Bachelor's Thesis

Crowdsourcing Instruction Data for Statistical Relational Learning

Crowdsourcing von Instruktionsdaten für Statistisches Relationales Lernen

Author	Kevin Scheck
Supervisor	Prof. Michael Beetz, Ph.D.
Second Supervisor	Prof. Dr. Ing. Johannes Schöning
Advisor	Mareike Picklum, M.Sc.

15th June 2017

Abstract

Crowdsourcing is a popular method to generate data sets for machine learning, however most of the crowdsourced tasks are short and of low complexity. We evaluate whether crowdsourcing is an adequate solution to obtain data for a Statistical Relational Learning (SRL) application, as its labeling tasks can require additional knowledge and multiple, interdependent annotations. We create a task design which guides crowdsourced workers through the annotation process and lets them interactively select relations in natural-language instructions. Using this design, we let workers annotate relations in over 900 instructions on Amazon Mechanical Turk. For quality assurance, we require workers to pass a test about the annotation rules before they can complete a task. The workers' submissions are of high quality, but their completion rate is lower than reported in related work. SRL models trained with crowdsourced data showed lower performance compared to models trained with expert data. This indicates that it is feasible to use crowdsourcing with more complex labeling tasks, but the usage of crowdsourced data can require additional effort to improve SRL models.

Contents

1 Introduction **1**

2 Foundations and Related Work **3**

2.1 Probabilistic Action Cores 3

2.2 Amazon Mechanical Turk as Crowdsourcing Platform 9

2.2.1 The Mechanics of Amazon Mechanical Turk 9

2.2.2 Related Usages 12

3 Implementation of a Web Application for Instruction Annotation **17**

3.1 Task Designs 17

3.2 Quality Assurance 19

3.3 Design of the Task's User Interface 22

3.4 Architecture of the Web Application 27

4 Evaluation **29**

4.1 Experiment Conduction 29

4.2 Evaluation of the Crowdsourced Annotations 34

4.2.1 Description of the Crowdsourced Annotations 34

4.2.2 Annotation Quality of Individual Workers 37

4.2.3 Training MLNs with Crowdsourced Data 43

5 Conclusions and Future Work **53**

A The Qualification Test of the Implemented Task Design **57**

B Estimating the Task Completion Time **61**

C Complete Inference Results of MLNs Trained With Expert and Crowdsourced Data **63**

C.1 Action Core Recognition 63

C.2 Property Extraction 66

C.3 Action Role and Word Sense Inference 67

C.4 Action Core Refinement 70

References **73**

Online References **76**

List of Figures

1 The reasoning pipeline of PRAC. 4
2 Visualization of the grammatical dependencies of the running example. . . . 4
3 The worker overview of the newest available HITs. 11
4 The HIT preview as seen by a worker. 12
5 A flowchart of the workflow if the PRAC data annotations are split up among workers. 18
6 The action subtask "Select the first actions". 23
7 The action subtask "Assign roles". 24
8 The action subtask "Refine generic actions". 25
9 The properties assignment page. 26
10 The word sense assignment page. 26
11 The architecture of the web application. 27
12 Posted and completed HITs of the evaluation. 30
13 A more complicated annotation of a sentence of the second batch. 31
14 Number of approved HITs for each worker. 32
15 Percentages of the number of correctly answered questions of the qualification test. 33
16 Correspondence between the number of completed steps and the effective wage of workers. 34
17 Number of selected ACs in the crowdsourced annotations. 36
18 Percentages of selected refinements by AC. 37
19 An example of wrong property selections. 42
20 Histogram of the total completion times of the single steps of the howtos annotated by the author. 61

List of Tables

1 Predicates of the grammatical dependencies and POS tags of the running example. 5
2 Predicates related to ACs and action roles for the running example. 6
3 Selected synsets for the word *"pot"* obtained from WordNet. 7
4 Predicates related to word senses for the running example. 7
5 Used qualifications in the posted HIT batches. 30
6 Descriptive statistics of the approved crowdsourced annotations. 35
7 Possible types of mistakes workers can make while annotating howtos. . . . 40
8 Descriptive statistics of the workers' mistakes measured on sampled annotations. 41
9 Recognized AC by MLNs trained with expert and MTurk data. 45
10 Recognized properties by MLNs trained with expert and MTurk data. . . . 46
11 Inferred action roles by MLNs trained with expert and MTurk data. 47
12 Recognized AC refinements by MLNs trained with expert and MTurk data. 49
13 Initial features used to estimate the task completion time for a single step. . 62
14 Complete inference results of PRAC's AC recognition module. 64
15 Complete inference results of PRAC's property extraction module. 66
16 Complete inference results of PRAC's action role and word sense module. . 67
17 Complete inference results of PRAC's AC refinement module. 71

List of Abbreviations

AC	Action Core
ACQ	Attention Check Question
FOL	First-order Logic
HIT	Human Intelligence Task
HTML	Hypertext Markup Language
ID	Identifier
JSON	JavaScript Object Notation
POS	Part-of-speech
ML	Machine Learning
MLN	Markov Logic Network
MTurk	Amazon Mechanical Turk
NL	Natural-language
NLP	Natural-language Processing
NLTK	Natural-language Toolkit
PCTL	Percentile
PRAC	Probabilistic Action Cores
SRL	Statistical Relation Learning
UI	User Interface
URL	Uniform Resource Locator
VQA	Visual Question Answering
WSD	Word Sense Disambiguation
WSI	Word Sense Induction
XML	Extensible Markup Language

1 Introduction

A large amount of the recent advancements in Machine Learning (ML) can be attributed to the massive volume of available data which is now generated by smart mobile devices, web services or internet-connected gadgets [GBC16, p. 19]. Even though more data is now accessible than ever, not all ML tasks can directly benefit from it. Supervised learning requires each data point to have a corresponding target label to train a model. However, in many domains, creating correct labels of data is time consuming, costly and may require human experts, even though a huge amount of unlabeled data is present.

An example for an application which suffers from this problem is the Probabilistic Action Cores (PRAC) framework [NB15a][Nyg+17], which transforms Natural-language (NL) instructions like recipes into fully parameterized robot plans [NB15a, p. 1]. It relies on Markov Logic Networks (MLNs), a Statistical Relation Learning (SRL) algorithm, to build joint distributions over actions, their roles and other properties contained in the instructions [NB15a, p. 11]. A labeled sample of PRAC consists of all modeled relations contained in an NL instruction, encoded in First-order Logic (FOL). For example, the sample of the instruction *"Cut the tomatoes."* must include predicates which denote that there is a *"Cutting"* action and that the tomatoes are the objects to be cut. Creating these samples is expensive and time-consuming, as the labels require expert knowledge and there are numerous relations contained in a single NL instruction. This severally inhibits the development of PRAC, as much of its authors' time is spent labeling data instead of advancing the application.

A possible solution for this problem is crowdsourcing [How06]. Instead of fulfilling the labeling tasks in-house, they are posted online to an anonymous crowd of workers. This brings major advantages for the practitioners, since they gain the flexibility to attain labels on-demand instead of labeling the data themselves. Additionally, crowdsourcing is regarded as cost-effective and efficient, as the majority of tasks are completed within a few days and for a fraction of the cost of dedicated workers [Ipe10, pp. 19-20]. A prominent example of a crowdsourcing platform is Amazon Mechanical Turk [MTURK], which was used to cheaply label data in many ML domains like Natural-language Processing (NLP) [Sno+08] [Akk+10], computer vision [SF08] and speech recognition [MBR10].

However, most tasks on Amazon Mechanical Turk (MTurk) do not require expert knowledge and only take seconds or a few minutes to complete [KCH11, p. 91]. Therefore, the labeling task for PRAC must be designed such that non-experts can create valid training data. Firstly, workers must be provided with the required knowledge about the relations of PRAC. Secondly, the task must be designed in such a way that workers can effectively apply this knowledge to annotate NL instructions in a short amount of time. However,

even if an adequate labeling task is designed, the following problems can still occur when using MTurk: Workers might not be interested in the labeling tasks, as they require a greater time investment than typical micro-tasks. Furthermore, the gathered data can be of low quality, as the workers must apply newly learned knowledge besides common everyday knowledge. Lastly, even if the quality of the gathered labels is high, they can result in a trained model which performs worse than a model trained with expert data.

This Bachelor's Thesis examines whether crowdsourcing is feasible to create labels for the PRAC framework. The goal is to crowdsource high-quality training data for PRAC and evaluate their usability for training MLNs. If crowdsourcing can be used to generate adequate annotations for PRAC, then not only can the development process of PRAC be severally eased, but crowdsourcing can also be a feasible option for other SRL applications with complex labels. The contributions of this thesis are the following:

1. We investigate whether crowdsourced workers are willing to fulfill more complex tasks by posting 241 labeling tasks to MTurk, in which they must annotate a recipe for the PRAC framework. In total, 87 recipes with over 900 NL instructions were annotated.

2. To ease the task of creating PRAC training data for non-experts, we create a task design which guides them through the annotation process and lets them interactively label relations in NL instructions. To ensure that workers know our annotation rules, we create a test which they must pass before they are allowed to submit our labeling tasks.

3. We evaluate the crowdsourced data in two ways: Firstly, we examine the annotation quality by reviewing sampled recipes for mistakes which workers can make during the labeling task. Secondly, we assess the usability of the annotations as training data. We retrain MLNs used by the PRAC framework with the gathered data and compare their performance with the models trained with expert data.

This Bachelor's Thesis is structured as follows: Section 2 provides an overview of the PRAC framework and MTurk as well as a review of related work. Building on these foundations, possible annotation task designs are first discussed in Section 3. Subsequently, the chosen design and its implementation are explained in greater detail. Section 4 describes the used procedure of posting tasks to MTurk and the gathered data. Furthermore, it presents the evaluation results of the review of the workers' annotation quality and the comparison of MLN models trained with crowdsourced and expert data. Section 5 summarizes this Bachelor's Thesis and outlines possibilities for future work.

2 Foundations and Related Work

This section provides an overview over the main topics which are required for the following thesis. As the main focus of thesis is to generate training data for the PRAC framework, its components and the resulting training data are outlined in Section 2.1. Subsequently, the foundations and related usages of MTurk are explained in greater detail in Section 2.2.

2.1 Probabilistic Action Cores

PRAC is a framework which generates action plans for robots from NL instructions [NB15a, p. 1]. For example, given the instruction *"Slice the tomatoes into small pieces and add them to the pot."*, it infers that there is an *"Adding"* as well a *"Cutting"* action in it. Furthermore, it infers which objects in this instruction are used to carry out the actions, e.g. the tomatoes should be cut in this instruction. These results are embedded in a robot plan and returned as output.

PRAC uses probabilistic knowledge bases which are implemented as MLNs [NB15a, p. 11]. MLNs model relations in FOL and attach a weight to each formula [RD06, p. 5], which denotes the formula's importance in the modeled domain. If a world becomes inconsistent with the knowledge base, it becomes less probable the larger the weights of the contradicting formulas are. This makes it possible to model domain knowledge with logical formulas while being able to handle uncertainty which is present in NL instructions. MLNs can be learned by determining their template formulas' weights which maximize the likelihood of some given training data [RD06, p. 15].

However, modeling PRAC's main inference task for finding an interpretation in a single probabilistic knowledge base results in an enormously large distribution [NB15a, p. 12]. Therefore, PRAC breaks down its inference task into subproblems which are processed in a pipeline shown in Figure 1. PRAC not only uses MLNs for probabilistic reasoning, but also uses a semantically indexed database to infer missing information in modules which are connected to the *"Database"* in Figure 1 [Nyg+17, p. 2]. To find information which is not included in the parameter instruction, it queries the database for already annotated NL instructions which are semantically the most similar to the given instruction.

The pipeline consists of modules which, apart from the first module, take a set of predicates as evidence, which are treated as given truth. They run a separated inference task and pass the result together with the evidence to the next module [Nyg+17, p. 6]. Each module therefore serves as an expert for certain inference tasks which are described in the following.

Parsing The first pipeline module takes the raw string instruction as input and returns a logical representation which is used as evidence in further inference tasks [NB15a, p. 6].

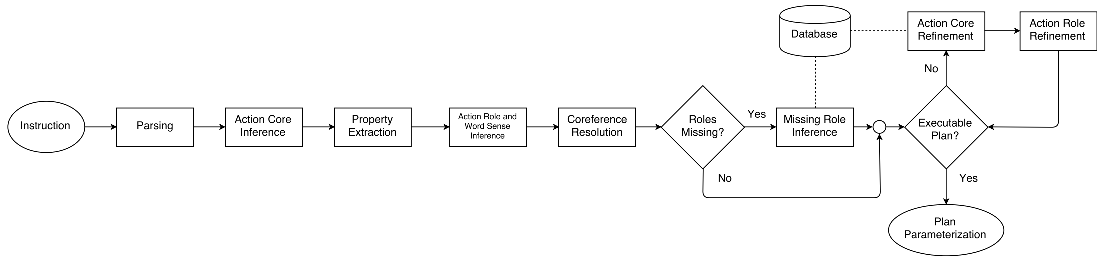

Figure 1: The reasoning pipeline of PRAC used for the inference task [Nyg+17, p. 6].

This logical form is a set of predicates about the grammatical relations of the sentence and is generated by the Stanford Parser [D+06]. There are two types of predicates which are returned by this module:

The first type are grammatical dependencies in the sentence which represent relations between its words and are encoded as a predicate together with the dependent words. The left column in Table 1 lists all grammatical dependencies obtained for our running example *"Slice the tomatoes into small pieces and add them to the pot."*. Figure 2 visualizes these dependencies in the sentence.

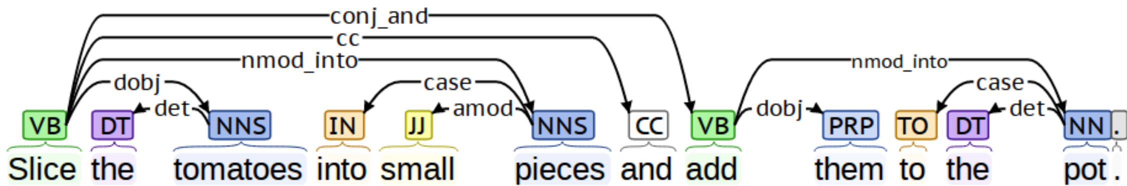

Figure 2: Visualization of the grammatical dependencies of the running example. In all predicates symbols, double points are replaced by underscores. For example, *nmod:in* from the Stanford Parser becomes *"nmod_in"*.

The predicate symbols provide information about the kind of dependency[1] the words have and are important for inferring the objects which are acted on in actions. As an example, the predicate *dobj* marks a relation between a verb and an object which is the *direct object* of the verb. In our running example, the two *dobj* predicates therefore indicate that the tomatoes should be sliced and the word referenced by *"them"* should be added somewhere.

The second type of predicates are Part-of-speech (POS) tags of the words in the NL instruction. They provide information about which part of speech the words in the sentences have, i.e. if they are nouns, verbs, adjectives and so on. For example, the word *"Slice"* is a

[1]An overview of the dependencies can be found on the website [UNIDEP].

Table 1: Predicates of the grammatical dependencies and POS tags of the running example.

Grammatical dependencies	**Part-of-speech tags**
root(ROOT-0, Slice-1)	has_pos(Slice-1, VB)
cc(Slice-1, and-7)	has_pos(the-2, DT)
det(tomatoes-3, the-2)	has_pos(tomatoes-3, NNS)
conj_and(Slice-1, add-8)	has_pos(into-4, IN)
dobj(Slice-1, tomatoes-3)	has_pos(small-5, JJ)
dobj(add-8, them-9)	has_pos(pieces-6, NNS)
case(pieces-6, into-4)	has_pos(and-7, CC)
case(pot-12, to-10)	has_pos(add-8, VB)
amod(pieces-6, small-5)	has_pos(them-9, PRP)
det(pot-12, the-11)	has_pos(to-10, TO)
nmod_into(Slice-1, pieces-6)	has_pos(the-11, DT)
nmod_to(add-8, pot-12)	has_pos(pot-12, NN)

This table lists all predicates of the grammatical dependencies and POS tags of the running example *"Slice the tomatoes into small pieces and add them to the pot."*.

verb and *"tomatoes"* is a noun in plural form. PRAC represents POS tags as predicates with the symbol *has_pos*, which takes the word in the sentence and an abbreviation of the POS as arguments. The right column of Table 1 lists all predicates of the POS tags[2] of the running example. In the visualization in Figure 2, the POS tags are also included in colored boxes above the words in the sentence. They are also crucial information, because they reduce the set of possible word meanings, which are inferred in the following modules of the pipeline.

Action Core Inference The second module uses the grammatical structure and POS tags to infer so called Action Cores (ACs) which are defined as *"the conceptualization of an action which constitutes an abstract event type and assigns a role to each entity that is needed in order to successfully perform the respective action."* [NB15a, p. 7]. In our running example, the two ACs *"Adding"* and *"Cutting"* are inferred. This results in two separate sets of evidences for the inferred ACs, which additional contain the two predicates *"action_core(Slice-1, Cutting)"* and *"action_core(add-8, Adding)"*.

Property Extraction The next module infers properties of objects mentioned in the instruction, such as sizes or colors. In our running example, the tomatoes should be sliced

[2]The website [POSTAGS] lists all POS tags obtainable by the Stanford Parser.

into *small* pieces. The module therefore yields the ground predicate *size(pieces-6, small-5)*, which gets added to the set of evidences.

Action Role and Word Sense Inference This module executes two inference tasks simultaneously [NB15a, p. 13]: Firstly, the action roles of the activated ACs must be inferred. They denote the required components to execute the action and must be assigned to words in the instruction. An action role assignment is encoded with the role's name as predicate symbol, which takes the assigned word and the role's AC as arguments. The left column of Table 2 lists the action roles which were assigned by this module for our running example.

Table 2: Predicates related to ACs and action roles for the running example.

Predicates related to ACs and action roles	
Adding	**Putting**
action_verb(add-8, Adding)	achieved_by(Adding, Putting)
goal(pot-12, Adding)	action_verb(add-8, Putting)
theme(pieces-6, Adding)	location(pot-12, Putting)
Cutting	obj_to_be_put(pieces-6, Putting)
action_verb(Slice-1, Cutting)	
obj_to_be_cut(tomatoes-3, Cutting)	

This table lists the predicates related the ACs *"Adding"*, *"Cutting"* and *"Putting"* and their roles for the running example. The *"Adding"* and *"Cutting"* ACs are activated by the *"AC Recognition"* module, while *"Putting"* is chosen as refinement for *"Adding"* in the *"Action Core Refinement"* module.

Secondly, word senses of the action role words are inferred. It is required that every action role is assigned to a word sense [NB15a, p. 8], since words can have varying meanings depending on their context. These word senses are provided by WordNet [Mil95] as so called synsets, which can be regarded as different meanings for the same word and therefore correspond to word senses. Table 3 lists a selection of synsets for the word *"pot"* obtained from WordNet. The first word sense *"pot.n.01"* is the best matching one for our running example, but other definitions may apply in other contexts. Furthermore, the sets of possible word senses get filtered by the words' inferred POS tags [NB15a, p. 7]. In the example of the word *"pot"*, the synset *"pot.v.01"* is not considered as possible word sense, since its POS is a verb, but the inferred POS of this word in our running example is a noun (see Table 1).

Table 3: Selected synsets for the word *"pot"* obtained from WordNet.

Name	POS	Definition
pot.n.01	Noun	metal or earthenware cooking vessel that is usually round and deep; often has a handle and lid
pot.n.03	Noun	the quantity contained in a pot
pot.n.04	Noun	a container in which plants are cultivated
batch.n.02	Noun	(often followed by 'of') a large number or amount or extent
pot.v.01	Verb	plant in a pot

This table lists selected synsets obtained by the WordNet for the word *"pot"*. The complete list of applicable word senses can be viewed by searching for the word on [WN].

Synsets of nouns and verbs are connected with an *is_ a* relation, representing a generalization [FMS03, p. 3]. WordNet builds a taxonomy over concepts, which can be seen as tree structure which root is the most general concept and the child nodes of a concept are interpreted as its specializations. PRAC uses so called Fuzzy-MLNs [NB15b] which exploit this taxonomy in order to reason about unseen concepts. They compute a measure of the similarity between the word senses s and concept c which is then given as evidence with the predicate *is_ a*(s, c). Using this predicate, formulas which contain similar concepts to the unknown one become more likely. Word senses are assigned to words with a *has_ sense(w, s)* predicate, denoting that a word w in the instruction has the word sense s [NB15a, p. 7]. Unlike the *is_ a* predicates, *has_ sense* predicates are not given as evidence, but are subject for inference. Table 4 lists these predicates for the words assigned to action roles in our running example.

Table 4: Predicates related to word senses for the running example.

has_sense predicates	is_a predicates
has_sense(add-8, add.v.01)	is_a(add.v.01, add.v.01)
has_sense(pot-12, pot.n.01)	is_a(pot.n.01, pot.n.01)
has_sense(pieces-6, piece.n.08)	is_a(piece.n.08, piece.n.08)
has_sense(Slice-1, slit.v.01)	is_a(slit.v.01, slit.v.01)
has_sense(tomatoes-3, tomato.n.01)	is_a(tomato.n.01, tomato.n.01)

This table lists the predicates related to word senses in the running example *"Slice the tomatoes into small pieces and add them to the pot."*.

Coreference Resolution The assigned roles are checked for coreferences by the next module. They occur when words are referenced by other words or following instructions. For instance, a possible NL instruction which can follow after our running examples is: *"Stir."*. This sentence does not contain any information about which objects should be stirred. However, it is obvious for humans that the tomato pieces of the previous sentences should be stirred. Another example for example of coreference is the word *"them"* in our running example. The predicate *dobj(add-8, them-9)* indicates that it is a suitable word for the action role *"obj_to be_cut"*, but it is referencing the word *"pieces"*. It is crucial to replace the referencing word with the full form of the referenced entity in action role assignments, because otherwise no meaningful word senses can be retrieved. Therefore, the action role assignment of *"them"* gets replaced with *"pieces-6"*. For inferring coreferences over multiple sentences, their ordering must be considered in the training data. The order of instructions is encoded with *distance* predicates, which take a word as first argument and the number of instructions between the word's instruction to the last instruction as second argument. Therefore, multiple consecutive instructions must be annotated to generate training data with *distance* predicates. For instance, in a training sample with the running example followed by the step *"Stir."*, the word *"Slice"* has the predicate *"distance(Slice-1, DIST1)"*. PRAC currently uses training samples of maximally three consecutive sentences. Lastly, the distance predicate does not need to be annotated manually, since it can be easily computed given an ordered list of sentences.

Missing Role Inference ACs have a set of required action roles and NL instructions often do not contain every required information to assign them. NL instructions often discard information required to execute actions, as it is obvious to humans, who usually have the required contextual information and background knowledge for this inference. If this is the case in the current evidence, this module infers the missing action roles. This requires common knowledge about which objects, tools and measures are used in different situations and would result in an infeasibly large distribution if it was encoded by MLNs [NB15a, p. 2]. Therefore, PRAC uses a semantically indexed database to infer the missing action role by querying instructions which are semantically the most similar to the evidence instruction, and its action role assignments [Nyg+17, p. 6].

Action Core Refinement PRAC can return a plan if every action role is assigned and their AC maps to a concrete plan schema [NB15a, p. 8]. However, some ACs, denoted as *generic* ACs, do not have a plan schema, as their actual execution vastly differs in different situations. For example, in the sentence *"Add 1 cup of chicken broth in the pot."*, the *"Adding"* action can be carried out by *pouring* the broth from one container into the pot, while in our running example, it is accomplished by *putting* the pieces in the pot. Such ACs can be refined in two ways: Firstly, refinement can be accomplished if there exists a

single AC which execution results in the desired goal as the generic AC [NB15a, p. 8]. In our running example, the generic AC *"Adding"* gets refined with the AC *"Putting"*. The resulting relation is encoded with the predicate *achieved_ by*(a_1, a_2), which denotes that a generic action a_1 is achieved by a more specific action a_2. It is not required that a more specific action must directly map to a plan schema, so multiple AC refinements may occur.

However, some actions are more complex and may require the execution of multiple consecutive actions to accomplish their goal. For example, the instruction *"Make a simple tomato sauce."* requires *cutting* tomatoes, *putting* a pot in the stove, *turning on* the heat and so on. In such cases, the database is queried for the most similar *howto*, which can be seen as a manual or recipe and consists of a list of consecutive instructions [Nyg+17, pp. 4-6]. The single abstract AC then gets replaced with the sequence of consecutive actions of the howto. To generate data for this step, NL sentences should therefore be annotated with the instructions in their howto, as only a completely annotated howto can be used to expand a complex action.

Action Role Refinement After a generic AC has been refined, the action roles of the new refined actions must be assigned. If the AC was refined by an *"achieved_ by"* relation, the action roles get refined by an MLN inference. In our running example, the action roles of *"Putting"* are listed in the right column of Table 2 and must also be provided in the training data. In the case of refinement where the generic AC was expanded into a sequence, the action roles of the new ACs are instead adjusted with the action role assignments of the generic action [Nyg+17, p. 6].

2.2 Amazon Mechanical Turk as Crowdsourcing Platform

Amazon Mechanical Turk is the platform which is used to crowdsource training data for PRAC. Firstly, the basic mechanisms of MTurk for requesters and workers are outlined. Subsequently, advanced methods for achieving high data quality and handling complex tasks are reviewed.

2.2.1 The Mechanics of Amazon Mechanical Turk

The main goods of MTurk are Human Intelligence Tasks (HITs), which are small jobs posted by *requesters* and fulfilled by *workers*. In order to create a HIT, requesters can either use the MTurk website, a provided Command Line Tool [MTCLT] or the web API [MTAPI]. The latter makes it easy to incorporate MTurk into programs, as many libraries exist which provide a wrapper around the API, e.g. "Boto" for Python [BOTO].

HITs have parameters which must be supplied by requesters [MTHIT]. Every HIT has a *title*, a more detailed *description* and a list of keywords to make HITS browsable and

discoverable by workers (see Figure 3). A HIT can have multiple *assignments*, for which the same HIT gets processed by a different worker. If a requester wants to ensure that no worker submits the same HIT multiple times, e.g. for a survey, a single HIT with numerous assignments should be posted. If this is not the case, multiple HITS with one assignment and different question contents should be posted, so that a worker can submit results on the same task multiple times. For example, one can post one HIT for each howto which should be annotated, such that a single worker can annotate multiple howtos. This is also desirable, as most work is done by a minority of workers [Ipe10, p. 21]. If a fixed set of parameters is equal between the HITs, they have the same *HIT Type* and are presented as one HIT to workers.

The *reward* of the HIT determines how much the workers is paid for each assignment. This is not the final amount which is paid by the requester, as Amazon currently charges a 20% fee for each paid reward [MTFEE]. This fee gets raised by an additional 20% if the HIT has more than nine assignments.

Requesters have to set up three time frames for a HIT: The *assignment duration* is the maximally allowed time in which a worker has to complete the HIT once it was accepted. The *auto-approval delay* determines how long a requester can review an assignment until it gets automatically approved. Lastly, the *lifetime* of a HIT sets how long a HIT is active on the market. If a HIT is not fully completed by its end, it is not longer available for new workers.

Requesters can require workers to have a set of qualifications to be allowed to accept their HITs [MTQUAL]. The most common qualifications include that the workers are in a particular location (e.g. in the US), have completed a minimal number of HITs or that a certain percentage of the workers' HITs have been accepted. Additionally, MTurk provides so called "master qualifications"[3] and more finer grained qualifications, for which additional fees are charged. Requesters can also create their own *Qualification Type* and require it in their HITs. It can be assigned to workers either manually or with a *Qualification test.* If this test only contains multiple choice questions, it can be automatically graded by MTurk. This way, the requester can ensure that the workers have the right skills to work on their HITs.

There are three main ways of creating the HIT's content [MTDATA]: Firstly, MTurk provides an Extensible Markup Language (XML) data structure called *QuestionForm* with which a requester can easily generate a Hypertext Markup Language (HTML) form consistent with MTurk. As only a simple HTML form gets generated, this option is not applicable for use cases which require advanced user interface features. The second option

[3]MTurk classifies workers as masters if they submitted many HITs with good performance [FTFAQ].

HTMLQuestion solves this problem by allowing the requester to supply a valid HTML string as the question's content. The only restriction is that the HTML code must eventually send a POST request to an MTurk Uniform Resource Locator (URL) with the worker's answers and a set of required Identifiers (IDs)[4]. In the third option called *ExternalQuestion*, the question's content is linked to a site hosted on an external server. This option provides the requester with the highest flexibility, as the HIT content can be changed even if the HIT is already posted to MTurk. Like *HTMLQuestions*, the external website must eventually submit a POST request to MTurk with identifiers and the worker's answers.

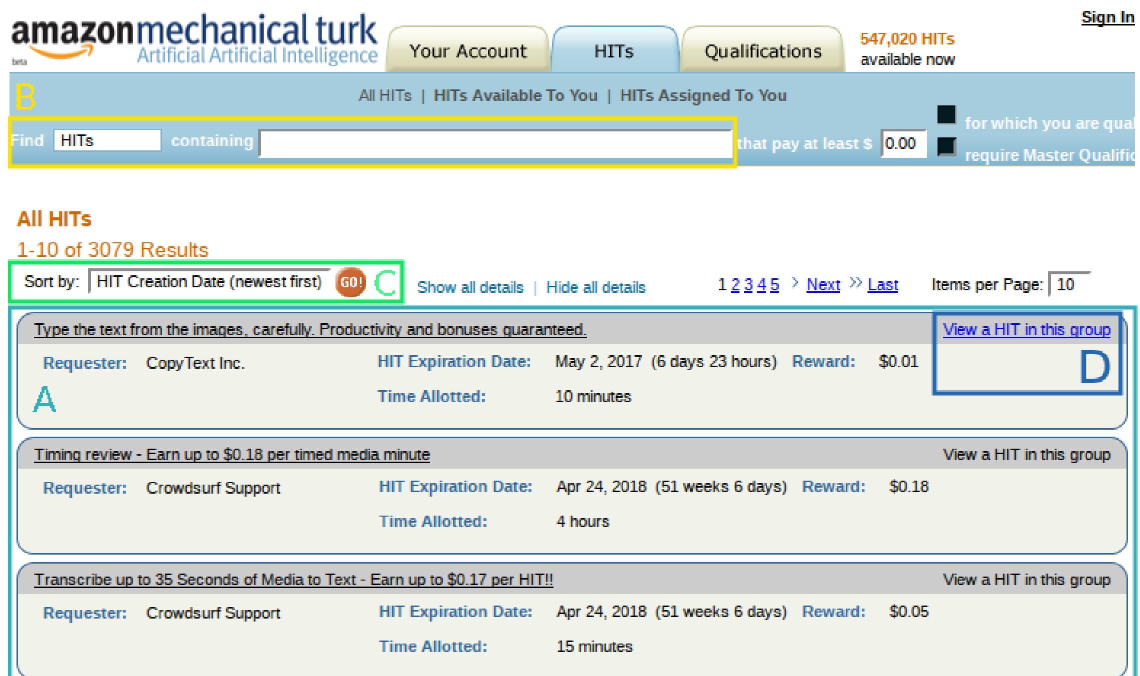

Figure 3: The worker overview of the newest available HITs, which can be seen in a list view (A). Workers can also search for HITS directly (B) or change the sort ordering of the HITs (C). By clicking on the link *"View a HIT in this group"* (D), workers can go to the preview page of a single HIT.

Regardless of how HITs were created, they are presented to workers the same way on the MTurk website (see Figure 3). They can either browse sorted lists of available HITs or search for individual HITs. HITs can be sorted in numerous categories, however, results from [Chi+10] indicate that workers prefer only two categories: Firstly the most recent HITs and secondly HIT types with the most available HITs [Chi+10, p. 1]. This preference also shows in the distribution of the completion times estimated by [Ipe10]. The measurements indicate that the completion time of HITs follows a power law [Ipe10, p. 20], such that

[4] These identifiers are supplied in the URL and consist of IDs of the HIT, the assignment and worker.

most of the HITs are completed in a short amount of time. However, as more time passes, the less likely a HIT will be worked on, because workers focus on more recent HITs. In the measurements of [Ipe10], the first 60% of the HIT types were completed in just under 16 hours, yet the completion time of the last decile spans from 1024 to 16,384 hours [Ipe10, p. 20]. Therefore it is advisable to have a short HIT lifetime and repost HITs if they were not fulfilled.

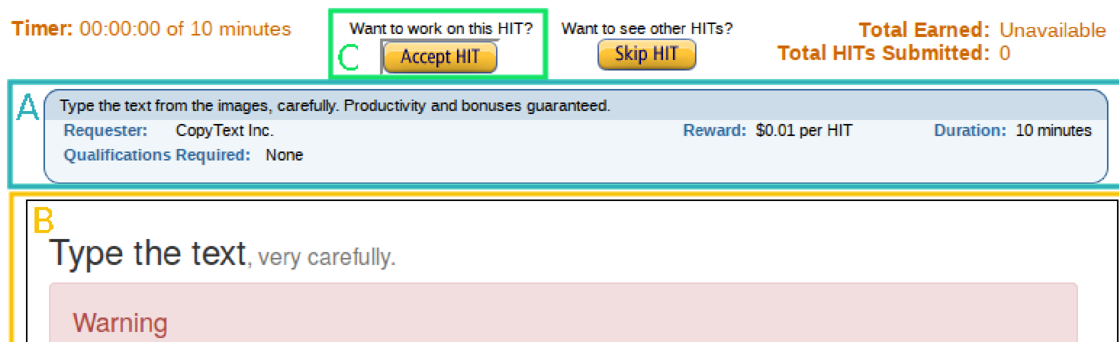

Figure 4: The HIT preview page as seen by a worker. It shows additional information about the HIT (A) and a preview of the HIT's content (B). By clicking on the "Accept HIT" button (C), the worker can start working on the HIT.

Workers can preview a HIT (see Figure 4), such that they can see more of its properties and an embedded view of its content. On the preview site, workers can *accept* the HIT and start working on it. Once the assignment has been submitted by the worker, the requester can review it by either *approving* or *rejecting* it. If the assignment is approved, the worker will be paid normally by MTurk. Besides paying the normal HIT rewards, requesters can pay an additional bonus to submitted assignments. This can be be used to compensate workers who submitted exceptional results or passed a qualification test. In case the assignment is rejected, the worker will not be paid which deteriorated the worker's statistic used for qualifications.

2.2.2 Related Usages

This subsection provides an overview over related work using MTurk. Firstly, work gets reviewed which focuses on how to accomplish tasks which require expert knowledge or a greater amount of effort on MTurk. Secondly, usages of MTurk to generate training data for NLP are reviewed.

Research in the field of human computation introduces methods to use MTurk to accomplish more complicated and interdependent work [Kit+11] [KCH12]. A common strategy to accomplish this is to create so called *workflows*, which divide complex tasks into

smaller subtasks, validate their results with automated verification tasks and eventually merge their partial solutions [KCH11, p. 2054]. This method looks promising for tasks for which a worker is required to have additional knowledge, as the main HIT task can be split up among workers, which drastically reduces the cognitive load and completion time of the HITs. However, experiments using this method show contradicting results: While some experiments resulted in multiple workers providing superior submissions than single workers [Kit+11, p. 46], other experiments required the active intervention of requesters in order to get adequate results [KCH11, p. 2057]. These experiments only posted a handful of HITs and using workflows may become unsuitable if numerous complex tasks are posted. As in some experiments the requesters had to intervene to get adequate results, posting many HITs may result in excessive monitoring effort for the requesters. Additionally, one has to implement strategies to prevent problems which can result from using workflows [KCH11, p. 2058]: The first problem is task *starvation*, i.e. no workers completed the tasks until they expired. Secondly, the verification tasks can fail. Thirdly, the tasks can *derail*, i.e. low-quality solutions get propagated into depending subtasks.

MTurk has been used in a vast number of domains for data collection. The following, selected usages of MTurk specifically for NLP are reviewed, as this domain most closely resembles the tasks performed by PRAC. [Sno+08] assessed MTurk for data acquisition in five multiple NLP tasks: Affect recognition, word similarity, the recognition of textual entailment, the temporal ordering of events and word sense disambiguation. In each task, they used a redundancy of ten annotations for each sample. To evaluate the acquired labels, they measured their agreement with gold standard data. Additionally, they investigated how many workers are needed to achieve a higher or equal agreement with the gold standard data than a single expert. Generally, using a redundancy of multiple workers achieves a high agreement with expert data similar to using a single expert. In the affect recognition tasks, only a few workers were necessary for five out of six sampled affects. Furthermore, they proposed a bias correction method which weights a worker's submission with the worker's accuracy on gold standard data. Using this measure, the accuracy of the text entailment could be improved by 4% on average and the accuracy of the temporal event ordering task by 3.4%. Lastly, they trained an bag-of-words model for affect recognition with gold standard and MTurk data. The model was trained on gold standard data of each single expert and its performance was then averaged over the experts. Subsequently, they investigated how many MTurk workers for each example were necessary to achieve a higher performance. Their findings are that it takes only one worker to achieve a higher performance averaged over all affects. A possible explanation for this is that multiple workers do not show as many biases as a single expert and therefore perform better.

[PE10] used MTurk for Word Sense Induction (WSI), which is the task of finding possible word senses for given words. It can also be seen as clustering dictionary definitions for a word. They evaluated the clustering with a data set of 50 words as follows: Firstly, they posted HITs to determine the number of clusters for each word. Workers were shown HITs with all definitions for a word and were instructed to enter the number of distinct meanings contained in the definitions. Thirteen submissions were obtained for each word and were used in a majority vote to determine the most likely number of clusters. Subsequently, they compared two HIT designs which were used to assign definitions to the resulting clusters. In "global view" HITs, workers were shown all definitions for a word and were instructed to drag and drop the definitions into boxes representing the clusters. In "local view" HITs, workers only saw two definitions for a word and must decide whether they have the same meaning. This requires the comparison of every pair of definitions for a word, so that local view HITs were vastly more expensive than global view HITs. Overall, the local view HITs costed $52.50 while global view HITs costed $7.50. To evaluate the results, a gold standard set of the 50 sampled words was created with four experts. The number of clusters and the cluster assignments of both methods showed high agreement with the gold standard data, indicating that the use of MTurk is feasible for WSI.

[AVC10] used MTurk in conjunction with active learning to train a statistical machine translation model for translations from Spanish to English. Active learning algorithms are ML algorithms which control the data they use for learning. They are useful when there are only a few labeled samples available and labeling costs are high, as they try to maximize the performance gain of getting an additional label. Their used procedure is the follows: Given an untranslated and translated corpus of sentences, the algorithm first selects a fixed number of untranslated sentences with the most n-grams, i.e. the most sequences of n words, which are not contained in the translated set. Three translation HITs get posted for each selected sentence, from which one translation is chosen by using measures of inner-worker agreement and worker reliability. The machine translation model is then retrained with the newly obtained translations. This process is iterated for a fixed number of times. The algorithm was evaluated with in two iterations, in which 1000 Spanish sentences were translated. After both iterations, the performance of the model trained with translations from MTurk was close to the same model trained with expert translations [AVC10, p. 2174]. This indicates that MTurk can be utilized to train translation models for languages for which no large translation corpora exist.

[NM10] translated English sentences into Spanish in so called *creation-verification cycles*, consisting of one translation task followed by multiple verification tasks. They documented problems which may occur using this method: The verification task did not ensure high data quality at first, as verification workers made major mistakes. 61% of the accepted

submission of the first run should have been rejected. In order to reduce the verification errors, they incorporated gold standard data in verification tasks. This greatly increased the time to complete the whole batch, as more HITs got rejected. Adequate completion time and data quality could be achieved by additionally using verifiable questions with gold standard data in the translation tasks, restricting the workers' location and doubling the reward of each HIT.

[Cal09] used MTurk to evaluate machine translation quality and utilized different measures to weight a worker's ratings. As first measure workers agreements with experts for a predefined set of translations was used. Secondly, the agreement of a single worker with the rest of the workers was measured. By incorporating weighted votes, the inner agreement of workers could be drastically improved, which indicates an increase in data quality. Using the votes of five workers with the same measure, the worker agreement was equal to the agreement of experts.

MTurk was used by [Ant+15] to create one of the most widely used data sets for Visual Question Answering (VQA) [Wu+16, p. 12]. VQA is an ML task where the algorithm is given an image with a NL question and must infer a correct answer for the question [Ant+15, p. 2425]. To develop the data set, they used MTurk to create questions and answers for images from the MS COCO data set [Lin+14] and abstract clipart images. In their questions HITs, they instructed workers to ask questions about the image that a "smart robot" would possibly not be able to answer. The intention of this phrasing was to prevent workers from providing simple questions answerable by normal computer vision algorithms. In answer HITs, workers were instructed to give as short answers as possible. This greatly simplifies the assessment of VQA algorithms, as single words or short phrases can be easily evaluated, compared to whole NL sentences. To account for ambiguity in the questions and answers, they used a redundancy of ten answers for each question. The accuracy of an answer was measured based on the number of workers who submitted it and capped its accuracy at 100% if at least three workers provided it. Their initial data set consists of about 800,000 questions and about 10 million answers for about 350,000 images.

3 Implementation of a Web Application for Instruction Annotation

This section describes the implementation of the web application "PRAC Data" which is used to annotate NL instructions and is embedded as an *External Question* in HITs of the evaluation of this thesis. Firstly, possible HIT design concepts are discussed based on HIT designs of related work (see Section 2.2.2). Subsequently, the quality assurance methods and the user interface of the chosen HIT design, as well as the architecture of the implemented web application is illustrated in the following subsections.

3.1 Task Designs

There are multiple criteria which influence the task design for generating training data for PRAC: Non-experts should be able to annotate the relations contained in the PRAC framework and need to be instructed appropriately before they begin the annotation tasks. As cost-effectiveness is one of the main motivations for crowdsourcing, the labeling expenses should not be higher than the costs of labeling the data in-house. Likewise, the time spent by the authors of PRAC to verify the HITs should be kept minimal. In the best case, the verification should happen automatically if an adequate quality assurance method is applicable. Above all, the resulting training data should be free from errors, as they can negatively impact the performance of models trained with the data. Lastly, the HIT design should be usable by the authors of PRAC even if MTurk is no longer used. Therefore, a design is favorable which can be used by the authors to effectively annotate howtos.

One of the first aspects of the HIT design is to decide whether only single steps of a howto or complete howtos are annotated in a single HIT. The first option results in a smaller HIT completion time and is therefore more typical for the micro-tasks of MTurk. However, annotating a single step for each HIT increases the time spent to label a complete howto and therefore the labeling costs. Workers might need to read the whole howto to understand the context required to annotate the sentence. This overhead is not existent if a single worker labels a complete howto. Analogously, the verification of the MTurk assignments becomes more laborious, as steps need to be reviewed individually. Lastly, annotating single steps does not correspond to the way the authors of PRAC create their training data, so the HIT design is inefficient to use without MTurk. In conclusion, we let workers annotate complete howtos, as this corresponds to the way howtos are usually labeled and leads to less overhead for both the workers and requesters.

Besides dividing HITs by steps, they can also be split by the type of annotations. This greatly decreases the cognitive load and completion time of HITs, as workers only need to

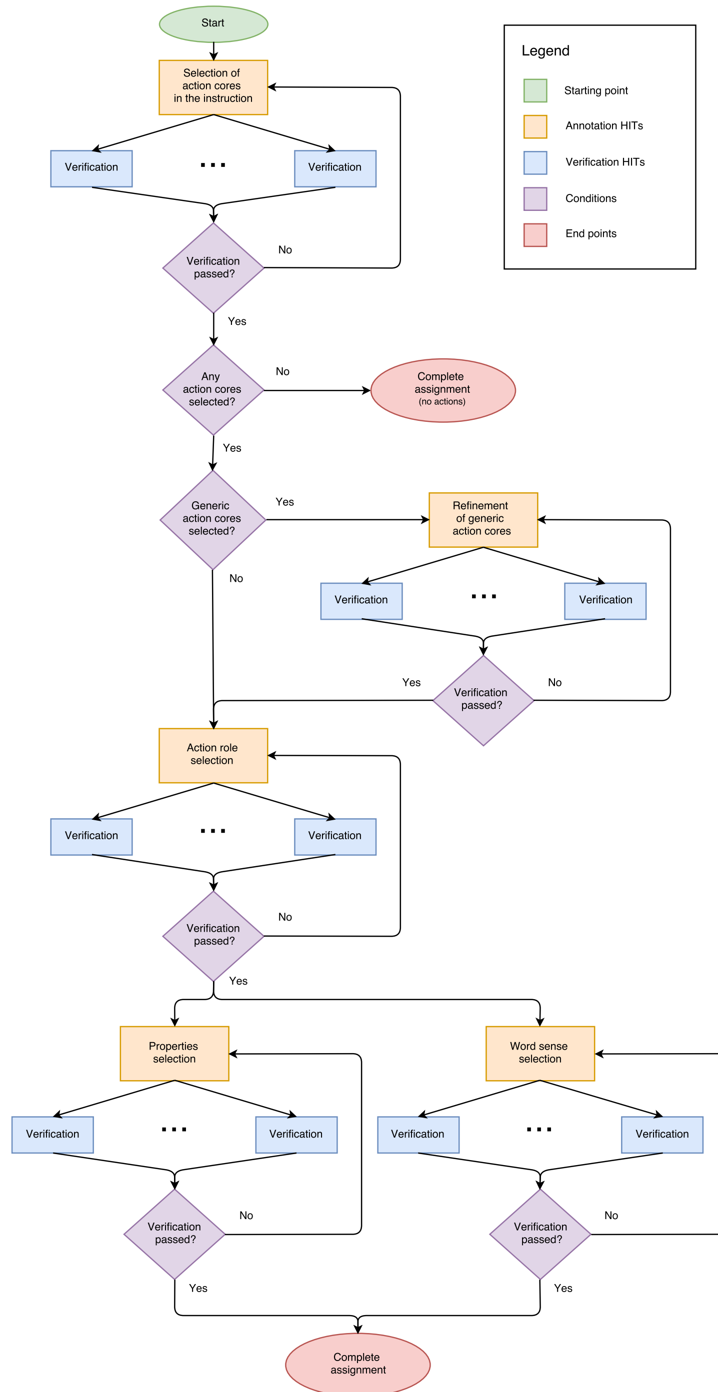

Figure 5: A flowchart of the workflow if the PRAC data annotations are split up among workers. For every annotation task, multiple verification tasks would be used to automatically review the assignment.

apply a subset of PRAC's annotation rules for each HIT. This option also corresponds to the methods used to handle complex tasks on MTurk [KCH11][Kit+11]. However, the time to review the assignments increases if done manually, as there are multiple assignments for each howto. A solution used by other workflow designs are verification tasks in which multiple workers rate the annotations of the assignment. Figure 5 shows a flowchart of the workflow where each annotation task is followed by multiple verification tasks. However, there are several problems resulting from this workflow: Firstly, there needs to be a policy on how to handle annotations which are rejected by the workers. Either they also get rejected on MTurk and are not paid or the they are still approved but not further considered. With the first option, wrong failing verification tasks result in an unfair treatment of the workers, as they might get rejected for correct work. For example, a worker could select the AC *"Adding"* for the sentence *"Season with salt and pepper."*, while the verification workers may think that *"UsingASpiceJar"* should be selected and therefore reject the submission, while both AC assignments are applicable. The latter option increases the labeling costs, as low-quality work or even spam gets paid. Moreover, additional logic needs to be implemented to prevent workers from taking advantage of the system, as they can coordinate to create annotation and verification loops in the workflow. Nonetheless, the usage of verification tasks increases the annotation costs, as many additional tasks are needed in order to complete a howto. Most of all, these verification tasks do not guarantee correct results, as they have failed to decline low-quality work, such that requesters had to intervene in the workflow [KCH12]. In conclusion, the implemented HIT design does not split the annotations by type, such that workers annotate every step of a howto and must know all rules for a correct labeling.

3.2 Quality Assurance

As one of the most important aspects of using MTurk is high data quality, a vast amount of quality assurance strategies is available. This subsection reviews common methods and discusses whether they are applicable for the chosen task design.

If an already labeled data set (gold standard data) exists, it can be used to verify the workers' answers [SF08, p. 2]. HITs then contain multiple samples from both the labeled and unlabeled data set. If a worker's performance is too low on samples of the labeled set, the submission is automatically rejected or at least not further considered [SF08, p. 2]. To utilize this strategy, it is required that there are numerous samples in the labeled set which are also indistinguishable from the unlabeled samples, as otherwise workers can recognize the verification samples and may not put as much effort in the other samples. Additionally, samples in the verification set must be unambiguous or else workers can get automatically

rejected for valid submission. These requirements make gold standard data infeasible to use in our HITs, as our annotations are ambiguous by nature. Furthermore, the verification samples must consist of whole howtos in order to be indistinguishable from unlabeled data, so that workers would need to annotate multiple howtos in one HIT, which results in an enormous increase in HIT completion time.

Similarly to the first method, HITs can include additional verifiable questions even if no gold standard data is available. These questions are additionally included in HITS and ask about subjects which are easily confirmable [KCS08, p. 455], e.g. their answers can be automatically computed. Workers who provide low-quality submissions likely fail these questions and their submission can be rejected or not further considered. However, such questions cannot be asked about the relations which should be annotated, as they are unknown or uncertain when the HIT gets posted. Properties of the howto which are certain when the HIT gets posted do not correspond to the annotation tasks main subject. Therefore, workers still can easily provide incorrect annotations while correctly answering questions about certain properties.

Another method to automatically verify work is to collect multiple submissions for a HIT and merge them into one answer. However, this method requires that multiple answers can be merged, which is not the case for most annotation tasks in our HIT. For example, there can be many possible AC assignments for an NL instruction which are different, but still correct, e.g. the ACs *"Adding"* and *"UsingASpiceJar"* for the previous example *"Season with salt and pepper."* Verification tasks work similarly by using multiple workers for an explicit review of answers. However, the use of verification tasks was shown to be unreliable, as workers accepted low-quality submissions which should have been rejected [NM10, p. 214][KCH11, p. 2058]. Additionally, they can result in additional costs and erroneous rejections.

As automatic verification methods are not applicable or have major drawbacks, we resort to manually reviewing the assignments. This approach gives us full control about rejections, but is only adequate if the time to review the assignments is only a fraction of the time to create them. We try to combat this problem by using the User Interface (UI) shown in Section 3.3 for reviews.

To reduce the amount of low-quality work to review, one can use additional methods which can improve the data quality. One method is to ensure that the workers read instructions carefully by embedding Attention Check Questions (ACQs) in HITs [PVA14, p. 1023]. The intention of an ACQ is that attention-paying workers read them and act in the correct way, while workers paying no attention fail them. ACQs can be questions for which there is only one sensible answer [PVA14, p. 1023], e.g. *"How often have you*

used Amazon Mechanical Turk in outer space during the last week?"[5]. ACQs can also be explicit instructions to the workers which contradict parts of the HIT, e.g. instructions to deliberately choose a wrong answer. However, ACQs do not ensure that workers submit correct annotations. Their usage is even contradictory to the HIT design, as workers are expected to *not* read the HIT instructions after they annotated a few steps and memorized the annotation rules.

Additionally, experiments show that ACQs are only useful if the workers have low qualifications [PVA14, p. 1030]. Highly qualified workers were not affected by them and submitted work which was superior to the work of lower-qualified workers. This indicates that requiring high qualifications in HITs may suffice to reduce the amount of low-quality work and we therefore set high requirements for our HITs. In particular, the workers must be based in the US, such that it is likely that they speak English fluently and can intuitively reason about the to be annotated NL instructions. Furthermore, we require that workers have already submitted a high number of assignments and 97% of these assignments must be approved. We first require that workers must have submitted 5000 assignments and lower this number by 2000 assignments each time no or only a few HITS are completed. If this number reaches 1000 assignments and no adequate number of HITS get completed, we decrease the percentage of approved HITs to 95%.

Furthermore, we use a qualification test to only allow workers who know the annotation rules to submit results. This test includes a video which explains the annotation rules and guides the workers through the HIT design shown in Section 3.3. Subsequently, they must answer six questions in which the annotation rules must be applied. The actual qualification test can be found in Appendix A. The video takes about ten minutes to watch and we expect the workers to spend additional five minutes for the questions. For compensation, workers are be paid 2$ as a bonus when they first submit an assignment and it gets approved[6]. This can increase the costs of annotations as in the worst case, every HIT costs additional 2$. However, it was shown that a majority of work is done by only a few workers [Ipe10, p. 21], so we expect the number of paid bonuses to be lower than the number of completed HITs. A drawback using high qualifications is that it greatly shrinks the available worker pool, so that it may take longer to annotate numerous howtos. However, as each assignment gets reviewed manually, we prefer fewer high-quality submissions over more submissions with lower quality, even if it takes longer to complete a howto batch.

[5]This ACQ assumes that no astronauts are using MTurk during their missions.

[6]We instruct in the beginning of the video that workers should ensure that enough HITs are available, so that they can still accept one HIT once they finished the qualification test.

3.3 Design of the Task's User Interface

The previous discussion concluded with a HIT design in which all steps of a howto are annotated in one task. In this subsection, the implementation of this design's UI is presented in greater detail.

The HIT starts with a modal dialog providing the workers with information about the context of the HIT, how to contact the requesters, a link to the instruction video of the qualification test and possible reasons why the HITs might get rejected. This dialog can be opened any time during the HIT by clicking on the *"Introduction"* link on the top bar of the HIT layout (see Mark C in Figure 6)

Once the worker accepts the dialog, the main view of the HIT appears, which layout is structured as follows (see Figure 6): The left side of the layout (see Mark A) consists of instructions as well as the current selections for the annotation tasks and is denoted as *selection view*. The right side (see Mark B) lists every step of the howto and is denoted as *howto view*. Only one step of the howto can be annotated at a time. It is marked as *"current step"* (see Mark F) in the howto view and only annotations for this step are displayed. The main annotation tasks are split by their type into three pages called *"Actions"*, *"Properties"* and *"Definitions"*, which are listed in the top bar of the layout (see Mark C).

The first page *"Actions"* is about assigning ACs and their respective roles. It is split up into subtasks, as many interdependent selections are necessary for these annotations. In the first subtask *"Select the first actions"*, the worker needs to select all ACs which are embedded in the current step. In order to reduce the amount of selections, some ACs are automatically selected, if their lemma[7] matches the lemma of at least one word in the current step. In our running example, this is the case for the AC *"Adding"*, as it matches with the word *"add"* in the instruction. Workers are instructed to ensure that automatically selected ACs are actions which should actually be executed in the current step, as otherwise this leads to incorrect training data. If a verb is written incorrectly, this automatic assignment is likely to fail, such that workers must use the search bar (see Mark D) to check whether there exists an AC with the correctly spelled action.

If an action verb occurs in the instruction for which no AC was assigned automatically, the worker must search through the checkbox list of ACs (see Mark E) to find an AC which is a synonym for this action. In our running example, the action verb *"Slice"* did not get automatically selected, such that the worker needs to find the *"Cutting"* AC manually. To generate more data for AC refinements, the worker is instructed to first search the generic ACs and prefer them over specific ACs. The selection of ACs can also be seen next to the

[7] A *lemma* denotes the base form of the words [Man+14, p. 57]. For example the lemma of *"Adding"* is *"add"*.

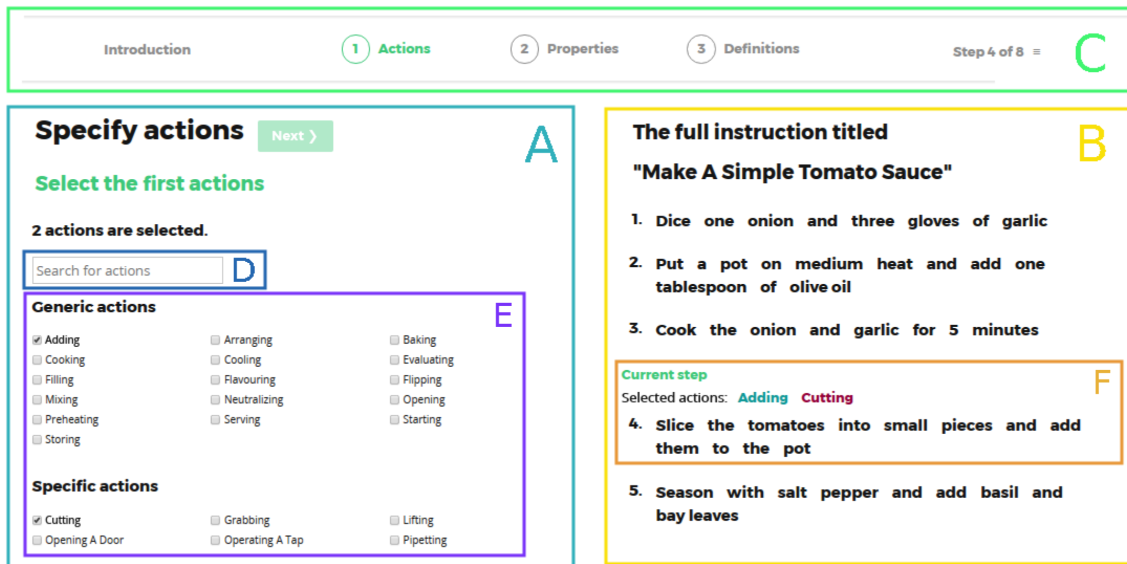

Figure 6: An excerpt of the first action subtask *"Select the first actions"* embedded in the main HIT layout. The worker sees annotation instructions on the left (A) and the to be annotated howto on the right (B). The top bar (C) provides information about the worker's progress in the HIT. In this subtask, the worker must select all ACs contained in the current step (F). ACs are displayed in a list of check boxes (E), which can be filtered by typing an AC's name in the search field (D). Selected ACs are listed over the currently edited step (F).

current step in the howto view (see Mark F). Every AC has a different color, such that the AC and its roles can be easily separable from other ACs in the howto view.

In the next *"Words with missing action items"* subtask, the worker is asked to select words in the current step which express actions, but for which no applicable AC could be found. This selection occurs by clicking on the corresponding words in the howto view. The selected words are then displayed in a simple list in the selection view. In our running example, an AC could be selected for every occurring action, so no selections need to be made. This subtask does not correspond to any training data, but is instead used to identify missing ACs.

Subsequently, the action page can progress in two different ways depending on the first AC selection. If no ACs were selected, no further annotations can be made for the current step. In this case, the worker is asked to confirm his AC selection and to continue with the next step of the howto.

Conversely, if the worker has selected ACs, the next action subtask *"Assign roles"* is displayed (see Figure 7). The worker is shown the respective action roles of the previously selected ACs and is instructed to assign them to words in the current or previous steps.

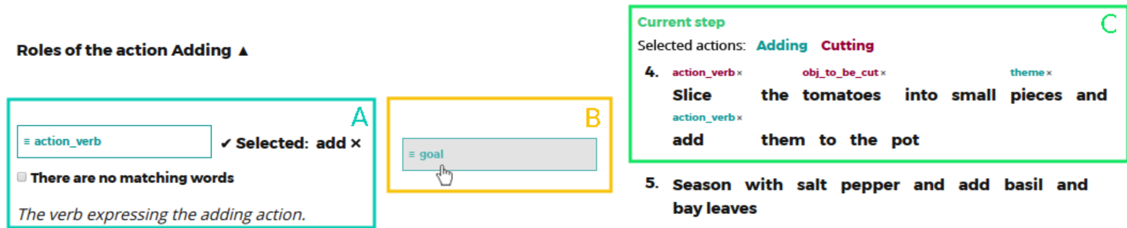

Figure 7: An excerpt of the action subtask *"Assign roles"*. In this subtask, workers can assign words to action roles by dragging the roles' boxes (A) and dropping them on words on the right. Currently, the *"goal"* action role is being dragged (B). When action roles are assigned, the corresponding words appear next to the roles' boxes and the roles are shown over their words (C).

Every role is listed as a box, colored in its AC's color, and its NL description in the selection view (see Mark A). To assign an action role, its box is dragged and dropped on an applicable word in the howto view (see Mark B). If there are no suitable words, the worker must instead tick a checkbox under its box, denoting that there are no matching words. The action role relations are displayed in two ways: Firstly, every word assigned to an action role is listed next to its box in the selection view (see Mark A). Secondly, the action roles are shown on top of their assigned words in the howto view (see Mark C). This way, the user gets a quick overview of the current state of the annotation. The action role titles in the howto view can also be dragged and dropped between words, so that a user can easily reassign action roles. The worker can only continue from this subtask if every action role is either assigned to at least one word or has a ticked checkbox. Furthermore, the worker is lead back to this subtask if an action role gets deselected in the following action subtask[8]. This makes sure that no action roles are overlooked or accidentally deleted by the worker, which can result in incomplete or inconsistent training data.

After all action roles have been assigned, the last action subtask *"Refine generic actions"* is displayed if the current step contains generic ACs. In this subtask, each selected generic AC is refined with a more specific AC and the respective roles of the refining AC are selected analogously to the previous subtask. As generic ACs can also be refined by other generic ACs, a chain of multiple refinements is possible. To navigate between the ACs in the refinement chain, they are displayed as a *"breadcrumbs"* menu in the selection view (see Mark A in Figure 8). For every refinement, the worker must choose one refining AC from a list of radio buttons in the selection view (see Mark B). If there is no applicable AC or multiple AC must be executed to accomplish the generic AC, the worker must instead

[8]The next subtask is the only part of the HIT in which the action roles could further change. In the *"Properties"* and *"Definition"* pages, the action roles are also displayed on top of their assigned words, but they cannot be changed by dragging and dropping.

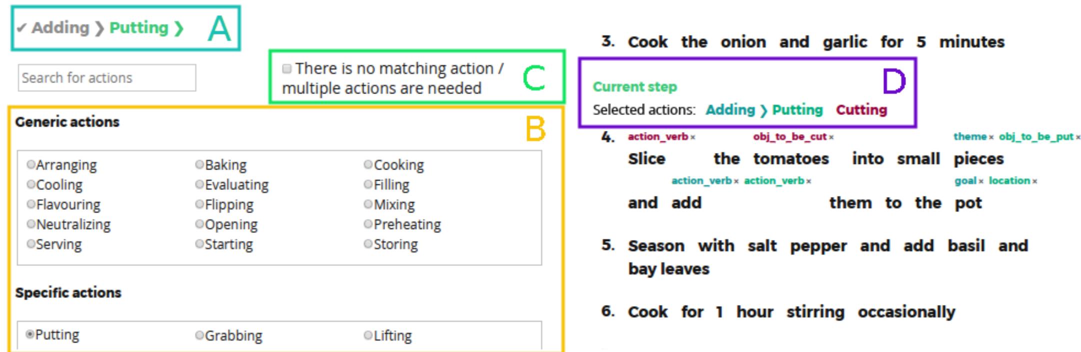

Figure 8: An excerpt of the action subtask *"Refine generic actions"*. A breadcrumb menu is used to navigate between multiple AC refinements in the selection view (A). Workers must refine a generic AC like *"Adding"* by selecting a more specific AC in a list of radio buttons (B) or toggle a checkbox if no AC applies (C). Selected refinement relations are marked with a ">" character in the howto view (D).

tick a checkbox specify to this (see Mark C). Refinement relations are also shown in the selected actions in the howto view and are denoted with a ">" character (see Mark D). Like the subtask for action role assignments, the worker can only progress if all refinement selections have been made for every selected generic AC.

When the worker has completed the last action subtask, the property page is displayed (see Figure 9). On this page, the worker is asked to select every property which references an action role word and has one of the four supported property types, which are colors, sizes, shapes and materials. To select a property, the user must drag the word denoting the property and drop it on the referenced word in the howto view. When the property is dropped, a popover appears in which one of the available property types must be chosen (see Mark A). Property assignments are displayed in a list in the selection view (see Mark B).

In the last annotation page named *"Definitions"*, the worker must select a word sense for every selected action role word (see Figure 10). The selection occurs with a drop down menu displaying definitions of WordNet synsets of the word filtered by its POS tag (see Mark B). Every menu has an additional item at the bottom which denotes that no provided synset applies for the word[9]. To reduce the number of selections made by the worker, word sense assignments are recommended for other words in the howto. Whenever a word sense is assigned, it is automatically selected for words in the howto, if they have the same lemma, the sense is applicable for them and they have no word sense selected yet. Using these recommendations, the worker has to assign the word sense to each action role word only

[9]This item is not displayed in the Figure 10, as it only shows a small excerpt of the annotation page.

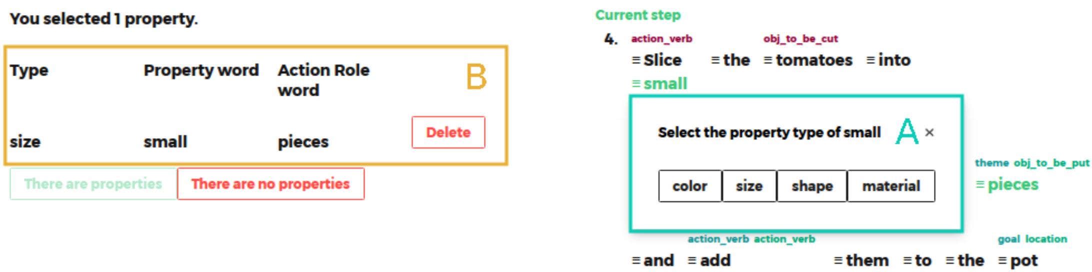

Figure 9: An excerpt of the properties assignment page. Properties can be selected by dragging the word expressing the property and dropping it on the referenced word. When the property word is dropped, a popover appears to select its property type (A). After a property relation has been selected, it appears in a list in the selection view (B).

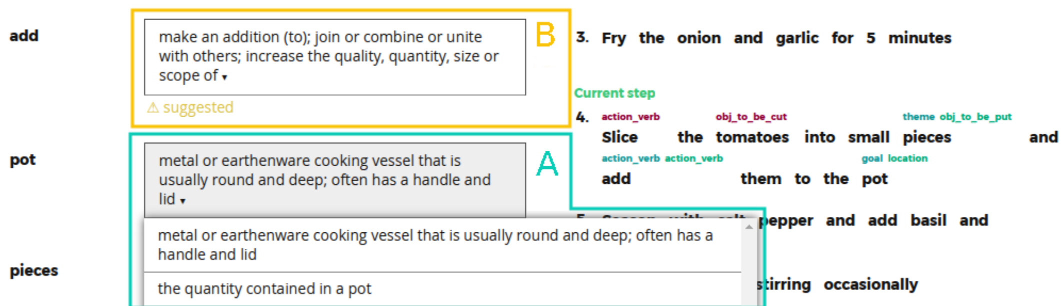

Figure 10: An excerpt of the word sense assignment page. For every word of the current step which is selected as an action role, the user is instructed to select a word sense using a drop down menu. Currently, the menu of the word *"pot"* is expanded (A). Word sense assignments which are automatically created are marked as *"suggested"* to the user (B).

once in the best case. However, as the meanings of words can still change during a howto, the workers are instructed to check recommended word sense assignments and correct them if they are not applicable. In order to signify automatically assigned word senses, they are marked with a yellow *"suggested"* text under to their drop down menu (see Mark B).

The tasks shown in this subsection get repeated for every step of the howto. If the howto was annotated in a MTurk HIT, the worker is asked for feedback on the final page. The text box for the feedback is embedded in the required HTML form which posts all gathered data encoded in JavaScript Object Notation (JSON) to MTurk. The same tasks are also used to annotate howtos on a separate website and to review posted assignments.

3.4 Architecture of the Web Application

The architecture of the web application is shaped by multiple circumstances: It needs to communicate with MTurk and should be capable of being integrated with the PRAC framework. Additionally, it needs access to WordNet for word senses as well as the Stanford Parser for grammatical dependencies and POS tags. As PRAC is written in Python and uses MongoDB as its knowledge base structure, the main application server uses the same technologies. The use of Python is also an adequate choice, as the library "Boto" [BOTO] gives access to the MTurk API and the library Natural-language Toolkit (NLTK) [NLTK] provides an interface to WordNet and the Stanford Parser. However, using the Stanford Parser directly from Python results in a significant overhead, as the Java Virtual Machine must be started every time a howto is parsed. To remove this overhead, the Stanford Parser is embedded in a server in Java which is accessed by the application server to parse howtos. Lastly, one important aspect for the architecture is that the HIT design requires advanced UI features. For example, various UI elements can be dragged and dropped into each other and the user can navigate between several pages without a page refresh. To make these features easily implementable, the UI is written as a standalone JavaScript application. The resulting architecture overview can be seen in Figure 11. Details about the architecture can be found in the documentation attached to this thesis.

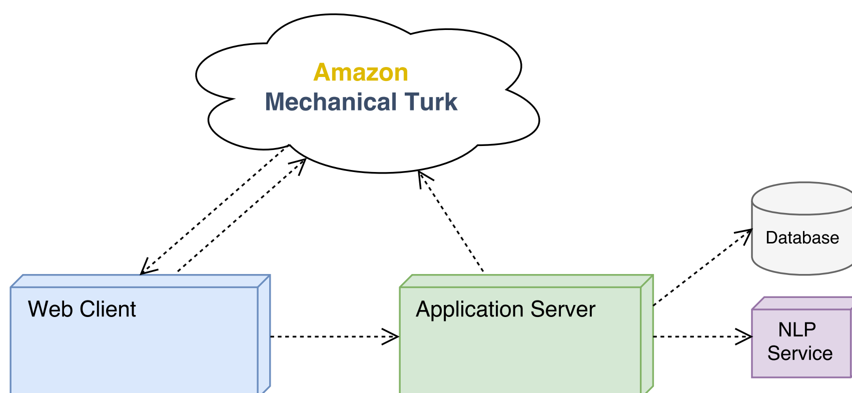

Figure 11: The architecture of the web application. The web client provides the UI of the HIT and is a standalone JavaScript application. The application server is written in Python and provides an API for the web client, performs database operations and communicates with MTurk. The NLP Service is a Java server which is used to efficiently parse howtos with the Stanford Parser.

4 Evaluation

This section provides the evaluation of using crowdsourcing to create training data for PRAC. Firstly, we describe the process of posting HITs to MTurk. Subsequently, the crowdsourced annotations are summarized and evaluated in greater detail.

4.1 Experiment Conduction

In the following subsection, we describe the used approach of posting HITs to MTurk. As source for unlabeled howtos, we use an existing corpus of 8786 manuals, which were mined by [NB12] from the "Food and Entertainment" section of the website wikiHow [WIKIHOW]. This data set is further reduced, as some manuals are unsuitable for our HITS: Firstly, the data set contains manuals which do not correspond to food preparation, e.g. *"Win a pie eating contest"*. Therefore, we only consider manuals if their title starts with a verb strongly indicating that it is a recipe, e.g. *"bake"* or *"cook"*. Secondly, the number of steps of some manuals is close to zero or greatly exceeds the median of 11 steps. We allow a range of 10 steps and keep the minimal number of steps per howto to five, such that we consider howtos with five to 15 steps for the evaluation. After filtering the initial data set, 4601 howtos are left as candidates for annotations.

We price the HITs with rewards which provide a fair wage to workers, but are not higher than needed in order to keep expenses low. Research on the influence of the reward on response rates of workers indicates that a higher wage has no significant influence on the data quality, but only results in a shorter completion time [MS12, pp. 9-10]. We therefore aim for an hourly wage of 8$, as this wage is greater than federal minimum wage in the US (7.25$) [USWAGE], but is less than the minimal amount paid for a dedicated worker in Germany (8.84€) [GERWAGE]. As the howtos have various numbers of steps, which results in various completion times, we use a linear regression model to estimate the time needed to annotate a howto (see Appendix B). For an initial estimate, the model is fit using 30 howtos annotated by the author, as no other completion time data exists before the first HIT batches are posted. To get an approximate of the difference in completion time on the users, a small number of college students were asked to annotate the same howto. On average, new users took twice as long to complete the howto as the author. Therefore, the estimated time gets multiplied by a factor of two until the model is refitted with data from MTurk.

As the HIT design described in Section 3.3 is untested for MTurk, we start pretests with smaller HIT batches, as this decreases the potential damage in reputation and money if unforeseen problems occur. For example, workers may have trouble with the HIT design or consistently provide incorrect answers. Furthermore, we use the logged completion times

from the pretests to retrain the linear regression model. When batches with more HITs are posted, the estimated completion times should reflect the actual time spent by MTurk workers. We use a lifetime of one day for each posted HIT, as newly posted HITS are more likely to be completed than older HITs. As previously described, we use four qualification requirements for each HIT, which are summarized in Table 5 for the posted batches.

Table 5: Used qualifications in the posted HIT batches.

Batches	**HITs submitted**	**HITs approved**	**Location**	**Qualification test**
1	5000	97%	US	5+ questions correct
2 - 6	3000	97%	US	5+ questions correct
7 - 8	1000	97%	US	5+ questions correct

This table lists the qualification requirements for the posted HIT batches. In all batches, 97% of the workers' HITs must be approved, the workers must be in the US and they must have answered at least five questions of the qualification test correctly. The number of HITs which must have been completed by workers was gradually lowered when the response rate was too low.

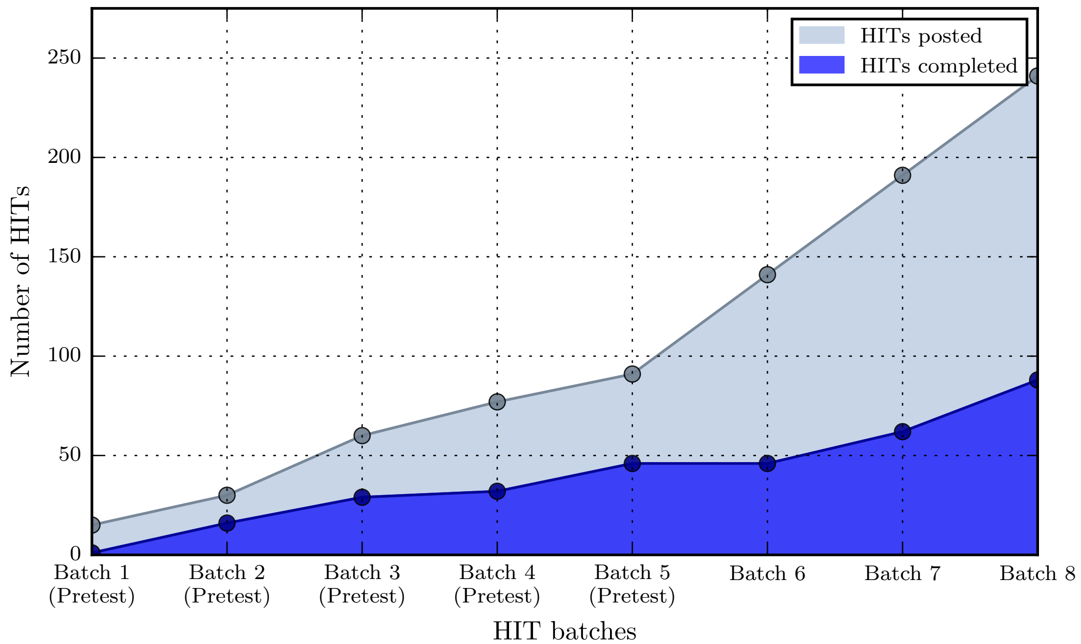

Figure 12: This figure shows the cumulative number of HITs which were posted to MTurk and completed by workers. The first five batches are pretests, which contain only 14 to 30 HITs each, while the last three HIT batches contain 50 HITs each.

The numbers of posted and completed HITS after every batch are shown in Figure 12. The initial batch consisted of 15 handpicked howtos for which only one worker posted an assignment. We therefore reduced the required number of posted HITs from 5000 to 3000 for the following batches. The second batch contained 14 of the previous howtos and one newly added howto. In contrast to the first batch, all 15 HITs were completed. Additionally, all posted assignments were of decent quality and therefore accepted. This batch also showed that workers were able to annotate more complicated instructions with multiple ACs, as the instruction shown in Figure 13. However, workers still made small mistakes in their HITs, making a manual review necessary. Furthermore, it is hard to justify rejecting HITs for a few mistakes, as the majority of their annotations are correct.

Selected actions: Removing Flipping ❯ Turning Putting

8. action_verb ×
After 50 minutes remove the
obj_to_be_removed × obj_to_be_flipped × obj_to_be_turned × obj_to_be_put × location ×
chicken from the oven
action_verb × action_verb × action_verb × location ×
flip it over and return it to the oven to finish roasting
breast-side up

Figure 13: A worker's annotation for a more complicated howto step from the second batch.

We doubled the amount of HITs in the next batch to evaluate whether all HITs of the batch would be submitted again. However, only 13 of the 30 HITS were completed. To obtain the last submissions to refit the linear regression model for larger HIT batches, the following batches contained the remaining howtos. Three of the 17 howtos were completed in the fourth batch and the last 14 HITs were submitted in a fifth batch by only one worker. Unfortunately, this worker misunderstood one of our annotation rules and made the same mistake consistently even after a warning email. We therefore rejected all of the worker's submitted HITs, but allowed the worker to correct the mistakes in additional reward-free HITS[10]. After the HITs were corrected, the originally rejected HITs were approved such that the worker was paid. This was also the only case in which we rejected HITs during our evaluation.

Following the pretests, the linear regression model was retrained with the annotation times of the MTurk workers and the previously used time estimation factor of two was removed. Additionally, the HIT batches were changed in two ways: Firstly, we raised the number of posted HITS for batch each to 50, as a possible reason for the low completion rate can be the low number of available HITs. Secondly, we determined the howtos for

[10]These HITs are not shown in not shown in Figure 12.

which the highest number of ACs was estimated[11] to maximize the annotations obtained from the HITs. In the first batch after the pretests, none of the 50 posted HITs were completed. We therefore lowered the required number of completed HITS to 1000 to increase the completion rate (see Table 5). In the next batch, 16 out of 50 HITs were completed. Unfortunately, we received negative feedback about underpayment. We therefore refitted the model with the completed HITS and increased the estimated time by a factor of 1.25 to prevent underpayment. In the last batch, 26 out of 50 howtos were annotated by the workers and we did not receive any negative feedback.

In total, 88 HITs were completed by only 20 workers. The number of HITs completed by each worker is shown in Figure 14. It can be seen that a minority of the workers submitted the most work. The five workers with the most assignments completed 65% of the total HITs. This corresponds to findings of related work [Ipe10, pp. 21] and is desirable, as this reduces the cost overhead of the qualification test.

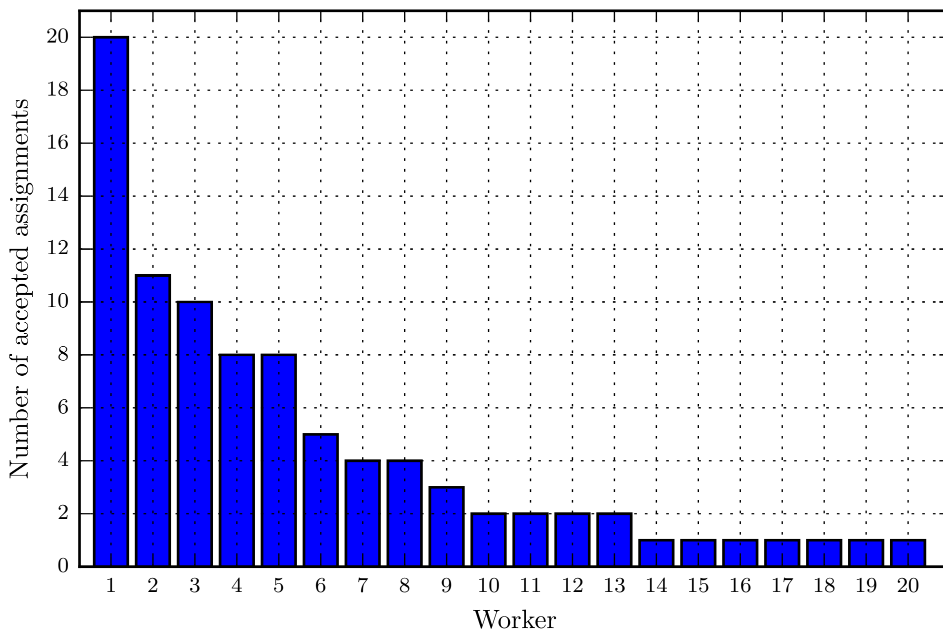

Figure 14: This figure shows the number of accepted HITs for each individual worker, sorted from most contributing to least contributing workers. One worker is excluded in this figure, as this worker was provided access to a special HIT with a short howto after a complaint.

[11]To approximate the number of ACs contained in a howto, we match its word's lemmas to the lemmas of ACs.

In contrast, the HIT completion rate is lower than reported in related work [Ipe10, pp. 20], as only 37% percent of the posted HITs were completed in total. This was expected, as workers can only accept our HITs with the right qualifications and an additional time investment to answer the qualification test. From the number of views of the qualification test video, it can be estimated that about 375 workers started the test. Out of these workers, 135 submitted an answer. The distribution of the number of correctly answered questions is displayed in Figure 15. Only 35% of the workers answered at least five questions correctly and were able to accept our HITs, such that the test can be one explanation for the lower completion rate. However, it can be argued that the test provides quality assurance for the cost of less completions, as without it, additional HITs would be submitted by workers who cannot apply our annotation rules well enough.

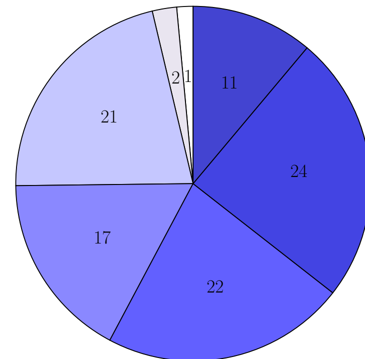

Figure 15: The percentages of the correctly answered questions in the qualification test, ordered from zero (lightest) to six (darkest) correct questions.

In total, the 88 posted HITS costed about 439$, from which about 48$ result from the qualification test. Using the logged completion times, the effective hourly wage of the workers is estimated as 9.86$ per hour, which exceeds intended hourly wage of 8$ per hour by about 23%. However, a higher effective wage is expected: The estimated time was doubled for pretest batches and was increased by a factor of 1.25 in the last HIT batch. As we received negative feedback about underpayment, estimated the effective hourly wage of individual workers with their time logs and HIT rewards. Figure 16 shows that workers who annotate more steps achieve a higher effective wage as workers who submit less. A possible explanation for this is that these workers get used to the HIT design and are therefore more efficient than newer workers. However, this poses a challenge for obtaining annotations in a cost-effective way: On the one hand, experienced workers are currently being paid far more than the intended wage. On the other hand, lowering the HIT rewards results in an unfair payment of new workers. A possible solution for this problem is lowering the overall HIT rewards, but paying new workers extra bonuses for their first annotated HITs, such that one can obtain cost-effective labeling from experienced workers while new workers are still paid fairly.

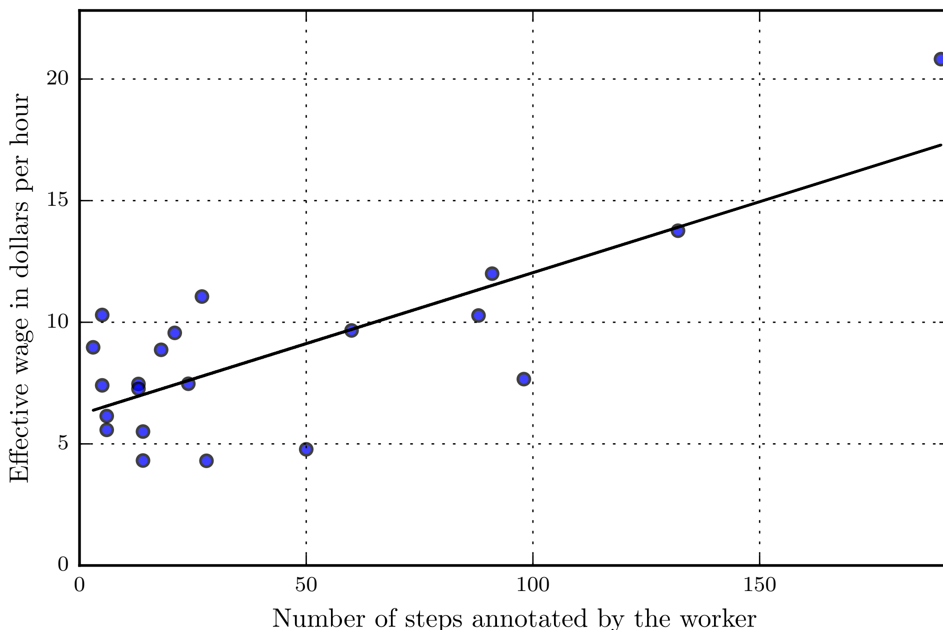

Figure 16: This figure shows the correspondence between the number of annotated steps to the effective hourly wage for each worker. The black line corresponds to the following linear regression: $r(s) = 0.058 \cdot s + 6.206$, where s is the number of annotated steps.

4.2 Evaluation of the Crowdsourced Annotations

The crowdsourced annotations are evaluated as follows: Firstly, they are characterized with descriptive statistics. Subsequently, we assess whether workers submit adequate annotations by reviewing one howto of each worker. Lastly, we examine whether the usage of the crowdsourced data achieves similar performance as expert data when it is used to train MLNs of PRAC modules.

4.2.1 Description of the Crowdsourced Annotations

The 88 completed HITs resulted in 87 annotated howtos. Unfortunately, one howto was annotated twice, as unlike the rest of its batch, it was ignored by the application server. The first annotation of this howto is not considered in the following evaluation, as it belongs to the only rejected HIT batch. Table 6 lists the descriptive statistics of the 87 howto annotations. In total, 1555 ACs, 4680 action roles, 248 properties and 2932 word senses were selected. On average, a howto step contains about 1.4 first ACs selections, 0.7 AC refinements, 5.2 action roles, 0.3 properties and 3.4 word sense selections.

Table 6: Descriptive statistics of the approved crowdsourced annotations.

Annotation type	**Total**	**Mean**	**Std. dev.**	**Min**	**Median**	**Max**
First AC selections	1261	1.40	0.93	0	1	6
Words with no ACs	292	0.32	0.59	0	0	5
Action role selections	4680	5.18	4.33	0	4	32
AC refinements	631	0.70	0.70	0	1	4
AC selected	294	0.33	0.54	0	0	3
None applies selected	337	0.37	0.56	0	0	3
Property selections	248	0.27	0.65	0	0	6
Word sense selections	3103	3.44	2.57	0	3	23
Sense selected	2932	3.25	2.48	0	3	23
None applies selected	292	0.32	0.59	0	0	5

This table lists descriptive statistics of the approved crowdsourced annotations. All decimal numbers are rounded to two decimal places. In total, 87 individual howtos with 903 instruction steps were annotated by MTurk workers. The descriptive statistics are computed using the annotation counts of single steps. Action role assignments in the AC refinement subtask are included in the *"Action role selection"* statistics. The annotation types *"AC refinement"* and *"Word sense selections"* are split up into two selection types, as for these annotations, workers must either select one applicable item or select that no item applies.

As ACs are the central pieces of PRAC, it is of interest which ACs were selected the most. Figure 17 shows the number of selections for each AC. It can be seen that the majority of AC selections concentrates on a few ACs. For example, the top five most selected ACs account for about 56% of all AC selections. In contrast, 26 of 40 ACs were selected less than 25 times in over 900 crowdsourced steps. This was expected, as [NB12] has already shown that the majority of actions contained in the used NL instructions concentrate on a few actions. Nonetheless, some ACs were almost never assigned, as the 10 least common ACs were only selected 17 times in total. This poses a challenge to crowdsource data for these ACs, as howtos containing them must be explicitly searched to generate enough training samples. A possible solution to increase the selections of such ACs is to redesign the AC refinement task, as many of the least selected ACs are highly specific, e.g. *"OperatingATap"*. Instead of showing all ACs as possible refinements in the subtask, one can show only a subset of applicable ACs depending on the to be refined AC. Furthermore, the shown ACs can be ordered from uncommon ACs like *"OperatingATap"* to more common ACs like *"Pouring"*, such that workers are more likely to select the uncommon ones.

Furthermore, workers selected a refining AC in only about 47% of the AC refinement selections. This percentage is much lower than in other subtasks in which workers can

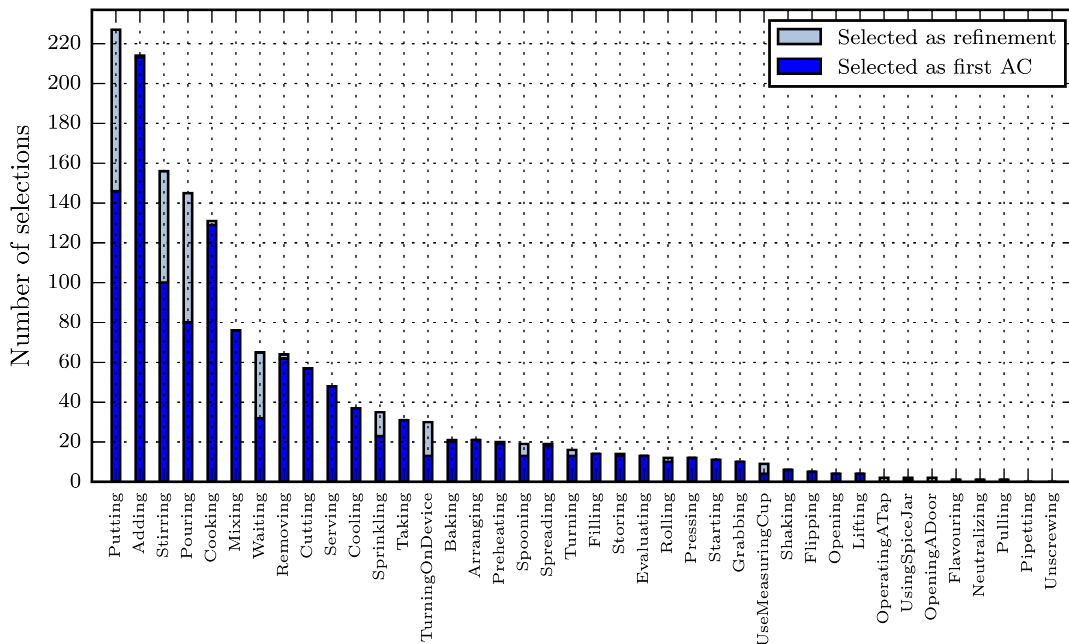

Figure 17: The number of selected ACs in the crowdsourced data, ordered from the most selected to the least select ones. The ACs *"TurningOnElectricalDevice"* and *"UsingAMeasuringCup"* are abbreviated as *"TurningOnDevice"* and *"UseMeasuringCup"*.

select that no option applies. For example, workers assigned word senses in about 94% of the word sense selections. A possible reason for this is that multiple actions are required to achieve a generic AC. However, this can also indicate that more additional ACs are needed. Figure 18 shows the percentages of AC refinement selections in which an AC was selected for each generic AC. It can be seen that the percentages of refinements range from 100% to 5%. The two ACs *"Flavouring"* and *"Neutralizing"*, which were refined in all selections, were also only selected once in all annotations. The *"Cooking"* AC stands out, as it is the fifth most selected AC, but is refined in only about 15% of its refinement selections. Furthermore, workers selected many actions related to cooking as words with missing ACs, for example the words *"melt"*, *"boil"* and *"fry"*. However, on one hand, a *"Cooking"* AC often denotes many consecutive actions, e.g. *"Cook some pasta."*, such that these actions should be represented as howtos in such cases. On the other hand, cooking can also denote a single action when the food and utensils are already in place, e.g. *"Cook the onions for two minutes."*. It is therefore debatable whether actions like *"frying"* should become ACs or should rather be represented with multiple steps. Besides words for cooking, the following words were under the most selected words with no ACs: *"drain"*, *"cover"*, *"peel"*, *"wash"*

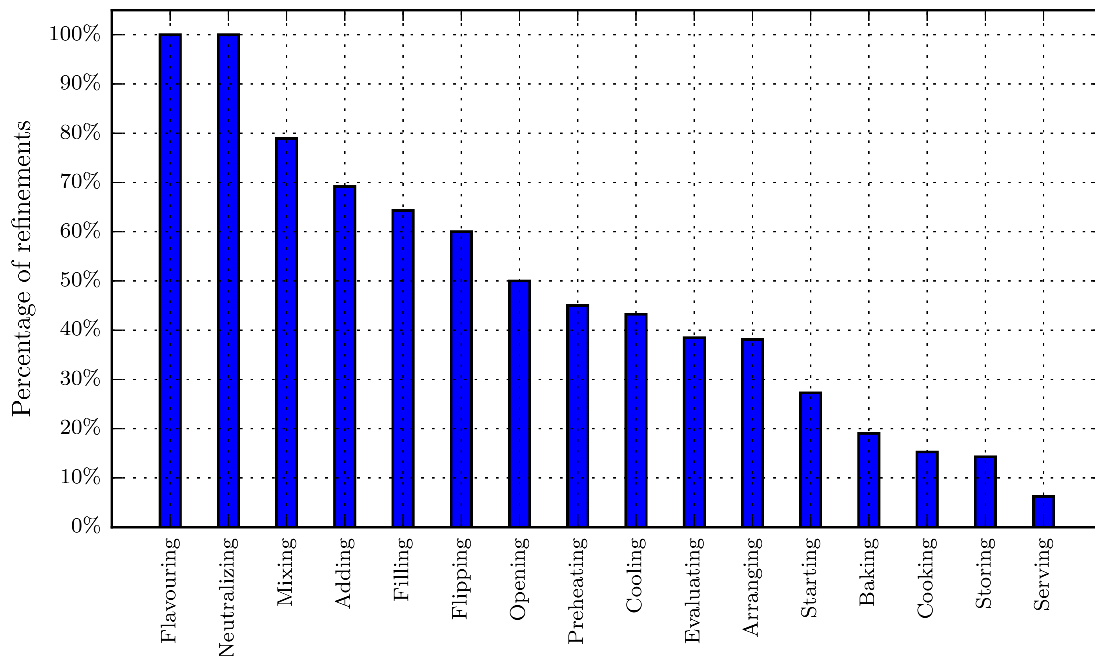

Figure 18: The percentages of AC refinement selections in which a refining AC was selected for each generic AC. The two ACs which were refined in all selections were selected only once by workers.

and *"reduce"*[12]. These actions can be incorporated as ACs in future versions of PRAC and the web application.

4.2.2 Annotation Quality of Individual Workers

The quality of the workers' submissions is of great interest: Firstly, the crowdsourced annotations are used as training data for MLNs, such that error-prone samples can result in low-performing MLNs. Furthermore, a high data quality in our HITs would be in favor of the assumption that MTurk can be used for long tasks requiring additional knowledge. We evaluate what kind of mistakes workers made in the HITs and how often these mistakes occur, such that a low occurrence of annotation mistakes indicates a high data quality. To generate statistics for the mistakes of individual workers, we sample one howto for each workers who submitted a howto in the pretests or larger HIT batches. Only one howto is sampled for each worker, as otherwise the mistakes and biases of the few workers who posted the majority of HITs mostly determine the results. The sampled howtos are reviewed manually for possible annotation mistakes workers can make (see Table 7). We

[12]In the sense of *"Reduce the heat to medium-low."*.

split annotation mistakes into more specific ones if possible, so one can find more precise reasons for why these mistakes were made. As an example, an incorrect AC selection can result from choosing a semantically incorrect AC or choosing a specific AC synonym while a generic AC synonym is applicable.

As NL instructions are ambiguous, more than one correct annotation can exist. Therefore, the workers' selections are only marked as incorrect if they are inconsistent with the HIT's rules or regarded as completely inapplicable by the reviewer. For example, this is the case if a worker selected an AC which is not applicable in any interpretation for the NL instruction, e.g. selecting the AC *"Preheating"* for *"Combine the ingredients."*. Furthermore, mistakes in previous subtasks are not propagated into following subtasks, as otherwise the statistics of these subtask do not reflect the corresponding worker selections. For example, if a worker missed an AC, the theoretical role assignments of that AC are not marked as missing in the action role selections of the same step.

Not all reviewed howto steps are used to compute the statistics for each type of mistake, as not every step contains possible selections for each subtask. Instead, for statistics of missing selections, only steps are considered which have at least one applicable item for the corresponding subtask. This is the case if the worker either made at least one correct selection of this type or at least one item is marked as missing in the review. For incorrect selections, only steps are used in which the worker selected at least one item in the corresponding subtask, regardless whether it is correct or not. The results are listed in Table 8. As nearly all HITs were accepted, the annotation quality of the workers is overall high: The median number of all types of mistakes in steps is zero and for the majority mistakes, a step contains maximally one to two mistakes of each type (see column *"Step max"* in Table 8). It is therefore of interest to evaluate in which subtasks workers made the most mistakes and to discuss possible reasons for this, such that the HIT design can be further improved.

In the first subtask, in which the first ACs must be selected, only about 6% of ACs were missed and about 10% of the workers' AC selections are incorrect. The majority of the incorrect selections result from unneeded selections, as workers kept recommended ACs selected even if they should not be executed. This is especially the case for the *"Starting"* AC, as it is often used in sentences without having to actually start an object, e.g. in the sentence *"As the milk starts to cool, add sugar and mix well."*. This is surprising, as this mistake was tested in the first question of the qualification test (see Appendix A). A possible approach to tackle this problem is to exclude this AC from the automatic selection and to include an extra warning about the meaning of the AC in the HIT instructions.

In contrast to the first subtask, workers made more incorrect selections in the second subtask *"Words with no AC"*. In this subtask, workers should select words which denote

actions but for which no applicable AC were previously found. Nearly 30% of the selections are incorrect and mostly result from the following mistakes: Firstly, actions were selected which cannot be executed by a robot like the words *"enjoy"* or *"taste"*. Secondly, some users understood that they should select all actions which did not get *automatically* assigned to ACs. Therefore, words like *"slice"* were selected, even though they were previously assigned to a *"Cutting"* AC. Since selections of this subtasks are used to find missing ACs, the high percentage of incorrect selections makes the examination more laborious, as many words must be manually evaluated to ensure that there were no applicable ACs for their sentence. A possible approach to tackle this problem is to update the HIT instructions of the subtask, as they currently do not emphasize non-executable actions and AC selections which were selected as synonyms.

The statistics of the *"Action role selection"* mistakes include role selections for the first ACs and also for refining ACs. The action role annotation quality is comparably high, as only about 2% of the action roles are missing and only about 3% of the workers' role selections are incorrect. In all of the 181 steps in which at least one role was selected, workers never assigned a role to a referencing word like *"it"* or *"them"*. Instead, all incorrect selections are semantic errors, for which two mistakes stood out: Firstly, some workers consistently missed applicable words in previous steps. Secondly, workers incorrectly assigned the *"amount"* role, which is described as the used amount of the *"unit"* role. For example, in the sentence *"Add two tablespoons of olive oil."*, the amount role is *"two"* and the unit is *"tablespoons"*. When there was no unit role in a step, workers selected ambiguous words like *"enough"* or *"to taste"* as amount, which greatly differs from the role's intended semantics. A possible way to prevent this mistake is to rename the action role to *"amount of unit"* on the web client.

The AC refinement subtask, in which workers must refine generic ACs to more specific ones, is the task with the most missed selections. 25% of the refinements are missing, but if ACs were selected for refinement, almost all selections were applicable. There can be multiple reasons for the high percentage of missed AC refinements: Firstly, workers can be inclined to select that no AC is applicable, so the roles of the refining AC do not need to be selected. Secondly, the AC refinement subtask greatly depends on how workers interpret the AC to be refined. For example, a *"Cooling"* AC in the instruction *"Set the cake aside to cool."* can be interpreted as only *putting* the cake aside, or *putting* the cake aside and also *waiting* for it to cool. In the first case, an AC refinement is applicable, while in the latter, no refinement should be selected, as multiple consecutive actions are needed. One possible way to tackle this problem is to make workers less inclined to select that there is no refining AC by making this option as time-consuming as selecting an AC refinement. For example, if an AC has no refinement, workers can be instructed to type short NL

Table 7: Possible types of mistakes workers can make while annotating howtos.

Type of mistake	Description
First AC selection	
Missing selections	An AC was not selected for an action explicitly mentioned in the instruction.
Incorrect selections	
Semantically incorrect	A selected AC is inapplicable for the corresponding action in the sentence.
Specific over generic	A specific AC was selected for an action as synonym, even though there exists a generic AC which is an applicable synonym for the action.
Unneeded action	An AC was selected for an action which is contained in the instruction, but should not be executed by the robot.
Words with no ACs	
Missing selection	A word which denotes an action with inapplicable ACs was not selected.
Incorrect selection	
Non-executable action	A word was selected which denotes an action which the robot cannot execute.
Action is applicable for AC	A word was selected for which an AC was selected in the previous task.
Action role selection	
Missing selection	An action role was not assigned to a word, even though an applicable word was in the instruction.
Incorrect selection	
Semantically incorrect	A role was assigned to a word which does not denote the role.
Referencing word	A referencing word like *"it"* was selected instead of the word it references.
AC refinement	
Missing selection	The user selected that there is no more specific AC for the action, while there is an applicable refining AC.
Incorrect selection	
Semantically incorrect	A refining AC was selected with which the generic AC cannot be refined.
Selection for consecutive actions	A refining AC was selected for an AC for which actually multiple consecutive ACs are needed.
Property selection	
Missing selections	A property for an action role word with an applicable type was not selected.
Incorrect selections	
Semantically incorrect	A property word was selected which does not denote a property.
Incorrect property type	The property type does not match the selected property.
Incorrect action role word	The selected action role word is not referenced by the selected property.
Word sense selection	
Missing selections	The user falsely selected that there is no applicable word sense for a word.
Incorrect selections	An inapplicable word sense was selected for a word.

This table lists various types of mistakes which users can make for every annotation subtask. Some mistakes are split up further to consider variations of them.

Table 8: Descriptive statistics of the workers' mistakes measured on sampled annotations.

Type of mistake	Percentage	Step mean	Step std. dev.	Step max
First AC selection				
Missing selections	5.82%	0.09	0.29	1
Incorrect selections	10.13%	0.17	0.41	2
Semantically incorrect	1.96%	0.03	0.18	1
Specific over generic	2.94%	0.05	0.22	1
Unneeded selection	5.23%	0.09	0.30	2
Words with no ACs				
Missing selections	3.77%	0.05	0.21	1
Incorrect selections	28.17%	0.34	0.54	2
Non-executable action selected	4.23%	0.05	0.22	1
Action is applicable for AC	23.94%	0.29	0.53	2
Action role selection				
Missing selections	2.49%	0.15	0.44	3
Incorrect selections	3.11%	0.19	0.72	8
Semantically incorrect	3.11%	0.19	0.72	8
Referencing word selected	0.00%	0.00	0.00	0
AC refinement				
Missing selections	25.00%	0.28	0.48	2
Incorrect selections	1.37%	0.02	0.12	1
Semantically incorrect	1.37%	0.02	0.12	1
Selection for consecutive actions	0.00%	0.00	0.00	0
Property selection				
Missing selections	4.65%	0.06	0.24	1
Incorrect selections	44.59%	0.70	1.35	6
Semantically incorrect	41.89%	0.66	1.29	6
Incorrect property type	1.35%	0.02	0.15	1
Incorrect action role word	1.35%	0.02	0.15	1
Word sense selection				
Missing selections	1.67%	0.06	0.24	1
Incorrect selections	1.07%	0.04	0.22	2

This table lists descriptive statistics for various mistakes which workers can make while annotating NL instructions. Some mistakes are split up to consider variations of them. For this evaluation, the mistakes in one howto annotation for each crowdsourced worker are examined. To compute statistics of missing selections, only steps with at least one selection object (e.g. ACs) are considered. Likewise, to compute statistics of incorrect selections, only steps in which a worker made at least one selection for the correspond subtask are used. All decimal numbers are rounded to two decimal places. Furthermore, the minimum and median number of mistakes in the steps are not listed, as all mistakes have a median of number zero per step.

instructions describing the more specific steps needed to accomplish the generic AC. These NL instructions can then be used as a new howto in the semantically indexed database.

Proportionally, the workers made the most mistakes in the properties task. Only about 5% of the properties are missing, but nearly 45% of the selected properties are incorrect. In the incorrect selections, users mostly selected words which do not denote properties as an adjective or adverb does.

Type	Property word	Action Role word	
size	2	Mix	Delete
size	cup	Mix	Delete
material	Coconut Milk	Mix	Delete
size	can	Mix	Delete
material	curry	Mix	Delete
material	paste	Mix	Delete

Current step

1. action_verb action_verb
≡ Mix ≡ 2 ≡ cup ≡ of
content content content
≡ Coconut Milk ≡ and ≡ can ≡ of
content content content
≡ curry ≡ paste ≡ in ≡ a ≡ pot
action_verb
≡ stir ≡ over ≡ medium ≡ heat ≡ until
≡ the ≡ mixture ≡ boils

Figure 19: This figure shows an extreme case of wrong property selections in the sampled howtos, as every selection semantically incorrect. Furthermore, the worker misunderstood that action role words always denote action verbs.

Figure 19 shows the extreme case of incorrect property annotations in which each property assignment is semantically incorrect. These mistakes can have many reasons: Firstly, users were shown examples for correct properties assignments in the qualification test and in the HIT instruction, but they were not shown examples for incorrect annotations. Secondly, workers were not given explicitly notice that most steps do not contain properties, as we expected that they continue this task if they do not find any properties as shown in the examples. Because of this, they might suspect that there should be various properties in the sentences, since most other subtasks have multiple selections per step, and therefore select items like compound nouns when no applicable adjectives are found. A possible way to improve the number of correct selections is to include examples of applicable and incorrect selections in the HIT instructions. Furthermore, the qualification test question of properties can be changed so it tests for this mistake.

Overall, workers performed best in the word sense task, as only about 2% of the word senses are missing or and only about 1% are incorrect. These results can have multiple reasons: Firstly, word sense disambiguation is natural to humans and does not require extra knowledge like other subtasks of the HIT. Secondly, word senses get recommended during the HIT, so workers are only required to actively select senses for each only one time word and therefore have less chances to make mistakes.

In summary, the data quality on most subtasks is high, such that it is feasible to use MTurk to crowdsource PRAC annotations. However, the subtasks *"Words with no ACs"*, *"AC refinement"* and *"Property selection"* show potential for improvement, as workers did more mistakes in them compared to other subtasks. Future iterations of the HIT design and the qualification test should therefore focus on these subtasks to improve the overall quality of the workers annotations.

4.2.3 Training MLNs with Crowdsourced Data

As the annotation quality of workers is overall high, it is of interest whether the crowdsourced data yields comparable or even better performance results than data created by experts. To evaluate this, the MLNs of various PRAC modules (see Section 2.1) are first retrained with data from crowdsourced annotations. For training, the template formulas of the original MLNs are used and only get modified if the crowdsourced data is incompatible with them.

After an MLN was trained with crowdsourced data, it is compared to the original MLN counterpart trained with expert data. However, a quantitative evaluation is challenging, as multiple inference results can be valid for a given NL instruction, such that checks for identical inferences can yield inadequate results. For instance, many ACs can be applicable for a sentence and many word senses can be appropriate for an action role word. We therefore evaluate the performance qualitatively using all example sentences listed on the PRAC website [PRACWEB]. These instructions yield adequate results with the currently trained models of PRAC, such that the evaluation criteria for the crowdsourced data is whether they also produce comparable results.

Only PRAC modules which are affected by training MLNs are considered, as there are modules which use database queries instead of MLNs. Only one module is changed in every evaluation, such that wrong inferences of previous modules do not influence the results of following ones. The crowdsourced data is not changed in any way, such that the produced training data can include annotation errors from the workers. We limit the number of training samples to 30 for every training set, as training with too many samples results excessive memory usage or inadequate training time. Moreover, using too many training samples results in an excessively large MLN model, which becomes less usable as its inferences take longer to complete. An exception to this is the AC recognition module, as we first allow one sample for each AC. All MLNs are trained with the exactly same optimization parameters and algorithms as the MLNs used for comparison.

AC Recognition The first evaluated module is used to infer ACs and only needs one trained MLN. The training data consists of one step for each AC for which the AC was selected under the *"First AC selection"* subtask at least once. The inference results of

this data set are shown in the column *"MTurk data (1x)"* of Table 9. The correct AC was recognized in 15 out of the 27 sentences. The wrong classifications can be attributed to semantically wrong annotations or unfulfilled requirements of the samples. For example, the *"Starting"* AC was often selected incorrectly by workers, such that the sampled step does not contain a correct usage of it. The sampled steps for *"Cutting"* and *"Pressing"* had an action verb for which no applicable word sense was selected, such that these samples were not embedded in the MLNs. The AC *"Neutralizing"* was selected only one time in over 900 crowdsourced steps and this selection is also semantically wrong. Therefore, this AC cannot be inferred using the crowdsourced data. However, this was expected, as a neutralizing action is not normally used in the kitchen domain, to which all crowdsourced howtos belong.

As one inapplicable step can lead to an incorrect inference for this MLN, it is of interest if its performance can be improved with more samples. We therefore train an MLN with two samples for each AC, resulting in a training set of 67 samples. The results are shown in the last column of Table 9. Using the larger training set, the ACs of 21 of 27 instructions were inferred correctly. The ACs which did not improve are *"Neutralizing"*, as it has only one sample, and *"Pressing"*, which was inferred as *"Cooking"*. The training set with two samples for each AC performs better than the first crowdsourced training set, but results in a large model with a slower inference runtime. For example, the MLN learned from the crowdsourced data set with two samples for each AC contains 5635 weighted formulas. In contrast, the MLN trained with expert data contains only 676 of such formulas and still provides superior performance.

Property Extraction This module infers properties contained in the NL instruction and uses one MLN. As the example sentences used by PRAC do not contain any properties, three additional sentences for the property types colors, sizes and shapes are used to assess this module[13]. For each property type, seven steps are sampled. Even though 28 samples are used, only one sample is embedded in the resulting MLN. This is caused by the used template formulas, which require specific grammatical dependencies between property and action role words. As the property assignments are the most incorrect in the crowdsourced data, most samples do not conform the formulas and are therefore not embedded in the resulting MLN. The result of this can be seen in the inference results in Table 10. The MLN trained with crowdsourced data performs considerably worse than the original MLN. While the original MLN can recognize colors and sizes, the MLN trained with MTurk data falsely recognizes sizes as colors. The reason for this is that the formulas containing color predicates have a considerably higher weight than other formulas in the learned MLNs, as the only

[13]PRAC currently does not query materials, such that no comparison for this property type is possible.

Table 9: Recognized AC by MLNs trained with expert and MTurk data.

NL instruction	**Expert data**	**MTurk data (1x)**	**MTurk data (2x)**
Add some cheese to the pizza.	Adding	Adding	Adding
Make an italian dinner.	Cooking	Cooking	Cooking
Slice the baguette.	Cutting	**Pouring**	Cutting
Fill a cup with water.	Filling	Filling	Filling
Flavour the tomato sauce with the oregano.	Flavouring	Flavouring	Flavouring
Neutralize 5 drops of hydrochloric acid.	Neutralizing	**Filling**	**Filling**
Open the fridge.	Opening	Opening	Opening
Preheat the oven.	Preheating	Preheating	Preheating
Press the start button.	Pressing	**Adding**	Cooking
Start the centrifuge.	Starting	**Adding**	Starting

This table lists recognized ACs by the PRAC framework which AC recognition module was trained with expert and crowdsourced. Only one sentence for each AC is displayed. For the full inference results, we refer to Appendix C.1. For this evaluation, two MLNs were trained with two crowdsourced data sets which contain one sample for each AC, denoted as (1x), and two samples for each AC, denoted as (2x). Wrong inference results are written in **bold**.

embedded sample contains this property. Furthermore, the MLN fails to recognize the color property in the sentence *"Pour the sauce over the brown meat."*. When the requirements of grammatical dependencies are removed from the formulas, unappropriated words like nouns get embedded in formulas of properties, as the crowdsourced property annotations often contain these words. A possible way to tackle this problem is therefore to filter howto steps which contain property selections, but also contain the required grammatical dependencies.

Action Role and Word Sense Inference This modules infers action roles and simultaneously assigns word senses to them. To keep the model size of the MLNs small, one MLN model is used for every AC which can be inferred by the AC recognition module. We therefore train an MLN for every AC contained in the example sentences with the exception of the AC *"Neutralizing"*. The MLN of this AC has, unlike many other MLNs, a hard constraint for the action roles, for which the only step with a *"Neutralizing"* AC is not applicable, as it is incorrectly annotated. Table 11 lists action role inference results which differ from the original MLNs.

The inference performance differs between the MLNs of various ACs. For the *"Adding"* AC, the roles of three out of five evaluated sentences were inferred correctly. The MLN

Table 10: Recognized properties by MLNs trained with expert and MTurk data.

NL instruction	Expert data	MTurk data
Sizes		
Slice the bread into **small** pieces.	size(pieces-6, small-5)	color(pieces-6, small-5)
Slice the **large** bread.	size(bread-4, large-3)	color(bread-4, large-3)
Fill a **large** pot with water.	size(pot-4, large-3)	color(pot-4, large-3)
Shapes		
Cut the sandwich into **triangular** pieces.	-	-
Press the dough into a **round** form.	-	-
Put the chicken on a **rectangular** tray.	-	-
Colors		
Press the **red** button.	color(button-4, red-3)	color(button-4, red-3)
Slice the **green** apples.	color(apples-4, green-3)	color(apples-4, green-3)
Pour the sauce over the **brown** meat	color(meat-7, brown-6)	-

This table lists the recognized properties by the property extraction module of the PRAC framework. The expert data column shows the inferred properties of the current MLN and MTurk data shows the MLN trained with sampled data from MTurk. The shown example sentences were additionally created to assess this module. The sentences are ordered by the type of their contained property which is written in **bold**.

trained with MTurk data made the same mistake for the other two sentences which involve adding quantities of objects. The module creates two *"Adding"* ACs instead of one, in which the unit is incorrectly assigned to the object to be added. In the example sentence *"Add 5 drops of the lysergic acid to the pyrimidine"*, the *"drops"* were incorrectly inferred as theme and were assigned the word sense *"cliff.n.01"*, which is described as *"steep high face of rock"*. However, 9 out of 30 samples in the training set contain a unit role, which are mostly assigned to the words *"cups"*, *"spoons"*, *"pieces"* and *"dashes"*. Changing the *"drops"* of the example sentence to *"teaspoons"* still results in two inferred ACs. The MLN of the *"Cutting"* AC shows the same errors, as the units and amounts of the example sentence *"Cut the pizza into 4 pieces."* are also not inferred correctly. However, the roles in the other three example sentences of the *"Cutting"* AC were inferred correctly, as they do not contain units and amounts.

All roles of *"Filling"* besides the action verb were inferred correctly. This role was consistently assigned to the word sense *"fill.v.09"*, which is described as *"plug with a substance"*, which vastly differs from the expected *"fill.v.01"* described as *"make full, also*

Table 11: Inferred action roles by MLNs trained with expert and MTurk data.

NL instruction	Expert data	MTurk data
Add 5 drops of the lysergic acid to the pyrimidine.	action_verb: add.v.01 theme: lysergic_acid.n.01 amount: five.n.01 unit: drop.n.02 goal: pyrimidine.n.01	**action_verb: add.v.01 (2x)** theme: lysergic_acid.n.01 **cliff.n.01** **amount: Unknown** **unit: Unknown** goal: pyrimidine.n.01
Cut the pizza into 4 pieces.	action_verb: cut.v.01 utensil: cutter.n.06, obj_to_be_cut: pizza.n.01 amount: four.n.01 unit: piece.n.08	action_verb: cut.v.01 **utensil: Unknown** obj_to_be_cut: pizza.n.01 **amount: Unknown** **unit: four.s.01**
Fill a cup with water.	action_verb: fill.v.01 stuff: water.n.06 goal: cup.n.01	**action_verb: fill.v.09** stuff: water.n.01 goal:cup.n.01
Flavour the tomato sauce with the oregano.	action_verb: season.v.01 goal: tomato_sauce.n.01 spice: marjoram.n.02	**action_verb: season.v.01 (2x)** **goal: tomato_sauce.n.01 (2x)** spice: marjoram.n.02 **tomato_sauce.n.01**
Preheat the oven.	action_verb: preheat.v.01 obj_to_be_heated: oven.n.01 temperature_setting: Unknown temperature_unit: Unknown	**Fails query**
Press the start button.	action_verb: press.v.01 obj_to_be_pressed: push_button.n.01 location: Unknown	action_verb: press.v.01 **obj_to_be_pressed: Unknown** location: Unknown
Put the solution onto the mixer.	action_verb: put.v.01 obj_to_be_put: solution.n.01 location: mixer.v.04	action_verb: put.v.02 obj_to_be_put: solution.n.01 **location: Unknown**
Start the centrifuge.	action_verb: begin.v.03 obj_to_be_started: centrifuge.n.01	**action_verb: get_down.v.07** **obj_to_be_started: Unknown**

This table lists the recognized action roles by the senses and roles module of the PRAC framework. The *"expert data"* column shows the inferred roles of the current MLNs and *"MTurk data"* shows the inferred results of the MLNs when trained with sampled crowdsourced data. Only sentences in which the action roles could not be correctly inferred are shown in this table. Additionally, only one sentence for each AC is shown if there were multiple failures for an AC. To get an example sentence which only contains roles of the *"Putting"* AC, the example sentence *"Put the solution onto the mixer and start it."* is reduced accordingly. For a table summarizing the full inference results, we refer to Appendix C.3.

in a metaphorical sense". Furthermore, this word sense is not applicable for instructions in which containers should be filled with liquids. Interestingly, only one sample of the MTurk training data contains *"fill.v.09"*, but the weight of its formula is higher than the weight of the formula with *"fill.v.01"*. Furthermore, *"fill.v.09"* is applicable in its sample sentence. This shows that a small derivations of the word senses of action verbs can lead to vastly different behavior in the MLN.

The *"Flavouring"* AC was selected by workers in only one step, such that the trained MLN is highly influenced by it. This MLN behaves similarly to the *"Adding"* MLN and does not correctly infer the roles in its example sentence *"Flavour the tomato sauce with the oregano"*. It creates two ACs, in which the tomato sauce and marjoram[14] are inferred as spices. Another problem besides having only a few training samples is that many instructions do not fulfill the grammatical requirements of the original template formulas. For example, the template of the *"Pressing"* AC requires that the word assigned to the *"location"* must have an *"nmod_on"* predicate[15]. However, none of the 12 samples for which this AC was selected contain this predicate, such that an MLN cannot be learned with the original template. However, if this requirement is removed, an MLN be learned and it correctly infers the roles of the example sentences.

The *"Preheating"* AC has only 18 applicable samples, however training its MLN demands too much memory or time when more than a few samples are used. Learning an MLN was only possible using three samples for which every action role was assigned to a word. However, the learned MLN does not contain any formulas for the action roles *"temperature_setting"* and *"temperature_unit"*, such that the query for the example sentence *"Preheat the oven"* fails. The reason for this is that the template formulas containing these roles require that both roles have applicable word senses, however this is not the case in any crowdsourced sample of this AC. There are two reasons for this: Firstly, the temperature settings are never assigned to a synset, as they are mostly numbers. Not every number has a synset in WordNet, and even if a synset exists for a number, e.g. *"one_hundred_eighty.s.01"* for 180, its POS tag is *"adjective satellite"*. In contrast, the Stanford Parser assigns a plural noun POS tag to the numbers in the sampled steps, such that their available synsets are filtered out. Secondly, the temperature settings rarely get an applicable word sense assigned, as they are mostly written in an abbreviated form, e.g. *"F"* instead of *"Fahrenheit"*. If the abbreviations are written with an ending dot or are at the end of a sentence, they get parsed to incorrect words, e.g. *"F."*, for which no synsets are available. A possible solution for these problems is using additional synsets for numbers and expanding abbreviated temperature units in a preprocessing step.

[14] The word sense *"marjoram.n.02"* was inferred for the word *"oregano"*.

[15] This predicate denotes an "on" dependency between nouns, e.g. *"Press the button on the mixer."*.

All action roles for the *"Starting"* AC were incorrectly inferred. This AC has many incorrect annotations, as it was assigned by workers in sentences in which no object should be started. As a result, the trained MLN assigns the word sense *"get_down.v.07"* to action verbs, which is described as *"take the first step or steps in carrying out an action"* which is not applicable for the example sentences. This AC is additionally evaluated for the *"Coreference Resolution"* module, as it is the only AC in which a coreference occurs in the example sentences. In contrast, the coreference resolution MLN for the *"Starting"* AC was able to resolve the coreference in the example sentence successfully.

AC Refinement This module refines generic ACs to more specific ones and one MLN is trained for the generic ACs which is embedded in the example sentences. However, the ACs *"Starting"* and *"Cooking"* are not evaluated: The sentence of the *"Cooking"* AC, *"Make italian dinner."* gets refined by PRAC into several substeps using the database instead of an MLN. Even if the corresponding MLN is trained and would perform worse than the original one, the decreased performance would not be reflected in the inference results, as the substeps can only be inferred with the database. Similarly, the example sentences of the *"Starting"* AC are also refined with the database. As the generic AC is not refined into multiple substeps, we first trained a *"Starting"* MLN with crowdsourced data and it yields the showed inference results as the original MLN. However, they can not originate from the training data, as the refining AC is not contained as refinement in the used data set. Lastly, as the *"Neutralizing"* AC has only one crowdsourced sample with an incorrect refinement, it is not evaluated.

Table 12: Recognized AC refinements by MLNs trained with expert and MTurk data.

NL instruction	**Expert data**	**MTurk data**
Add 5 drops of the lysergic acid to the pyrimidine.	Pipetting	**Pouring**
Fill a cup with water.	OperatingATap	**Putting**
Open the test tube.	Unscrewing	**OpeningADoor**
Flavour the tomato sauce with the oregano.	UsingASpiceJar	**Cooking**

This table lists the recognized AC refinements by the module of the PRAC framework. The expert data column shows the inferred refinements of the current MLNs while the MTurk data column shows results of the MLNs trained with data from MTurk. The example sentences of the *"Cooking"* and *"Starting"* ACs are refined by PRAC with a database instead of an MLN, such that the correct inference results are not determined by a correctly learned MLN. These sentences are therefore not included in the evaluation of this module. Furthermore, example sentences for *"Neutralizing"* are not included, as the only sentence with this AC has an incorrect refinement. For a table summarizing the full inference results, we refer to Appendix C.4.

The incorrect inference results are listed in Table 12. Overall, the ACs of 10 of the 14 evaluated sentences were refined to applicable ACs, however it is questionable whether some refinements are valid. Furthermore, the MLNs show monotone inference behavior: The MLN of the *"Adding"* AC always inferred *"Pouring"* as refinement for every example sentence, even though other ACs are contained in the training set. However, in almost all example sentences, the substances can be added by pouring them, such that this behavior is not negatively reflected in the overall inference results. Even in the example sentence *"Add some cheese to the pizza."*, one can add grated cheese by "pouring" it from a package. Nonetheless, for the example sentence *"Add 5 drops of the lysergic acid to the pyrimidine."*, this refinement is not applicable, as one must use a pipette to precisely add five drops of the acid. One possible reason for this monotone inference behavior is that more specific ACs like *"Pipetting"* are not included in the sampled steps and therefore cannot be inferred, as they were not as nearly as much selected as the former refinements (see Section 4.2.1).

The MLN of the *"Filling"* AC showed the best results for AC refinement, as only the sentence shown in Table 12 was refined incorrectly. For the other three sentences, the generic AC *"Adding"* was inferred as refinement. These refinements differ from the refinements of the original MLNs, but they are still valid, as filling a container can be achieved by adding the object into it. Lastly, the MLN of *"Opening"* always inferred *"OpeningADoor"* as refinement, as this is the only refining AC which was selected in the the training set, which consists of only two steps. Therefore, the expected *"Unscrewing"* AC for the example sentence *"Open the test tube."* can never be inferred.

In conclusion, the crowdsourced data achieves similar performance as PRAC for the AC recognition module, but has lower performance on most other modules. One reason for the superior performance of the MLN of the AC recognition module is that it corresponds to the first subtask in HITs, such that the training samples are not affected by previous wrong selections. Additionally, the template of the MLN is not restrictive, as it only contains one formula which requires that all actions denoting ACs are assigned to a word sense.

In contrast, the template formulas of the property extraction module are far more restricting, as they require specific grammatical dependencies between words, such that only one sample of the crowdsourced training set was used to build the MLN. Furthermore, the property extraction module suffers from noisy data, as about 45% of the property selections in the sampled howtos of the evaluation in Section 4.2.2 are incorrect.

There are multiple reasons why the MLNs of the action role and word sense module perform badly when trained with crowdsourced data. Firstly, this module is directly affected by wrong selections of previous subtasks: If a worker incorrectly selected an AC, then its action role assignments are most likely semantically wrong. This could be seen in the completely wrong inference results MLN of *"Starting"* AC, as this AC was mostly selected

incorrectly by workers. Secondly, the MLNs of this module are sensitive to the word senses selected by the workers. In the case of the *"Filling"* AC, only one differing word sense assignment for the action verb in the training data resulted in a model which infers this word sense for every example sentence. Thirdly, the evaluated performance of the models greatly depends on the word senses which are in the training data and the evaluation sentences. If the training data does not contain word senses similar to the senses in the evaluation sentences, then the MLN will likely yield inadequate results. Lastly, the MLNs fail inferences with more complex sentences. For instance, the *"amount"* and *"units"* roles were never inferred correctly. There are two possible reasons for this: Firstly, many NL instructions do not contain these roles and secondly, workers often assigned the amount role incorrectly (see Section 4.2.2).

When MLNs of the AC refinement module are trained with crowdsourced data, they tend show repetitive inference behavior. This can be attributed to the distribution of selected ACs in the crowdsourced data, as a minority of ACs have the majority of ACs selections. Therefore, the training samples consists of the more common ACs, such that the inference of uncommon ACs is impossible, as these ACs were never selected, or become less likely. This problem does not exist in the expert data, as samples with the more uncommon ACs can be simply manually added to the training set.

The results show that using crowdsourced data for training MLNs of PRAC modules requires additional effort: Firstly, either only samples conforming the template formulas can be used for training or the template formulas must be changed to include more complex sentences. Secondly, some ACs only get selected a few times, even if more than 900 instructions were annotated. To generate training samples for these ACs, practitioners need to submit additional HITs containing howtos with specifically these ACs. Furthermore, some ACs like *"Neutralizing"* and *"Pipetting"* almost never occur in the domain of the posted howtos, such that howtos of new domains, like the chemistry domain, must be used to get annotations with these ACs. Thirdly, even if the data quality is high, mistakes in the workers' annotations need to be corrected during the review, as they were shown to lower the inference performance when used to train MLNs. Lastly, practitioners must carefully choose the steps for the training data, as the number of usable samples is restricted due to training limitations. The used steps should have a diverse set of concepts, as otherwise no adequate inferences can be made on steps which strongly differ from the training samples.

5 Conclusions and Future Work

This Bachelor's Thesis evaluated whether crowdsourcing can be an adequate solution to generate training data for PRAC. To examine this question, we used the crowdsourcing platform MTurk to annotate over 900 NL instructions. We chose a task design in which the workers must annotate every relation of the PRAC framework in a sequence of NL instructions. To ensure high data quality, we required high worker qualifications and used a custom qualification test such that only workers who know the annotation rules can accept our HITs.

We posted 241 HITs in multiple batches, from which only 88 were completed. This results in a completion rate of about 37%, such that the used approach is not adequate if it is required that all annotations must be gathered in a short amount of time. Nonetheless, this outcome shows that a few workers on crowdsourcing platforms are willing to invest 15 minutes for a qualification test and accept annotation tasks which are longer and more complex than typical micro-tasks. Therefore other SRL applications can use crowdsourcing to obtain training data if their task is similar to the one used in this thesis. However, it is unclear whether workers are also willing to make a larger time investment before a HIT, e.g. 30 minutes, if labeling tasks require more knowledge.

Even though we posted long HITs with interdependent annotations, a detailed review of a sample of the crowdsourced data showed that workers made just a few mistakes which are mostly concentrated in a few subtasks. This is evidence for the assumption that one can crowdsource high-quality data with the use of high qualifications and a custom qualification test, even when the tasks are more complex. One possible reason for this is that workers who did not know our annotation rules were filtered out by our qualification test. However, we have never posted a HIT which did not require this test, such that we could not directly measure the influence the qualification test on the data quality.

To investigate whether the gathered labels can be used as training data, we trained MLNs of various PRAC modules with the crowdsourced data and compared their performance with their original MLN counterpart. Only the MLN of one module showed comparable results to the original model with the cost of a larger model size. MLNs of other modules performed worse, as their samples did not conform the required grammatical structures, contained mistakes from workers or did not embed all possible inference results as labels. Therefore, expert data is superior to a random sample of crowdsourced data for training MLNs from PRAC.

This provides opportunities for future work: Firstly, the template formulas of PRAC modules often require specific grammatical dependencies, which not every used sample does fulfill. However, this indicates that the formulas need to be changed, as the used samples

originate from everyday NL instructions and otherwise no adequate inference results can be achieved for such instructions.

Secondly, MLNs of the AC refinement module showed monotone inference behavior since many specific ACs were only rarely selected. The UI of the AC refinement task can be changed such that workers are more likely to select such ACs, e.g. by showing only a selection of ACs which are sorted from the least to he most common AC.

Thirdly, many other improvements on the UI and qualification test can be made to raise the overall data quality. As many selections of the AC refinement module are missing, one can first evaluate the effectiveness of the previously mentioned improvement of this task. If there are still many missing refinement selections, one can make the selection of no adequate refinements more laborious by requiring workers to write short NL instructions describing how an AC is achieved. Furthermore, workers made more incorrect selections in the subtasks *"Words with missing AC"* and *"Properties"*. To improve the quality of the assignments of these subtasks, one can change the instructions of their subtasks, such that they include examples of correct and incorrect selections. Additionally, the qualification test can be modified such that it specifically tests for the most common mistakes made in these subtasks.

Furthermore, we experienced several pain points in our approach of using MTurk: Firstly, more experienced workers of our HITs earned considerably more than the intended wage while some newer workers earned less than this wage. A possible solution to this problem is lowering the overall HIT reward and paying extra bonuses to new workers as compensation. Another pain point is that the workers' assignments needed to be manually reviewed by requesters. This showed to be time-consuming, as we were required to skim through all workers' annotations. Because of this, only about one fourth of the available budget was used in this Bachelor's Thesis. This problem can be tackled by changing the HIT design for reviews, such that all annotations can be edited in one page instead of three.

In conclusion, this Bachelor's Thesis showed that it is possible to use crowdsourcing to obtain high-quality data labels for PRAC. However, in the case of PRAC, the use of this data requires additional effort such that it yields high performing models. The crowdsourced data gathered by this Bachelor's Thesis can be a basis for further investigations on the design of the template formulas for MLNs. However, even if no adequate MLNs can be trained with crowdsourced data, the use of MTurk to generate labels for PRAC is not unreasonable, as PRAC needs an abundance of annotated NL instructions for its database. It is therefore worthwhile to improve the HIT design, the qualification test as well as the procedure of posting and reviewing HITs, such that practitioners can obtain high-quality labels more efficiently.

A The Qualification Test of the Implemented Task Design

To ensure that workers who work on our HITs can apply the annotation rules of PRAC, a qualification test was created. This test includes a video which explains the worker the annotation rules and guides them through the UI shown in Section 3.3. The workers must answer five out of six multiple choice questions about this video correctly to get access to our HITs. Three of these questions are in the form of exercises which can be solved by applied PRAC's annotation rules. The other questions directly ask about some rules of the annotation tasks. The video takes about 10 minutes to watch and we calculate five minutes for all questions. Workers are paid for this qualification test with a 2$ bonus when they submit a HIT and we approve it. The rest of this section consists of a direct excerpt[16] of the qualification test shown to workers. Lists with a ☐ symbol are check boxes from which the worker must choose one answer for each questions. The correct answers of each question are in **bold**.

PRAC Data Qualification Test

This is a qualification test for HITs in which you must annotate instructions like recipes or chemical experiments.

Please follow these steps:

- View the following instruction video: `https://vimeo.com/206304499`
- Answer all multiple choice questions

You must answer 5 from 6 questions correctly in order to accept our HITs. In order to find more HITs from us, search for HITs with the keyword "prac data".

This question is about the first action subtask "Selecting the first actions"

If some actions in the sentence did not automatically get an appropriate action item and are correctly spelled, you must first check if there are synonyms in the specific actions, then in the generic actions.

- ☐ True
- ☐ **False**

[16] Whenever two lists of bullet points followed in the original HIT, the phrase *"The following options are available"* was additionally included for readability.

This question is about the first action subtask "Selecting the first actions"
Which action items would you select (or leave selected) for the following sentence?
"When the water starts to boil, add the rest of the ingredients and stirr."
The "Adding" and "Starting" actions have been automatically selected, but may be deselected by you. (The starting action means that the robot should start something.)
Also note that the last word has a typo.
The following actions can be selected:

- Generic actions: Adding, Mixing, Starting
- Specific actions: Pouring, Stirring

The following options are available:

☐ Adding, Stirring, Starting

☐ **Adding, Stirring**

☐ Adding, Mixing, Starting

☐ Adding, Mixing

This question is about the action subtask "Assigning roles of actions"
You are in the second step of a recipe and selected the actions "Adding" and "Shaking". The two steps of the recipe are:

1. Fill a shaker halfway with ice.
2. Add 200 ml of orange juice, 100ml of peach nectar and shake it well.

Which words do you select for the following action roles?

- [goal] - a role of Adding - Where the objects should be added to.
- [obj_to_be_shaken] - a role of Shaking - The object which should be shaken.

The following options are available:

☐ [goal]: No word applies / [obj_to_be_shaken]: it

☐ [goal]: shaker / [obj_to_be_shaken]: No word applies

☐ [goal]: shaker / [obj_to_be_shaken]: it

☐ **[goal]: shaker / [obj_to_be_shaken]: shaker**

☐ [goal]: No word applies / [obj_to_be_shaken]: shaker

This question is about the action subtask "Refining actions"
You selected the generic actions "Adding" and "Cooking" for the sentence:
"Add onions and olive oil to a pan and cook on low heat for 10 minutes."
How would you refine the "Adding" and "Cooking" actions? Note that in this step during the recipe, there is currently no pan on the stove and the stove is turned off. Only use the following action items:

- Generic actions: Adding, Cooking
- Specific actions: Putting, Preheating, Waiting

The following options are available:

☐ "Putting" refines "Adding", and "Cooking" is refined by "Preheating", "Putting", and "Waiting"

☐ "Putting" refines "Adding", and "Cooking" is refined by "Preheating"

☐ **"Putting" refines "Adding", and there are no applicable refinements for "Cooking"**

☐ There are no applicable refinements for both actions

This question is about the task "Selecting properties"
You must select every property of action role words, but only if the property is a color, shape, size or material.

☐ **True**

☐ False

This question is about the task "Selecting definitions"
If you have assigned a definition to a certain word in the previous step, this definition will be recommended if the same word appears again. If you change the recommended definition for the new occurrence, the definition of the previous word also changes.

☐ True

☐ **False**

B Estimating the Task Completion Time

A major component of designing HITs is setting a fair reward for the workers. However, using a fixed reward for every assignment is inadequate for the chosen HIT design: Since the number of steps of howtos differ, they result in different completion times. Additionally, single steps also show a great variance in completion times as seen in Figure 20. To obtain more detailed data about annotation times of our users, the web client tracks the total active time of the whole HIT, the time spent on the introduction modal as well as the active time of every single page for every step. These logs are posted to MTurk along with the remaining assignment data.

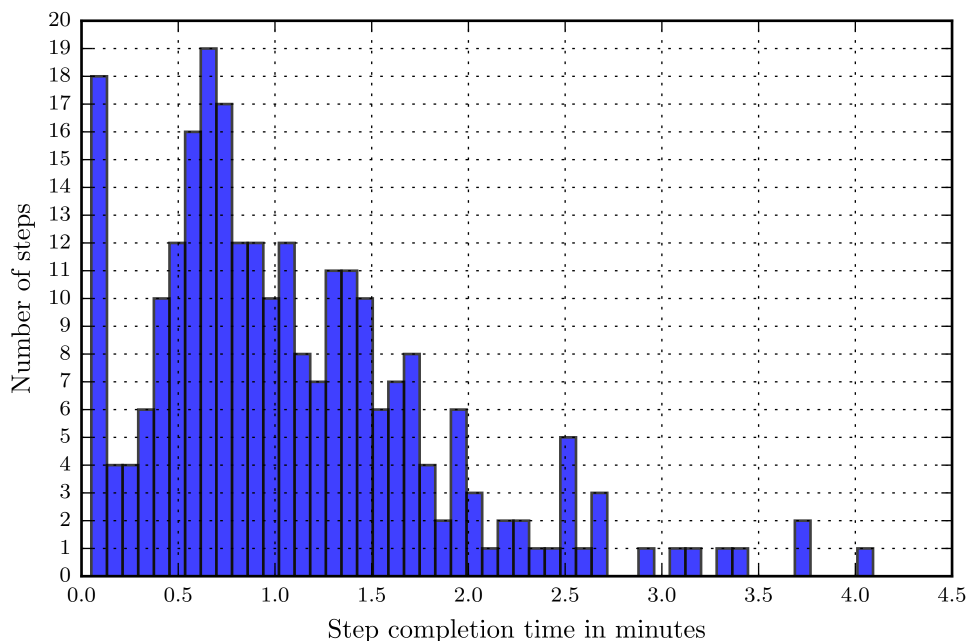

Figure 20: Histogram of the total completion times of the single steps of the howtos annotated by the author.

With the more detailed time logs, estimating the HIT completion time reduces to training a regression model which predicts the total step annotation time in milliseconds given the step and its howto. To estimate the completion time of a single step, one could only use the number of its words as explaining variable. However, various word types increase the completion time differently. For example, verbs like *"add"* indicate ACs and should therefore increase completion time to a greater extend than common words like *"the"*. To estimate the complexity of the instruction regarding the annotation task, we assign its words the categories listed in Table 13. We use the word counts of these categories

Table 13: Initial features used to estimate the task completion time for a single step.

Feature	**Mean**	**Std. dev.**	**25th PCTL**	**Median**	**75th PCTL**
1. Generic AC	0.40	0.59	0.00	0.00	1.00
2. Specific AC	0.40	0.60	0.00	0.00	1.00
3. Other verbs	1.27	1.30	0.00	1.00	2.00
4. Nouns	2.90	1.95	2.00	2.00	4.00
5. Adjectives	0.71	0.95	0.00	0.00	1.00
6. Conjunctions	0.42	0.59	0.00	0.00	1.00
7. Other words	4.54	3.22	2.00	4.00	6.00
8. Rel. pos.	0.44	0.29	0.20	0.44	0.67

This table lists the features which were initially used for the linear regression in conjunction with their descriptive statistics for the sampled howtos. The first seven features count specific word occurrences in the sentences. If multiple of these features apply for a word, the feature which is listed first in this table is chosen. The first two features count the words which lemma matches with a lemma of a generic or specific AC. For the conjunction category, only the words *"and"* and *"then"* are considered, as other conjunctions like *"or"* usually do not denote the execution of multiple actions in NL instructions. Furthermore, the last feature denotes the relative position of the step in the howto and is defined as the step's index divided by the number of steps in its howto.

as features of a linear regression model, such that words in categories which largely impact the annotation time obtain a higher weight. The estimed time $\hat{t}_s$ of a step s is computed as follows:

$$\hat{t}_s = \boldsymbol{w}^T \boldsymbol{x_s} + b \tag{1}$$

$\boldsymbol{x}_s \in \mathbb{R}^m$ is the feature vector of instruction s, $\boldsymbol{w} \in \mathbb{R}^m$ is the vector of coefficients, and $b \in \mathbb{R}$ is the intercept. The vector $\boldsymbol{x}_s$ initially contains the features listed in Table 13. However, if a feature becomes statistically insignificant when applied to the time data of MTurk users, it will be removed during the evaluation. The estimated completion time $\hat{t}_h$ of a howto h results from the sum of predicted step times and the scalar $\hat{t}_{modal}$, which approximates the time to read the initial introduction modal shown in the beginning of HITs:

$$\hat{t}_h = \hat{t}_{modal} + \sum_{s \in h} \hat{t}_s = \hat{t}_{modal} + \sum_{s \in h} \left(\boldsymbol{w}^T \boldsymbol{x_s} + b\right) \tag{2}$$

C Complete Inference Results of MLNs Trained With Expert and Crowdsourced Data

In Section 4.2.3, the performance of MLNs trained with crowdsourced data was compared to the performance of PRAC's original MLNs, which are trained with expert data. This subsection mostly showed an excerpt of the inference results for which the newly trained MLN made mistakes. In this appendix, the complete inference results of the evaluation are listed. For a more detailed description and interpretation of the evaluation, we refer to Section 4.2.3.

C.1 Action Core Recognition

This module infers ACs in NL instructions given their grammatical dependencies and POS tags. The inferred ACs for the example sentences are listed in Table 14. To evaluate this module, two training sets of crowdsourced data were used. The first training set contains one sample for each AC and is denoted as *"MTurk data (1x)"*. The second set contains two samples for each AC and is therefore denoted as *"MTurk data (2x)"*.

Table 14: Complete inference results of PRAC's AC recognition module.

NL instruction	Expert data	MTurk data (1x)	MTurk data (2x)
Adding			
Add some water to the purine.	Adding	Adding	Adding
Add some arsenic acid to the imidazole.	Adding	Adding	Adding
Add 5 drops of the lysergic acid to the pyrimidine.	Adding	Adding	Adding
Add 1 liter of water to the chlorous acid.	Adding	Adding	Adding
Add some cheese to the pizza.	Adding	Adding	Adding
Cooking			
Make an italian dinner.	Cooking	Cooking	Cooking
Cutting			
Slice the baguette.	Cutting	**Pouring**	Cutting
Slice the italian bread.	Cutting	**Pouring**	Cutting
Slice the bread.	Cutting	**Pouring**	Cutting
Cut the pizza into 4 pieces	Cutting	**Neutralizing**	Cutting
Filling			
Fill a cup with water.	Filling	Filling	Filling
Fill a blender with apples.	Filling	Filling	Filling
Fill a mixer with pineapples.	Filling	Filling	Filling
Fill a glass with wine.	Filling	Filling	Filling
Flavouring			
Flavour the tomato sauce with the oregano.	Flavouring	Flavouring	Flavouring

Complete inference results of PRAC's AC recognition module.

NL instruction	Expert data	MTurk data (1x)	MTurk data (2x)
Neutralizing			
Neutralize the methacrylic acid with 100 milliliters of cyanuramide.	Neutralizing	**Filling**	**Filling**
Start with neutralizing the pytworidine with 4 drops of hydrofluoric acid.	Neutralizing	**Filling**	**Filling**
Neutralize 5 drops of hydrochloric acid.	Neutralizing	**Filling**	**Filling**
Neutralize 100 ml of hydrochloric acid.	Neutralizing	**Filling**	**Filling**
Neutralize 100 ml of acid.	Neutralizing	**Filling**	**Filling**
Opening			
Open the fridge.	Opening	Opening	Opening
Open the test tube.	Opening	Opening	Opening
Open the cupboard.	Opening	Opening	Opening
Preheating			
Preheat the oven.	Preheating	Preheating	Preheating
Pressing			
Press the start button.	Pressing	**Adding**	**Cooking**
Starting			
Start the centrifuge.	Starting	**Adding**	Starting
Put the solution onto the mixer and start it.	Putting, Starting	Putting, **Adding**	Putting, Starting

This table lists the inference results of the AC recognition module of the PRAC framework. The inference results of the module with the original MLN are written in the *"Expert data"* column. To evaluate this data from MTurk for this module, two MLNs were trained with two crowdsourced data sets which contain one sample for each AC, denoted as (1x), and two samples for each AC, denoted as (2x). Wrong inference results are written in **bold**.

C.2 Property Extraction

This module recognizes properties in NL instructions which are colors, sizes, shapes and materials. PRAC currently does not query for materials in this module, therefore only the former three types are evaluated. Furthermore, the example sentences used for other modules are not applicable to evaluate this module, as they do not contain any properties, such that three new sentences for each evaluated property type are used. The training set contains seven samples for each property type. Table 15 lists the complete inference results for properties of MLNs trained with expert data and crowdsourced data[17].

Table 15: Complete inference results of PRAC's property extraction module.

NL instruction	Expert data	MTurk data
Sizes		
Slice the bread into **small** pieces.	size(pieces-6, small-5)	color(pieces-6, small-5)
Slice the **large** bread.	size(bread-4, large-3)	color(bread-4, large-3)
Fill a **large** pot with water.	size(pot-4, large-3)	color(pot-4, large-3)
Shapes		
Cut the sandwich into **triangular** pieces.	-	-
Press the dough into a **round** form.	-	-
Put the chicken on a **rectangular** tray.	-	-
Colors		
Press the **red** button.	color(button-4, red-3)	color(button-4, red-3)
Slice the **green** apples.	color(apples-4, green-3)	color(apples-4, green-3)
Pour the sauce over the **brown** meat	color(meat-7, brown-6)	-

This table lists the recognized properties by the property extraction module of the PRAC framework. The expert data column shows the inferred properties of the current MLN while the MTurk data column shows the inferred properties of the MLN trained with crowdsourced data. The normal evaluation sentences do not contain properties, such that the shown example sentences were additionally created to assess this module. The sentences are ordered by the type of their contained property which is written in **bold**.

[17] This table corresponds to Table 10 in Appendix C and is only included in this appendix for completeness.

C.3 Action Role and Word Sense Inference

This module infers action roles of ACs which are given as evidence. Furthermore, it assigns word senses to the inferred roles. For this module, a MLN is trained for every AC contained in the example sentences. An exception to this is the AC *"Neutralizing"*, as its template formulas contain hard constraints which the only crowdsourced sample containing does not fulfill. Table 16 lists the inferred action roles with inferred their word senses.

Table 16: Complete inference results of PRAC's action role and word sense module.

NL instruction	Expert data	MTurk data
Adding		
Add some water to the purine.	action_verb: add.v.01 theme: water.n.06 goal: purine.n.02 amount: Unknown unit: Unknown	action_verb: add.v.01 theme: water.n.06 goal: purine.n.02 amount: Unknown unit: Unknown
Add some arsenic acid to the imidazole.	action_verb: add.v.01 theme: arsenic_acid.n.01 goal: imidazole.n.01 amount: Unknown unit: Unknown	action_verb: add.v.01 theme: arsenic_acid.n.01 goal: imidazole.n.01 amount: Unknown unit: Unknown
Add 5 drops of the lysergic acid to the pyrimidine.	action_verb: add.v.01 theme: lysergic_acid.n.01 goal: pyrimidine.n.01 amount: five.n.01 unit: drop.n.02	**action_verb: add.v.01 (2x)** theme: lysergic_acid.n.01 **cliff.n.01** goal: pyrimidine.n.01 **amount: Unknown** **unit: Unknown**
Add 1 liter of water to the chlorous acid.	action_verb: add.v.01 theme: water.n.06 goal: chlorous_acid.n.01 amount: one.n.01 unit: liter.n.01	action_verb: add.v.01 theme: water.n.01 **liter.n.01** **goal: chlorous_acid.n.01 (2x)** **amount: Unknown** **unit Unknown**
Add some cheese to the pizza.	action_verb: add.v.01 theme: cheese.n.01 goal: pizza.n.01 amount: Unknown unit: Unknown	action_verb: add.v.01 theme: cheese.n.01 goal: pizza.n.01 amount: Unknown unit: Unknown

Complete inference results of PRAC's action role and word sense module.

NL instruction	Expert data	MTurk data
Cooking		
Make italian dinner.	action_verb: cook.v.02 obj_to_be_cooked: dinner.n.01	action_verb: make.v.03 obj_to_be_cooked: dinner.n.01
Cutting		
Slice the baguette.	action_verb: slit.v.01 obj_to_be_cut: baguet.n.01 amount: Unknown unit: Unknown utensil: Unknown	action_verb: slice.v.03 obj_to_be_cut: baguet.n.01 amount: Unknown unit: Unknown utensil: Unknown
Slice the italian bread.	action_verb: slice.v.03 obj_to_be_cut: italian_bread.n.01 amount: Unknown unit: Unknown utensil: Unknown	action_verb: slice.v.03 obj_to_be_cut: italian_bread.n.01 amount: Unknown unit: Unknown utensil: Unknown
Slice the bread.	action_verb: slice.v.03 obj_to_be_cut: bread.n.01 amount: Unknown unit: Unknown utensil: Unknown	action_verb: slice.v.03 obj_to_be_cut: bread.n.01 amount: Unknown unit: Unknown utensil: Unknown
Cut the pizza into 4 pieces.	action_verb: cut.v.01 utensil: cutter.n.06, obj_to_be_cut: pizza.n.01 amount: four.n.01 unit: piece.n.08	action_verb: cut.v.01 **utensil: Unknown** obj_to_be_cut: pizza.n.01 **amount: Unknown** **unit: four.s.01**

Complete inference results of PRAC's action role and word sense module.

NL instruction	Expert data	MTurk data
Filling		
Fill a blender with apples.	action_verb: fill.v.01 stuff: apple.n.01 goal: blender.n.01	action_verb: fill.v.09 stuff: apple.n.01 goal: blender.n.01
Fill a cup with water.	action_verb: fill.v.01 stuff: water.n.06 goal: cup.n.01	**action_verb: fill.v.09** stuff: water.n.01 goal:cup.n.01
Fill a mixer with pineapples.	action_verb: fill.v.01 stuff: pineapple.n.02 goal: mixer.n.04	action_verb: fill.v.09 stuff: pineapple.n.02 goal: mixer.n.04
Fill a glass with wine.	action_verb: fill.v.01 stuff: wine.n.01 goal: glass.n.02	**action_verb: fill.v.09** stuff: wine.n.01 goal: glass.n.02
Flavouring		
Flavour the tomato sauce with the oregano.	action_verb: season.v.01 goal: tomato_sauce.n.01 spice: marjoram.n.02	**action_verb: season.v.01 (2x)** **goal: tomato_sauce.n.01 (2x)** spice: marjoram.n.02 **tomato_sauce.n.01**
Opening		
Open the cupboard.	action_verb: open.v.01 obj_to_be_opened: cupboard.n.01	action_verb: open.v.01 obj_to_be_opened: cupboard.n.01
Open the fridge.	action_verb: open.v.01 obj_to_be_opened: electric_refrigerator.n.01	action_verb: open.v.01 obj_to_be_opened: electric_refrigerator.n.01
Open the test tube.	action_verb: open.v.01 obj_to_be_opened: test_tube.n.01	action_verb: open.v.01 obj_to_be_opened: test_tube.n.01
Preheating		
Preheat the oven.	action_verb: preheat.v.01 obj_to_be_heated: oven.n.01 temperature_setting: Unknown temperature_unit: Unknown	**Fails to query**

Complete inference results of PRAC's action role and word sense module.

NL instruction	Expert data	MTurk data
Pressing		
Press the start button.	action_verb: press.v.01 obj_to_be_pressed: push_button.n.01 location: centrifuge.n.01	action_verb: press.v.01 **obj_to_be_pressed: Unknown** location: Unknown
Putting		
Put the solution onto the mixer.	action_verb: put.v.01 obj_to_be_put: solution.n.01 location: mixer.v.04	action_verb: put.v.01 obj_to_be_put: solution.n.01 **location: Unknown**
Starting		
Start the centrifuge.	action_verb: begin.v.03 obj_to_be_started: centrifuge.n.01	**action_verb: get_down.v.07** **obj_to_be_started: Unknown**
Put the solution onto the mixer and start it.	action_verb: begin.v.03 obj_to_be_started: Unknown	action_verb: get_down.v.07 **obj_to_be_started: Unknown**

This table lists the recognized action roles by the Action Role and Word Sense Inference module of the PRAC framework. The expert data column shows the inferred roles of the current MLNs and MTurk data shows the inferred results of MLNs trained with sampled MTurk data. To get an example sentence which only contains roles of the *"Putting"* AC, the example sentence *"Put the solution onto the mixer and start it."* is reduced accordingly. Incorrect inferences are highlighted in **bold**.

C.4 Action Core Refinement

This module refines generic ACs to more specific ones. For evaluation, we train a MLN for generic ACs contained in the example sentences. However, the evaluation of some ACs is not adequate: Firstly, the AC *"Neutralizing"* is contained in only one crowdsourced sample for which a semantically wrong refinement was selected. Additionally, the example sentence of *"Make italian dinner."* of the *"Cooking"* AC gets refined into several sub steps using a database. Therefore, changes in the MLN are not reflected in the final inference results. Similarly, the example sentences of the *"Starting"* AC are also refined with the database. Even if a MLN is trained with crowdsourced data, it yields the same refinements as the original MLN, even though these refinements are not in the used training sets. Table 17 lists the inference results of the other generic ACs in the example sentences.

Table 17: Complete inference results of PRAC's AC refinement module.

NL instruction	Expert data	MTurk data
Adding		
Add some water to the purine.	OperatingATap	Pouring
Add some arsenic acid to the imidazole.	UsingMeasuringCup	Pouring
Add 5 drops of the lysergic acid to the pyrimidine.	Pipetting	**Pouring**
Add 1 liter of water to the chlorous acid.	OperatingATap	Pouring
Add some cheese to the pizza.	Spooning	Pouring
Flavouring		
Flavour the tomato sauce with the oregano.	UsingASpiceJar	**Cooking**
Filling		
Fill a blender with apples.	Putting	Adding
Fill a cup with water.	OperatingATap	**Putting**
Fill a mixer with pineapples.	Putting	Adding
Fill a glass with wine.	Pouring	Pouring
Opening		
Open the fridge.	OpeningADoor	OpeningADoor
Open the test tube.	Unscrewing	**OpeningADoor**
Open the cupboard.	OpeningADoor	OpeningADoor
Preheating		
Preheat the oven.	TurningOnElectri-calDevice	TurningOnElectri-calDevice

This table lists inference results of the AC refinement module of the PRAC framework. The expert data column shows the inferred refinements of the original MLNs while the MTurk data column shows results of MLNs trained with crowdsourced data.

References

[Akk+10] Cem Akkaya et al. “Amazon Mechanical Turk for Subjectivity Word Sense Disambiguation”. In: *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon’s Mechanical Turk*. Association for Computational Linguistics. 2010, pp. 195–203.

[Ant+15] Stanislaw Antol et al. “VQA: Visual Question Answering”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 2425–2433.

[AVC10] Vamshi Ambati, Stephan Vogel and Jaime G. Carbonell. “Active Learning and Crowd-Sourcing for Machine Translation”. In: *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC’10)*. 2010, pp. 2169–2174.

[Cal09] Chris Callison-Burch. “Fast, Cheap, and Creative: Evaluating Translation Quality Using Amazon’s Mechanical Turk”. In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*. Association for Computational Linguistics. 2009, pp. 286–295.

[Chi+10] Lydia B. Chilton et al. “Task search in a human computation market”. In: *Proceedings of the ACM SIGKDD workshop on human computation*. ACM. 2010, pp. 1–9.

[D+06] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D. Manning et al. “Generating Typed Dependency Parses from Phrase Structure Parses”. In: *Proceedings of LREC*. Vol. 6. 2006. Genoa. 2006, pp. 449–454.

[FMS03] Kostas Fragos, Yannis Maistros and Christos Skourlas. “Word Sense Disambiguation Using WordNet Relations”. In: *First Balkan Conference in Informatics, Thessaloniki*. 2003.

[GBC16] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[How06] Jeff Howe. “The Rise of Crowdsourcing”. In: *Wired magazine* 14.6 (2006), pp. 1–4.

[Ipe10] Panagiotis G. Ipeirotis. “Analyzing the Amazon Mechanical Turk Marketplace”. In: *XRDS: Crossroads, The ACM Magazine for Students* 17.2 (2010), pp. 16–21.

[KCH11] Anand P. Kulkarni, Matthew Can and Bjoern Hartmann. "Turkomatic: Automatic Recursive Task and Workflow Design for Mechanical Turk". In: *CHI'11 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2011, pp. 2053–2058.

[KCH12] Anand Kulkarni, Matthew Can and Björn Hartmann. "Collaboratively Crowdsourcing Workflows with Turkomatic". In: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM. 2012, pp. 1003–1012.

[KCS08] Aniket Kittur, Ed H. Chi and Bongwon Suh. "Crowdsourcing User Studies with Mechanical Turk". In: *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM. 2008, pp. 453–456.

[Kit+11] Aniket Kittur et al. "Crowdforge: Crowdsourcing Complex Work". In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. 2011, pp. 43–52.

[Lin+14] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision*. Springer. 2014, pp. 740–755.

[Man+14] Christopher D. Manning et al. "The Stanford CoreNLP Natural Language Processing Toolkit". In: *ACL (System Demonstrations)*. 2014, pp. 55–60.

[MBR10] Matthew Marge, Satanjeev Banerjee and Alexander I. Rudnicky. "Using the Amazon Mechanical Turk for Transcription of Spoken Language". In: *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE. 2010, pp. 5270–5273.

[Mil95] George A. Miller. "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[MS12] Winter Mason and Siddharth Suri. "Conducting behavioral research on Amazons Mechanical Turk". In: *Behavior research methods* 44.1 (2012), pp. 1–23.

[NB12] Daniel Nyga and Michael Beetz. "Everything Robots Always Wanted to Know about Housework (But were afraid to ask)". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vilamoura, Portugal, Oct. 2012.

[NB15a] Daniel Nyga and Michael Beetz. "Cloud-based Probabilistic Knowledge Services for Instruction Interpretation". In: *International Symposium of Robotics Research (ISRR)*. Sestri Levante (Genoa), Italy, 2015.

[NB15b] Daniel Nyga and Michael Beetz. “Reasoning about Unmodelled Concepts – Incorporating Class Taxonomies in Probabilistic Relational Models”. In: *Arxiv.org*. Preprint: `http://arxiv.org/abs/1504.05411`. 2015.

[NM10] Matteo Negri and Yashar Mehdad. “Creating a Bi-lingual Entailment Corpus through Translations with Mechanical Turk: $100 for a 10-day Rush”. In: *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon’s Mechanical Turk*. Association for Computational Linguistics. 2010, pp. 212–216.

[Nyg+17] Daniel Nyga et al. “Instruction Completion through Semantic Information Extraction”. In: *International Conference on Robotics and Automation (ICRA)*. Accepted for publication. Singapore, 2017.

[PE10] Gabriel Parent and Maxine Eskenazi. “Clustering dictionary definitions using Amazon Mechanical Turk”. In: *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon’s Mechanical Turk*. Association for Computational Linguistics. 2010, pp. 21–29.

[PVA14] Eyal Peer, Joachim Vosgerau and Alessandro Acquisti. “Reputation as a sufficient condition for data quality on Amazon Mechanical Turk”. In: *Behavior research methods* 46.4 (2014), pp. 1023–1031.

[RD06] Matthew Richardson and Pedro Domingos. “Markov Logic Networks”. In: *Machine learning* 62.1-2 (2006), pp. 107–136.

[SF08] Alexander Sorokin and David Forsyth. “Utility data annotation with Amazon Mechanical Turk”. In: *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW’08. IEEE Computer Society Conference on*. IEEE. 2008, pp. 1–8.

[Sno+08] Rion Snow et al. “Cheap and Fast—But is it Good?: Evaluating Non-Expert Annotations for Natural Language Tasks”. In: *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics. 2008, pp. 254–263.

[Wu+16] Qi Wu et al. “Visual Question Answering: A Survey of Methods and Datasets”. In: *arXiv preprint arXiv:1607.05910* (2016).

Online References

[BOTO] *Boto 3 - The AWS SDK for Python.* URL: https://github.com/boto/boto3 (visited on 26/04/2017).

[FTFAQ] *Amazon Mechanical Turk FAQs.* URL: https://requester.mturk.com/help/faq (visited on 26/04/2017).

[GERWAGE] *Minimum Wages in Germany with effect from 01-01-2017.* URL: http://www.wageindicator.org/main/salary/minimum-wage/germany (visited on 20/05/2017).

[MTAPI] *Amazon Mechanical Turk API Reference.* URL: http://docs.aws.amazon.com/AWSMechTurk/latest/AWSMturkAPI/Welcome.html (visited on 26/04/2017).

[MTCLT] *Amazon Mechanical Turk Command Line Tools.* URL: https://requester.mturk.com/developer/tools/clt/ (visited on 26/04/2017).

[MTDATA] *Amazon Mechanical Turk - Question and Answer Data.* URL: https://docs.aws.amazon.com/AWSMechTurk/latest/AWSMturkAPI/ApiReference_QuestionAnswerDataArticle.html (visited on 26/04/2017).

[MTFEE] *Amazon Mechanical Turk Pricing.* URL: https://requester.mturk.com/pricing (visited on 26/04/2017).

[MTHIT] *Amazon Mechanical Turk API Reference - The CreateHit Operation.* URL: http://docs.aws.amazon.com/AWSMechTurk/latest/AWSMturkAPI/ApiReference_CreateHITOperation.html (visited on 26/04/2017).

[MTQUAL] *Amazon Mechanical Turk - Creating and Managing Qualifications.* URL: https://docs.aws.amazon.com/AWSMechTurk/latest/AWSMechanicalTurkRequester/Concepts_QualificationsArticle.html (visited on 26/04/2017).

[MTURK] *Amazon Mechanical Turk.* URL: https://www.mturk.com/mturk/welcome (visited on 08/04/2017).

[NLTK] *Natural Language Toolkit for Python.* URL: http://www.nltk.org/ (visited on 26/04/2017).

[POSTAGS] *Alphabetical list of part-of-speech tags used in the Penn Treebank Project.* URL: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html (visited on 10/04/2017).

[PRACWEB] *Probabilistic Action Cores Website*. URL: http://prac.open-ease.org:5001/prac/ (visited on 31/05/2017).

[UNIDEP] *Universal Dependencies*. URL: http://universaldependencies.org/docs/en/dep/ (visited on 10/04/2017).

[USWAGE] *United States Department of Labor - Minimum Wage*. URL: https://www.dol.gov/general/topic/wages/minimumwage (visited on 20/05/2017).

[WIKIHOW] *wikiHow - How to do anything*. URL: http://www.wikihow.com/Main-Page (visited on 01/06/2017).

[WN] *WordNet Search*. URL: http://wordnetweb.princeton.edu/perl/webwn (visited on 10/04/2017).