Using Task Descriptions for Macro Action Transfer

Trevor Houchens
Brown University
trevor_houchens@brown.edu
https://github.com/houchenst/NLPMacroActionTransfer

Abstract

Using reinforcement learning methods to learn how to perform new tasks from scratch does not closely mimic the human learning process. As humans, we rely on a wealth of widely applicable motor skills and high level actions that allow us to quickly pick up complicated new tasks. Within the reinforcement learning paradigm, this process of re-using learned abstract actions is modeled through the commonly used options framework. While options have proven to speed up learning in related tasks, too many options can lead to slow learning by bloating the action space. This poses a problem for building truly general agents that are capable of performing many high level tasks. This work proposes a method for identifying useful high level actions using natural language task descriptions, with the goal of minimizing the time necessary to learn new tasks.

1. Introduction

Humans are capable of learning complex new tasks remarkably quickly, in large part due to our ability to transfer skills from previously learned tasks. Consider the skill of grasping an object. Grasping, in itself, is fairly complicated and requires positioning the hand correctly while simultaneously manipulating multiple fingers. In spite of the difficulty of this action, adults are easily able to learn new tasks, like hammering nails or swinging a tennis racket, which rely on being able to grasp an object. The explanation for this is that humans spend time as infants learning to pick up and hold objects, and this grasping ability can then be transferred to new tasks when necessary.

In the reinforcement learning community, there has been significant work towards learning higher level actions that can be transferred between tasks instead of being re-learned each time. This is commonly done using the options framework, which allows agents to choose to execute a fixed sub-policy until some end condition is met [1]. A variety of methods have been proposed to identify useful options, which range from minimizing the the cover time - the expected time to visit every node on a graph during a random walk - of the state space, to finding state bottlenecks in successful runs, to looking for common pieces of policy among successful runs [2–4]. While work in this area has shown that well-chosen options can speed up learning in related tasks, it has also revealed that too many options can slow down learning. Since each new option is a new potential action for an agent, adding a large number of options leads to a large state space which makes learning more difficult.

Interestingly, humans do not seem to suffer from slow learning despite having a large number of abstract actions at their disposal. This can be attributed to our ability to identify a small subset of our learned actions which might be relevant to a new task. Language is commonly used as a tool to help us identify which actions are relevant and which are not. For example, if I'm teaching somebody how to play soccer, I might tell them to ”Run to the ball and kick it towards the goal.” The verbs ”run” and ”kick” in this statement correspond to well-known abstract actions that are useful when playing soccer. The person learning soccer might be capable of many other high level actions like writing, speaking, eating, and driving, but they probably won't try out these actions because they haven't been told that they are relevant.

We aim to mimic this process of relevant skill identification within a reinforcement learning framework. High level skills are learned as action sequences across a set of different tasks within the same state/action space. Natural language task descriptions, along with action sequences from successful runs, are then used to train a model to predict which macro actions are useful for a task based on its description.

2. Related Work

2.1. Options

The options framework has been an established part of RL for some time now. Options consist of a partial policy, defined over some subset of states, as well as a set of initialization states and a set of termination states, specifying where the option can be executed, and where it must terminate, respectively. There have been many proposed methods for generating useful options. A common approach is to identify bottleneck states (i.e. states that are frequently passed through on successful runs) [4]. Intuitively, these can be thought of like doorways. Doors are small openings connecting two rooms, and anybody who wants to move between the two rooms must stand in the doorway at some point. Similarly there frequently exist states that must be passed through on the way to the goal. Another approach is to identify sections of policy that are shared between agents trained on similar tasks [2]. For example, if agent 1 has the policy $\pi_1(1) \rightarrow 4, \pi_1(2) \rightarrow 3, \pi_1(3) \rightarrow 8$ and agent 2 has the policy $\pi_2(1) \rightarrow 4, \pi_2(2) \rightarrow 3, \pi_1(3) \rightarrow 11$, then a useful option might consist of the sub-policy $\pi_2(1) \rightarrow 4, \pi_2(2) \rightarrow 3$ since it is shared by both learned policies. Further work has gone into identifying options that minimize cover time (the time necessary to cover the state space via random walk), since the time needed to find a goal state is related to cover time [3].

2.2. Action Sequences

Similar to options are action sequences. While an option defines a policy over different states, an action sequence is a fixed series of actions that will be executed sequentially over multiple time steps. Action sequences are somewhat less complex than options since they define a policy that is independent of state, while options are conditioned on state. A downside of this is that with stochastic transitions, a fixed series of actions will not always have the same effect. Finding useful action sequences is often done by identifying common sequences of actions among successful runs. Some work has successfully selected action sequences using methods borrowed from compression since the concept of reducing an series of successful actions into a shorter list of high level actions is analogous to the problem of compression [5].

The main inadequacy in the prior work in options and action sequence selection is the limit on the number of macro actions that can be added before learning is slowed due to the larger state space. By choosing only task-relevant actions, we should be able to add a much larger set of macro actions, that can be pruned according to descriptions of novel tasks.

3. Approach

3.1. Markov Decision Processes

A Markov Decision Process (MDP) is a common format for RL problems. A typical MDP can be expressed as a tuple in the following manner:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$$

- $\mathcal{S}$ represents the state space of the problem
- $\mathcal{A}$ is the set of actions available to the agent
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the transition probability from a state, action pair into a new state.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ defines the reward given to the agent from moving from one state to another via a specific action.

Posed with an MDP, an agent seeks to learn a policy, π, which defines which action to take in a given state.

$$\pi(s) \rightarrow a$$

$$a \in \mathcal{A}, s \in \mathcal{S}$$

We seek to learn a set of macro actions, M, each of which defines a series of actions to take at subsequent time steps. These macro actions will then be added to the action space of our agent, so that it learns a new policy, π^*

$$\pi^*(s) \rightarrow a^*$$

$$a^* \in \mathcal{A}^* = \mathcal{A} + M, s \in \mathcal{S}$$

The objective here is to identify a set of macro actions, M, that are useful for the tasks at hand, and maximize the cumulative reward over a large number of time steps.

3.2. Defining a Class of MDPs

One of the more challenging aspects of learning high-level actions is designing a problem space that is both complex enough to have meaningful high-level actions, and small enough that common reinforcement learning algorithms can find an optimal policy in a reasonable time. In this work, we define a new class of MDPs that is similar to the well-known ”GridWorld”, but with additional complexities. As in GridWorld, an agent moves around a 2D grid. However, instead of directly picking a direction to move, the agent must change it's position or orientation by moving it's two rear legs. While GridWorld has a reward for reaching a goal state, our MDPs give out a reward when the agent carries an object to the goal state. All other states get a reward of -1. The agent has the following 10 actions available to it:

1. Move rear left leg up

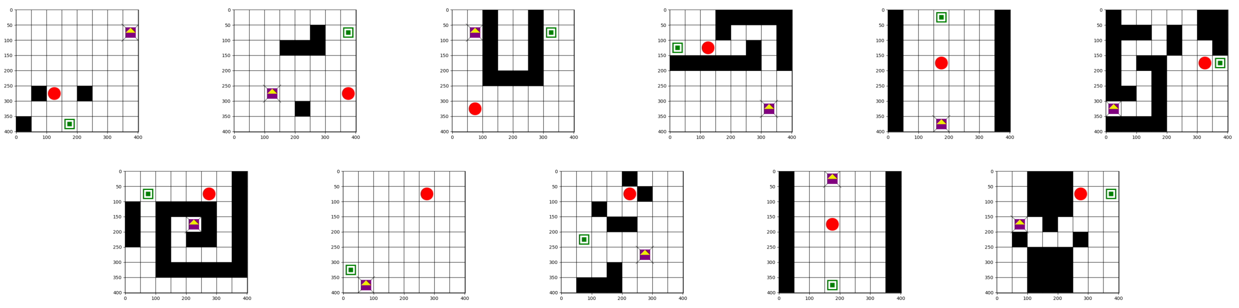

Figure 1. The initial states of the 11 tasks used. The two rightmost tasks in the second row were not used during macro proposition. The agent is purple, the object to be carried is red, and the target location is green.

2. Move rear left leg down
3. Move rear right leg up
4. Move rear right leg down
5. Move both rear legs up (causes the agent to move backwards, if possible)
6. Move both rear legs down (causes the agent to move forwards, if possible)
7. Move rear legs clockwise (causes the agent to turn CCW, if possible)
8. Move rear legs counter-clockwise (causes the agent to turn CW, if possible)
9. Move forelimbs up (grasps the object if it is in front of the agent)
10. Move forelimbs down (releases the object if it is grasped)

This class of MDPs was designed with action abstraction in mind. There are several obvious abstract actions in this space that do not exist in GridWorld; walking (making several leg movements to move forward, turning (making several leg movements to change orientation), and carrying (grasping an object and then moving with it) are a few.

The eleven tasks that are used in this work can be seen above in Figure 1. As an example, the caption for the task in the upper left is:

> Move west then south to the object. Carry the object south then east to the target. The object is far away.

Most descriptions contain cardinal directions which aren't directly related to macros, but are somewhat correlated with whether the agent needs to turn or how far it needs to walk. Some other captions describe high-level actions directly with statements like "Make a sharp turn."

3.3. Proposing Macros

To propose macros that might reduce our training time, we can utilize successful action sequences that are generated from our learned policies. This is accomplished via the following process:

1. Consider a set of MDPs, T. For each task, $t \in T$, we learn an optimal policy, π^t, using the SARSA-λ algorithm.
2. For each policy, π^t, we can generate state and action sequences, $S_{0:n}$ and $A_{0:n-1}$. We are interested in successful action sequences, so S_n should be a goal state. m action sequences are generated per policy. Since MDPs have stochastic transitions, each action sequence is not necessarily the same.
3. We now have the following set of $m \times |T|$ action sequences:

   $$\{A_{0:n-1}^{i,t} | i \in [0, m), t \in T\}$$

   From this set of action sequences, we count the number of occurrences of each subsequence with length between 2 and 10. This count is then scaled by the length of the subsequence to give us an estimate of the total number of actions from our original sequences that we could encode by adding this subsequence as a macro. The subsequence with the greatest scaled count is added as a macro. Note that by choosing macros with this heuristic, we also significantly reduce the distance between the initial state and the goal states for each task. This can lead to faster convergence.
4. After a new macro is added, the matching subsequences are replaced by this macro action. Then, an-

other macro is extracted using the same procedure until the desired number of macros are obtained.

In this work, 50 macros are proposed from a set of 9 tasks. There are 20 action sequences generated for each task.

3.4. Using Task Descriptions to Identify Relevant Macros

As previous work has demonstrated, adding too many abstract actions can lead to slower learning since it bloats the action space of the MDP [2]. However, in order to accomplish a variety of complex tasks, it is useful to have a large number of abstract actions available. For each task, we have a natural language description, d, that explains how to do the task. We train a model to predict the probability that each of our proposed macros is the best macro for a new task, based on the task's description. Given a set of proposed macros, M, and a description, d, we define the model as:

$$F(d) \to [0,1]^{|M|}.$$

Our model is implemented as a Naive Bayes classifier. The task descriptions are tokenized and transformed into a vector that stores the number of times each word appears in the description. Each macro corresponds to a "class" within the Naive Bayes framework. Each time a macro appears in the action sequence of a task, the vectorized task description and the macro are added to the training data. It is important that another training point is added for *each* occurrence of a given macro, and not just once per sequence.

After the Naive Bayes model is trained, the posterior can be used to estimate which proposed macro is most likely to appear in the optimal action sequence of a task described by a new descriptor.

4. Results

4.1. Training with Different Numbers of Macros

Prior work has shown that adding some abstract actions tends to speed up convergence, while adding too many can slow it down. In order to determine the optimal number of macros to add to our tasks, we re-trained with the top-n macros, for $n \in [1, 3, 5, 10, 30, 50]$. This was done for both the full set of macros, and for the model-predicted macros. The results of using different numbers of macros can be seen in Figure 2. The number of steps to the goal state at each episode is shown for a task that was used to train the model and a novel task. Since most curves were quite noisy, all plots represent the average of several trials with smoothing applied over a window of 50 episodes.

The charts clearly indicate that while a using a few macros converges as fast or faster than using no macros, having too many macros slows the convergence significantly. When 30 or 50 macros were added it took many more time steps to reach the goal state. Adding 1,3,5 or 10 macros seemed to offer some improvements. Based on these results, 5 macros was chosen as the optimal number of macros and was used in further experiments.

4.2. Predicting Macros From Descriptions

Figure 3 shows the results of re-training on one of the tasks that was used to propose macros, and the results of re-training on two novel tasks. These results were obtained by adding the top 5 macros to each of the tasks. For the state that was used for macro proposal, we took the top 5 macros from the full set of macros, the top 5 macros that our model predicted would be best, and the top 5 task-specific macros. The task-specific macros were extracted in the same way as the set of all macros, except that only one task was used to generate the training sequences, rather than 9. We would expect that macros extracted from the optimal policy of a particular task would be especially useful when re-training on that task. Therefore, this is a good benchmark to compare to when we are evaluating the macros that our model predicted.

We can see that for the task that was seen during training, the task-specific macros and model-predicted macros outperform the original macros. For the novel tasks, there are two very different results. The novel tasks are visualized in Figure 4. For Task 6, the model-predicted macros performed similarly to the original macros, and both outperformed the agent trained with no macros. For Task 9, the model-predicted macros did poorly relative to the original macros, and both of these performed worse than the agent trained with no macros.

4.3. Macros By Keyword

By using our Naive Bayes model to suggest macros from a single word, we can see which macros are associated with that word. In the training tasks, the phrase "sharp turn" was used to describe problems where the agent was required to make a U-turn around an obstacle. When the model was used to propose macros for the keyword "sharp", the first long (length > 5) macro corresponded to a partial U-turn. This macro is shown in Figure 5. It causes the agent to move forward, turn clockwise, and then move two spaces forward. There were a number of smaller macros proposed before this macro, but this can be attributed to the fact that it becomes increasingly uncommon to repeat longer action sequences due to the stochastic nature of MDPs.

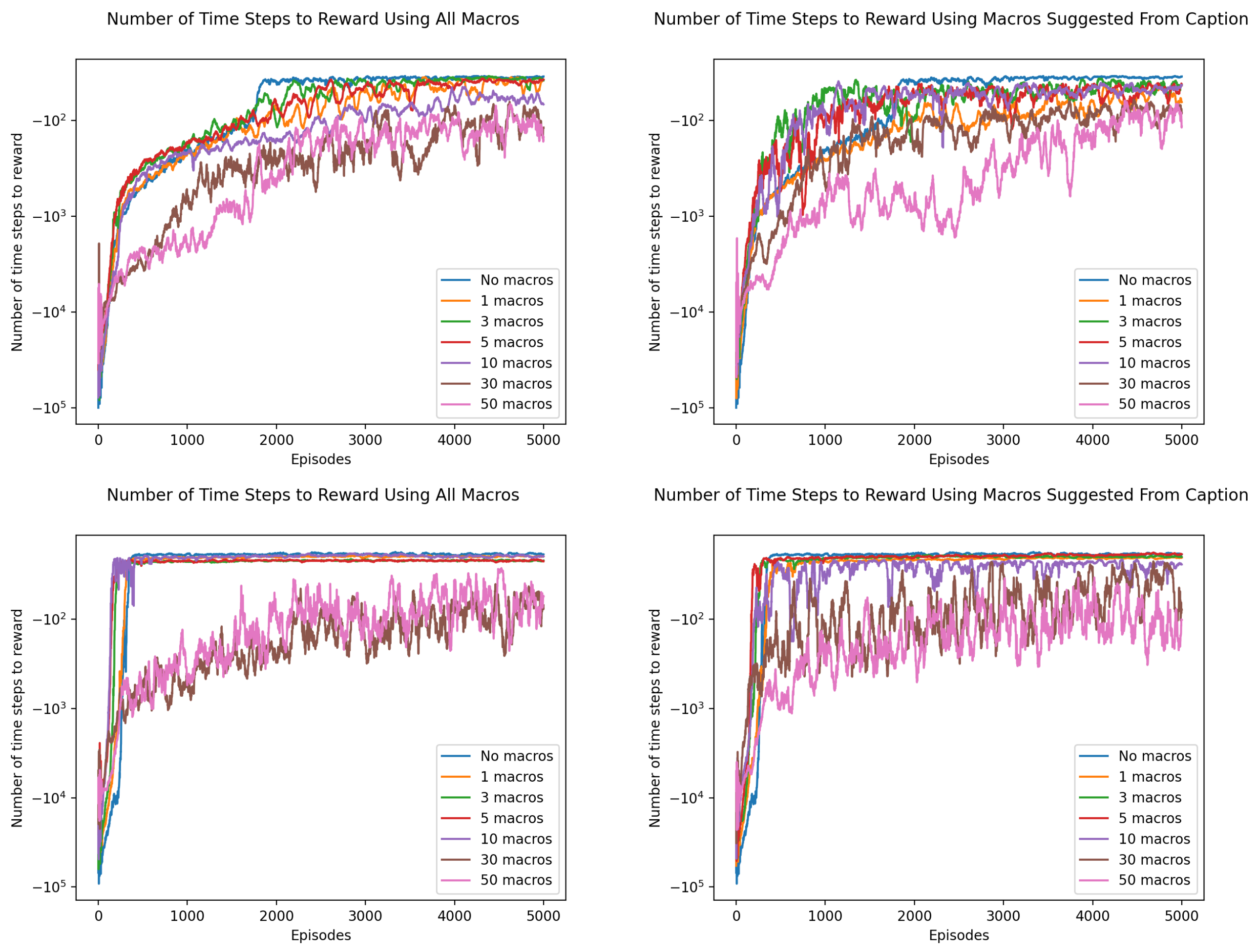

Figure 2. These graphs show the number of time steps to get to the goal state of an MDP during each episode of learning. The top row shows results from one of the tasks used to generate the macro proposals, and the bottom row shows results on a novel task (Task 6).

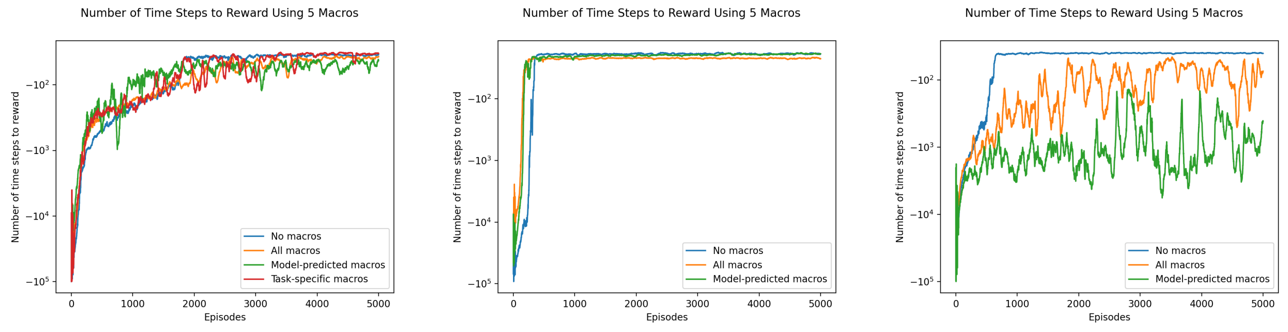

Figure 3. On the left are reward curves for one of the tasks used to train the macro prediction model. On the right are two novel tasks (Task 6: middle, Task 9: right)

4.4. Hierarchy of Macros

5. Discussion

The goal of this project was twofold, to learn useful action sequences, and to predict the best action sequences

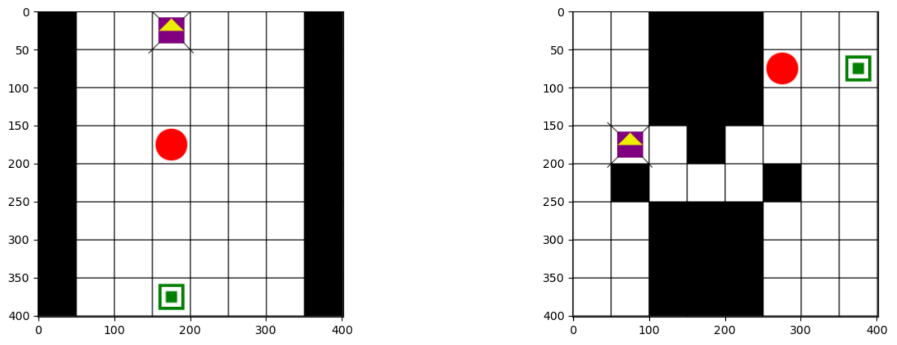

Figure 4. The two novel tasks. On the left is Task 6. On the right is Task 9.

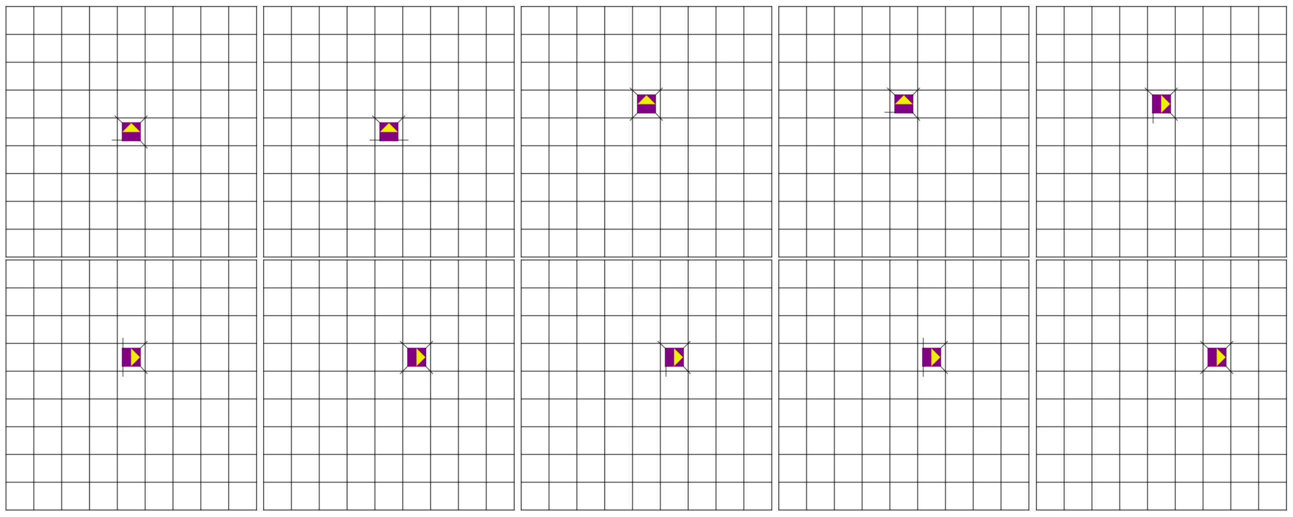

Figure 5. The first macro longer than 5 actions that was suggested for the keyword ”sharp”. This macro completes part of a u-turn, which is common in tasks that have the phrase ”sharp turn” in their description.

given a task description.

5.1. Learning Action Sequences

We were successful in learning action sequences that helped our RL algorithm converge to an optimal policy in fewer episodes. In Figure 2, we show that we are able to converge more quickly with macros than without macros on two different tasks. This result was expected, as there are some very commonly repeated patterns, like walking, that occur in all tasks. By adding a ”walking” macro, the policy that the agent has to learn can be less complex, since the macro can be executed, rather than sequentially taking each primitive action individually.

5.2. Macro Action Transfer

The second objective of this work was to predict which macros, from a set of proposed macros, would be relevant to a task, based on a natural language description. For one of the tasks used to generate macro proposals, the model-predicted macros seemed to perform better than the task-specific macros, originally-proposed macros, and no macros (Figure 3). However, some of this can be attributed to noise, since the task-specific macros should be the most relevant macros for a given task, and it is unlikely that the model-predicted macros truly outperformed these.

The performance of the model-predicted macros on the two novel tasks was less straightforward (Figure 3). On Task 6, the model-predicted macros and the original macros performed similarly well, and both outperformed the agent trained with no macros. On Task 9, no macros performed the best and the model-predicted macros performed the worst.

The results indicate that predicting macros from task descriptions using a learned model was not more effective than simply extracting the best macros from a set of action sequences. This doesn't mean that abstract actions can't be predicted from task descriptions in general. These results can be attributed, in part, to the number of tasks that were used and the formulation of our abstract actions as action sequences. The goal in predicting macros from task descriptions was to be able to maintain a large set of useful abstract actions that could be narrowed down for specific tasks. Since it is difficult to learn long action sequences that are useful, we were limited to modeling more low-level actions. The space of low-level actions is inherently smaller than that of high-level actions and we didn't have very many tasks to begin with, therefore it is likely that there were not very many useful macro action sequences. Most of these useful sequences were likely in the original macro proposals. Predicting relevant macros using task descriptions would probably be most effective when there are a large number of high-level actions that are useful for different tasks, something that we did not have.

The number of tasks used to train the Naive Bayes classifier also likely contributed to the lackluster results of the predicted macros. Since most words in the task descriptions occurred at most a few times, it was easy for the classifier to overfit or make false connections between words and macros. When predicting macros for novel tasks, this could lead to suggesting poor macros that had some correlation with the description words, but no meaningful connection.

The decision to model the relationship between task descriptions and useful macros using a Naive Bayes model was influenced by the limited number of tasks available. While treating a sentence as a bag of words loses a lot of information, the bias it introduces makes it less prone to overfitting than a more complex model. With more tasks and an abstract action formulation that allows for higher-level actions, like options, it might make more sense to use an LSTM or a Transformer. The results of our macro prediction from a keyword suggest that while a Naive Bayes model isn't perfect, it was able to correctly model some of the connections between the task descriptions and macros (Figure 5).

6. Conclusion

We have shown that it is possible, and straightforward, to extract action sequences from optimal policies that can speed up learning when added to the action space as macros. The results of this work suggest that learning to predict useful abstract actions using a description of a task is a viable approach, even though it was not very effective within our framework. Future work should consider modeling abstract actions using options, using a larger set of tasks, and analyzing how task descriptions are created.

References

[1] S. et. al., "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, 1999. 1

[2] M. Pickett and A. Barto, "Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning," 2002. In: ICML (2002). 1, 2, 4

[3] Y. Jinnai, "Discovering options for exploration by minimizing cover time," 2019. In: ICML (2019). 1, 2

[4] O. Simsek, "Skill characterization based on betweenness," 2008. In: NeurIPS (2002). 1, 2

[5] P. T. Francisco Garcia, Bruno da Silva, "A compression-inspired framework for macro discovery," 2019. In: AAMAS (2019). 2