Semantic parsing from English to AMR using Imitation Learning

James Goodman

Supervisors: Andreas Vlachos & Jason Naradowsky

This report is submitted as part requirement for the MSc Degree in CSML at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

September 4, 2015

Abstract

The Abstract Meaning Representation (AMR) is a recent standard for representing the semantics of English text that is not domain-specific and provides a machine-interpretable format that encapsulates key parts of the meaning of a sentence. An automated parser to convert English sentences to AMR would be very helpful in machine translation, human-computer interaction and other areas.

We develop a novel transition-based parsing algorithm to obtain results in this English to AMR translation task that are close to state-of-the-art using simple imitation learning, in which the parser learns a statistical model by imitating the actions of an expert on the training data.

We attempt to improve upon this with more sophisticated imitation learning algorithms that permit the learned parser to ask for advice from the expert in situations that the expert by itself would not visit. This approach creates new sequences of actions that extend the training data into regions of search space that better represent the states that the learned parser is likely to visit in practice, and hence should improve performance. We take two existing algorithms for comparison (V-DAgger and LOLS), and find that taking elements of each leads to a hybrid algorithm (DI-DLO) with higher performance than either of the existing algorithms on the AMR parsing problem.

Acknowledgements

I would like to thank both my supervisors, Andreas Vlachos and Jason Naradowsky, for their time, invaluable insights and helpful review comments. These frequently stopped me veering too far into the undergrowth. Jason provided the initial Scala code for V-DAgger and AROW implementations that I proceeded to mangle. Thanks also to Ian Kirker for his public provision of a LaTeX thesis template.

All the code for this dissertation is publicly available in two repositories:

- Implementation of the imitation learning algorithms (LOLS, DAgger, DI-DLO and variants) is at `https://github.com/hopshackle/mr-dagger`
- Implementation of the transition-based parser is at `https://github.com/hopshackle/dagger-AMR`

Contents

1 Introduction **10**

1.1 AMR background 10

1.2 Previous work on English to AMR translations 14

1.3 Our contribution 16

2 Background **17**

2.1 Previous Work on AMR to English translation 17

2.2 AMR parsing as Structured Prediction 21

2.3 Transition-based parsing 23

2.4 Imitation Learning background 26

2.5 Cost-sensitive and confidence-weighted Classification 32

3 Data **36**

3.1 Training corpus 36

3.2 F-Score 37

4 A novel transition-based graph parsing algorithm for AMR **39**

4.1 Fragments 39

4.2 Pre-processing and initialisation 40

4.3 Action Space 42

4.4 Features used 52

4.5 Classifier 52

4.6 The Expert Policy 53

4.7 Alignment impacts 55

4.8 Unseen Lemma replacement 58

5 Imitation Learning approach 61

5.1 Combinations of LOLS and V-DAgger 61

5.2 Smatch as a non-decomposable loss function 66

6 Experiments 74

6.1 Validation Results 74

6.2 Final Results 83

7 Conclusion and Future Work 85

7.1 AMR performance 86

7.2 Imitation Learning 88

Bibliography 93

List of Figures

1.1 AMR Graph of Example I 11

1.2 Dependency Tree for Example I 13

2.1 Example of AMR ‘fragment’ I 18

2.2 Example of Transitions in Wang et al. algorithm 19

2.3 Example of AMR ‘fragment’ II 20

4.1 NATO AMR fragment 40

4.2 AMR example graph II 41

4.3 Example of NextNode and NextEdge actions 44

4.4 Example of Swap action 44

4.5 Example of Reattach action 45

4.6 Example of ReplaceHead action 45

4.7 Example of Insert action 46

4.8 Example of ReversePolarity action 46

4.9 Example of Reentrance action 47

4.10 Sample output AMR graph in training showing result of an ‘Insert Loop’ 50

4.11 Correct AMR output for NATO/network sentence 56

5.1 Example of constructing an AMR graph for “Estonia” 66

5.2 Alternative example of constructing an AMR graph for “Estonia” . . 67

5.3 Plot of Smatch score against Naive Smatch 72

6.1 Impact of F-Score on Training Set with AROW Regularisation . . . 77

6.2 Impact of F-Score on Validation Set with AROW Regularisation . . 77
6.3 Loss Function performance comparison 78
6.4 Algorithm performance comparison 80
6.5 Algorithm performance comparison (Validation data only) 80
6.6 Impact of V-DAgger sample size 81
6.7 Training error by V-DAgger sample size 81
6.8 Impact of β decay on training performance 82
6.9 Impact of β decay on validation performance 82
6.10 Impact of a constant β rate on validation performance 83

List of Tables

2.1 Comparison of previous work on the AMR task 20
2.2 RollIn and RollOut guarantees 32

3.1 Training and Test data split 36

4.1 Action Space for the transition-based graph parsing algorithm . . . 43
4.2 Features used 51
4.3 Accuracy of expert on training set 55

5.1 Comparison of RollIn and RollOut policies by algorithm 64
5.2 Smatch performance with algorithmic changes 70

6.1 Cross-validation results for parameters in simple Imitation Learning 75
6.2 Simple imitation learning results compared with state-of-the-art . . 84

7.1 Concept Identifications Actions in Werling et al. [2015] 87

List of Algorithms

1 Simple Imitation Learning 25

2 V-DAgger (Vlachos Data Aggregation Algorithm) 29

3 LOLS (Locally Optimal Learning to Search) 31

4 Improved Aligner 57

5 Batch LOLS (Locally Optimal Learning to Search) 62

6 Generic outline for SEARN, V-DAgger, LOLS 63

7 Smatch calculation of F-Score 68

Chapter 1

Introduction

In this dissertation we use imitation learning and a transition-based graph-parsing algorithm to predict the Abstract Meaning Representation (AMR) graph of an input English language sentence.

In this first chapter we outline the natural language processing background to the problem of English to AMR translation, previous work on the problem, and our contributions in terms of novel algorithms for a transition-based parser and the imitation learning framework used to learn the parser. In Chapter 2 we provide background detail on transition-based parsing, and on imitation learning algorithms. Chapter 4 documents our novel adaptations in transition-based parsing, and Chapter 5 does the same for imitation learning algorithms.

Chapter 3 covers the training and test datasets used here and in previous AMR parsing work; Chapter 6 provides experimental results; and Chapter 7 concludes and considers future directions.

1.1 AMR background

The Abstract Meaning Representation (AMR) into which we translate English sentences was introduced by Banarescu et al. [2013]. The AMR graph of a sentence represents part of the underlying semantic information it carries in a standardised format, stripped of the inherent ambiguities and syntactic idiosyncrasies of human language. AMR is designed to help create large databases of semantically annotated sentences to be used in statistical machine translation, and does not include

all information; most obviously tense information is ignored. Banarescu et al. note that AMR remains an improvement over many previous semantic representation formalisms, which have focused on specific elements such as named entity identification or labelling of semantic roles (for example the Conference on Computational Natural Language shared task of 2009 [Hajič et al., 2009]), and less on whole-sentence representation. Figure 1.1 shows an example of an AMR graph.

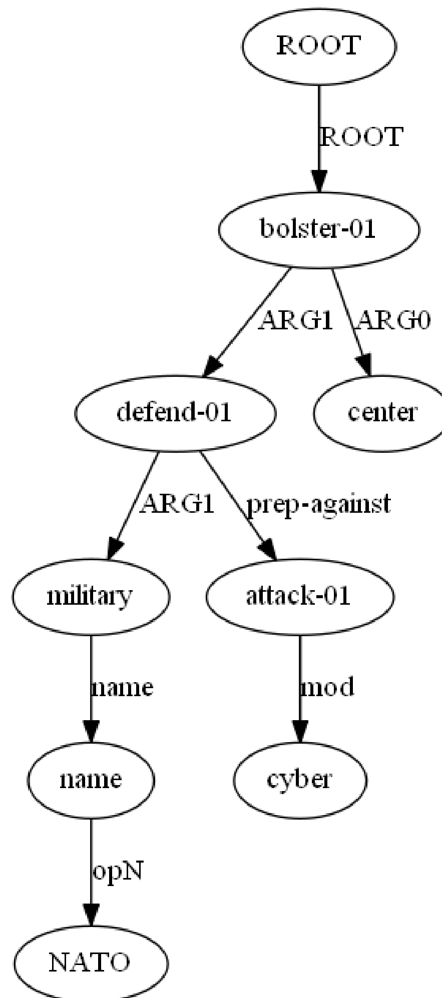

Figure 1.1: AMR Graph of the sentence "The center will bolster NATO's defenses against cyber attacks".

Reduction of a sentence in natural language to a standardised machine-interpretable form that contains the semantic meaning enables a suite of applications. For example Vlachos and Clark [2014] use a domain-specific meaning representation language (MRL) to automatically respond to requests by tourists in Ed-

inburgh looking for directions to hotels, restaurants and sites of interest. In this example the domain-specificity of the MRL reduces the vocabulary and semantic structures to be learned, which makes learning easier, but also restricts the results to the tourism domain. Similar issues apply to ATIS for air-travel bookings [Dahl et al., 1994] and GeoQuery for database queries [Zelle and Mooney, 1996]. The ambition of AMR is that it is not domain-specific and applies to any expression in the English language [Banarescu et al., 2013].

'AMR' is also used in a general sense to refer to any abstract system of semantic notation; *an* Abstract Meaning Representation, rather than the specific version developed in Banarescu et al. [2013]. See for example Langkilde and Knight [1998] and Dorr et al. [1998]. For the purposes of this dissertation, AMR always refers to the specific version of Banarescu et al. [2013]. There are other broad-based SemBank initiatives with similar goals to AMR, such as the Groningen Meaning Bank [Basile et al., 2012], UCCA [Abend and Rappoport, 2013] and the Semantic Treebank [Butler and Yoshimoto, 2012]. A comparison of these initiatives is outside the scope of this dissertation.

The linguistic analysis from sentence to AMR can be considered in layers, with the semantics that represent the actual meaning of the sentence on top. The lowest level labels each word (or token) in the input text with the part-of-speech (POS) that it represents. For example in the Penn Treebank nomenclature `JJ` indicates that a word is an adjective, `NN` a singular noun and `NNS` a plural noun [Marcus et al., 1993]. A layer up from this constructs syntactical connections *between* words, to form a parse-tree of a sentence. Using the Stanford dependency nomenclature [De Marneffe and Manning, 2008], we have `nsubj` to indicate a nominal subject, or `dobj` for a direct object; and these dependencies represent edges between nodes. An example of a dependency parse-tree is depicted in Figure 1.2, for the same sentence as the AMR graph in Figure 1.1.

Lexical databases then provide a dictionary of different meanings of words. In the example given, 'bolster' can (ignoring its possible usage as a noun), have a number of different meanings depending on context; the WordNet database [De Marn-

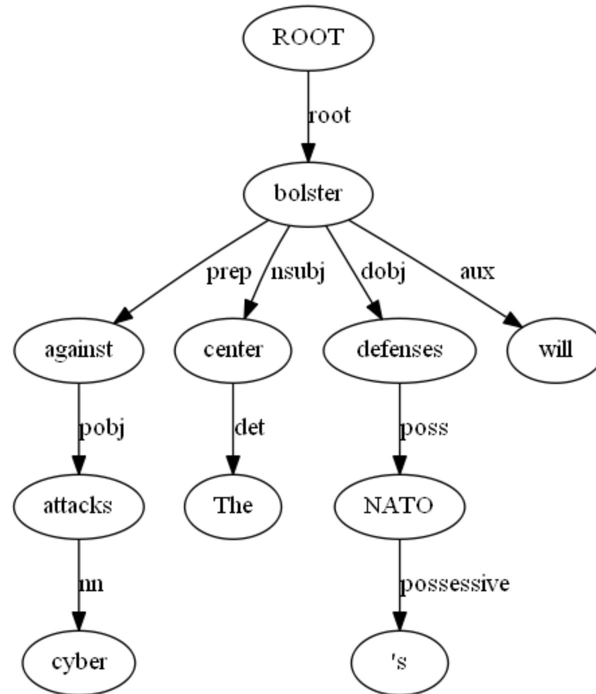

Figure 1.2: Stanford Dependency Tree of the sentence "The center will bolster NATO's defenses against cyber attacks". The root, or focus, of the sentence is the single verb, bolster, which has a subject of 'center' and a direct object of 'defenses'.

effe and Manning, 1998] has three:

1. support and strengthen

2. prop up with a pillow or bolster

3. add padding to

The first of these is the one used in our example sentence. To semantically represent our sentence we need to clearly indicate which meaning of 'bolster' is meant, as well as the fact that it is a verb rather than a noun. The same goes for the other words, such as 'center', with multiple possible meanings. AMR makes use of the Penn PropBank [Kingsbury and Palmer, 2003] as this dictionary.

In PropBank each distinct meaning of a verb is a FrameSet. This documents the meaning, and the mandatory and optional arguments that the verb takes. For example the FrameSet `leave-02` has the meaning "give" (in the sense of leaving money in a will), with ARG0 being the giver, ARG1 the thing given and ARG2

the beneficiary of the gift [Babko-Malaya]. This is different to the FrameSet of `leave-01` for the more common meaning of departing a location; this has ARG0 as the leaver, ARG1 as the location or entity being left, and no ARG2.

Once we have annotated the sentence to resolve ambiguities in word meanings, we are ready for the final layer of the semantic parse, in which we seek to encapsulate the full meaning of the text. This is not just a matter of replacing the words in the dependency tree with the appropriate PropBank reference because we also need to abstract the underlying meaning away from the syntactic form. For example the sentences, “The supervisor regarded the thesis as a disaster.” and, “The thesis was a disaster from the supervisor’s point of view.” should have the same semantic graph as there is little difference in the *meaning* (for most practical purposes) even though they have very different syntactic structures.

Figures 1.1 and 1.2 show the AMR graph and dependency tree for the same sentence. As well as specifying the PropBank FrameSet (bolster-01), with its subject (ARG0) and object (ARG1) relations identified, the AMR graph has removed several features. These includes the determinant “The”, the auxiliary verb “will”, and the possessive “’s”. It has also replaced the noun “defenses” with a PropBank FrameSet (defend-01), and expanded the single word “NATO” to three nodes, to represent an abstraction of a military organisation called “NATO”.

1.2 Previous work on English to AMR translations

The first published work on the task was Flanigan et al. [2014] with the JAMR system that achieved an F-Score metric of 0.58. (The details of the training data and performance metrics are in Chapter 3.) Flanigan et al. split the task into two distinct sub-tasks; firstly *concept identification* and secondly *graph creation*. The sub-tasks are learned independently, and exact inference is used to find highest-scoring maximum spanning connected acyclic graph that contains all the concepts identified in the first stage.

Parallel work by Wang et al. [2015a] adopted a different strategy based on the insight that there is significant similarity between the dependency parse-tree of a

sentence, and the semantic AMR graph. A comparison of Figures 1.2 and 1.1 helps make this insight clear. Wang et al. [2015a] start from the dependency tree, and apply a transition-based parsing algorithm that converts this into an AMR graph by applying a sequence of actions to the starting tree. One key advantage they cite for this approach is that it enables the initial stage of generating a dependency tree to be trained using a much larger corpus of data than that for which we have AMR annotations. For this they use the Stanford NLP Parser v3.3.1 to generate the dependency trees which is trained on a separate and much larger corpus from the Wall Street Journal [Manning et al., 2014]. Wang et al. achieved an improvement over JAMR with an F-Score of 0.63. They were able in July 2015 to increase this to 0.71 by incorporating further features from additional sources, including a semantic role labeller and co-reference resolver [Wang et al., 2015b].

Wang et al. [2015a] create an expert policy that is run on the training data to generate a set of ideal trajectories (a sequence of ordered ⟨state, action⟩ pairs) that take the input dependency tree and output the AMR graph. The parser (or policy) is trained from these generated expert trajectories using an averaged Perceptron [Collins, 2002] and a binary expert loss function that scores the action chosen by the expert policy as correct, and all other actions as equally incorrect. Each trajectory is broken down into *instances*, with one instance per state. Each instance is a ⟨state, actionList, losses⟩ triple; *actionList* contains all possible actions that can be taken from *state*, and *losses* contains the correspondings loss experienced for taking each action. In the case of Wang et al. [2015a] one of the possible actions will have a loss of 0 (the one chosen by the expert), and all others will have a loss of 1. These instances are then used to train the Perceptron that can be used on unseen states to predict the losses of actions. The learned parser then takes the action with the lowest predicted loss.

This can be considered as a form of imitation learning, although it is not usually described as such in the transition-based parsing literature. Imitation learning has a long history in the field of process and robotic control [Silver et al., 2008, Schaal, 1999] and requires an expert demonstrator that takes the optimal set of ac-

tions on the training data to generate the (known) gold output on a training set. The observed trajectories of $\langle$state, action$\rangle$ pairs are then used to train a learned controller, which learns by imitating the expert. This has also been referred to as 'inverse reinforcement learning' [Abbeel and Ng, 2004], since we start with an optimal (or at least very good) expert, and seek to learn the action costs/rewards in an inversion of the usual reinforcement learning paradigm of learning policy from observed rewards. This is necessary because the expert is only defined on the training data, and by learning the implicit reward function underlying its behaviour, we hope to generalise the learned controller to function well with unseen data.

1.3 Our contribution

We follow the general strategy formulated by Wang et al., pre-processing input sentences to a dependency tree, applying a transition-based algorithm to convert this tree into an AMR graph, and then training a parser on these trajectories. Our contribution is three-fold:

1. We use a novel transition-based algorithm to convert from dependency tree to AMR graph that can Insert new nodes, unlike the original algorithm of Wang et al. [2015a], and hence generate AMR graphs that they cannot (Chapter 4)

2. We apply existing imitation learning algorithms (V-DAgger, LOLS) to the problem that explore beyond the trajectories generated by the expert policy and permit use of a more sophisticated loss functions (Chapter 5)

3. We develop a novel imitation learning algorithm (DI-DLO) and find that it performs better than either V-Dagger or LOLS on the AMR problem (Chapter 5)

The intuition for the second point is that by incorporating some level of exploration (i.e. error) in the generation of trajectories, the learned parser will observe the expert recovering from mistakes, which should help it to generalise better. The detailed background and motivation to this is covered in Section 2.4.

Chapter 2

Background

In this chapter we provide greater detail on previous AMR work (2.1), and general background in the two distinct areas of transition-based parsing (2.3) and imitation learning (2.4), outside of the context of the AMR problem.

2.1 Previous Work on AMR to English translation

The first work on the AMR parsing problem in the JAMR system by Flanigan et al. [2014] had two discrete stages. The first stage of *concept identification* is implemented using a semi-Markov model with a basic set of features to obtain the best-scoring set of AMR concepts that align to the words in the input sentence. The second stage of *graph creation* then uses a Lagrangian relaxation of the optimisation problem to find a minimum cost spanning acyclic graph that incorporates the identified AMR concepts. The AMR concepts considered in the concept identification stage are those found in the training data. The one exception to this is that JAMR uses some hard-coded rules to construct fragments for temporal expressions and some common named entity structures (e.g. people's names) identified in unseen data. For examples see Figure 2.1.

Flanigan et al. [2014] note that if they manually specify the correct concepts in the first stage then the second stage gives an F-Score of 0.80, which is comparable with observed inter-annotator agreement of 0.79-0.83 that provides the effectively performance ceiling for any machine learning approach. Given that 38% of the AMR concepts in the validation data set do not exist in the training set, the restric-

tion to AMR concepts that exist in the training data is a major problem identified by the authors.

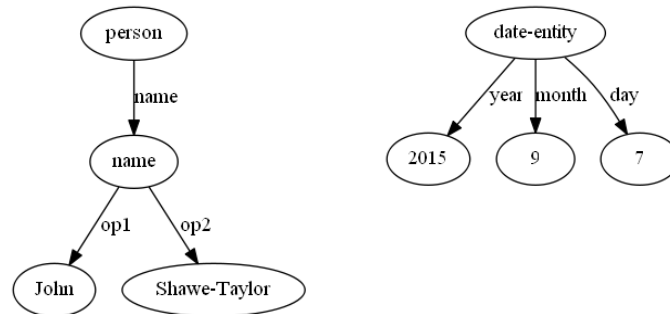

Figure 2.1: Example fragments that JAMR [Flanigan et al., 2014] would create from the unseen strings ”John Shawe-Taylor”, and ”20150907” based on a named entity recognition system applied as a pre-processing step.

This suggests that the major area for improvement is in concept identification and further work by Werling et al. [2015] focused on the concept identification stage, while retaining the JAMR graph creation algorithm. This allowed AMR concepts to be used that were not in the training data, in particular using the raw English word itself as the AMR concept (this would work for ‘center’ in the NATO example), or via a dictionary lookup to the most common Penn PropBank FrameSet for a verb (which could translate ‘attacks’ to `attack-01`). This increased the overall F-Score to 0.62.

As the sentence in Figures 1.2 and 1.1 shows, mapping from words to AMR concepts is not 1:1. “NATO” is mapped to a fragment of three distinct AMR nodes of `military`, `name` and `NATO` for example. Figure 2.1 has an example of four AMR concept nodes that from a fragment mapping to the two words “John Shawe-Taylor”. In the JAMR system it is these AMR ‘fragments’ which are used as the atomic building blocks rather than single AMR concept nodes directly. Flanigan et al. [2014] note that about 20% of all AMR fragments used have more than one node. In Chapter 4 we explain how, and why, we move away from this fragment approach.

The Wang et al. approach does not split the problem into distinct steps of concept identification followed by graph generation, and their single-stage transition-

based algorithm can interleave concept labelling actions with others that mutate the shape of the graph. This interleaving of steps allows information from graph generation to affect concept identification, which is not possible in the JAMR approach. A simple example of successive actions converting part of a dependency tree into an AMR Graph is shown in Figure 2.2.

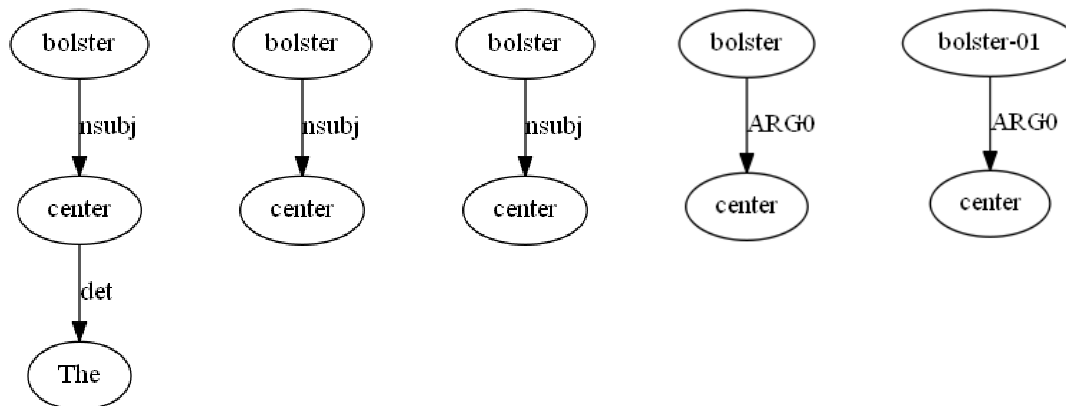

Figure 2.2: The five graphs show parts of successive states in a transition-based parse of a starting dependency tree (on the left). The actions from left to right are to Delete “The”; Label the “center” node as `center`; Label the “nsubj” edge as `ARG0`; Label the “bolster” node as `bolster-01`

Wang et al. still require a dictionary of AMR concepts that can be used when a node or edge is labelled. The concepts that are permitted are those that occur in the training data, and the same AMR fragments are used as in JAMR. Each fragment is treated as a composite node with hidden structure that is only unfolded once the algorithm completes (see Figure 2.3).

The scoring in the *concept identification* phase of Flanigan et al. [2014] requires an alignment between AMR fragments and words in the training set. This is not part of the corpus, each entry of which consists of just the raw English text and the AMR graph. Hence JAMR provides an aligner to pre-process the *training set* to provide this information. This aligner is not learned in any way, and uses a number of heuristic rules, regex expressions, and partial string matches to lemmas[1] in WordNet. On a sample of 200 hand-aligned sentences it achieved an F-Score of 0.90. Wang et al. [2015a] use the JAMR aligner for the training data without

[1] A lemma is the base form of a word without syntactic modification. For example ‘attack’, ‘attacking’, ‘attacks’, ‘attacked’ all have the same lemma of ‘attack’.

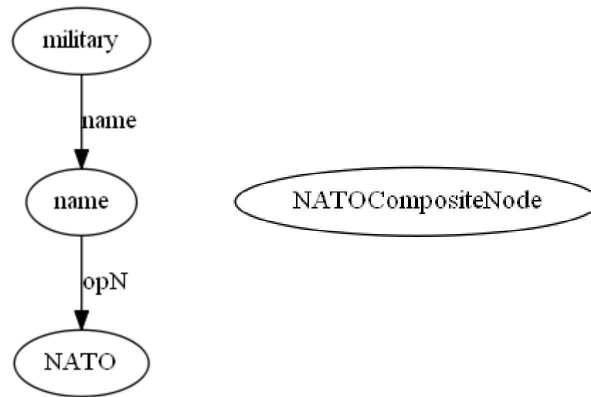

Figure 2.3: Three-node fragment on the left is represented by a single composite node by Wang et al. [2015a]. On termination of the algorithm, the `NATOCompositeNode` is replaced with the three-node fragment in the final AMR graph.

Table 2.1: Comparison of previous work on the AMR task. The last three items were unpublished (and unknown) when this dissertation was started.

Authors	AMR-English alignment	Algorithmic Approach	F-Score
Flanigan et al. [2014]	JAMR Aligner	Two stages. Concept identification with semi-markov model to identify set of AMR fragments. Followed by optimisation of constrained graph that contains all of these.	0.58
Werling et al. [2015]	JAMR Aligner	Same stages as Flanigan et al. [2014], but with enhancements to concept identification to use dictionary lookups and concepts not in the training set	0.62
Wang et al. [2015a]	JAMR Aligner	Single stage using transition-based parsing algorithm	0.63
Pust et al. [2015]	Pourdamghani Aligner	Single stage within framework of System-Based Machine Translation	0.66
Peng et al. [2015]	Enhanced JAMR Aligner	Hyperedge replacement grammar	0.58
Wang et al. [2015b]	JAMR Aligner	Extensions to action space of original transition-based algorithm, and to features used	0.71

modifications.

Separately Pourdamghani et al. [2014] developed a learned English to AMR aligner using Expectation-Maximisation methods that aligns English words to the edges of an AMR graph as well as the nodes; however they do not use the same dataset as Flanigan et al. [2014], nor compare against JAMR to provide a comparable F-Score. The Pourdamghani aligner is used in contemporaneous unpublished work by Pust et al. [2015] that attains an F-Score of 0.66 using string-to-tree translation within the framework of Syntax-Based Machine Translation (SBMT). This treats AMR and English as 'a language-pair indistinct from any other', and uses some semantic features but not the full dependency tree as Wang et al. do. Another recent approach by Peng et al. [2015] uses Hyperedge Replacement Grammars, which they find to be especially sensitive to missing or incorrect alignments in the training data.

2.2 AMR parsing as Structured Prediction

The parsing of English to AMR is an example of structured prediction, in which we search for the best structured output for a given input. This provides some theoretical insight, and helps clarify the role imitation learning is expected to play.

Following Daumé III et al. [2009], a structured prediction problem $\mathscr{D}$ is defined as a cost-sensitive classification problem where $\mathscr{Y}$, the output space, has structure: elements $\mathbf{y} \in \mathscr{Y}$ decompose into variable-length vectors $(y_1, y_2, y_3, \dots y_T)$. $\mathscr{D}$ is a distribution over inputs $x \in \mathscr{X}$ and cost vectors $\mathbf{c}$, where $|\mathbf{c}|$ is a variable in k^T, and k is the cardinality of y_i.

The output, $\mathbf{y}$, that we are predicting in the AMR case is one possibility amongst the (infinite) space of all possible AMR graphs. We have structure in two senses. Any AMR graph can be broken down into sub-graph fragments, each with its own structure; alternatively, if we consider the space of action sequences, we are predicting a sequence of actions that directly ties in with Daumé's "decomposition into variable-length vectors". From this second perspective the same output $\mathbf{y}$ can be generated by different action sequences, and there may be multiple optimal

vectors $(y_1, y_2, y_3, \dots y_T)$.

The objective of a structured prediction problem is to find a function $h : \mathscr{X} \to \mathscr{Y}$ that minimises J, the expected cost c_y of the output $y = h(x)$ given inputs x that follow a distribution $\mathscr{D}$:

$$J(\mathscr{D}, h) = \mathbb{E}_{(x \sim \mathscr{D})}(c_{h(x)}) \tag{2.1}$$

Classic methods to finding h construct a function $F(\mathbf{y}|x, \theta)$ that given an input x and (possibly learned) parameters θ provides a score for any given output $\mathbf{y}$. This requires an enumeration over all $\mathbf{y} \in \mathscr{Y}$ to find the optimum. This for example is solved in HMM or MEMM approaches to labelling a sequence, which rely for tractability of inference on only short-range label dependencies being modelled in F. Each element y_i of Daumé's output vector is predicted independently of the others; or more generally with some k-order Markov assumption so that only $y_{i-k} \dots y_{i-1}$ influence the prediction of y_i. See for example Punyakanok et al. [2004] for the independent case, or McCallum et al. [2000] for Maximum Entropy Markov Models that cover the general-order case. These approaches work well if the problem can naturally be phrased as a *bounded* sequence of decisions with a chain structure. This is natural in labelling the part-of-speech tags of words in a sentence, but less so in the case of graph construction. If the length of action sequences (and space of possible graph output) is unbounded this angle does not seem attractive. We explain in Chapter 4 why this is the case for our transition-based parser. For the JAMR system of Flanigan et al. [2014] that provides the first AMR parsing baseline results the problem *is* tractable as the space is bounded in both stages. In stage one the bound on AMR concepts is provided by the words in the sentence, with no support for AMR fragments not linked to a specific set of words. In stage two the bound is the finite set of AMR concepts from stage one.

When exhaustive search is intractable, the standard response is selective search of the output space using for example, a greedy heuristic, local hill-climbing or Monte Carlo search [Daumé III et al., 2009]. As Nivre [2003] points out, transition-based parsing algorithms that process monotonically node by node, and edge by edge, amount to a highly pruned, near-deterministic search of possible outputs,

where output space is defined by all possible graphs. A more exhaustive search of this output space would enable better solutions to be found at the cost of (exponential) increases in running time. Hence transition-based algorithms can be considered as selective searchers and imitation learning uses an expert policy to direct this selective search to fruitful areas of $\mathscr{Y}$; the *starting points* for exploration in training are provided by the trajectories the expert policy generates.

In the syntactic dependency-parsing literature, a similar distinction has been made between transition-based parsing techniques and graph-based techniques [Nivre and McDonald, 2008]. Graph-based techniques use exact inference with an implicit enumeration over all outputs to find the best-scoring graph directly, usually by assuming that the score of a graph can be factorised into additive sums from each arc, or sometimes from factors of pairs of edges [McDonald, 2006]. This has the advantage that exact inference algorithms can be used to find the highest scoring graph, but a disadvantage that in order for this to be tractable the factorisation of the score cannot take into account long-range features. JAMR uses this approach in the *graph construction* phase. Transition-based parsing has the advantage that a much richer feature-set can be used by breaking the model assumption that graph-scoring factorises at the edge level required for tractability of exact inference, but has the downside of potential error propagation due to the greedy nature of decisions at each action step. The parser may greedily take an action at step n that causes issues at step $n+100$. The issues around error propagation in transition-based parsing form a major strand of the later discussion.

2.3 Transition-based parsing

Transition-based parsing algorithms have been used extensively in dependency tree parsing; taking an input sentence of words and generating a tree structure as in Figure 1.2. This is a very effective method of processing programming languages during compilation into a functioning program [Aho et al., 1986]. The extension of this to natural language is hampered by the innate ambiguity of the input. Despite this issue the approach works well in practise, with the first left-to-right such parser

on English described by Yamada and Matsumoto [2003].

The standard transition-based parsing approach starts with a graph (e.g. all word-tokens from the input sentence as fully disconnected nodes), and then applies a sequence of actions until a terminal state is reached. A learned parser will use an appropriate classifier to make a decision at each stage. Yamada and Matsumoto [2003] used an SVM , while perceptron-based linear classifiers are popular in more recent work Collins [2002]. We review classifiers in Section 2.5.

A transition-based parsing algorithm requires three components [Nivre, 2003]:

- a set of actions (including a definition of when an action is permissible based on the current state)
- a set of features
- a classifier to select the best action at each step based on the features

The classic dependency parsers use either three (LEFT, RIGHT, SHIFT) or four (LEFT, RIGHT, SHIFT, REDUCE) actions, and are *incremental* (to use the terminology of Nivre [2003]) in that once an action is taken we cannot revisit previous steps. It is also *monotonic* (to use the closely-related terminology of Honnibal et al. [2013]) in that all later actions must be consistent with the actions taken hitherto. Training of the parser occurs as outlined in Section 1.3 by applying an oracle to the training data to construct trajectories of $\langle$state, action$\rangle$ pairs that represent perfect decisions. These are then used to train the classifier using Algorithm 1.

A number of limitations of the simple parsing approach have been identified in the literature. A central problem is that each decision is made greedily, and cannot take account the effect future actions might have on the final output. We might need to see the impact of the decision on future states and permissible actions to be able to make a decision between two actions now. An error made by a parse at an early stage in the process can then propagate, as explained by McDonald and Nivre [2007]. This problem avoided by graph-based approaches that score on the final output only. A number of strategies have been used to mitigate this. Goldberg and

```
Data: data D, expert policy π*, feature function f
Result: learned classifier C
Initialise E = ∅;
for d ∈ D do
    Initialise L = ∅;
    Predict trajectory ω = ŷ_{1:T} using π*;
    for ŷ_t ∈ ω do
        Extract features Φ_t = f(d, ŷ_{1:t−1});
        y_t^0 = ŷ_t;
        y_t^{1...} = all other possible actions;
        foreach possible action y_t^j ≠ y_t^0 do
            L_t^j = 1.0;
            /* Loss on non-expert actions is 1        */
        end
        L_t^0 = 0.0;
        /* Loss on expert action is 0                 */
        Add (Φ_t, L_t) to E;
    end
end
Train C using full set of experienced data E;
```

Algorithm 1: Simple Imitation Learning

Elhadad [2010] remove the determinism of the sequence of actions to create *easy-first* parsers, which postpone uncertain decisions if possible until more information is available; but once a decision is taken it is still fixed. This contrasts with the classic *static* approach in which we work inflexibly left-to-right along a sentence, or bottom-to-top up a tree. Zhang and Clark [2008] use beam search through state-space for each action choice; for each possible action the parser conducts searches through the exponentially branching set of actions to find a better approximation of the long-term score of the action.

Goldberg and Nivre [2012, 2013] introduce *dynamic* experts that are both *non-deterministic* in generating a *set* of best actions in a given state; and *complete* in that they will respond from any state, not just those on the perfect trajectory. Hence if mistakes have been made previously, a complete oracle will still be able to instruct on what to do next. They refer to this approach as "Learning with exploration". During training instead of *always* taking the expert action to generate a trajectory, occasionally an exploratory action will be taken according to the output of the cur-

rent learned parser. The trajectory will then continue from this sub-optimal position. The authors link their work to the general imitation learning literature and note similarities between their approach and the DAgger algorithm which we cover in more detail in section 2.4. It is not standard practice to refer to Algorithm 1 as 'imitation learning' in the transition-based parsing field. We do so here to make clear the natural progression to the later algorithms of Section 2.4.

Honnibal et al. [2013] use a transition-based parser that is *non-monotonic*, allowing actions that are formally inconsistent with previous actions. When such an action is taken it also amends the results of previous actions to ensure post-hoc consistency. They find there is a need to favour monotonic steps over non-monotonic ones when both are equally scored. Otherwise training results in a poor classifier, because the difference between good and bad actions early in the trajectory cannot be distinguished, as the bad actions can be fixed later. Despite this 'undo' ability, the algorithm does not increase the overall length of the trajectory, with repair executed as part of the overriding action. As a result Honnibal's non-monotonic algorithm works in linear time like the monotonic algorithms on which it is based [Honnibal et al., 2013] [Nivre, 2003].

2.4 Imitation Learning background

Daumé III et al. [2009] noted the common ground between robotic/process control setting and structured prediction such as the parsing algorithms summarised in the previous section - see also Vlachos [2012]. In both fields there is a common issue arising from the difference between the actions of the expert policy/demonstrator being imitated and the actions of the learned policy/controller during test that we touched on when discussing the work of Goldberg and Nivre [2013] on *dynamic* graph-parsing algorithms.

Even when the learned policy acts on the training set, it may not take exactly the same actions as the expert demonstrator/policy. This will be because the policy's feature set is not rich enough to distinguish between two states that the expert can. Consider a learned policy that takes a single non-optimal action on a training

example and finds itself in a state for which there are no exact training examples as it has deviated from the expert trajectory. This is not necessarily a problem, and is no different to the learned policy acting on an unseen test instance for which there are similarly no exact training examples. The learned policy may generalise similarly well to off-trajectory states on the training data. However there can be a deeper problem since the states encountered during training are themselves dependent upon the expert policy, and this training distribution may not be representative of the states encountered by the learned policy even when executed on the training data.

For a somewhat contrived example, a shopping robot demonstration that always buys milk before bread will never encounter states in which there is bread in the basket, and milk still on the list of required items. This will hinder learning and generalisation as if a mistake is made in dairy products, this might propagate to later groceries. The 'exploratory actions' in Goldberg and Nivre [2013] would allow this bread and no milk state to be reached; and the expert could then demonstrate the corrective action of popping back for a pint.

Ross et al. [2011], Ross and Bagnell [2010] formalise this intuition in with theoretical results for simple imitation learning using just the expert trajectory data (Algorithm 1). In this simple case with a cost function bounded on $[0,1]$, we have

$$J(\pi) \leq J(\pi^*) + T^2\varepsilon \tag{2.2}$$

where $J(\pi)$ is the loss incurred following policy π (from start state to terminal state); ε is the expectation of the $0-1$ loss using policy π under the distribution of states encountered using the expert policy π^*; T is the total number of actions taken in the trajectory. The problem arises because the distribution of states the policy encounters in test is not that encountered in training - the normal assumption of i.i.d. over inputs is very far from holding, as the states in any trajectory (the inputs used in training) are dependent on the policy used. To address this problem we need to observe a distribution of states during training that are representative of those the trained policy will encounter in the wild.

A number of algorithms have been outlined in recent literature that do this. Daumé III et al. [2009] introduced SEARN in 2009, which generates a stochastic mixture of policies. Ross et al. [2011] introduced DAgger (Dataset Aggregation) which outputs a deterministic policy that is generally more stable and more accurate than that of SEARN [Vlachos, 2012]. DAgger, SEARN and similar algorithms allow the learned classifier to influence the states used in training. Sometimes we take an exploratory step (under the learned classifier's control) off the expert trajectory, and ask the expert what it would do in this situation. In sample empirical work, Vlachos and Craven [2012] show a 8.3 F-point gain from SEARN over simple imitation learning.

2.4.1 V-DAgger

In this dissertation we look in detail at two imitation learning algorithms. The first of these is V-DAgger, a modification by Vlachos and Craven [2012] of the original DAgger by Ross et al. [2011].

At each iteration, V-DAgger generates a 'RollIn' trajectory ω for each item in the training set. This uses the expert policy with probability β at each step, and the currently learned policy with probability $(1-\beta)$. β starts at 1.0 and decays at a rate δ so that the expert policy has progressively less and less impact on training. For each step on this baseline trajectory, all possible actions other than the one taken are considered, and for each of these a 'RollOut' trajectory to a terminal state is generated using the same process. These RollOut trajectories explore 'what if' scenarios and provide the information of the loss incurred for the exploratory action. Training of the learned policy then takes place using the *regret* (i.e the difference between the loss on the exploratory action and the loss on the best action, which may or may not be the one taken during the RollIn trajectory) as the training loss.

The RollOut trajectories involve step-wise stochastic choice between the expert and currently learned policies. This means that, especially for long trajectories, we can obtain very different terminal states and consequent losses depending on where and how we deviate from the expert. Hence we use Monte Carlo sampling of multiple RollOut trajectories and take the mean loss across all of these. This is the

```
Data: data D, expert policy π*, Loss function F(y), sampleSize, feature
      function f, decay rate δ
Result: learned policy π̂_{N+1}
Initialise E = ∅;
Initialise π̂_1 = π*, β = 1.0;
for i = 1 to N do
    for d ∈ D do
        Initialise L = ∅;
        π_i = βπ* + (1 − β)π̂_i  randomised at each step;
        Predict RollIn trajectory ω = ŷ_{1:T} using π_i;
        for ŷ_t ∈ ω do
            foreach possible action y_t^j ≠ ŷ_t do
                Extract features Φ_t^j = f(d, y_t^j, ŷ_{1:t−1});
                for s = 1 to sampleSize do
                    e_s = RollOut trajectory with starting actions ŷ_{1:t−1}, y_t^j
                    through to a terminal state using π_i;
                end
                Calculate mean loss L_t^j = average of F(e_1)...F(e_sampleSize);
                Calculate regret L_t^j = L_t^j − F(ω);
            end
            Add (Φ_t, L_t) to E;
        end
    end
    Train π̂_{i+1} using full set of experienced data E;
    β = (1 − δ)β;
end
```

Algorithm 2: V-DAgger (Vlachos Data Aggregation Algorithm)

purpose of the *sampleSize* parameter in Algorithm 2. The differences between V-DAgger and Dagger are that DAgger does not RollOut from each step on the RollIn trajectory, but uses binary expert loss (as per Wang et al. [2015a]); and DAgger trains a policy on each iteration over the data and then averages them to give the final policy (also as per Wang et al. [2015a]).

Ross et al. [2011] prove that using DAgger for N iterations where N is $O(uT)$ and in the limit $N \rightarrow \infty$, under the same other assumptions as 2.2 then

$$J(\hat{\pi}) \leq J(\pi^*) + uT\varepsilon_N + O(1) \qquad \text{for some } \hat{\pi} \in \hat{\pi}_{1:N} \tag{2.3}$$

$\hat{\pi}_i$ is the output classifier from the ith V-DAgger iteration; ε_N is the expectation

of the true loss under the best policy in $\hat{\pi}_{1:N}$ in hindsight. Given the previously noted problem of unbounded T in our AMR transition-based parsing algorithm, this is an attractive theoretical point and improvement on the quadratic loss (in T) of (2.2) using simple imitation learning[2]. Provided that they are generated using the policy being learned (as in V-DAgger), the 'exploratory actions' help move the distribution of training events towards that the learned policy will encounter, and as training continues the pure expert actions will take a decreasing role through the decay of β.

Using V-DAgger or similar algorithms for the English to AMR problem provides two potential benefits over the simple imitation learning approach of Wang et al. (Algorithm 1). First is the ability to learn from mistakes, correct errors, and search areas of output space not visited by the expert on the training set. Secondly Algorithm 1 as written just uses a binary expert loss - the expert's action is correct, all other actions are equally wrong. It would also be possible to use a loss function that can quantify the extent to which the individual action will affect the final loss; i.e. the loss function would need to be *decomposable* over all actions in the trajectory with the total loss calculable by adding up the individual action losses. V-DAgger does not require a decomposable loss function as it calculates the loss for each permissible action from a RollOut trajectory to a terminal state. Hence for an ultimate target metric that is non-decomposable (such as F-Score), these imitation learning algorithms allow this to be used directly in training.

Vlachos and Clark [2014] use V-DAgger to show a total benefit of 4.8 points of F-Score from these factors in a domain-specific semantic parsing problem similar to the AMR case. In this example there is also a lack of alignment in the training corpus. This is learned as part of the main parser, with actions linking words in the input to one of the 35 node types - and a completely random 'expert' policy is used for this step; this works precisely because of the benefit from a non-decomposable loss function. We do not need to know whether any specific action is correct or

[2]This result is only proved by Ross et al. [2011] for the original DAgger, and has not been formally extended to V-Dagger. It arises from the decreasing role of the expert policy in trajectory generation, which is unchanged.

optimal immediately and we take our learning signal from the final impact this has on the terminal state.

2.4.2 LOLS

As well as V-DAgger, we also look in detail at LOLS (Locally Optimal Learning to Search) as a recent algorithm from Chang et al. [2015] (see Algorithm 5) . This was developed for the scenario where the expert policy is not optimal, and we want to learn how to improve performance beyond the starting policy. To this end LOLS provides a guarantee of low regret compared to one-step deviations from the learned policy *as well as* to deviations from the expert policy.

Data: data D, expert policy π^*, Loss function $F(\mathbf{y})$, feature function f, mixing rate β
Result: learned policy $\hat{\pi}$
$\hat{\pi}_0 = \pi^*$;
for $d \in D$ **do**
  Initialise $E = \emptyset$;
  Predict trajectory $\omega = \hat{y}_{1:T}$ using $\hat{\pi}_{d-1}$;
  for $\hat{y}_t \in \omega$ **do**
    Initialise $L = \emptyset$;
    foreach *possible action* $y_t^j \neq \hat{y}_t$ **do**
      Extract features $\Phi_t^j = f(d, y_t^j, \hat{y}_{1:t-1})$;
      $\pi_{RollOut} = \pi^*$ with probability β or $\hat{\pi}_{d-1}$ otherwise;
      $e =$ trajectory with starting actions $\hat{y}_{1:t-1}, y_t^j$ through to a terminal state using $\pi_{RollOut}$;
      $L_t^j = F(e) - F(\omega)$;
    end
    Add (Φ_t, L_t) to E;
  end
  $\hat{\pi}_{d+1}$ = Train$(\hat{\pi}_d, E)$;
  `/* Update old learned policy by training with newly gathered data */`
end
$\hat{\pi} =$ average of $\hat{\pi}_{1:D}$

Algorithm 3: LOLS (Locally Optimal Learning to Search)

LOLS introduces a type of consistency in trajectory generation. We always use the learned policy to generate the RollIn trajectory, and then *either* the expert *or* the learned policy on the whole of the RollOut, without randomising at each step as

V-DAgger does. Chang et al also introduce the terminology of RollIn and RollOut that we use in this dissertation.

Table 2.2: RollIn and RollOut guarantees

	RollOut		
RollIn	Expert	Mixture	Learned
Expert	Inconsistent	Inconsistent	Inconsistent
Learned	Locally non-optimal	Good [LOLS]	Reinforcement Learning

Table 2.2 is reproduced from Chang et al. [2015], and summarises the results of the detailed theoretical analysis in that paper. The LOLS guarantees apply when the Learned policy is used to RollIn, and a mixture of expert and learned policies used to RollOut. RollIn with the expert policy is problematic as it leads to an "unrealistically good" distribution of encountered states during training so that the learned policy never learns to correct for mistakes. Using the learned policy for RollIn and RollOut reduces to Reinforcement Learning, and we gain no benefit at all from an expert policy to help direct our search of the trajectory and output spaces.

RollIn with the learned policy, and RollOut with the expert meets the guarantee of low regret compared to deviations from the expert policy; but only if we use a mixture of expert and learned policies do we also obtain this guarantee with respect to the learned policy. If we compare LOLS to V-DAgger we see that V-DAgger also uses a mixture on the RollOut, but with the mixing occurring at each *step* rather than the level of the entire trajectory. V-DAgger similarly uses a mixture on the RollIn, which is not explicitly analysed by Chang et al. [2015].

2.5 Cost-sensitive and confidence-weighted Classification

Once we have a set of trajectories, whether these come from the expert policy, learned policy, or some combination of the two, the training step must take this as input to produce a new learned policy. Training instances consisting of ⟨state, actionList, losses⟩ triples are generated from the trajectories as discussed in Section 1.2. These training instances are used to train a classifier c able to score an

action a from a state s, i.e. $score = c(s,a)$. Specifically we generate a set of weights $\mathbf{w}$ that calculate the *score* as a simple linear predictor $score = \mathbf{w}^T \phi(a,s)$, where ϕ is a set of features calculated from s and a. From state s the learned policy then takes the action a^* with the highest score.

$$a^*(s) = \underset{a \in \mathscr{A}}{\operatorname{argmax}} (\mathbf{w}^T \phi(a,s)) \tag{2.4}$$

The use of Perceptron-style on-line learning has a long history from the original work by Rosenblatt [1958], and has become widely used in natural language processing since it was popularised by Collins [2002]. These methods try to find a separating hyper-plane between classes of data in feature-space and have advantages of making very few statistical assumptions about the data and being very efficient Crammer et al. [2009a]. The central idea is that for each instance of training data, the scores for each action are calculated using the current $\mathbf{w}$, and if the selected action is not the same as the correct (i.e. the observed) action, then $\mathbf{w}$ is updated in a direction to make the correct action more likely. This is repeated for a number of iterations over the whole set of instances. It has also been found useful to conduct this training a number of times, with the order of the training data randomised, and then average the resultant weight vectors $\mathbf{w}$ [Collins, 2002]. This averaging is used in Wang et al. [2015a], and is also part of the original LOLS algorithm (Algorithm 3).

The original Perceptron algorithm, and many variants derived from it, were designed for binary classification tasks. In our case we require a multi-class classifier, with each possible action being a class. We also wish to use cost-sensitive classification, and not be restricted to a simple zero-one loss. This enables the training process to distinguish between an incorrect action which is only slightly sub-optimal (for example with just one node mislabelled in the terminal state), and one which causes many nodes to be wrong. Intuitively we wish to update $\mathbf{w}$ more if we chose the latter action than had we chosen the former.

Another problem with classic perceptron-style algorithms is that they do not take into account variations in the frequency with which features are observed. This is a particular issue in natural language processing (NLP), where a feature that rep-

resents a bi-gram of two specific consecutive words (such as "dog" followed by "ate") is going to be seen very rarely compared to a feature for two specific consecutive parts-of-speech (such as NOUN-VERB). Rare features are often very discriminative between classes in NLP, and we should like to update their weights significantly more on the rare occasions that they occur and would help predict the correct action. This is the intuition behind confidence-weighted (or adaptive) algorithms [Crammer et al., 2009a, Shalev-Shwartz et al., 2003] with extensions in AROW [Crammer et al., 2009b] or AdaGrad [Duchi et al., 2011]. These can be extended to the multi-class case [Crammer et al., 2013, 2009a, Chiang et al., 2013].

These have adaptive learning rates that take into account the confidence on specific components of $\mathbf{w}$. Components of $\mathbf{w}$ corresponding to rare features will have a low confidence (and hence high learning rate), while weights for commonly observed features will be known with higher confidence, and a lower learning rate will be used when updating them. In AROW this is implemented by maintaining a Gaussian distribution over $\mathbf{w}$, with mean $\mathbf{w}$ and covariance Σ. The objective function minimised to obtain updated values of $\mathbf{w}$ and Σ for one training instance is:

$$C(\mathbf{w_t},\Sigma_t) = KL(\mathcal{N}(\mathbf{w_t},\Sigma_t) \parallel \mathcal{N}(\mathbf{w_{t-1}},\Sigma_{t-1})) + \frac{Loss(\mathbf{w_t},\phi_t)}{2C} + \frac{{\phi_t}^T \Sigma \phi_t}{2C} \tag{2.5}$$

where KL is the Kullback-Leibler divergence and $Loss$ is the Loss function. Minimising the first term reduces the movement in $\mathbf{w}$, with changes to components with higher confidence particularly damped. The second term minimises the actual loss, introducing cost-sensitivity. Minimising the third term encourages high-confidence in $\mathbf{w}$, since as $\Sigma \to \mathbf{0}$ we have a point distribution. C is the AROW regularisation parameter, and large values will emphasise the first term, so that we only make small changes to $\mathbf{w}$ at each step. Additionally, Σ is constrained to be a diagonal matrix, with no covariance between components of $\mathbf{w}$ [Crammer et al., 2013, Chiang et al., 2013].

To encourage a wide margin between the scores for correct and incorrect actions we can use *margin-scaling* [Crammer and Singer, 2003, Chiang et al., 2013]. This introduces a component for the margin between the scores of the predicted and

correct actions into the loss function. The Loss function we actually use for AROW in this dissertation is

$$\text{Loss} = \sqrt{\mathrm{l}(s,a)} + \mathbf{w}^T \phi(s,a) - \mathbf{w}^T \phi(s,a^*) \tag{2.6}$$

where a^* is the correct action for the training data from state s, a is the action selected by the classifier and l is the non-negative underlying loss we are trying to minimise (detailed in Chapter 5). Note that since $a^* \in \mathscr{A}$, the set of all actions, Loss is also non-negative. The incorporation of margin-scaling means that we may update $\mathbf{w}$ even if the classifier predicted the correct action, but another high-loss action was too close in score. This uses the 'Top-1' AROW variant [Crammer et al., 2013], by only considering a and a^*, and the square root of the loss is taken from Crammer et al. [2006], who find this helpful in reducing error for this family of algorithms.

Chapter 3

Data

3.1 Training corpus

The dataset used is the newswire section of LDC2014T12 [Knight et al., 2014]. This contains sentences of news reporting from a number of sources, predominantly Agence France Presse[1]. The data from years 1995-2006 form the training data, with 2007 as the validation set and 2008 as the test set. The data split is summarised in Table 3.1, and is the same as that used in all previous work on the AMR parsing task.

Table 3.1: Training and Test data split

Years	Role	Sentences
1995-2006	Training	3988
2007	Validation	2132
2008	Test	2132

The generation of these sets from the main corpus was achieved using the scripts provided by Wang et al. [2015a][2]. Additionally the first 577, and also the first 2120, sentences in the full training set were used as 'Small' and 'Medium' training sets respectively when comparing algorithms and parameter settings. Their use is made clear in the relevant experimental sections.

[1]Flanigan et al. [2014], Wang et al. [2015a] use the pre-release version of this dataset (LDC2013E117). Werling et al. [2015] conducted comparative tests on the two versions, and found only a very minor increase of 0.1 to 0.2 points of F-score when using the final release (from 0.621 to 0.622 for their system, and 0.591 to 0.593 for an improved JAMR).

[2]At http://goo.gl/vA32iI

3.2 F-Score

The F-Score used is the traditional F_1 score; the harmonic mean of the precision (proportion of output entries that are correct, and exist in the input) and recall (proportion of all correct input entries that also appear in the output)[3]. If P is the Precision, R the recall, C the total number of correct entries in the output, O the total number of entries in the output and I the total number of entries in the input, then:

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} \tag{3.1}$$

Alternatively, using $P = \frac{C}{O}$ and $R = \frac{C}{I}$ gives

$$F_1 = \frac{2C}{I + O} \tag{3.2}$$

In the context of AMR parsing, we use the Smatch score [Cai and Knight, 2013]. The input set consists of all logical statements defined in the gold AMR graph. For example just considering the nodes for 'defend-01' and 'military', plus their connecting arc, we have three statements:

- a – defend-01
- b – military
- a – ARG1 – b

There exists a node, a, with AMR concept 'defend-01', and a separate node, b, with concept 'military'. There is a relation of ARG1 connecting them in the direction a to b. This set of statements, taken across the whole graph, is what Smatch uses to calculate the F-Score. Consider an output graph that is an attempt to recreate the gold AMR graph and which contains:

- x – defend-01
- y – university

[3] https://en.wikipedia.org/wiki/F1_score

- x – ARG1 – y
- z – thing
- y – mod – z

The maximum F-Score of 0.50 occurs when we have a mapping of $a \rightarrow x, b \rightarrow y, \emptyset \rightarrow z$. We then have a precision of 0.40, since 2 of the 5 statements in the output example can be found in the gold graph, and a recall of 0.67, since 2 of the statements in the target can be found in the output. Formally we consider all possible mappings of nodes in the first graph to nodes in the second graph, and evaluate the F-Score for each, taking the maximum over all mappings. However this is an NP-hard problem and would become prohibitive as graph-size grows. The algorithm specified in Cai and Knight [2013] executes a heuristic search of mapping space, which is faster and empirically leads to no loss of accuracy compared to brute-force enumeration. We use the Smatch F-Score calculation for all evaluations in this dissertation, and we discuss the Smatch algorithm in more detail in Section 5.2 where we consider its use as a Loss function during training.

Chapter 4

A novel transition-based graph parsing algorithm for AMR

In this chapter we detail the algorithm we use to convert a dependency tree into an AMR graph. This is based on Wang et al. [2015a], and we make clear the changes made. We start in Section 4.1 by reviewing the AMR 'fragment' approach of Flanigan et al. [2014], Wang et al. [2015a], and our motivation for modifying this and introducing an Insert action as our major innovation over Wang et al. [2015a]. We then cover the initial pre-processing that takes place to produce the input to the main algorithm in Section 4.2.

As discussed in Section 2.3 a transition-based parsing algorithm requires three components: a set of actions (including a definition of when an action is permissible based on the current state), a set of features, and a classifier to select the best action at each step based on the features. We cover each of these in turn in the next three sections.

4.1 Fragments

As detailed in Section 2.1, Flanigan et al. [2014], Wang et al. [2015a] use AMR fragments as their smallest unit, which may consist of more than one AMR concept. The example in Figure 4.1 shows an example for the fragment that represents "NATO", reproduced from Chapter 2. The internal structure to this fragment is invisible to either algorithm during learning or execution.

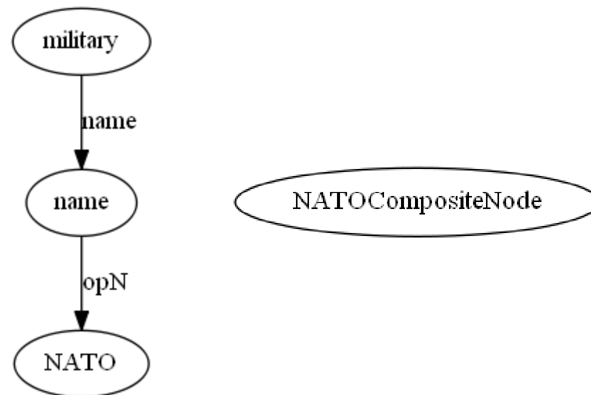

Figure 4.1: Three-node fragment on the left is represented by a single composite node by Flanigan et al. [2014], Wang et al. [2015a]. On termination of the algorithm, the `NATOCompositeNode` is replaced with the three-node fragment in the final output.

We do not use AMR 'fragments' as our building blocks, but instead work at the level of the individual AMR node. In the NATO example, this means that we do not align the single English word 'NATO' to the three node fragment of `NATO-name-military`, but simply to the AMR concept `NATO`. We then learn to build up the graph, by inserting `name` and `military` concepts. We use a novel parsing algorithm that incorporates an Insert action to do this.

This more granular approach means that we are not as restricted as previous work to the fragments that existed in the training set, as we should be able to learn to use just parts of 'fragments', and learn the underlying patterns to these. Figure 4.2 is another example, from which we could potentially learn the general situations in which the `university` and `organization` concepts would need to be inserted, even if the specific named examples are different. In both the JAMR and Wang & Xue initial approaches this is not possible, as the "Naif Arab Academy for Security Sciences", and "Arab Interior Ministers' Council" are only available as self-contained graph fragments, or as fixed rules to generate fragments from common named entities, with no generalisation of their contents possible.

4.2 Pre-processing and initialisation

While we do not match sequences of words to AMR graph fragments we do undertake some pre-processing on the English sentences, primarily to deal efficiently

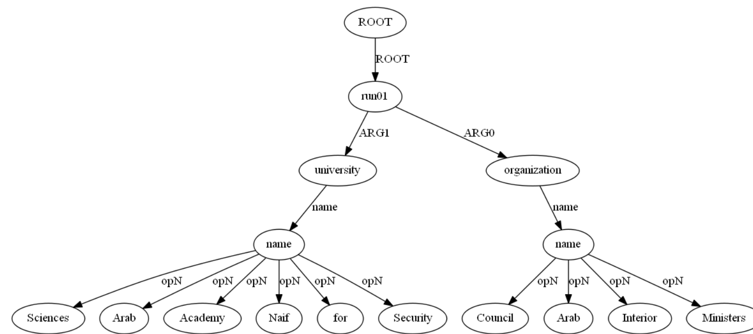

Figure 4.2: AMR Graph of the sentence "Naif Arab Academy for Security Sciences is run by an Arab Interior Ministers' Council".

with dates and numbers. The pre-processing steps are:

- Pass the full sentence through the Stanford Dependency Parser to construct a dependency tree [Manning et al., 2014], including annotation on parts-of-speech, named entity recognition, lemmas and dependency labels (all used as Features in Section 4.2). We use v3.3.1 of the Stanford Parser for comparability with Wang et al.

- Any tokens representing punctuation marks are then removed. (During cross-validation, it was determined that leaving punctuation marks in the starting dependency tree did not improve performance - see section 6.1.1.)

- Match any month name ('January', 'Jan', 'March' etc.) and replace it with the month number `mm`.

- Match any numeric strings of format `ddmmyy` or `dd-mm-yy` and change these to `dd mm yyyy` to provide a set of three numeric tokens from which the AMR `date-entity` structure can be learned.

- Match any number string between 'one' and 'twelve' and replace it with the relevant numeric digits.

- Match any string of 'thousand', 'million' or 'billion' immediately preceded by a number, and replace both word tokens with the numeric amount - e.g. "2 thousand" becomes "2000".

In AMR the convention is that any amount is expressed in digits, regardless of the form in the text, and this pre-processing enables these amounts to be used directly in the AMR graph.

We follow Wang et al. [2015a] and initialise the main algorithm with a stack of the nodes in the dependency tree, with the root node at the bottom of the stack, and the leaf nodes furthest from the root at the top. This stack is termed σ. A second stack, β is then initialised with all children of the top node in σ (hence at initialisation, β will be empty as we always have a leaf node at the top of σ). The state at any time is described by σ, β, all previous actions taken, and the current graph (which starts as the dependency tree). Each action will manipulate the top nodes in each stack, which we call σ_0 and β_0 respectively. We reach a terminal state when σ is empty.

It is important to understand that we take a dependency tree and mutate it into an AMR graph. At any stage before termination some of the nodes will be labelled with words from the sentence, and others with AMR concepts. In an extreme case it is admissible for the algorithm to take no actions and output (as an AMR graph) the input dependency tree, treating all the words as AMR concepts, and all the dependency labels on the edges as AMR relations. When the algorithm refers to edges or nodes they refer to the current state of the graph being manipulated, which will be a hybrid of a dependency tree and an AMR graph. Unless clearly stated otherwise, any mentions of edge or node labels can therefore be either from the dependency tree or the AMR vocabularies.

4.3 Action Space

The actions in the space are summarised in Table 4.1, with details in the following sections.

4.3.1 NextNode, NextEdge and Delete

NextNode and NextEdge form the core of the algorithm. We progress over all nodes from the bottom of the tree up, first labelling the outgoing edges with an AMR relation using NextEdge, and then labelling the node with an AMR concept

Table 4.1: Action Space for the transition-based graph parsing algorithm

Action Name	Param.	Pre-conditions	Outcome of action
NextEdge	l_r	β non-empty	Set label of edge (σ_0, β_0) to relation l_r. Pop β_0.
NextNode	l_c	β empty	Set concept of node σ_0 to concept l_c. Pop σ_0, and re-initialise β.
Swap		β non-empty	Make β_0 parent of σ_0 (reverse edge) and its sub-graph. Pop β_0 and insert β_0 as σ_1.
ReplaceHead		β non-empty	Pop σ_0 and delete it from the graph. Parents of σ_0 become parents of β_0. Other children of σ_0 become children of β_0. Insert β_0 at the head of σ and re-initialise β.
Reattach	κ	β non-empty	Pop β_0 and delete edge (σ_0, β_0), and attach β_0 as a child of the node κ. If κ has already been popped from σ then re-insert it as σ_1.
DeleteNode		β empty; σ_0 is leaf node	Pop σ_0 and delete it from the graph.
Insert	l_c		Insert a new node δ with AMR concept l_c as the parent of σ_0, and insert δ into σ.
RevPolarity			Inserts a new node δ with AMR concept "-" as the child of σ_0, and label the edge (σ_0, δ) with `polarity`.
Reentrance	κ	Phase 2	Insert new edge (σ_0, κ). Then apply the best NextEdge action.

with NextNode before moving to the next node in the σ stack. Figure 4.3 shows an example of NextNode and NextEdge actions, and these are unchanged from Wang et al. [2015a]. Each NextNode action is parameterised with l_c, the AMR concept to be used as the label, and each NextEdge action is parameterised with l_r, the AMR relation to be used. We discuss this parameterisation in Section 4.3.5.

Delete removes a leaf node completely from the graph where the word does not map to any AMR concept. An example is included in Figure 4.3.

4.3.2 Swap, Reattach and ReplaceHead

These actions change the overall structure of the graph, but always retain a tree structure provided they start from a tree.

- Swap reverses the direction of an edge, so that β_0 becomes the parent of σ_0 and its sub-graph. An example is shown in Figure 4.4.

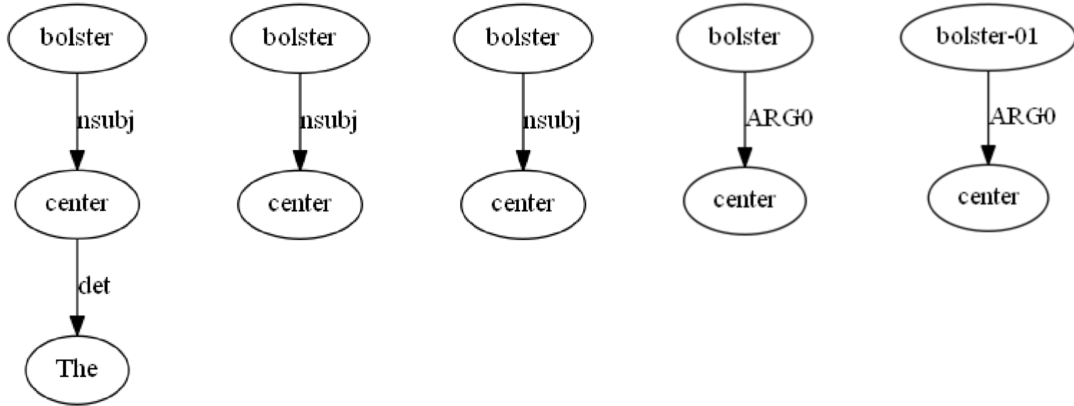

Figure 4.3: The five graphs show parts of successive states starting from a dependency tree (on the left). The actions from left to right are to Delete "The"; Label the "center" node as `center` with NextNode; Label the "nsubj" edge as `ARG0` with NextEdge; Label the "bolster" node as `bolster-01` with NextNode

- Reattach takes the sub-graph starting at β_0, detaches it from σ_0 and moves it to a new parent κ. An example is shown in Figure 4.5.
- ReplaceHead removes a node from the graph that is not a leaf node (in which case the Delete action would be used). An example is shown in Figure 4.6.

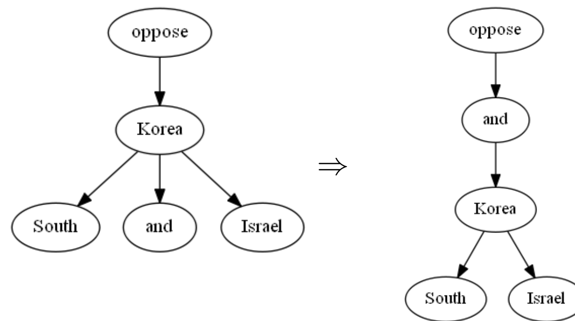

Figure 4.4: Example of Swap action for "...oppose South Korea and Israel". We need "and" to be the parent of South Korea and Israel, and Swap moves it to be the parent of the whole sub-graph.

Unlike Wang et al. we do not parameterise Swap or Reattach actions with a relation label. We leave that decision to a later NextEdge action. We permit a Reattach action to use parameter κ equal to any node within a distance of six edges from σ_0, excluding any node in the sub-graph of β_0 to avoid disconnecting the graph and creating loops. This is slightly more than Wang et al. use, and we found

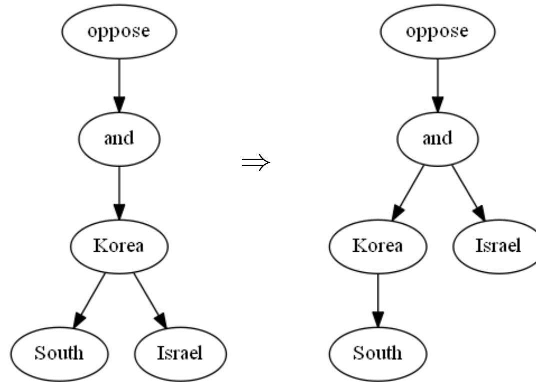

Figure 4.5: Example of Reattach action. This follows on from Figure 4.4, and we Reattach the "Israel" node to be a direct child of "and".

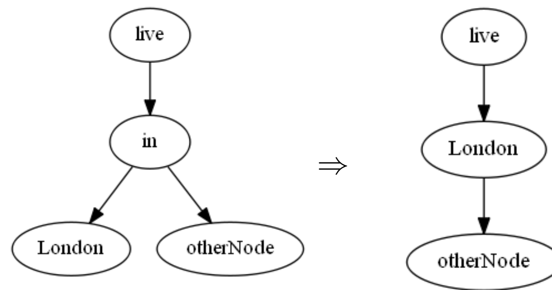

Figure 4.6: Example of ReplaceHead. "in" is not in the final AMR graph, and needs to be removed. It is not a leaf node, so we use ReplaceHead to merge it into "London".

the increase helpful to cope with the larger graphs we have given the avoidance of 'fragments' that merge multiple AMR Concept nodes in the parsed graph.

Our ReplaceHead covers two distinct actions in Wang et al. [2015a]; Replace-Head and Merge. The Merge action merges σ_0 and β_0 into a composite node, that keeps all the words represented in the final AMR graph. This is not required in our approach as we do not have composite nodes and retain a 1:1 mapping between nodes and AMR concept.

4.3.3 Insert and ReversePolarity

The Insert action inserts a new node as a parent of the current σ_0. The action is parameterised with l_c, the AMR concept for the inserted node. When a node is inserted, we set the lemma equal to the AMR concept, to be used in features for future actions. An example is shown in Figure 4.7.

Negation in AMR is represented by a concept of "−" and a relation label of

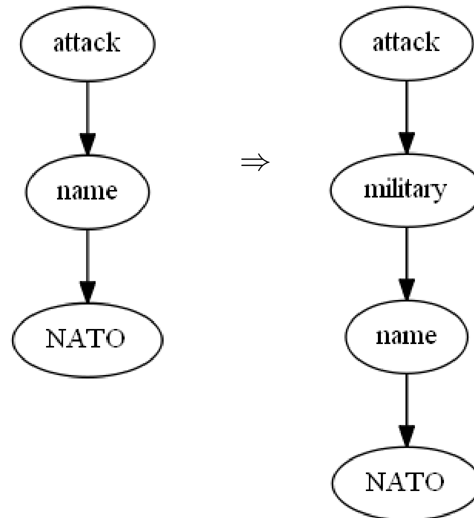

Figure 4.7: Example of Insert. We insert a new node with a label of `military` to create the full AMR representation of "NATO".

`polarity`, and this often requires a new leaf node to be inserted. JAMR and Wang et al. resolve this by labelling words like "not" with the AMR "−" concept; or by converting prefixes of "un-", or "non-" into distinct word tokens during the starting dependency tree in pre-processing that can then be labelled. We instead use the ReversePolarity action to negate a concept. An example is shown in Figure 4.8.

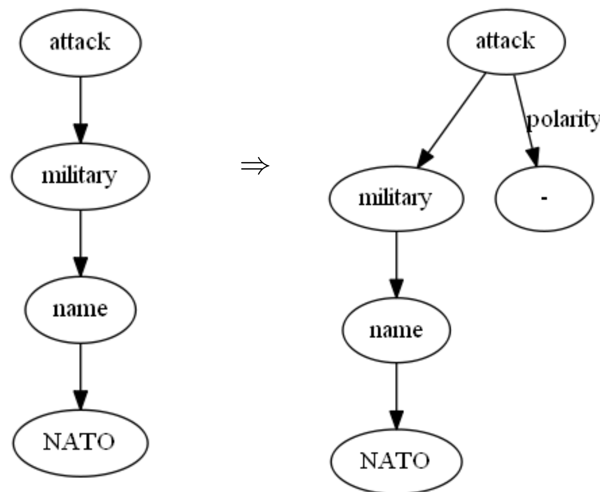

Figure 4.8: Example of ReversePolarity. Inserting a "−" node indicates that the attack node is negated.

There are other leaf nodes that would ideally be inserted to obtain the correct AMR graph from a training sentence, and these cannot be perfectly constructed by

us, or by Wang et al. A review of these scenarios shows they are predominantly due to alignment issues and an example is shown in detail in Section 4.7. Generalising ReversePolarity to insert any AMR concept as a leaf is a possible extension of the Action Space for future work.

In follow-up work Wang et al. [2015b] introduce a new 'Infer' action that was not in their original action space, and that is very similar to our Insert action. Infer inserts an AMR concept node above the current node as Insert does, but is restricted to nodes that occur outside of AMR 'fragments', which continue to be the base building block. The Wang et al. Infer action would not therefore insert the node in the NATO example of Figure 4.7 as this is part of an AMR fragment.

4.3.4 Reentrance

Reentrance is the one action that will turn a Tree into a non-Tree graph, and an example is given in Figure 4.9.

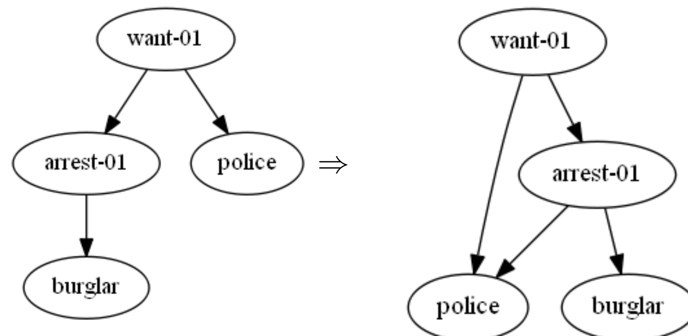

Figure 4.9: Example of Reentrance. The "police" both want the burglar to be arrested, and will make the arrest, so it is an argument of both "want" and "arrest"

Unlike Wang et al. we do not consider Reentrance actions at the same time as the other actions, but have a second pass ("Phase Two") through the final AMR graph once the first pass has reached a terminal state. In this second pass we consider each node as σ_0 in turn, and each node within a range of four as a possible κ for which we should insert a new edge (σ_0, κ). We do not label the edge in the same action (another difference to Wang et al.), but immediately follow any Reentrance decision with a NextEdge action to label the created arc. This second pass is implemented primarily to simplify the first pass, during which we can assume

that the graph is always a Tree. We also found that Reentrance makes only a small difference in the final F-Score, and many of the main experiments were run without the second Reentrance phase switched on; this is made clear in the relevant experimental results.

4.3.5 Parameters used in actions

We differ from Wang et al. in terms of which AMR relations and concepts we consider as parameters (l_c and l_r). They use all AMR relations that appear in the training set, and all AMR concepts that appear in any training set sentence that contained the same lemma as that of the word represented by σ_0. Given our use of imitation learning algorithms that generate RollOuts for each possible action that could have been taken at *each* step of the RollIn trajectory as detailed in Section 2.4, adopting this label set creates severe time performance issues, as on the full training set it gives 600+ possible actions for a large proportion of nodes, and about 80 relations that need to be considered for each edge. Trajectory lengths are commonly in the range 20-140, and the need to explore every possible one of these options at every step takes a prohibitively long time.

Instead we run the expert policy over the training set as part of pre-processing, and track for each lemma in the input which AMR concepts the expert used as labels. This sub-set for each lemma provides the possible set of parameters l_c that will be used for NextNode actions for that lemma. Similarly we track the lemmas at head and tail of each expert-assigned AMR relation, and compile the possible l_r parameters from all AMR relations used by the expert as the outgoing edge from any node with the same lemma as σ_0, plus the incoming edges to any node with the same lemma as β_0. We also allow an `UNKNOWN` parameter for l_c, which uses the actual word from the dependency tree node as the AMR concept; this is particularly helpful in correctly processing new named entities such as people or organizations not present in the training data.

For the l_c parameters on Insert actions, we use all AMR concepts that the expert inserted above any node in the training set with the same lemma as σ_0.

4.3.6 Additional action constraints

Transition-based parsing algorithms have classically relied on a fixed length of trajectory T for guarantees on performance, or at least a bounded T. This is the case in all of Goldberg and Elhadad [2010], Honnibal et al. [2013], Sartorio et al. [2013], McDonald and Nivre [2007]. A problem in our approach is the existence of 'Insert' that can increase the number of nodes, and other actions that revisit previous parts of the graph. This means that unlike the classic labelling case we do not have a fixed T for the number of decisions required, and is theoretically unbounded since the algorithm could Insert nodes *ad infinitum*. Wang et al. do not have 'Insert' in their original action space, but they still note Wang et al. [2015a] that in the worst-case their parser is $O(n^2)$ in the number of nodes; although in practice they observe near linear time to process an input.

Early experiments indicated a problem with training trajectories never completing. This stems from the lack of a fixed number (T) of actions, and of any guarantee that the algorithm will terminate. Two common examples observed in practice involved 'Reattach battles', where a node would be reattached to another point of the graph, and then reattached back to its origin, which could repeat *ad infinitum*; and 'Insert loops', with the constant insertion of new nodes to create chains of unlimited length. An example of the result of an 'Insert loop' is shown in Figure 4.10, with the chain of inserted `person` and `have-org-role-91` concepts continuing without cease. In the terminology of Honnibal et al. [2013] we are not *monotonic*, and later actions can undo earlier actions.

Wang et al. [2015a] had a similar problem with Swap actions, and inserted a feature for the number of times a given edge had been Swapped to control this. We found that while these types of features did mitigate the problem, they did not do so sufficiently and training iterations failed to finish. The alternative approach adopted is to use hard constraints to prevent these pathological situations. Specifically:

- A Swap action cannot be applied to a previously Swapped edge
- Once a node has been moved by Reattach, then it cannot be Reattached again

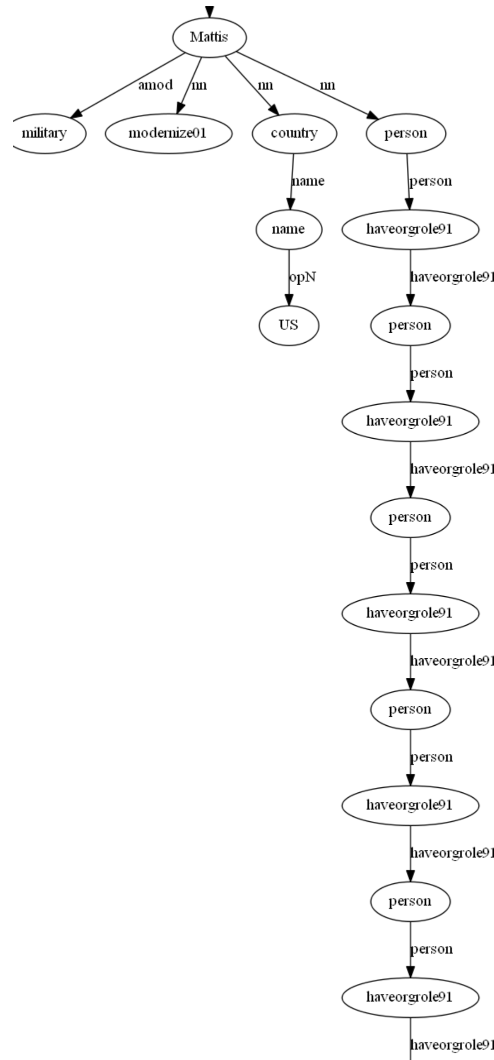

Figure 4.10: Sample output AMR graph in training showing result of an 'Insert Loop'

- An Insert action is only permissible as the first action on a node
- An Insert action is not permissible if it would insert an AMR concept that is already in use as any of the parent, children, grand-parents or grand-children of σ_0

The last of these restrictions does prevent some correct AMR graphs from being formed, but in cross-validation (see Section 6.1.1) this improved performance on the validation set and also prevented training iterations from entering an 'Insert Loop' as shown in Figure 4.10.

The original JAMR work of Flanigan et al. [2014] used a linguistically inspired

Table 4.2: Features considered during cross-validation. *Italicised* items are additions to those used by Wang & Xue. All features are zero-one indicator functions.

Type/Context	Features
σ_0	lemma, dependency label (dl), named entity (ner), part-of-speech (POS), *inserted*
σ_0's parent	lemma, dependency label (dl), named entity (ner), part-of-speech (POS), *inserted*
β_0	lemma, dependency label (dl), named entity (ner), part-of-speech (POS), *inserted*
κ	lemma, dependency label (dl), named entity (ner), part-of-speech (POS), *inserted*
$(\sigma_0 \to \beta_0)$	(lemma, POS), (lemma, dl), (POS, lemma), (dl, lemma), (ner, ner), *(inserted, inserted), relation*, distance, path, distance-path, lemma-path-lemma
$(\beta_0 \to \kappa_0)$	(lemma, POS), (lemma, dl), (POS, lemma), (dl, lemma), (ner, ner), *(inserted, inserted), relation*, distance, path, distance-path, lemma-path-lemma
$(\sigma_0\text{parent} \to \sigma_0)$	(lemma, POS), (lemma, dl), (POS, lemma), (dl, lemma), (ner, ner), *(inserted, inserted), relation*
Actions	*If A_0 is the previous action and A_n is last but n action; A_0, A_1, A_2, A_3, $A_0A_1A_2$, A_0A_1*
Words	Word of $\sigma_0, \beta_0, \kappa, \sigma_0$'s parent if different to the corresponding lemma
Merged nodes	Lemma of all nodes merged into σ_0 or β_0 by a previous ReplaceHead action
Deleted Nodes	*Lemma of all children nodes of this node previously the subject of a Delete action*
Children	*inserted, lemma, relation, (lemma, relation) of each child of σ_0 excluding β_0*

constraint that prevented duplication of argument relations, so that no concept could have two outgoing `ARG1` edges for example. We do not have this constraint, but do only allow actions which preserve acyclicity as JAMR did not (although in practise they found that cycles did not occur). The JAMR constraint that all AMR concepts from the concept identification phase are included in a single connected graph is not relevant to us, as we do not have a separate concept identification phase. We start with a fully connected graph, and none of the available actions can disconnect it.

4.4 Features used

The bulk of the features used are exactly as used by Wang et al. as the most directly comparable work. Additional features added to this set are shown in *italicised* text in Table 4.2. The Wang et al. features using length of span (i.e. the number of word tokens that a given AMR fragment covers) are not used, as this is uniformly one with our non-fragmental approach. Instead we have features to indicate whether a node was inserted, which was not possible for Wang et al.'s parsing algorithm. 'Relation' refers to the AMR relation used to label an edge in a previous action - and is distinguished from the dependency label (dl), which only applies to nodes that were in the original dependency tree from the Stanford parser. The 'path' refers to the sequence of dependency labels traversed to move between the two relevant nodes in the starting dependency tree, and is not defined when either of the two end-nodes is the result of an Insert action.

4.5 Classifier

We use an AROW classifier for all our experiments, without averaging of **w** across iterations. The impact of different classifiers is not a focus of this dissertation. Flanigan et al. [2014] use AdaGrad [Duchi et al., 2011] in the JAMR system, which is a later and potentially more sophisticated development relative to AROW. Wang et al. [2015a] use the simpler averaged Perceptron [Collins, 2002]. We consistently use batch generation of trajectories and instance data, despite the fact that LOLS and DAgger algorithms are on-line algorithms. V-DAgger is a batch version of DAgger, and we formally present LOLS in batch form in Chapter 5.

We also used 'Shenanigan' features as a means of regularisation that have been found to be useful by one of my supervisors (Jason Naradowsky) in some previous imitation learning work. These are features that are unique to a training instance and can never have a value other than zero on any future data. They therefore cannot provide any useful information when training the classifier. They act as an additional form of regularisation on top of the AROW regularisation parameter C, as the training process will assign them part of the credit for any particular action

and update their weights, which will then never be used on future data. Consider the objective function that AROW minimises (2.5). A feature unique to the instance will never have been encountered previously, and will have a very low confidence on its coefficient weight. Changing the 'Shenanigan' feature by a large amount will therefore help optimise (2.5) with little change to the KL term compared to the others that contribute to the decision, and will correspondingly take on a lion's share of the update in training. AROW will treat it no differently to the genuine rare features that provide discriminative power for which the algorithm was designed - the difference here is that it will never be used again. The net effect is a smaller update to the weights that do matter than would otherwise be the case, akin to regularisation.

4.6 The Expert Policy

The expert policy used in training applies a number of heuristics to determine the next action from a given state. It uses the alignments from the JAMR aligner to construct a mapping between nodes in the starting dependency tree, and nodes in the target AMR graph. Any unmapped nodes in the dependency tree will then need to be deleted by the expert, and any unmapped nodes in the AMR graph will need to be inserted. The JAMR aligner maps a sequence of words to an AMR graph fragment, so there is an additional alignment stage local to the expert that maps individual nodes in the AMR fragment to words in the sequence. This is done by calculating the Jaro string distance [Jaro, 1989] between each pairing of AMR concept and word, and then greedily assigning pairs starting with the best match. The Jaro distance is a number in the range $[0,1]$ that indicates how similar two strings are, with 1.0 indicating identity.

From any given state, the expert takes an action from the following rules listed in priority order. In these rules, 'AMR' refers to the Gold AMR graph that the expert is aiming to produce. 'Current' refers to the state of the graph during processing.

1. If we are in Phase Two (the Reentrance phase), and there is an edge to the current node σ_0 in the gold AMR graph that is not in the current graph, then

use the Reentrance action to insert this edge

2. If the current node, σ_0, is mapped to an AMR node, and this AMR node has an unmapped AMR node as parent, then Insert a new node and map this to the unmapped parent AMR node

3. If the current node σ_0 needs to have a "−" node inserted to indicate negation, then apply ReversePolarity

4. If β is empty (i.e. all outgoing edges from σ_0 have already been labelled), and σ_0 has a mapping to an AMR node, then label σ_0 with the appropriate concept using NextNode

5. If σ_0 is a leaf node and has no mapping to an AMR node, then Delete it

6. If σ_0 has no mapping to an AMR node, but β_0 does, then apply ReplaceHead to merge σ_0 into β_0

7. If both σ_0 and β_0 map to AMR nodes, and there is an AMR relation $\sigma_0 \to \beta_0$, then label with the appropriate relation using NextEdge

8. If both σ_0 and β_0 map to AMR nodes, and there is an AMR relation $\beta_0 \to \sigma_0$ then apply Swap to reverse this relationship

9. If β_0 is mapped to an AMR node with a parent that is not mapped to σ_0, then Reattach β_0 to the correct node according to the mapping

10. If β is not empty, then apply NextEdge to label the relation using the current label on the edge $(\sigma_0 \to \beta_0)$]

11. Use NextNode to label the node using an AMR concept equal to the word of the node

The last two actions ensure an action is always possible and may result in relations and concepts in the final AMR graph that are not in the AMR vocabulary. For example an edge might be labelled `nsubj` from the starting dependency label. This will simply reduce the final F-Score.

Table 4.3 details the performance of the expert policy on the 3,988 instances in the training set. Even without Reentrance actions, so that we are restricted to Tree outputs, the expert policy achieves an F-Score of 0.94 using the JAMR aligner. The Recall of this baseline of 0.90 indicates that the major issue is failing to include all of the nodes and edges from the gold target. Permitting re-entrant arcs increases this to 0.92 recall; and using an improved aligner (see Section 4.7) provides a further, independent 0.02 increase in recall. Finally, lifting the restriction on Reattach so that it can explore the whole graph instead of only nodes within a distance of six, increases Precision by 0.01 to effectively a perfect score, and a further 0.01 gain to Recall. All three of these improvements are independent of each other.

Table 4.3: Accuracy of expert on training set

Conditions	F-Score	Precision	Recall
Baseline - JAMR aligner, no Reentrance action and limit on Reattach to a distance of six	0.941	0.985	0.901
Improved Aligner	0.953	0.984	0.923
Reentrance actions included	0.960	0.986	0.920
Reattach distance restriction lifted	0.952	0.996	0.911
Improved Aligner and Reentrance actions	0.963	0.985	0.942
Improved Aligner, Reentrance actions, and no limit on Reattach distance	0.973	0.996	0.951

These results indicate that our expert, while not perfect, is good enough to train our parser via imitation learning. This is especially true when we take into account that we cannot expect to achieve an F-Score of more than about 0.83 on completely unseen test data; as this is the F-Score between the AMR produced by different trained human annotators [Banarescu et al., 2013, Cai and Knight, 2013]. Wang et al. [2015a] report an F-Score of 0.99 on the training data, but do not provide explicit details on their expert policy.

4.7 Alignment impacts

Figure 4.11 shows an AMR example where the JAMR greedy alignment prevents the expert policy from producing the correct AMR output. The English sentence in question is “NATO CONSIDERS cyber attacks a threat to military and civilian

computer networks after the Estonian Government was struck by cyber attacks in 2007." In this case there are *two* `military` nodes in the target AMR: one referring to networks, and one referring to NATO's type of organisation. The English sentence only contains the word "military" *once*, in the context of networks.

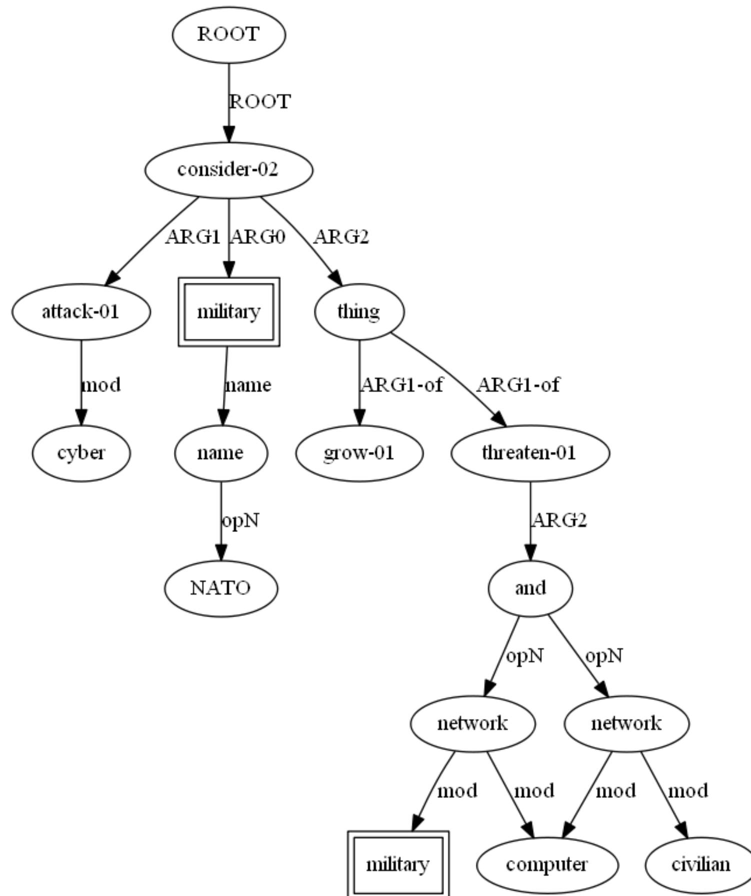

Figure 4.11: Gold AMR graph for "NATO CONSIDERS cyber attacks a threat to military and civilian computer networks after the Estonian Government was struck by cyber attacks in 2007." The two `military` nodes are highlighted as boxes.

The JAMR aligner greedily links "military" in the English sentence to the first matching AMR concept found in a root-first graph search, which in this case is (incorrectly) the NATO-context `military`. As a result the expert Reattaches the "military" node in the Dependency Tree to be the parent of the "NATO" sub-graph, rather than the correct action of Inserting a brand new `military` node as the parent. With the JAMR aligner we therefore obtain a terminal state that is missing the leaf `military` node in Figure 4.11.

As well as getting this specific training example wrong, the action of moving a node half-way across the sentence is unlikely to provide any useful information that can be generalised to unseen examples. Additionally, as noted in previous work by Flanigan et al. [2014], Pust et al. [2015], Peng et al. [2015], improving the alignments can increase performance. We therefore develop a new improved aligner (Algorithm 4) by updating the greedy assignment of words to AMR concepts to take into account their relative positions in the AMR and dependency tree graphs. Intuitively we expect words that are close together in the dependency tree (representing syntactic links) to be aligned to concepts that are close together in the AMR graph (representing semantic links).

Specifically we consider each alignment of dependency tree node (DT) to AMR node (AMR), and calculate the Jaro string distance between them [Jaro, 1989]. We then consider any pairing with a Jaro distance of at least 0.75. For an initial pass we assign DT to AMR node mappings greedily by Jaro distance. This is very similar to the JAMR aligner, which uses a perfect string match, and then a 'fuzzy' match based just on at least the first four characters of the DT word and AMR concept.

Data: Graphs A, B with nodes $a \in A$, $b \in B$
Result: M a set of mappings $(a \rightarrow b)$ between the graphs
Initialise $M =$ greedy mappings based on Jaro distance;
for $i = 1$ **to** 3 **do**
  for $(a \in A, b \in B)$ **do**
    score$(a,b) =$ similarity$(a,b|M)$
  end
  Initialise $M' = \emptyset$;
  for *score(a,b) in descending order* **do**
    if $(a \rightarrow *) \notin A$ *and* $(* \rightarrow b) \notin B$ **then**
      $M' \leftarrow (a \rightarrow b)$
    end
  end
  $M \leftarrow M'$
end
Output M

Algorithm 4: Improved Aligner. Graph A is the dependency tree and Graph B the AMR graph.

After this initial pass we calculate a score for aligning each possible pair of

DT-AMR node based on topological similarity (TS); we take the neighbours of the DT node, consider their *current* mappings in the AMR graph, and calculate the mean distance in the graph (for those neighbours which have such a mapping) to these AMR nodes from the AMR candidate mapping. We then calculate an overall similarity for the DT-AMR candidate pairing:

$$\mathrm{TS}(a,b|M) = \frac{1}{|\mathrm{neA}(a)|} \sum_{n \in \mathrm{neA}(a)} \mathrm{distance}(\mathrm{B}(n|M), b) \tag{4.1}$$

where M is the current set of mappings $(a \rightarrow b)$ from DT to AMR nodes, B($b|M$) returns the AMR nodes to which x maps in M, neA(a) returns all neighbours of the node a in DT, and distance($b1, b2$) returns the number of edges between nodes $b1$ and $b2$ in the AMR graph.

$$\mathrm{similarity}(a,b|M) = \left(1 + \frac{10}{TS(a,b|M)}\right) * \left(1 + 10j(a,b)^3\sqrt{d(b)}\right) \tag{4.2}$$

Where $j(a,b)$ is the Jaro distance between AMR concept and DT word, and $d(b)$ is the number of duplicates of the AMR concept. The left-most bracketed term encourages AMR-DT mappings that put nodes close together in AMR if they are close together in the dependency tree, and right-most term encourages close string matches. The constants in 4.2 were set using experiments on the training set. The d term was found to be helpful due to a number of sentences with lists of several countries. When there are six copies of the `country` concept, but only one `Estonia`, then we want the unique concept to be weighted more heavily in scoring.

These changes correct the examples of Figure 4.11, aligning the “military” in the sentence to the *second*, leaf, `military` node in the gold AMR. The impact this improved aligner has on our results is detailed in Section 6.1.1.

4.8 Unseen Lemma replacement

One of the main identified problems when learning to parse an English sentence into AMR is the paucity of AMR concepts and English words in the training data. A learned parser can only learn to predict an AMR concept if it has training data

that includes this as Flanigan et al. [2014] noted in their original work. We partly address this with the `UNKNOWN` parameter for NextNode, which uses the actual English word as the AMR Concept. Werling et al. [2015] use a similar approach, and extend it to a dictionary lookup to cope with verbs, which are one key area where the English word will not suffice directly, due to the '-nn' post-fix to indicate the relevant FrameSet (e.g. the AMR concept for 'leave' will be of the form `leave-01`).

We try a variant to investigate whether there may be some information in words in the validation data that had not been present in the training data. The underlying idea is that a rare word in the test data, say 'vacate', could be replaced with the word from the training data that is closest to it in meaning - so that 'vacate' could be replaced with 'leave', before processing the sentence. This potentially allows some contextual information to be included by the parser on the basis that sentences with 'leave' as the verb in the training data have some structural similarities to a sentence with 'vacate' as the verb. Given the absence of any information at all on 'vacate', our hypothesis is that this should not make matters worse.

We try two separate approaches for this replacement of unseen words. In both cases we focus on the lemma of the word rather than the word itself, because the lemma is used in feature calculation while in the final selected feature set the words are not. After lemmatisation by the Stanford parser in the pre-processing step (see Section 4.2) we replace any lemma in the test or validation data with the closest lemma from the training data. We do not change the word itself, so that a NextNode(UNKNOWN) action will still use the original word.

Firstly we use GloVe, which provides vector representations of words based on word-word co-occurrence training on billions of words taken from Wikipedia [Pennington et al., 2014], so that words close together in GloVe space should be close together in meaning and usage patterns. GloVe combines elements of matrix factorization and the local context window methods used in the original Word2Vec approach [Mikolov et al., 2013]. We use the lowest (50) dimensional set of GloVe vectors[1], as tests showed no difference in the replacement words selected when

[1] http://www-nlp.stanford.edu/data/glove.6B.50d.txt.gz

using higher dimensional vectors.

Secondly we use data from the WordNet database [De Marneffe and Manning, 1998]. In this case we look up the missing lemma in WordNet to obtain its SynSets, each of which is a list of synonyms for a particular word use. We then pick the most common SynSet, and replace the unseen lemma with any previously seen lemma in the training data that is also in the SynSet. This second approach replaces many fewer lemmas in test data, as the WordNet vocabulary is much smaller than GloVe's and even lemmas that do exist in WordNet may not have any previously seen element in their most common SynSet. In contrast GloVe is trained on the contents of Wikipedia in 2014, and any unseen lemma will always have a seen lemma that is nearest to it, even if the match is not brilliant. In practise this means that GloVe replaces the lemmas for a large proportion of named entities, while the WordNet approach only replaces unseen nouns, verbs and other standard parts of speech. The results of lemma replacement are reported in Section 6.1.1.

Chapter 5

Imitation Learning approach

5.1 Combinations of LOLS and V-DAgger

5.1.1 LOLS in batch mode

Algorithm 5 documents our implementation of LOLS and is amended slightly to fit the same pattern as V-DAgger. The changes from the original Algorithm 3 are batch learning with no averaging of policies, and a decay of β, the probability that a RollOut trajectory uses the expert policy.

Our key motivator to investigate LOLS as well as V-DAgger for the AMR task is the switch from step-wise to trajectory-wise stochasticity. Early experiments using V-DAgger observed that if we took three samples for each RollOut trajectory from the same starting state and exploratory action, then these could give three very different terminal states (and hence losses incurred) due to the step-wise stochasticity. This is despite both component policies being deterministic. This gives a high variance in the loss signal used in training, which early experiments suggested was harmful to effective learning. Some results are included in Chapter 6.

A potential advantage of LOLS is that each trajectory uses a single deterministic policy, so we do not have this high variance in the loss signal for RollOuts with the same starting $\langle$state, action$\rangle$. Daumé III et al. [2009] have a similar problem, and note that an approximate cost function outperforms single Monte Carlo sampling, “likely due to the noise induced following a single sample”. We may be able to use LOLS to reduce noise, as an alternative to cranking up the sample size in V-DAgger.

Data: data D, expert policy π^*, Loss function $F(\mathbf{y})$, feature function f, decay rate δ
Result: learned policy $\hat{\pi}_{N+1}$
Initialise $E = \emptyset$;
Initialise $\hat{\pi}_1 = \pi^*, \beta = 1.0$;
for $i = 1$ **to** N **do**
  for $d \in D$ **do**
    Initialise L = $\emptyset$;
    Predict trajectory $\omega = \hat{y}_{1:T}$ using $\hat{\pi}_i$;
    for $\hat{y}_t \in \omega$ **do**
      foreach *possible action* $y_t^j \neq \hat{y}_t$ **do**
        Extract features $\Phi_t^j = f(d, y_t^j, \hat{y}_{1:t-1})$;
        $\pi_{RollOut} = \pi^*$ with probability β or $\hat{\pi}_i$ otherwise;
        $e =$ trajectory with starting actions $\hat{y}_{1:t-1}, y_t^j$ through to a terminal state using $\pi_{RollOut}$;
        $L_t^j = F(e) - F(\omega)$;
      end
      Add (Φ_t, L_t) to E;
    end
  end
  Train $\hat{\pi}_{i+1}$ using full set of experienced data E;
  $\beta = (1-\delta)\beta$;
end

Algorithm 5: Batch LOLS (Locally Optimal Learning to Search)

5.1.2 Combinations

Algorithm 6 is constructed to highlight the aspects shared between SEARN, DAgger and LOLS. When generating the trajectories on the training data that will be used for training the learned parser, each of the following need to be specified:

- the policy used to generate the RollIn trajectory (the *RollInPolicy*)
- the policy used to generate RollOut trajectories from each step on the RollIn trajectory, for each possible action that *could* have been taken at that step (the *RollOutPolicy*)
- rules for changing β at each iteration (parameter used to determine impact of expert policy)

Data: data D, expert policy π^*, Loss function $F(y)$, $sampleSize$
Result: learned classifier C
Initialise $C = \emptyset$;
for $n = 1$ **to** N **do**
  Initialise $E_n = \phi$;
  $\pi_{Rollin} = RollInPolicy(\pi^*, C, n)$;
  $\pi_{Rollout} = RollOutPolicy(\pi^*, C, n)$;
  for $d \in D$ **do**
    Initialise L = $\emptyset$;
    Predict trajectory $\omega = \hat{y}_{1:T}$ using π_{Rollin};
    for $\hat{y}_t \in \omega$ **do**
      foreach *possible action* y_t^j **do**
        Extract features $\Phi_t^j = f(d, y_t^j, \hat{y}_{1:t-1})$;
        for $s = 1$ **to** $sampleSize$ **do**
          Predict outward trajectory $\hat{u}_{t+1:T}$ using $\pi_{Rollout}$;
          $L_t^j = F(\hat{u}_T)$;
        end
      end
      $LowestCost = \min L_t^j$;
      foreach j **do**
        $Regret_t^j = L_t^j - LowestCost$
      end
      Add $(\Phi_t, Regret_t)$ to E;
    end
  end
  Train C using experienced data $E_1 \ldots E_n$;
  $C = Train(C, E_1 \ldots E_n)$;
end

Algorithm 6: Generic outline for SEARN, V-DAgger, LOLS

- which RollOut data are fed into the classification learning algorithm to generate a learned policy (part of the *Train* function)

LOLS uses a single policy for each RollOut trajectory - either the learned parser, or the expert. If these two give very different results, then very different losses from very similar starting states will be included in the trajectory data used for training. This will happen if an exploratory step takes the learned parser into an unseen state from which it makes very sub-optimal decisions, which then propagate as suggested by Ross et al. [2011] (2.2) and result in a very high final loss. The expert policy in contrast will take the best action to get back to the target, and give

Table 5.1: Comparison of RollIn and RollOut policies by algorithm

Algorithm	RollIn	RollOut	Train
Simple Imitation Learning	Use the expert policy only	RollOut is not required, as we use the expert binary loss function.	Use all the data from the first (and only) iteration.
SEARN	At each step use current trained policy with probability γ, previous trained policy with probability $\gamma(1-\gamma)$ ...etc., with residual probability for expert policy	As RollIn	Use the data from the current run only to generate a new current trained policy
V-DAgger	At each step with probability β use the expert policy, and with probability $1-\beta$ use the current trained policy	As RollIn	Use all the data from all previous iterations to train the current policy
LOLS	Use the current trained policy only (the expert policy in the first iteration)	With probability β use the expert policy for all steps, otherwise use the current trained policy for all steps	Use all the data from all previous iterations to train the current policy
DetLOLS	As LOLS	Execute one trajectory using the expert only, and one using the trained classifier only. Then take the weighted average of the two Losses - expert Loss weighted by β and trained by $1-\beta$)	Use all the data from all previous iterations to train the current policy
DI-LO	As DAgger	As LOLS	As LOLS
DI-DLO	As Dagger	As DetLOLS	As LOLS
LI-DO	As LOLS	As DAgger	As LOLS

a relatively low final loss for a single-step error. This could give a high variance in loss signal as observed with V-DAgger.

In initial experiments with LOLS, this large discrepancy between expert and learned parser losses was found still to be an issue, especially on large AMR graphs. To reduce this variance in the calculated loss we introduce DetLOLS (Deterministic LOLS) as a simple tweak to LOLS. What DetLOLS does is generate every RollOut trajectory twice; once for the expert, and once for the learned parser. It then takes a weighted average of the two losses using the β parameter to weight the result towards the expert early in training, and reduce this weighting as training proceeds.

We also mix-and-match the RollIn and RollOut policies to form hybrid algorithms. Specifically we try DAgger RollIn combined with LOLS RollOut (giving DI-LO), and LOLS RollIn combined with DAgger RollOut (giving LI-DO). Combining V-DAgger RollIn with DetLOLS RollOut in the same way gives a DI-DLO composite algorithm. The differences between all these algorithms are summarised in Table 5.1 and Algorithm 6 implements all of them. Experimental results comparing all these variations are in Section 5.1.

5.1.3 Expert policy updates

With all these algorithms the expert needs to make decisions based on *any* state, including those where previous actions might have been taken by a learned policy. This was not required in the simple imitation learning of Algorithm 1, as we never ventured off the expert trajectory. In the terminology of Goldberg and Nivre [2012] our expert must be *complete*.

This is achieved by updating the mapping between current graph and the target AMR after each action regardless of whether the action was taken by the expert or not; this then provides the data the expert needs on a later step to make a decision. The only mapping update that is required is after an Insert action, and the newly inserted node is mapped to any unmapped target AMR node with the same concept that is also a parent of σ_0. If no such node exists, then the newly inserted node is left unmapped.

5.2 Smatch as a non-decomposable loss function

To apply V-DAgger, or any approach that fits the pattern of Algorithm 6 to the AMR problem, we need a loss function that can be calculated for any terminal state. This is in contrast to the simple imitation learning of Algorithm 1, which uses a binary expert loss function - a loss of 0.0 for the exact action taken by the expert, and a loss of 1.0 for any other action. In this case of binary expert loss, we do not need to unroll each possible action to a terminal state.

A downside of binary expert loss is that we cannot distinguish between non-expert actions with different terminal states; some mistakes may cause much less harm than others to the end result, and there may be multiple equally valid actions leading to different paths to the same correct answer. For example, starting from the word 'Estonia' in the dependency tree, the expert policy will first insert a new `name` node, then label the leaf node as `Estonia`, then insert a `country` node above the previously inserted `name` node. However it would be equally correct to insert `country` first, and then immediately insert `name` before labelling the leaf node as `Estonia`. Binary expert loss will mark insertion of `country` as incorrect, even though were we to unroll to a terminal state this action could give the correct answer. See Figures 5.1 and 5.2.

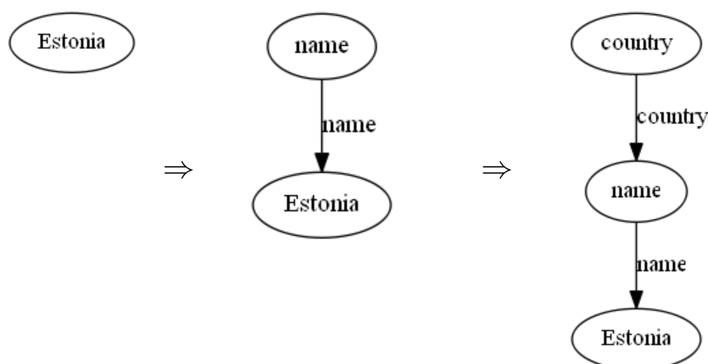

Figure 5.1: Example of Insertion of nodes starting from the word "Estonia" to obtain the AMR country representation

Another disadvantage of binary expert loss is that it is difficult to learn to be better than the expert policy, as the expert policy also defines the loss function used during training. Using a loss function independent of the expert would allow us to

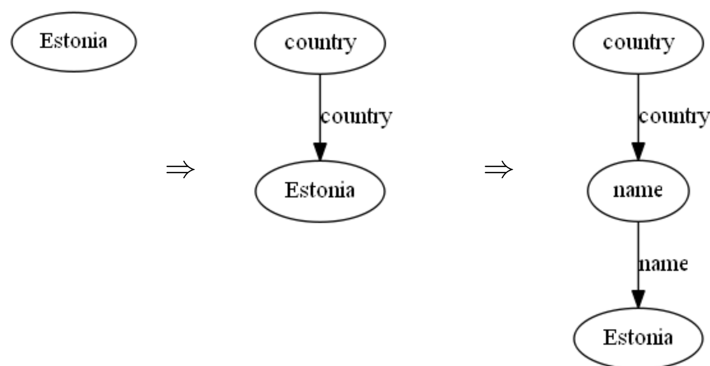

Figure 5.2: Alternative processing of "Estonia" to the same AMR representation

learn actions that are better than the expert's suggestion for a given state. This may not be a problem if the expert is optimal (or close to), but for sub-optimal experts this will provide an upper limit on the success we can achieve.

Calculating (1.0 − F-Score) for the terminal state is the obvious loss function to use, as the F-Score is exactly what we are trying to maximise. However this is NP-hard to calculate, with complexity exponential in the size of the input graphs as discussed in Section 3.2. The heuristic short-cuts of the Smatch algorithm of Cai and Knight [2013] partially get around this, and Smatch is perfectly adequate for approximating the final F-Score on the test and validation sets once training is complete.

However we encounter difficulties when we use Smatch directly as the training loss function. A RollIn trajectory is of the order of 100 steps to get to a 30-node AMR graph, and for each step we have up to 20 actions (once we have reduced the size of the permissible actions as discussed in Section 4.3). For each of these we have to generate a RollOut trajectory to a terminal state. Hence processing one training sentence can require 2000 or more Smatch calculations. This is half the size of the full training set, which takes about 16 minutes to process via Smatch on a single core (see Table 5.2). The full training set would therefore take of the order of 32,000 minutes for one training iteration just to calculate the loss function results, or 20 to 25 core-days. This performance rapidly worsens when a trajectory leads to large graphs, such as the pathological example shown in Figure 4.10; or if we need to take multiple samples to reduce variance in the estimated loss. As a

result it is necessary to find a faster loss function than Smatch for training.

The original Smatch algorithm of Cai and Knight [2013] is shown as Algorithm 7. The central idea is that we start with a reasonable initial match of nodes between the two AMR graphs by aligning greedily nodes that have the same concept. We link other nodes in this initial mapping randomly. We then consider all one move changes to M (see Algorithm 7 for *possibleMoves*), and calculate the F-Score improvement of each such change. We greedily select the best move, apply this to M to obtain an improved candidate mapping and iterate until there is no improvement in F-Score. Given the stochastic nature of the initial assignment, we repeat a number of times with different initialisations. Four iterations is the number that Cai and Knight [2013] settle on as providing near-perfect accuracy on their datasets compared to using brute-force enumeration of all possible mappings.

Data: AMR Graphs A and B; wlog A is the larger of the two graphs
Result: F-Score F of best mapping of nodes between A and B
for $i \leftarrow 1$ **to** 4 **do**
  Initialise Mapping M as set of pairings $a \rightarrow b$ between nodes in A and B. We link nodes based on greedy alignment for exact matches on labels, and then randomly link remaining nodes;
  repeat
    possibleMoves = all possible swaps in M: $(a1 \rightarrow b1), (a2 \rightarrow b2) \Rightarrow (a1 \rightarrow b2), (a2 \rightarrow b1)$;
    possibleMoves = *possibleMoves* + all possible assignments of unmapped nodes in A to a node in B: $(a1 \rightarrow \phi), (a2 \rightarrow b) \Rightarrow (a1 \rightarrow b), (a2 \rightarrow \phi)$;
    for $move \in possibleMoves$ **do**
      Calculate improvement in F-Score of $M + m$ (see Section 3.2);
    end
    $m = move$ with best improvement in F-Score ;
    Update M with m ;
  until *m does not increase F-Score*;
  $F(i)$ = F-Score of M ;
end
F = maximum of all $F(i)$;

Algorithm 7: Smatch calculation of F-Score

We can modify the heuristics to reduce the amount of exploration undertaken in various ways. A first change we can consider is to only look at N of the total *possibleMoves*, picked randomly, and then stop if none of these increase the score.

Secondly, the calculation of F-Score for *every* possible move only to pick one of them seems wasteful, so we try a greedier policy of applying a move m as soon as we find one that improves the score, and break the **repeat** loop early, in a process of *satisficing*.

A third change is to improve the initial mapping to take account of topological information. We adopt an approach similar to that for our improved AMR to English aligner in Section 4.7 with the same Algorithm 4, except that in the first pass use the Smatch initialisation. In further iterations we amend this based on the string similarity between node labels, and a measure of their topological similarity (TS). This gives us different functions for TS$(a,b|M)$ and similarity$(a,b|M)$, listed below. For a potential pair mapping $(a \rightarrow b)$ in the context of a previous full mapping M, we define TS as the count of the neighbours of a in graph A that map to neighbours of b in graph B. If neA(x) is a function returning the neighbours of node x in graph A, and B(x) is a function returning the node in graph B that maps to x in graph A, then

$$TS(a,b|M) = |\mathrm{B}\left(\mathrm{neA}(a)\right) \cap \mathrm{neB}(b)| \tag{5.1}$$

In each subsequent iteration, we consider every possible pairing $(a \rightarrow b)$ and score each using:

$$\text{similarity}(a,b|M) = (1.0 + TS(a,b|M)) * \left(1.0 + 5\frac{\delta(a,b)}{2^{c(a)-1}}\right) \tag{5.2}$$

where $\delta(a,b)$ is 1 if the labels of a and b are identical and 0 otherwise, and $c(a)$ is the count of the duplicates of the label of a across both graphs. We then greedily add pairings with the highest similarity scores to form a new mapping M'. Once a node from either graph has been included, we ignore any lower-scored pairings that include it. The constants and form of (5.2) were determined from experiments on the training set.

The results of these amendments to the vanilla Smatch algorithm are in Table 5.2. Smatch was run on all 3,988 instances of the Training Set, comparing the gold AMR with that generated by running the expert policy of Section 4.6 on

Table 5.2: Smatch performance with algorithmic changes

Algorithm	Iter	F-Score	Time taken
Vanilla Smatch	4	0.965	16.7 min
+ satisficing	4	0.965	6.5 min
+ improved initialisation	4	0.963	11.4 min
+ $N = 1000$	4	0.965	11.5 min
+ all three of above	4	0.963	8.8 min
Vanilla Smatch	1	0.960	4.1 min
+ satisficing	1	0.959	1.7 min
+ improved initialisation	1	0.963	2.8 min
+ $N = 1000$	1	0.958	2.9 min
+ all three of above	1	0.962	2.2 min
Naive Smatch	-	-	0.5 sec

the English sentence. Vanilla Smatch provides the highest accuracy score we can hope to achieve, as this conducts the most complete search of possible mappings to find the one that maximises F-Score. Accuracy of the algorithmic changes is hence measured in the drop of F-Score compared to vanilla Smatch.

Both the satisficing and reduced search of possible moves (via N) improve performance (by a factor of 2.5 for satisficing) without any loss of accuracy when we use four random initialisations. The improved initialisation method also provides a performance improvement despite the increased overhead, presumably because less search is required from a more accurate starting mapping. However it is the only one of the three changes that reduces the accuracy on the full training set from 0.965 to 0.963, because the starting mapping is deterministic and stops exploration of the full space of possibilities. We can see this as when using just a single iteration the improved initialisation sees no drop in accuracy compared to multiple random starts, while vanilla Smatch and the other tweaks all do.

As a result of these changes, we can obtain a performance improvement of about 8-fold over the standard Smatch, with relatively little loss of accuracy. This is helpful, but we can improve matters much more if we use an approximation to the Smatch, which we term ‘Naive Smatch’.

5.2.1 Naive Smatch

Even with these improvements, Smatch is still very slow if it is to be run to evaluate every possible RollOut at every step of every RollIn trajectory. We try 'Naive Smatch' as an approximation to Smatch evaluations during training. This skips the mapping of nodes in graph A to nodes in Graph B. Instead for each graph we compile a list of:

- Node labels, e.g. `name`
- Node-Edge-Node label concatenations, e.g. `leave-01:ARG-0:room`

We then calculate precision, recall and F-Scores between the two lists directly. Precision being the proportion of list entries in graph A (the actual output) that can be found in the list for graph B (the target output); Recall being the complement; and F-Score being the harmonic mean of the two (see Section 3.2).

This is lightning-fast relative to the full Smatch calculation as it avoids the combinatorics, taking just 0.5 seconds versus the 16 minutes of vanilla Smatch (see Table 5.2)[1]. This is an improvement of 2,000-fold, and shifts the performance bottleneck from loss function calculation to generation of RollOut trajectories.

To confirm that Naive Smatch is a reasonable approximation, we trained a parser on the Small Training set, and ran this on the full validation set of 2,132 sentences. We calculated Smatch and Naive Smatch scores for the output AMR graphs against the gold AMR, and obtained a positive correlation of 0.97 between Smatch and Naive Smatch scores. The result is shown graphically in Figure 5.3.

If the only labelling problems occur in the names given to edges, then both Smatch and Naive Smatch will give the same result, as both will treat an incorrectly labelled edge as a single mistake. When there are mistakes in a node label Smatch will count this as a single mistake, while Naive Smatch counts an additional error for each edge into or out of the mislabelled node. This accounts for the main structure visible in Figure 5.3. The line of perfect correspondence consists of graphs which only have edges mislabelled, with the majority of points below this line due to the

[1]No F-Score is provided for Naive Smatch in Table 5.2, as this is not comparable to vanilla Smatch as is the case for the other variants

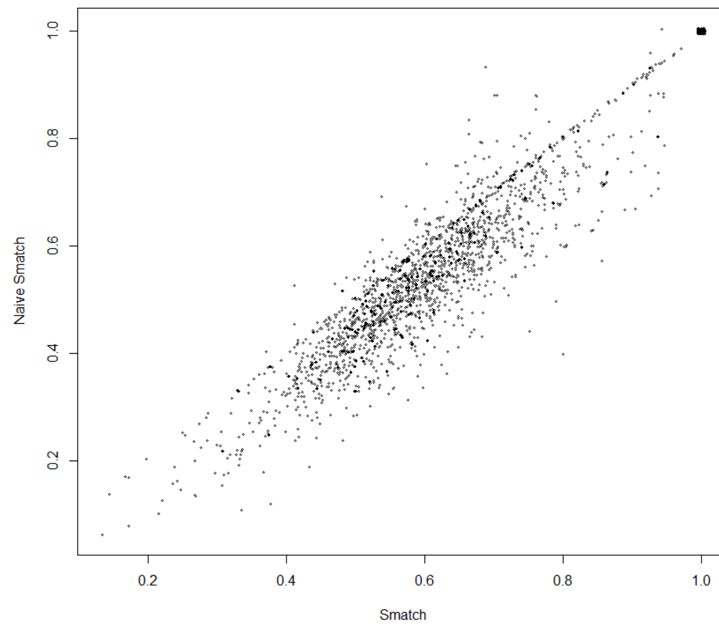

Figure 5.3: Plot of Smatch score against Naive Smatch after parsing the full validation set with a learned parser. Jitter has been added for clarity.

over-counting of node label mistakes by Naive Smatch. The few points where Naive Smatch gives a greater result than Smatch are for graphs with many identically named nodes, such as with a list of countries, for which Naive Smatch will match nodes with the same name, even if topologically dissimilar.

5.2.2 Absolute Smatch

A further potential problem is that Smatch (and Naive Smatch) provides a loss in the range $[0, 1]$, regardless of the size of the AMR graphs being compared. The training corpus contains graphs of widely varying sizes, from four nodes up to around 50; and correspondingly wide variations in the lengths of trajectories, T. This means that the loss is not scaled to the length of the trajectory, and the number of actions that contribute to it. For example a single mislabelled node in a graph of four nodes will give a much higher Smatch loss than a single mislabelling in a graph of 50. The action that mislabels the node is equally wrong in both cases, but using Smatch directly will ascribe a much lower loss to the mistake that takes place in the larger graph.

To control for this effect, we use an absolute Smatch loss measure that is simply the number of mistakes made, without scaling to the $[0,1]$ range. We count the incorrect entries in the output graph (once converted to triples as discussed in Section 3.2), plus the entries in the input graph (once converted to triples) that are not in the output. This number is proportional to the size of the graph.

5.2.3 Penalty Smatch

Smatch and Naive Smatch as discussed so far do not take account of T, the length of the trajectory, but just the accuracy of the final terminal state. Hence a trajectory with $T = 10$ is just as correct as one with $T = 100$ if they reach the same terminal state. In practise we would prefer the first of these as a more frugal use of resources. This is a particular issue due to the lack of any guarantees of termination, and for mistakes to be corrected by later actions. For example 'Reattach Battles' (Section 4.3.6) can increase T arbitrarily while reaching the same terminal state.

In early testing there were examples of this happening, with Smatch or Naive Smatch giving equal losses to RollOuts from many different exploratory actions, as any mistakes could be repaired later to give the same terminal state. Hence we introduce a penalty to the Smatch loss for taking each action. This is implemented simply as:

$$\text{Penalty Smatch} = \text{Absolute Smatch} - \frac{T}{5} \tag{5.3}$$

The coefficient of $\frac{1}{5}$ was selected after experiments on the training set.

Chapter 6

Experiments

This chapter is split into two sections. In Section 6.1 we provide the results of experiments using the training and validation sets to determine the features and parameters to use in the final runs. In all cases we test various settings by training parsers on the training set, and evaluating the result using the performance of the trained parser on the unseen validation set. In Section 6.2 we then use the optimal settings as determined by these validation experiments in final experiments using the full training and validation sets for training, and the previously unseen test set to give a final F-Score. Details of the Small, Medium and Full training sets, and of the validation and test sets are in Chapter 3.

6.1 Validation Results

6.1.1 Feature and Parameter validation

We used the Medium training set and full validation set to determine the sets of parameters to use in Algorithm 1, using an AROW classifier and binary expert loss. The parameters we considered were the inclusion of the various sets of features in Table 4.2, the use of the improved aligner of Section 4.7, the AROW regularisation parameter C (see Section 6.1.2), the inclusion of the optional Insert prohibition of Section 4.3.6, the inclusion of punctuation tokens (commas, semi-colons, brackets etc.) in the dependency tree and the replacement of unseen lemmas in the test set (Section 4.8). All of these results were obtained without the Reentrance action.

'F-Train' is the percentage F-Score that the trained parser obtains on the train-

Table 6.1: Cross-validation results for parameters in simple Imitation Learning

Parameter settings	F-Train	F-Val
Baseline - improved aligner, base features, $C = 1.0$	90.2	62.3
+ Shenanigan Features	-0.2	+2.9
$C = 100$	-0.6	+2.3
+ Action Features	+0.2	+2.2
JAMR Aligner	-3.3	+2.1
+ Insert Prohibition	-1.7	+1.9
+ Child Features	-0.2	+1.3
+ Delete Features	-0.3	+0.7
+ Punctuation tokens	-1.3	+0.4
- Word features	-0.9	+0.1
GloVe Lemma replacement	0.0	-1.8
GloVe Lemma replacement (II)	0.0	0.0
WordNet Lemma replacement	0.0	0.0
+ Shenanigans, $C = 100$	-1.7	+3.6
+ Deletion + Punctuation	-0.7	0.0

ing data used, and 'F-Val' is the percentage F-Score it obtains on the unseen validation set. For clarity, the impact of parameter settings are shown as the change in F with respect to the baseline. The baseline features used are those in the first eight lines of Table 4.2, and the table is sorted in descending order of beneficial impact on the validation score.

The regularisation effect of the 'Shenanigan' features is at least partly independent of that from the smoothing parameter C, as their combination shows. The alignment changes made do improve performance on the training data relative to the JAMR aligner, but worsen it on the validation data. The reason for this is not clear, but for all future experiments the default JAMR aligner was used. We expected punctuation tokens to provide some additional useful information, but the effect is small. Since they also increase the size of the starting dependency tree and the length of parsing trajectories, they were not included in our final settings. Word features have little effect, and were dropped from the final runs.

The final set of parameters selected from these results were: JAMR aligner, Insert Prohibition, Action, Child, Shenanigan and Delete Features with $C = 100$.

The attempt to extract some information from lemmas in the validation data that had not been present in the training data using a GloVe lookup to find the

closest match did not help, and resulted in a large drop in performance. This is due to the high proportion of organisation and individual names in the training/test corpus. For example, the unseen word 'Israel' was replaced with 'Syria', as the closest match from the training data. However this led the trained parser to use the label 'Israel' as the AMR concept, when otherwise it would default correctly to the word itself (using the `UNKNOWN` parameterisation for the NextNode action). One modification was made to the expert policy and action space to counter this, with the expert amended to preferentially use the `UNKNOWN` parameterisation if the English word was a perfect match for the AMR concept. The results after this change are shown as "GloVe Lemma Replacement (II)" in Table 6.1 - and while avoiding the previous problem, do not provide any benefit.

6.1.2 AROW Parameters

Figures 6.1 and 6.2 show the effect of changing the AROW regularisation parameter on a full V-DAgger run, with decay parameter $\beta = 0.10$, and 10 training iterations for each V-DAgger iteration. Each line represents the F-Score the trained parser achieves after the nth iteration on the training set and validation sets respectively. The lighter the line, the heavier the smoothing. We naturally expect the performance on the training set to be higher, and the regularisation effect should reduce performance here as it increases. Assuming that we have a sufficiently rich set of features we expect the performance on the validation set to be low for small C due to overfitting to the training set, improve to an optimum level as regularisation increases, and then decline beyond this.

As the data show, a regularisation parameter of 10 or greater is required to improve performance by the learned parser on the unseen validation set. Regularisation up to at least 10^5 still gives good results, but gives starkly worse performance on the training set as expected. We choose $C = 100$ as our selected parameter, as there is no clear improvement for higher values.

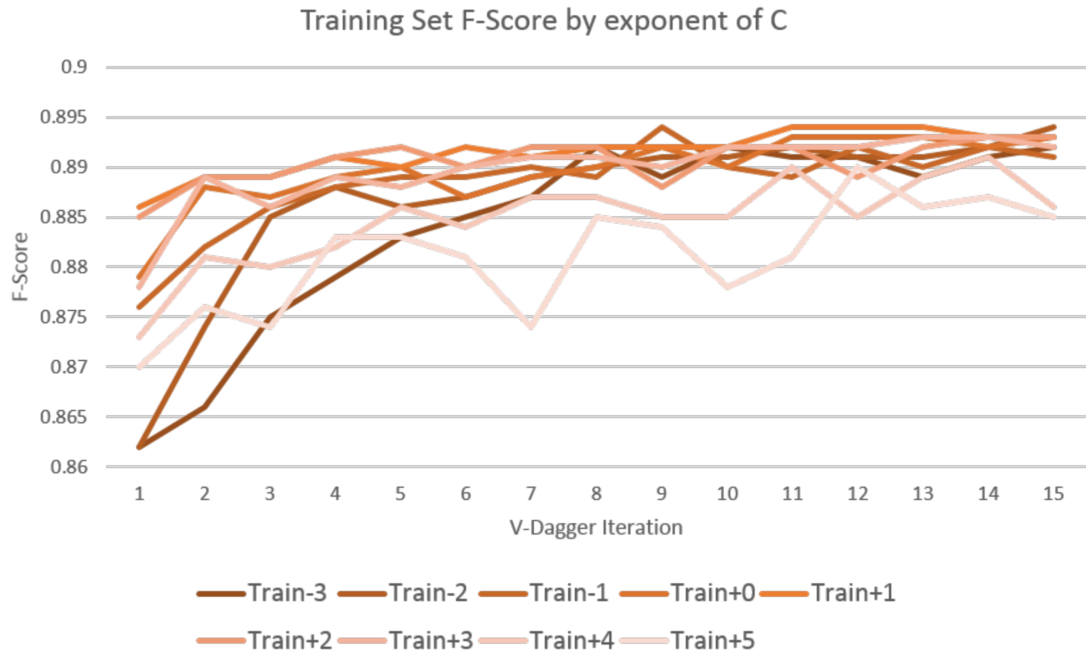

Figure 6.1: Impact of F-Score on Training Set with AROW Regularisation parameter. The line for Train-n represents $C = 10^n$.

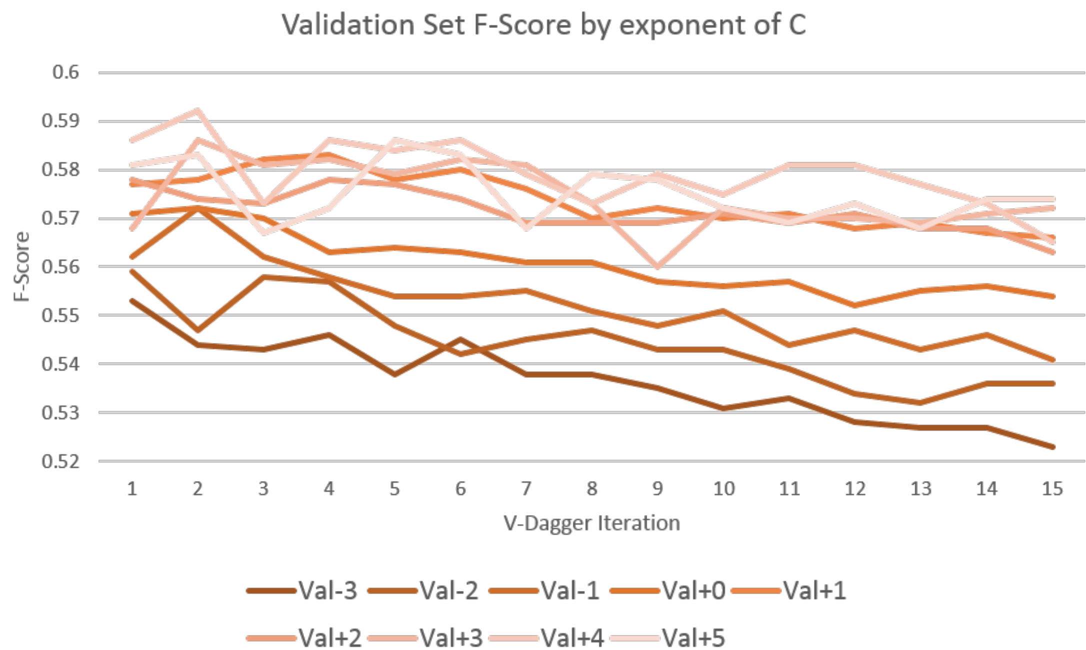

Figure 6.2: Impact of F-Score on Validation Set with AROW Regularisation parameter. The line for Val-n represents $C = 10^n$.

6.1.3 Loss function comparison

Figure 6.3 shows results on DI-DLO runs using Naive Smatch, Naive Absolute Smatch, and Naive Absolute Penalty Smatch (see Section 5.2). A decay rate of 0.03 was used, and all parameters and features according to the results of Section 6.1.1. There is little difference between Naive Smatch, and Naive Absolute Smatch apart

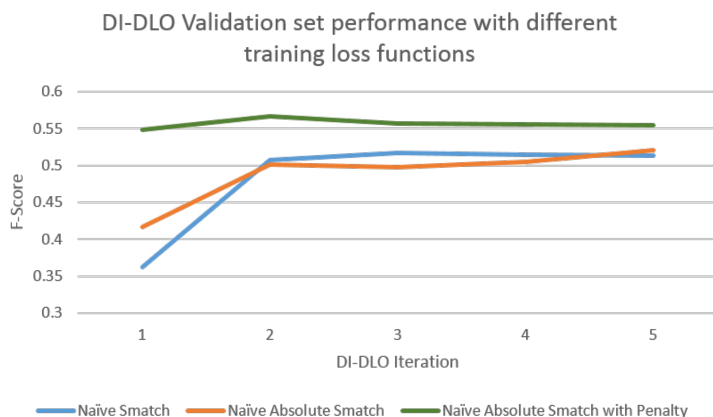

Figure 6.3: Comparison of Loss Functions (Naive Smatch, Naive Absolute Smatch, and Naive Ansolute Penalty Smatch) with DI-DLO. F-Score evaluated on full validation set.

from the first iteration, when the Absolute version is better. Including the penalty term gives a large increase in performance, and Naive Absolute Penalty Smatch is used as the training loss function for all other experiments. The full Smatch of Cai and Knight [2013] is used to calculate the final F-Scores; Naive Smatch is *only* used during training.

6.1.4 Algorithm choice: combinations of V-DAgger and LOLS

We compared V-DAgger, LOLS and the variant algorithms documents in Table 5.1. These experiments were all run over 6 iterations using the Small training set, a decay parameter of $\delta = 0.02$, and evaluated on the full validation set. Parameter settings, other than ones specific to the imitation learning algorithm, were those selected from validation using simple imitation learning in Section 6.1.1 with the exception that Shenanigan features were not used. DI-DLO provides the best results as shown in Figures 6.4 and 6.5. We therefore use DI-DLO for the final run.

Note that the three algorithms (LI-DO, LOLS, DetLOLS) that use the LOLS

RollIn approach (learned parser from previous iteration only with no expert involvement) all perform worse in the second iteration than the three that use the V-DAgger RollIn (which takes 98% of the steps using the expert policy and just 2% using the learned policy). LOLS RollIn has the capacity to venture much further from the expert trajectory, and will explore distant parts of output space that provide less useful information about states commonly encountered by the next iteration of the learned parser. Where the training signal improves the performance on the Training Set (i.e. DetLOLS), this drop-off in performance is temporary while for the others it does not recover. A conclusion is that for DetLOLS the RollIn trajectories improve, exploring less unproductive output space. As to why only DetLOLS achieves this, we suggest the cause is reduced variance in the loss signal. Less noise means better training.

Early experiments using V-DAgger had shown that we needed to keep β high to obtain any benefit from imitation learning. Since DI-DLO is our final selected algorithm, we repeated experiments to explore the impact of β levels using DI-DLO. These are reported in Section 6.1.5.

6.1.4.1 V-DAgger variance

In general we note in Figures 6.4 and 6.5 that the algorithms with lower variance of loss on the RollOut do better. To test this hypothesis we ran comparisons using V-DAgger with increasing sample size of RollOut trajectories for each exploratory action. The parameter settings and data used are otherwise the same as in Section 6.1.4. The only change in this set of experiments is the increase in sample size; the higher the sample size, the lower the variance expected in the loss signal.

As Figure 6.6 shows, the higher the sample size, the better the performance of the trained parser. We also plot in Figure 6.7 the error rate during training of the AROW classifier at the end of five iterations; i.e. the percentage of training instances for which the incorrect action is predicted. This indicates the reduction in noise levels for greater sample sizes.

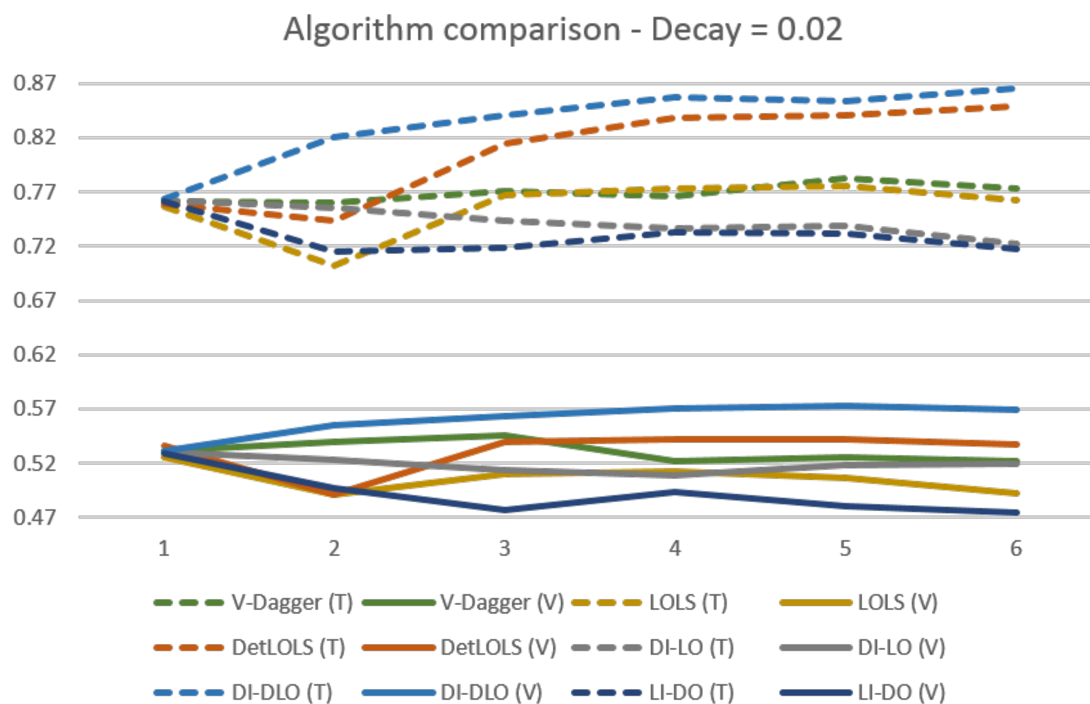

Figure 6.4: Comparison of Algorithm combinations, with F-Score on y-axis and number of iterations on x-axis. This shows performance of the trained parser on the training data (T) and the unseen validation data (V)

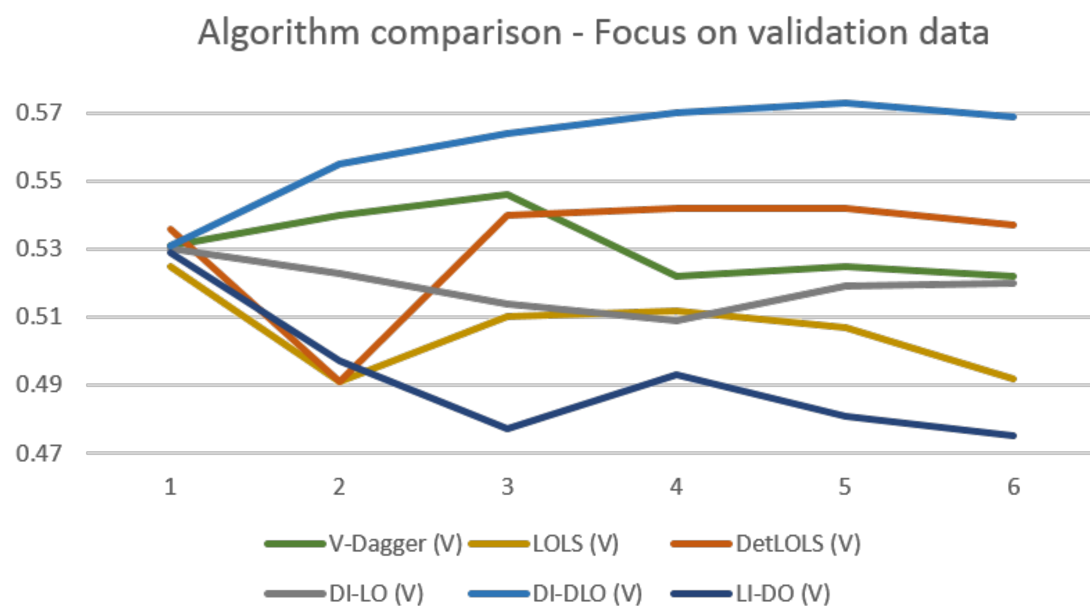

Figure 6.5: Comparison of Algorithm combinations, with F-Score on y-axis and number of iterations on x-axis. This shows performance of the trained parser on the unseen validation data only

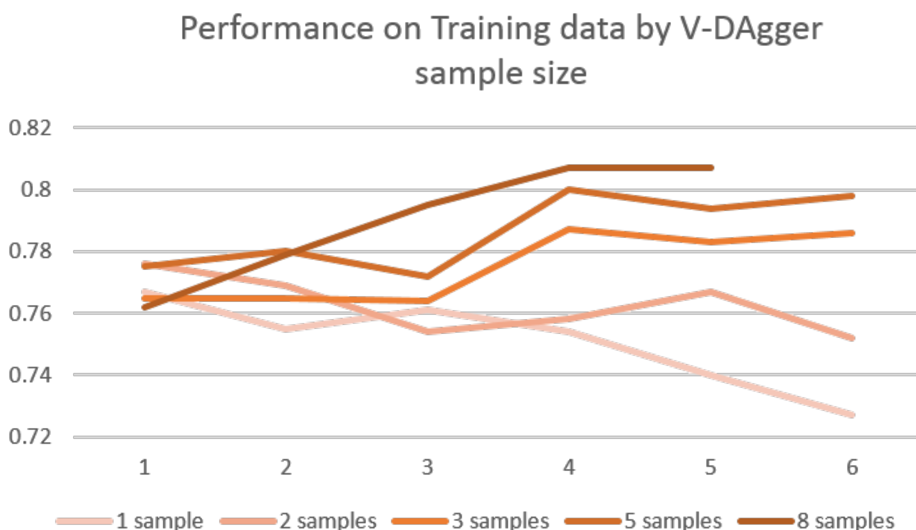

Figure 6.6: Effect of V-DAgger sample size on F-Score (y-axis) by number of iterations on x-axis. This shows performance of the trained parser on the training data

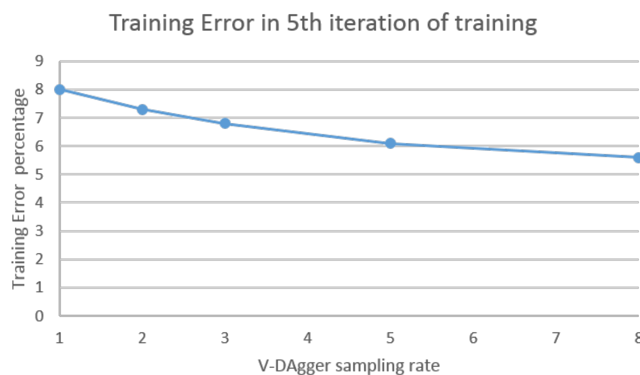

Figure 6.7: Effect of V-DAgger sample size on training error after five iterations.

6.1.5 Selecting β and δ rates

To select the decay parameter for β used in DI-DLO we ran validation experiments on the Small training set using the same parameters as in Section 6.1.4. The complete validation set was used, and the results for varying the rate of decay of β are shown in Figures 6.8 and 6.9.

For any decay rate of more than about 0.05, the performance of the learned classifier drops off rapidly on both the training and validation sets and that we need to keep the expert heavily weighted in our RollIn and RollOut trajectories. A decay rate of $\delta = 0.02$ provides the optimum, but these results are from only a single

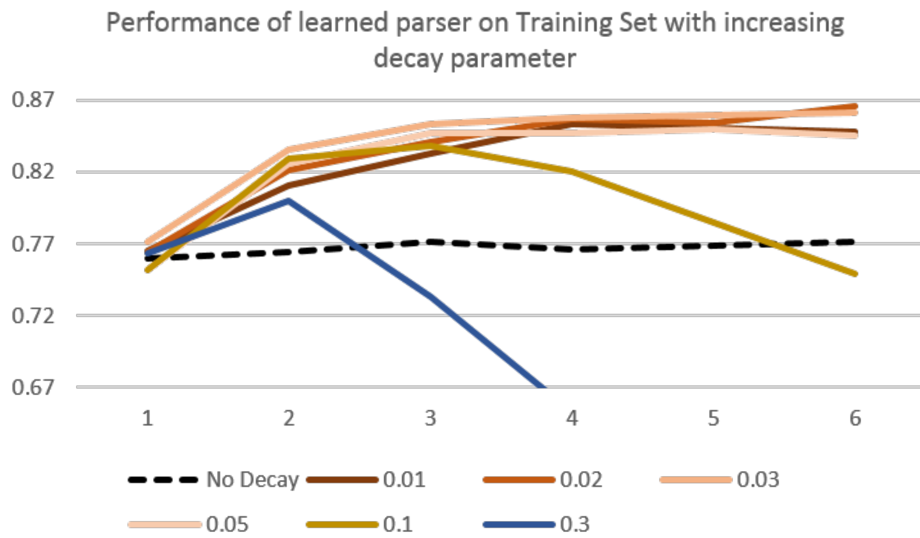

Figure 6.8: Six iterations, with learned classifier run over training set after each to give F-Score for varying decay rates.

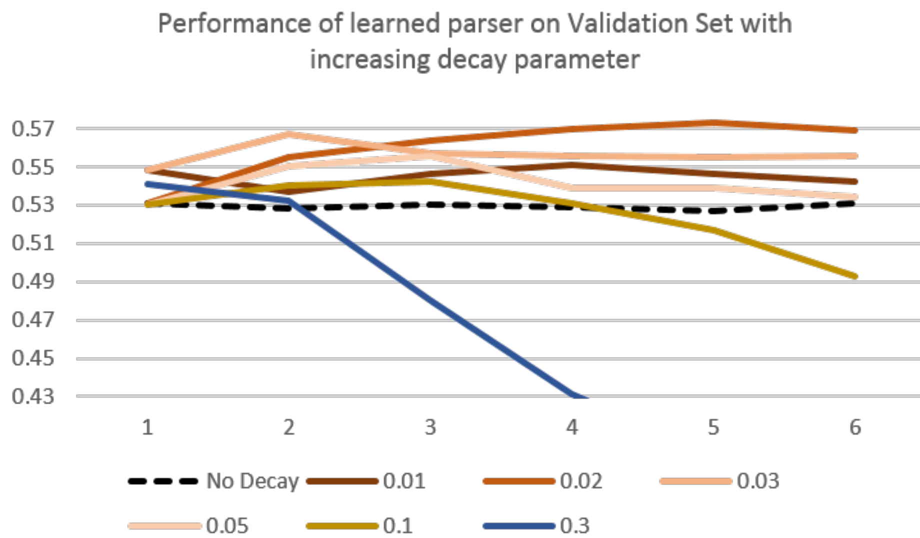

Figure 6.9: Six iterations, with learned classifier run over validation set after each to give F-Score for varying decay rates.

run of each setting. It is also clear that at least some decay of β is necessary as if we continue to use the expert policy the whole time (the "No Decay" series), then we never make any progress from the performance of the first iteration. Setting a small β between about 0.02 and 0.05 gives a clear improvement over this baseline for approximately 2-5 iterations before performance declines, and this is not due to using too few AROW training iterations given the improvement over zero decay.

We also experimented with a constant value of β beyond the first iteration, and the results are shown in Figure 6.10. There is not much difference between them, with all reaching a peak score on the validation set of about 0.56, albeit at different points. Again we conclude that some use of the learned classifier is essential; but not very much.

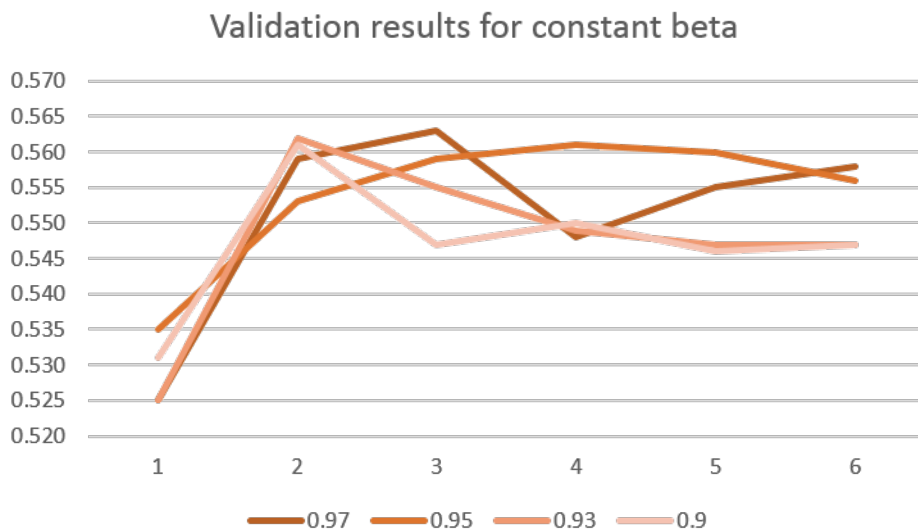

Figure 6.10: Six iterations of DI-DLO, with learned classifier run over validation set after each to give F-Score for varying constant rates of β and no decay.

6.2 Final Results

6.2.1 Simple Imitation Learning

We used the full training plus validation sets to train a parser, and tested this on the previously unused test data set. This gives a final F-Score of 0.68, and a comparison to previous work is shown in Table 6.2. Five final runs were executed with different

random seeds, and all gave the same result to two significant figures. While we surpass all previously published and other known work prior to July 2015, we do not achieve the same performance as Wang et al. [2015b].

Table 6.2: Simple imitation learning results compared with state-of-the-art

Author	F-Score on test	Precision	Recall
Flanigan et al. [2014] (JAMR)	0.58	0.66	0.52
Werling et al. [2015]	0.62	0.66	0.59
Wang et al. [2015a]	0.63	0.64	0.62
Pust et al. [2015]	0.66	-	-
Peng et al. [2015]	0.58	0.59	0.57
Wang et al. [2015b]	0.71	0.72	0.69
This dissertation	0.68	0.71	0.66

6.2.2 DI-DLO

We ran one full run using the full training set using DI-DLO, a decay rate of 0.03, and all other parameters and settings as in Section 6.1.4. Evaluation used the full validation set. This gave a peak in the Validation F-Score on the 3rd iteration. We therefore ran three iterations of DI-DLO with the same settings on a combination of the full training and validation sets. Applying the resultant learned policy to the unseen test set gave an F-Score of 0.63. We also tested the result of the first iteration, which gave an F-Score of 0.60, 8-points short of the result with one iteration of simple imitation learning in Section 6.2.1.

Chapter 7

Conclusion and Future Work

The main conclusion from the results in Table 6.2 is that by using a novel transition-based algorithm that incorporates an Insert action we are able to surpass all previous results for English to AMR translation prior to Wang et al. [2015b]. This gain comes at the expense of trajectory lengths T that are technically unbounded, and measures to prevent T exploding during training (see Section 4.3.6) are required. This is in line with the observations of Honnibal et al. [2013] that *non-monotonic* repair actions need to have an additional cost during training, otherwise the later repair facility means that all actions are equal earlier in the trajectory. This unbounded T also necessitates a loss function that applies a penalty per action taken. We consider future work aimed at increasing the F-Score beyond the 0.71 achieved by Wang et al. in Section 7.1.

Unfortunately the benefits we anticipated from more sophisticated imitation learning algorithms to explore beyond the simple expert trajectories fail to improve upon the result of simple imitation learning using binary expert loss. Theoretically we expected improvements both from the reduction in error propagation to avoid $O(T^2)$ losses on test data (Equations 2.2 and 2.3), and from the use of a non-decomposable loss function (Naive Smatch) that considers the ultimate loss of an action, taking into account future actions to mitigate local greediness. From the results in Sections 6.1.4 and 6.1.5 we see that performance does increase over the first few iterations for some V-DAgger-based algorithms provided that the β parameter is high enough. The problem is that it never reaches the level that simple imitation

learning achieves without exploration. This suggests that reduction in error propagation is providing a benefit, but that using a non-decomposable loss function is not working as anticipated. We consider some possible reasons for this, and future work to investigate these, in Section 7.2.

7.1 AMR performance

The advance of 8-points of F-Score between Wang et al. [2015a] and Wang et al. [2015b] is broken down by the authors as:

- 2 points from using a different parser to construct the dependency tree (the Charniak parser [Charniak and Johnson, 2005] instead of the Stanford one)
- 4 points from the addition of the Infer action (see Section 4.3.3)
- 2 points from use of additional semantic label features that utilise knowledge of the FrameSets in the underlying PropBank [Palmer et al., 2005, Kingsbury and Palmer, 2003]

We expect our Insert action to provide at least as much benefit as Infer, as it defines a strict super-set of graph transformations. Using the Charniak parser and augmenting the feature set to include semantic labels might therefore provide a similar boost to the final result if applied to our algorithm.

Werling et al. [2015] use JAMR for the graph creation phase, with a modification to initial concept identification. They retain the mapping of AMR fragment to a span of text, but learn a classifier that takes an input sentence and returns a set of AMR concept mappings that can include concepts not in the training set. While this is very different to our transition-based algorithm, they both address the same underlying problem of fragility of a memorized mapping based on a small training set - 38% of the concepts in the validation set do not occur in the training set, and cannot ever be generated by simple memorisation [Werling et al., 2015].

For each span of words they have a vocabulary of 9 action types, as listed in Table 7.1, alongside the percentage of total tokens that are processed with the action type in the JAMR aligned training data. As the table makes clear, the action

Table 7.1: Concept Identifications Actions in Werling et al. [2015] The Percentage is the proportion of total concepts in the training set that were matched by each Action.

Action	%age	Description	Our Equivalent
NONE	36.2	Word token is ignored, and not mapped to any AMR concept	Delete and ReplaceHead actions
DICT	26.1	JAMR lookup is used - i.e AMR graph fragment from training set aligned to the same set of word tokens	NextNode and NextEdge actions, which only use concepts from the training set
IDENTITY	16.6	Use the word token itself as the AMR concept	NextNode action with a parameter of `UNKNOWN`
VERB	10.2	Find the nearest verb (by Jaro-Winkler distance) in PropBank [Kingsbury and Palmer, 2003] that provides AMR with its FrameSets, and then select the most frequently used sense (e.g. left to `leave-01`)	No equivalent
LEMMA	4.5	Use the lemma of the word token as the AMR concept directly	No equivalent if Lemma is different to Word
NAME	3.9	Construct an AMR fragment by attaching a `name` node as the parent to the span	Insert action
DATE	1.1	Convert the span into an AMR `date-entity` fragment	Pre-processing step splits date fields, and then Insert permits construction of `date-entity`
PERSON	1.1	Insert `person` and `name` nodes above the span	Insert action
VALUE	0.1	Convert text to its integer value	Pre-processing numeric string conversion

space of our parsing algorithm covers all the concept identification actions of Werling et al. [2015], with the exception of LEMMA and VERB, and incorporation of knowledge of the PropBank FrameSets to augment the possible vocabulary of NextNode actions is an area of future extension we could apply to our work. This is likely to have some overlap with the 2-point gain in F-Score from using semantic label features derived from the PropBank found by Wang et al. [2015b].

Our attempt to improve the initial alignment between English words and AMR concepts in Section 4.7 does the opposite of what it was intended to, and reduces performance on validation data (Table 6.1). This contrasts with the work of Pour-

damghani et al. [2014]. It would be interesting to investigate this, and determine what causes the problem. Another approach would be to drop the alignment step completely, and learn it from the data. This is the approach of Vlachos and Clark [2014] to semantic parsing, albeit in the context of a much smaller vocabulary of concepts to be learned with a restricted action (i.e. vocabulary) space of 35 node types and 32 argument types; compared to approximately 6000 node concepts and 80 relations in the AMR training corpus.

7.2 Imitation Learning

7.2.1 State Distribution in training

More sophisticated imitation learning algorithms have not helped performance despite our theoretical hopes; and are also more time-consuming given the need to generate RollOuts for each possible action from each step of the RollIn trajectory.

If we look at Figure 6.4 we see that variants of V-DAgger and LOLS can improve performance over a few iterations, but that this benefit is much more pronounced when measured on the training set. This supports the underlying theory that exploring off-trajectory states and asking the expert for advice from these newly explored states does improve performance. The theoretical benefit of the exploratory versions of imitation learning comes from a training distribution of states that is more representative of the distribution the parser will experience in practice than the distribution of states the expert policy visits. In the case of the training set the gap between these two distributions is being bridged. The parser trained by the expert alone often deviated from the expert trajectory, and additional training data using input from the expert policy in these situations does improve the final result.

There is some benefit on the validation data, but to nothing like the same extent as Figure 6.4 shows. We speculate that the state distribution from the training data is highly dissimilar to that experienced on unseen data, reducing the value of the exploration. Considering the space of all possible trajectories for English sentences, the training set creates islands of closely clustered examples, with one cluster (roughly) per sentence. The validation sentences form their own set of data

points, not necessarily 'close' to the training examples.

The additional trajectories provided by the exploratory moves during RollIn do provide information that can be generalised, but focus on a narrow penumbra of space around the training sentences; so it is unsurprising that performance improves for these. The more distant validation sentences benefit less from this extra information because the gap between the observed distribution of states in training and the observed distribution of states experienced on the validation data remains high. We should recall that in the context of natural language, we have a very small training set of 4,000 sentences (6,000 including the validation set) and that many words and AMR concepts appear a handful of times, or just once. If we had AMR-labelled data of the size of the Wikipedia corpus used to train GloVe or the Stanford Dependency Parser, then we might expect this issue to be ameliorated. We could investigate this hypothesis by looking at the relative performance on test sentences with respect to the proportion of word and AMR concepts they contain that were not in the training set. Test sentences containing only words that are common in the training set should be 'near' the trajectories gathered in training, and accuracy on these should be much better.

7.2.2 Variance in loss signal

This argument about the relative paucity of training data applies equally well to the simple imitation learning approach using binary expert loss. The experimental results in Figure 6.4 for different algorithms suggest that the higher the variance in the loss calculated for different RollOuts from the same starting $\langle$state, action$\rangle$, the worse is the performance. DI-DLO is much better than V-DAgger, and DetLOLS is much better than LOLS. Between both these pairs the main difference is the shift from stochastic to deterministic calculation of RollOut trajectories and final loss. Removing this noise reduces variance in the loss signal and improves performance. This conclusion is reinforced by the results from varying the V-DAgger sample size in Figure 6.6, where it is the only variable changed and improves performance as it increases, and reduces variance in the loss signal.

While DI-DLO and DetLOLS cut down on the variance observed for a *given*

classifier, each iteration trains a new classifier, which will give different trajectories from the same state. Since the full set of training data includes trajectories from all previous iterations (and hence different classifiers), variance/noise levels will increase with iterations. We speculate that this might be a contributory factor in the fall-off in performance as the number of iterations increases, with training data originating from an ever growing number of policies. Experiments that use just the trajectory data from the most recent iteration of imitation learning might improve on this.

7.2.3 Expert optimality

The second point from these results is that we benefit from staying close to the expert trajectory. LOLS and LI-DO that use the learned parser to RollIn perform less well than DAgger and DI-DLO, which use expert actions for the vast majority of RollIn steps (given the high values of β used). This contrasts with other researchers who frequently report that low β values are helpful. Ross et al. [2011] find $\beta = 0$ after the first run often provides the best results ($\delta = 1.0$), and Vlachos and Clark [2014] use a decay rate of $\delta = 0.3$ for optimal results. We find completely the opposite and have to keep $\beta > 0.95$ for any benefit.

It is possible that with a less good expert, we might see more of an impact. We should recall that LOLS was designed for situations in which the expert is quite poor (Section 2.4.2), and this may not be an ideal choice for the AMR task. The potential to improve on the expert is limited given how good it is (see Table 4.3); and any benefit from this is swamped by the variance in the training signal inherent to the long roll-out trajectories before a terminal state is reached. In contrast, the simple imitation learning approach with binary expert loss has a crystal-clear training signal, and this lack of variance appears to provide more than enough benefit to outweigh the theoretical problems of an unrealistic distribution of states during training and the lack of a non-decomposable loss function to take account of future actions. The first iteration with any of the algorithms of Section 5.1 only uses the expert policy for RollIn and RollOut, and hence is the same as simple imitation learning except that the binary expert loss function is replaced with Naive Smatch.

From Section 6.2, the first iteration of DI-DLO with the full training set gives an F-Score of 0.60 when evaluated on the test set, compared to 0.68 for simple imitation learning. This 8-point drop must be due to the change in loss function, and supports the hypothesis.

We did run some experiments using binary expert loss with DI-DLO (and the other algorithms of Section 5.1), but these showed no benefit - there was usually a very small improvement on training set performance, and always a deterioration in performance on the validation set.

7.2.4 Future Work

From this discussion we believe that the imitation learning algorithms of Chapter 5 provide advantages from an improved distribution of training states (seen in this dissertation), and a non-decomposable loss function (not seen in this dissertation due to high variance in the loss signal). The initial focus of future work should be to address this variance problem. Three possible options to be investigated are:

- to use partial RollOuts at each step, say for M steps, so that there is less opportunity for major deviations to take place on a long trajectory. The loss could then be calculated against the graph generated by the expert policy after M steps. This would mean the loss was no longer fully independent of the expert, but the gain in reduced variance could be worthwhile, especially given the quality of our expert. This is the approach of *focused costing* from Vlachos and Craven [2011].

- modify the algorithms of Section 5.1 to use only the expert policy in the RollOuts, while retaining the learned policy during RollIns. Similarly we could use the original DAgger of Ross et al. [2011] which does not RollOut at all, but uses binary expert loss with step-wise stochastic RollIn trajectories. This might gain some of the benefit of improved state distribution with binary expert loss.

- start training just using small AMR graphs, in analogy with layer-wise training methods used in Neural Networks [Knerr et al., 1990, Bengio et al., 2007].

It might help to train a parser to work efficiently on small elements of graph sub-structure before putting together the larger elements. The smaller the graph, the less room there is for single-steps to lead to a large difference in the final terminal state and so we expect the variance problem to be reduced for these given their shorter trajectories. Feasibly we could train an initial parser on small sentences, or sentence-fragments, and learn common action sequences that could be included as a single meta-action when we look at longer sentences. This would reduce the trajectory length, and potentially the noise of the training signal.

Additionally it is worth investigating the theoretical implications of our LOLS/V-DAgger hybrids, given the theoretical bounds that exist for both. This could help us understand the general sorts of problems for which they are most suited.

Bibliography

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

Omri Abend and Ari Rappoport. Ucca: A semantics-based grammatical annotation scheme. In *Proc. IWCS*, 2013.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

Olga Babko-Malaya. Guidelines for propbank framers. *Unpublished manual, http://verbs.colorado.edu/ mpalmer/projects/ace/FramingGuidelines.pdf*. URL `http://verbs.colorado.edu/~mpalmer/projects/ace/FramingGuidelines.pdf`.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Phillip Koehn, Martha Palmer, and Nathan Schneider. Abstract Meaning Representation for Sembanking. *Proceedings of the 7th Lingustic Annotation Workshop & Interoperability with Discourse*, pages 178–186, 2013.

Valerio Basile, Johan Bos, Kilian Evang, and Noortje Venhuizen. Developing a large semantically annotated corpus. In *LREC*, volume 12, pages 3196–3200, 2012.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy

layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

Alistair Butler and Kei Yoshimoto. Banking meaning representations from treebanks. *Linguistic Issues in Language Technology*, 7(1), 2012.

Shu Cai and Kevin Knight. Smatch: an evaluation metric for semantic feature structures. In *ACL (2)*, pages 748–752, 2013.

Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé III, and John Langford. Learning to search better than your teacher. 32, 2015.

Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180. Association for Computational Linguistics, 2005.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. Parsing graphs with hyperedge replacement grammars. In *ACL (1)*, pages 924–932, 2013.

Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics, 2002.

Koby Crammer and Yoram Singer. Ultraconservative online algorithms for multiclass problems. *The Journal of Machine Learning Research*, 3:951–991, 2003.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585, 2006.

Koby Crammer, Mark Dredze, and Alex Kulesza. Multi-class confidence weighted algorithms. In *Proceedings of the 2009 Conference on Empirical Methods in*

Natural Language Processing: Volume 2-Volume 2, pages 496–504. Association for Computational Linguistics, 2009a.

Koby Crammer, Alex Kulesza, and Mark Dredze. Adaptive regularization of weight vectors. In *Advances in neural information processing systems*, pages 414–422, 2009b.

Koby Crammer, Alex Kulesza, and Mark Dredze. Adaptive regularization of weight vectors. *Mach Learn*, 91:155–187, 2013.

Deborah A Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. Expanding the scope of the atis task: The atis-3 corpus. In *Proceedings of the workshop on Human Language Technology*, pages 43–48. Association for Computational Linguistics, 1994.

Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine learning*, 75(3):297–325, 2009.

Marie-Catherine De Marneffe and Christopher D Manning. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

Bonnie Dorr, Nizar Habash, and David Traum. *A thematic hierarchy for efficient generation from lexical-conceptual structure*. Springer, 1998.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A Smith. A discriminative graph-based parser for the abstract meaning representation, 2014.

Yoav Goldberg and Michael Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750. Association for Computational Linguistics, 2010.

Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *COLING*, pages 959–976, 2012.

Yoav Goldberg and Joakim Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1:403–414, 2013.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, et al. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–18. Association for Computational Linguistics, 2009.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. A non-monotonic arc-eager transition system for dependency parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172. Citeseer, 2013.

Matthew A Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

Paul Kingsbury and Martha Palmer. Propbank: The next level of treebank. In *Proc. Workshop Treebanks and Lexical Theories*. Citeseer, 2003.

Stefan Knerr, Léon Personnaz, and Gérard Dreyfus. Single-layer learning revisited: a stepwise procedure for building and training a neural network. In *Neurocomputing*, pages 41–50. Springer, 1990.

Kevin Knight, Laura Baranescu, Claire Bonial, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Daniel Marcu, Martha Palmer, and Nathan Schneider. Abstract meaning representation (amr) annotation release 1.0. *Linguistic Data Consortium Catalog. LDC2014T12*, 2014.

Irene Langkilde and Kevin Knight. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, pages 704–710. Association for Computational Linguistics, 1998.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014. URL `http://www.aclweb.org/anthology/P/P14/P14-5010`.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

Andrew McCallum, Dayne Freitag, and Fernando CN Pereira. Maximum entropy markov models for information extraction and segmentation. In *ICML*, volume 17, pages 591–598, 2000.

Ryan McDonald. *Discriminative learning and spanning tree algorithms for dependency parsing*. PhD thesis, University of Pennsylvania, 2006.

Ryan T McDonald and Joakim Nivre. Characterizing the errors of data-driven dependency parsing models. In *EMNLP-CoNLL*, pages 122–131, 2007.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*. Citeseer, 2003.

Joakim Nivre and Ryan T McDonald. Integrating graph-based and transition-based dependency parsers. In *ACL*, pages 950–958, 2008.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106, 2005.

Xiaochang Peng, Linfeng Song, and Daniel Gildea. A synchronous hyperedge replacement grammar based approach for amr parsing. *CoNLL 2015*, page 32, 2015.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 12:1532–1543, 2014.

Nima Pourdamghani, Yang Gao, Ulf Hermjakob, and Kevin Knight. Aligning english strings with abstract meaning representation graphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 425–429, 2014.

Vasin Punyakanok, Dan Roth, Wen-tau Yih, and Dav Zimak. Semantic role labeling via integer linear programming inference. In *Proceedings of the 20th international conference on Computational Linguistics*, page 1346. Association for Computational Linguistics, 2004.

Michael Pust, Ulf Hermjakob, Kevin Knight, Daniel Marcu, and Jonathan May. Using syntax-based machine translation to parse english into abstract meaning representation. *arXiv preprint arXiv:1504.06665*, 2015.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics*, pages 661–668, 2010.

Stéphane Ross, Geoffrey J Gordon, and J Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011.

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. A transition-based dependency parser using a dynamic parsing strategy. In *ACL (1)*, pages 135–144, 2013.

Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

Shai Shalev-Shwartz, Koby Crammer, Ofer Dekel, and Yoram Singer. Online passive-aggressive algorithms. In *Advances in neural information processing systems*, page None, 2003.

David Silver, James Bagnell, and Anthony Stentz. High performance outdoor navigation from overhead data using imitation learning. *Robotics: Science and Systems IV, Zurich, Switzerland*, 2008.

Andreas Vlachos. An investigation of imitation learning algorithms for structured prediction. In *10th European Workshop on Reinforcement Learning*, pages 143–154. Citeseer, 2012.

Andreas Vlachos and Stephen Clark. A new corpus and imitation learning framework for context-dependent semantic parsing. *Transactions of the Association for Computational Linguistics*, 2:547–559, 2014.

Andreas Vlachos and Mark Craven. Search-based structured prediction applied to biomedical event extraction. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*, pages 49–57. Association for Computational Linguistics, 2011.

Andreas Vlachos and Mark Craven. Biomedical event extraction from abstracts and full papers using search-based structured prediction. *BMC bioinformatics*, 13(Suppl 11):S5, 2012.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. A transition-based algorithm for amr parsing. *North American Association for Computational Linguistics, Denver, Colorado*, 2015a.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. Boosting transition-based amr parsing with refined actions and auxiliary analyzers. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 857–862, Beijing, China, July 2015b. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-2141`.

Keenon Werling, Gabor Angeli, and Christopher D. Manning. Robust subgraph generation improves abstract meaning representation parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 982–991, Beijing, China, July 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-1095`.

Hiroyasu Yamada and Yuji Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206, 2003.

John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1050–1055, 1996.

Yue Zhang and Stephen Clark. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In

Proceedings of the Conference on Empirical Methods in Natural Language Processing, pages 562–571. Association for Computational Linguistics, 2008.