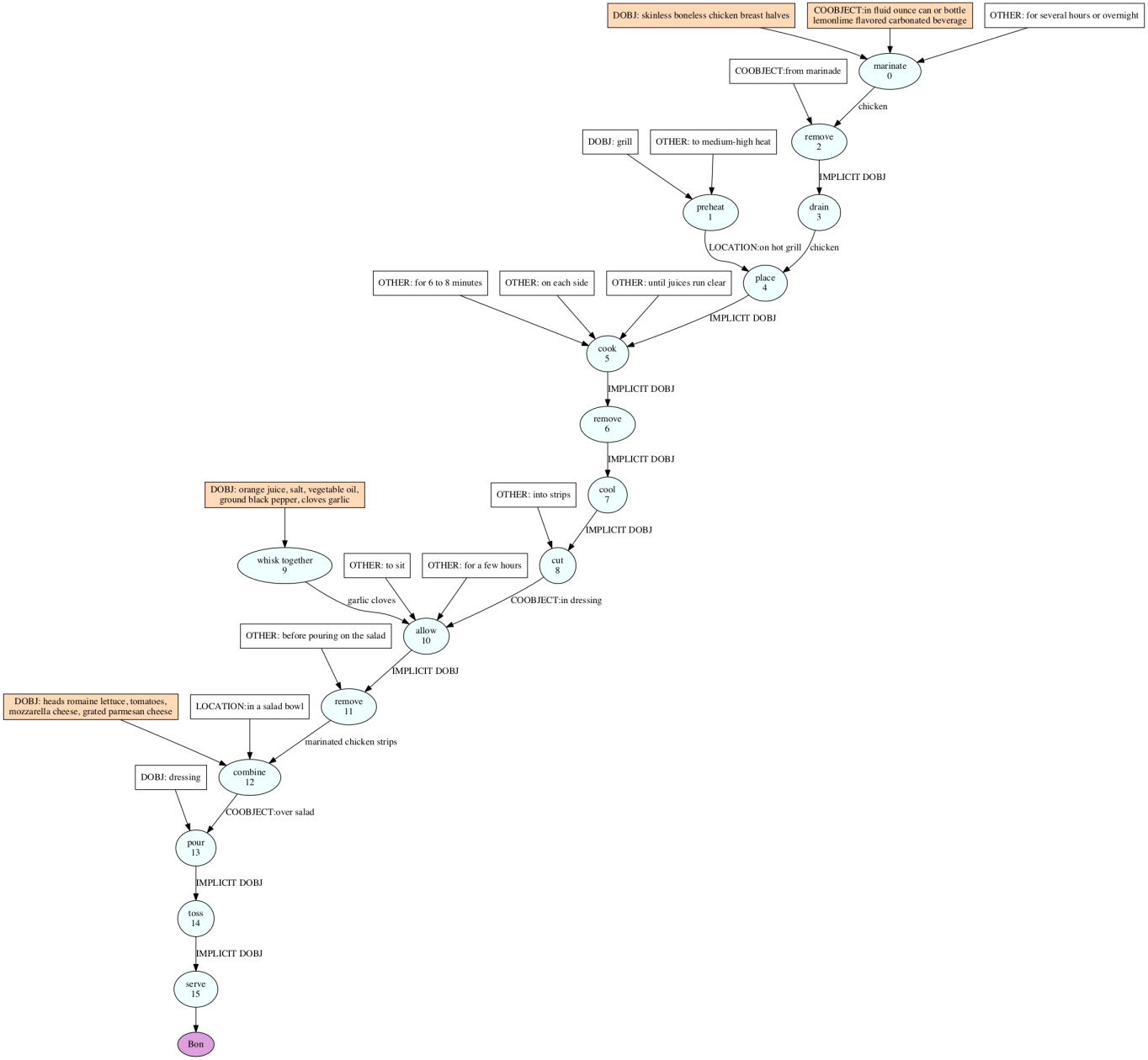


Figure A.6: Generated action graph for “Teriyaki chicken salad” using the gold-standard segmentation

A.4 Beer and bourbon pulled pork sandwiches

Steps

1. Combine paprika, onion powder, garlic powder, oregano, thyme, and salt in a small bowl; season with black pepper.

2. Blot pork chops dry with paper towels, then rub with paprika mixture.

3. Heat about 2 tablespoons canola oil in a non-stick skillet over medium-high heat. Fry pork chops in batches until browned, about 5 minutes per side. Transfer browned pork chops to a slow cooker.

4. Wipe skillet clean and heat remaining 1 1/2 teaspoon canola oil and butter over medium heat; cook and stir onions, 1/2 bottle beer, and a pinch of salt until onion is tender and slightly browned, about 10 minutes. Add liquid smoke. Spread onions over pork.

5. Mix barbeque sauce, remaining beer, Worcestershire sauce, garlic, bourbon, and hot sauce in a bowl; pour over pork.

6. Cook pork on Low until very tender, about 8 hours. Shred and divide pork over rolls to make sandwiches.

Ingredients

- 1 tablespoon paprika
- 2 teaspoons onion powder
- 2 teaspoons garlic powder
- 2 teaspoons dried oregano

- 2 teaspoons ground thyme
- 1 teaspoon salt
- 1 pinch ground black pepper, or to taste
- 1 (3 pound) pork roast, cut into 3-inch chops
- 2 1/2 tablespoons canola oil, divided
- 1 1/2 teaspoons butter
- 2 onions, sliced
- 1 (12 fluid ounce) can or bottle wheat beer, divided
- 1 pinch salt
- 1 teaspoon liquid smoke flavoring
- 3/4 cup barbeque sauce
- 1 1/2 teaspoons Worcestershire sauce
- 5 cloves garlic, minced
- 2 (1.5 fluid ounce) jiggers bourbon whiskey
- 3 dashes hot pepper sauce
- 6 crusty bread rolls, split

Data Parsed from this [URL](http://allrecipes.com/recipe/beer-and-bourbon-pulled-pork-sandwiches)

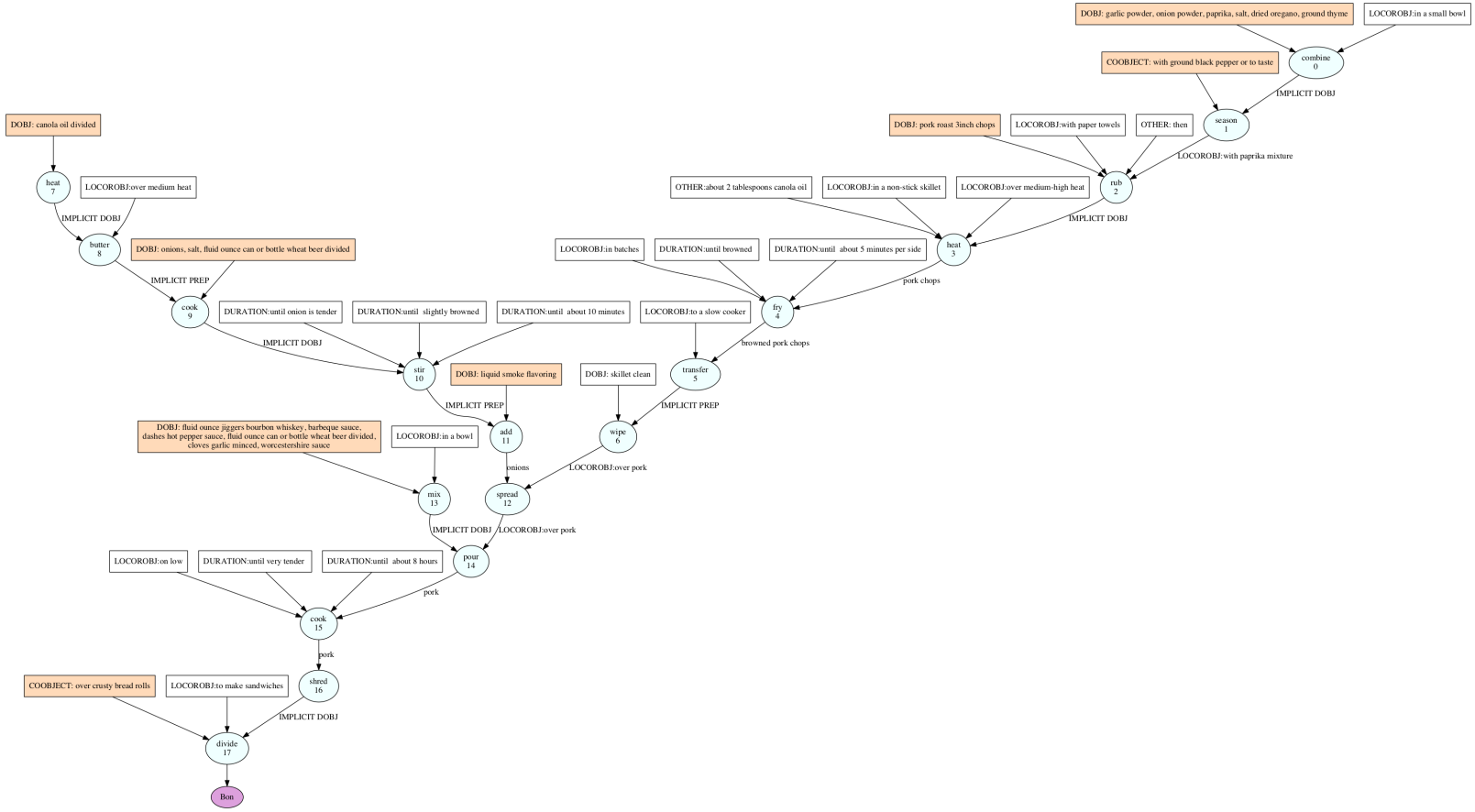


Figure A.7: Generated action graph for “Beer and bourbon pulled pork sandwiches” using the automatically generated segmentation

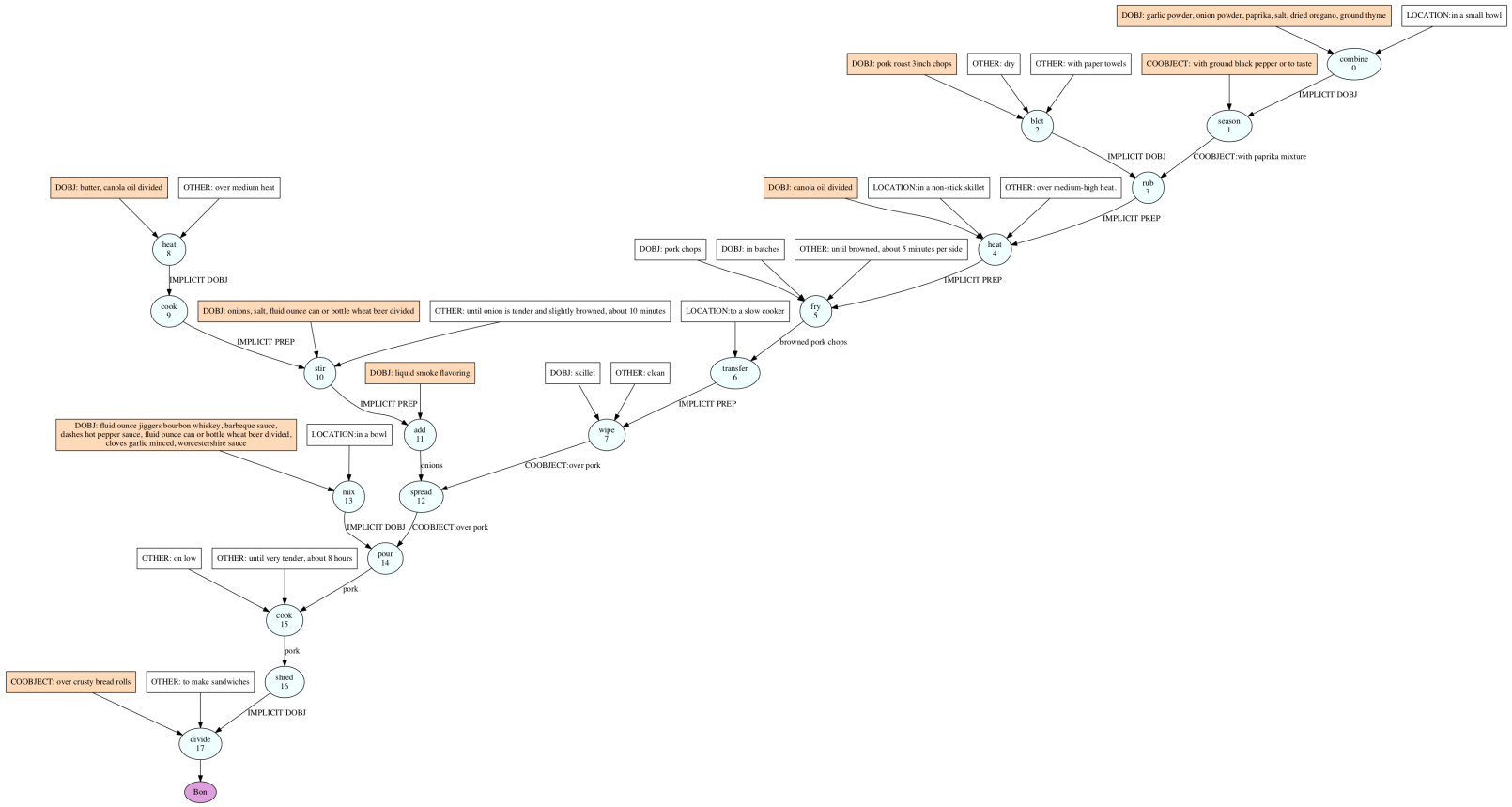


Figure A.8: Generated action graph for “Beer and bourbon pulled pork sandwiches” using the gold-standard segmentation

A.5 Corn cheese chowder

Steps

1. Using a saute pan over medium heat, saute onion in butter until tender.

2. Add flour and stir, it will form a paste like consistency.

3. Add milk and stir until thickened. Add corn, cheese and season with salt and pepper.

4. Heat through, until the cheese melts and then serve hot.

Ingredients

- 1/4 cup butter, melted
- 1/4 cup chopped onion
- 1/4 cup all-purpose flour
- 4 cups milk
- 2 (15 ounce) cans creamed corn
- 1 1/2 cups shredded American cheese
- 1 teaspoon salt
- 1/4 teaspoon white pepper

Data Parsed from this [URL](http://allrecipes.com/recipe/corn-cheese-chowder)

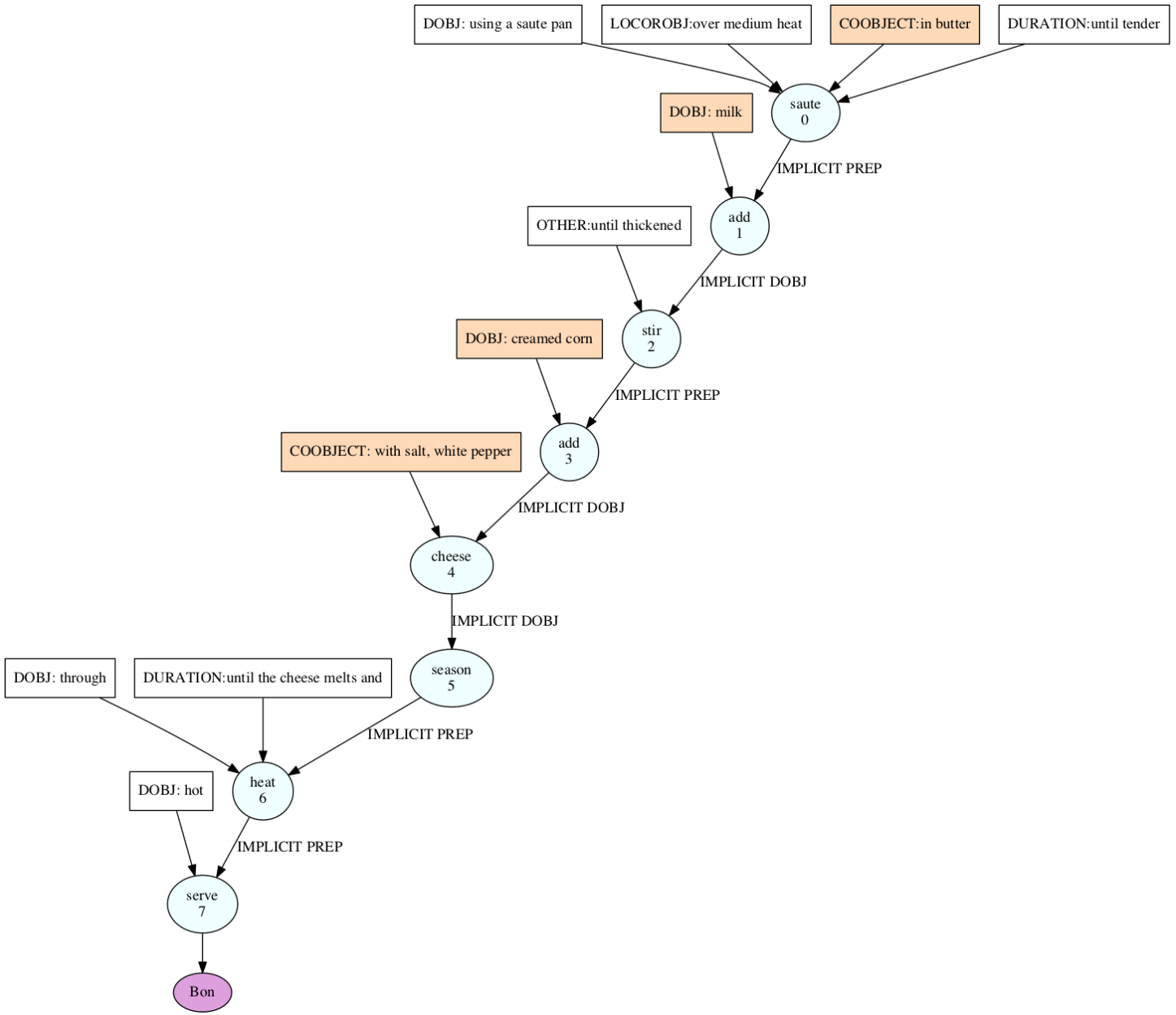


Figure A.9: Generated action graph for "Corn cheese chowder" using the automatically generated segmentation

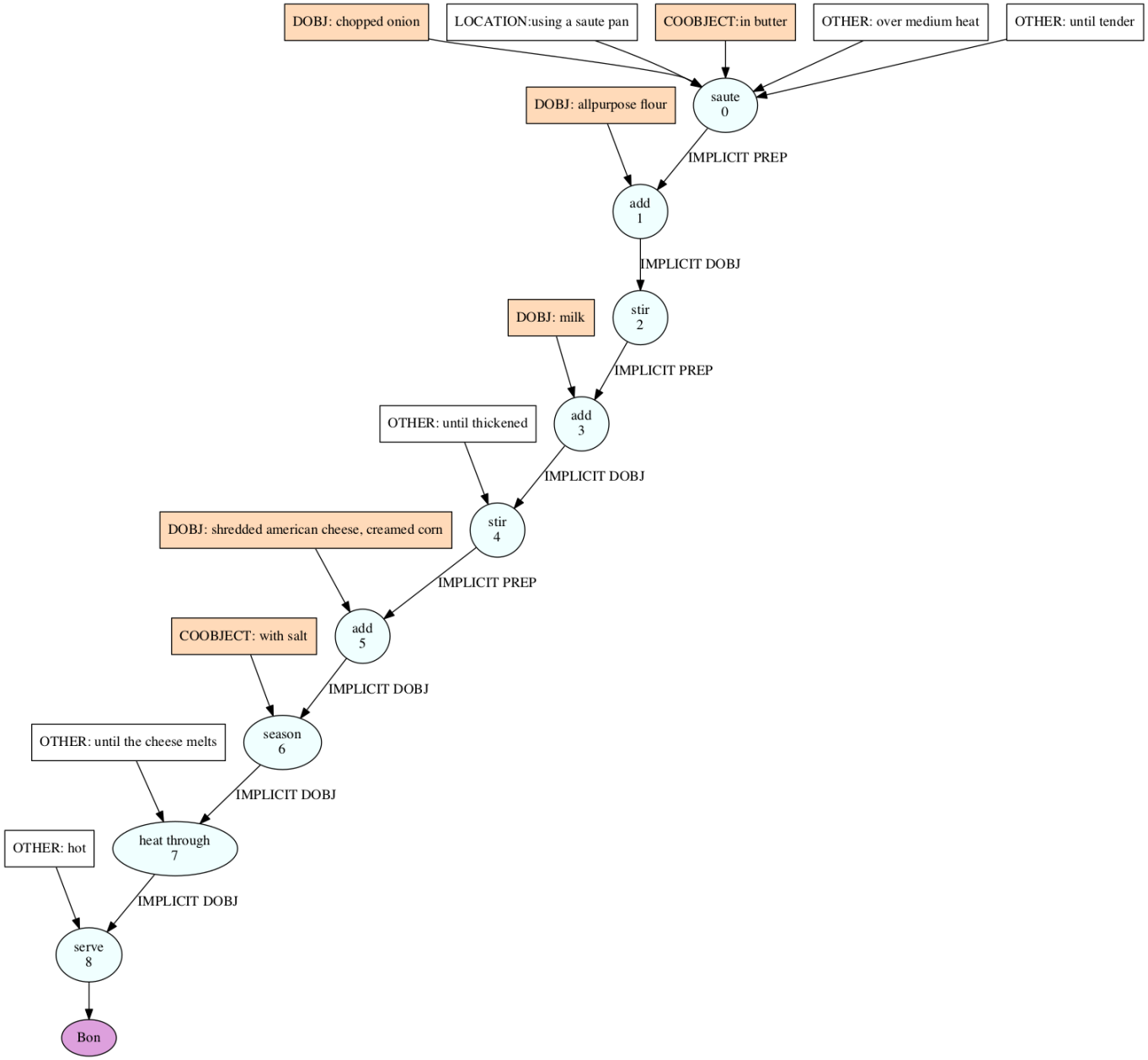


Figure A.10: Generated action graph for "Corn cheese chowder" using the gold-standard segmentation

VITA

Chloé Kiddon is a Ph.D. candidate in Computer Science & Engineering at the University of Washington advised by Professors Yejin Choi and Luke Zettlemoyer. Her research interests include building interpretable models and neural network architectures for natural language understanding and natural language generation, with a recent focus on instructional texts. She received an NSF Graduate Research Fellowship in 2010. Chloé previously received her Bachelors Degree with Honors in Computer Science at Stanford University in 2008 while working in the Stanford Natural Language Processing Group with research advisor Christopher Manning.