Feature Selection for Learning from Demonstration in Minecraft

Brandon Packard, Santiago Ontañón
Drexel University
Philadelphia, PA, USA
{btp36,so367}@drexel.edu

Abstract

Learning from Demonstration has the potential to enable the crafting of behavior for non-player characters, allies, and enemies without requiring programming knowledge. This paper focuses on addressing two key problems of LfD when applied to games. The first is data sequentiality, when actions might be influenced by previous environmental states/actions, instead of just the current state. The second is having structured representations of data, where data is provided as an arbitrary number of predicates instead of a fixed-length vector. In this paper, we evaluate a collection of feature selection strategies to address these problems in the context case-based learning algorithms in the domain of Minecraft.

Introduction

This paper focuses on Learning from Demonstration (LfD). Given sequential traces of the behavior of an expert performing a task on a given environment, the goal of LfD is to learn the function that determines the expert actions given the observed states. Specifically, this paper focuses on feature selection in LfD, i.e., trying to derive which pieces of data from the state should be used for learning which can both increase prediction accuracy, with an emphasis on structured representations, and on considering the sequential nature of training data in LfD.

This structured representation setting for LfD is important because of two reasons. First, in many domains of interest (e.g., computer games), it is more natural to represent the world state using structured representations than using a feature-vector approach. For example, in the popular game *Minecraft*, in a given world state there might be an arbitrary number of enemies. It can be hard to represent all of these enemies using a fixed-length feature-vector, but it is trivial to do so using a structured representation such as Horn clauses (Nienhuys-Cheng and Wolf 1997). Second, learning agents might need to remember information from past states (data sequentiality). For example, if an agent sees a treasure map on the wall but cannot take it, they would need to remember the map when looking for the treasure, since it would not show up later on as part of the agent perception.

We focus on comparing strategies for feature selection, both structured and sequential, in the context of case-based learning algorithms. We first evaluate a collection of algorithms based on adapting standard feature selection algorithms for propositional representations (Kohavi and John 1997), then evaluate filter-based strategies (Guyon and Elisseeff 2003), and finally present a new approach called Naive Sampling Feature Selection (NSFS), based on Monte Carlo sampling. Time windows (Dietterich 2002) are also tested to address data sequentiality. We use the popular videogame Minecraft as our evaluation domain.

Learning from Demonstration

Argall et al. (2009) formally define LfD as follows. Let S be the set of states the world can be in, and Y the set of actions an agent can perform. The mapping between states by way of actions is defined by a probabilistic transition function $T(s'|s, y) : S \times Y \times S \rightarrow [0, 1]$. If the state is not fully observable, an observation function $M : S \rightarrow Z$ maps states to observed states Z. A policy $\pi : Z \rightarrow Y$ selects actions based on observations of the world state, and can range from low-level motions to high level behaviors. A demonstration $T_i = [(z_1, y_1), ..., (z_n, y_n)]$ is defined as a sequence of observation-action pairs, where $z_i \in Z$, and $y_i \in Y$. The goal of LfD is, given training data consisting of a set of demonstrations $T = \{T_1, ..., T_n\}$, derive a policy for choosing an action based on the current observed world state.

Problem Statement

The problem that we address in this paper is that of feature selection in structured representations of sequential data:

- Structured Representations: we represent both actions and world states as logical clauses of the form $z_t = p_0 \wedge p_1, \wedge ... \wedge p_m$. Each predicate p_i takes the form $h(a_1, ..., a_r)$, where h is the functor and each a_i represents the value that each of the attributes of the predicate takes (note that both h and all the arguments a_i are constants), and represents some aspect of the world. For example, $health(20)$ would have $health$ being the functor and 20 being the value of the only attribute for that functor. Let $\mathcal{H} = \{h_0, ..., h_n\}$ be the set of all functors and $\mathcal{A}^{h_i}$ the set of all attributes for the functor h_i. Actions are also represented as logical clauses, where each individual predicate corresponds to a Minecraft action, and if the player executed more than one action at a time (e.g., walking while swinging a sword), the clause representing

$z_{162} = holding(17, 1, 0) \wedge rotation(sw, up) \wedge selectedBlock(17)$
$\wedge target(none) \wedge block(1, extreme) \wedge block(2, extreme)$
$\wedge block(3, extreme) \wedge block(4, high) \wedge block(17, high)$
$\wedge block(18, extreme) \wedge level(0) \wedge health(20) \wedge food(20)$
$\wedge pMob(pig, left, far) \wedge aMob(zombie, center, near),$
$\wedge pMob(pig, right, far) \wedge groundItem(17, right, near),$
$\wedge groundItem(17, left, far) \wedge cluster(17, 3, left, far),$
$\wedge closestStone(none, none) \wedge closestWorkbench(none, none)$
$y_{162} = rotating(sw), choppingWood(-1)$

Figure 1: An example world state and corresponding action.

Figure 2: A screenshot of Minecraft.

the action will have more than one predicate. An example of an observation-action pair can be seen in Figure 1.

- Data Sequentiality: for some behaviors, all of the relevant information might not be available in the current state, in which case data from past states should be taken into account. For example, if an agent passes a sign saying "turn right at the next intersection", when they approach the intersection they need to turn right, but the sign would no longer be in the current state information.

In structured representations, such as Horn Clauses (Nienhuys-Cheng and Wolf 1997), one equivalent of selecting features is to select which functors and attributes of each functor should be used. Specifically, we define the "set of features" F for a structured domain as the set of functors being used plus the set of attributes for each functor. Consequently, we define a subset of features $F' \subseteq F$ as a subset of all the functors and attributes in F. Any subset of features $F' \subseteq F$ must satisfy the condition that no attributes can be present if their corresponding functor is not present, *i.e.*: if $h_i \notin F'$, then $\mathcal{A}^{h_i} \cap F' = \emptyset$ (see above).

Minecraft

Our motivating domain, as well as our testbed, is the game of Minecraft (Figure 2), an open ended game that focuses on exploration and building. The game imposes no true goal on the player, instead letting them set their own. The players move about in a 3-dimensional world divided into blocks that (with a few exceptions) the player can pick up and put down as they please. Players are able to kill enemies, build/destroy structures, and collect items to craft new tools. Due to these attributes, Minecraft requires short term, almost reflexive decisions as well as long-term planning.

Feature Selection Methods

We evaluated the performance of seven different feature selection approaches compared against three baselines. Methods were also compared for time windows of size greater than one where feasible.

We explored a family of algorithms called *wrapper* methods (Kohavi and John 1997), which iteratively invoke a base learning algorithm with different features subsets and evaluate the performance of each in order to find the feature set that maximizes performance. Wrapper methods split the training set T into two parts: the *subtraining set* S and the *cross-validation set* C. Then, the wrapper method iteratively tests different feature subsets by training the base learning method using S and evaluating the resulting loss using C.

Moreover, some methods and baselines used assume a propositional representation of the data. In addition to the structured representation of our dataset, we created a propositional version in the following way (methods requiring this propositional representations are marked by the word "Propositional" in their names). We translated the structured representation of the world state to a fixed-size vector that contains a value for every attribute $a \in \mathcal{A}^{h_i} | h_i \in \mathcal{H}$, resulting in 98 propositional features in our dataset. When more than one predicate with the same functor appears in a world state (see for example $pMob$ in Figure 1, which appears twice in the game state), only the predicate which represents the enemy/object closest to the player is used to compute the corresponding feature value. When the predicate does not appear in a world state, its associated feature is set to 0.

Handling Sequential Data

For most standard supervised approaches, only the current world state is taken into account when learning. For LfD problems however, knowledge of past states is often required to fully capture the demonstrator's behavior. One common way of addressing this is time windows (Dietterich 2002), or simply passing in information from the past $k-1$ states as input in addition to the current state. This can improve results, but it also increases the computational cost significantly even for low values of k, so many of the feature selection methods that we employed were computationally unfeasible to run with a time window of size greater than 1.

Baselines

We compared against 3 baselines:

All Features Propositional (AFP): uses a propositional representation where all features are used for learning.

All Features Structural (AFS): uses a structured representation using all predicates and attributes.

Random (R): This method randomly selects an action from the training data.

Wrapper Methods

Propositional Wrapper (PW): This method uses a greedy search to attempt to find the best possible subset of features (Algorithm 1) (Kohavi and John 1997). The algorithm starts with an empty feature set, and iteratively adds the feature which will minimize loss. Once all possible features are added, the subset that yielded the lowest loss is returned.

Algorithm 1 PW(F, S, C)

1: $F_{best} = F_{subset} = \emptyset, F_{left} = F, l_{best} = 1,$
2: **while** $F_{left} \neq \emptyset$ **do**
3: $\quad f_{best} = argmin_{f \in F_{left}} loss(F_{subset} \cup f, S, C)$
4: $\quad l = loss(F_{subset} \cup f_{best}, S, C)$
5: $\quad F_{subset} = F_{subset} \cup f_{best}$
6: $\quad F_{left} = F_{left} \setminus f_{best}$
7: $\quad$ **if** $l < l_{best}$ **then**
8: $\quad\quad l_{best} = l, \quad F_{best} = F_{subset}$
9: $\quad$ **end if**
10: **end while**
11: **return** F_{best}

Algorithm 2 FSW$(\mathcal{H}, S, C)$

1: $F_{best} = F_{subset} = \emptyset, \mathcal{H}_{left} = \mathcal{H}, l_{best} = 1,$
2: **while** $F_{left} \neq \emptyset$ **do**
3: $\quad h_{best} = argmin_{h \in \mathcal{H}_{left}} loss(F_{subset} \cup h \cup \mathcal{A}^h, S, C)$
4: $\quad l = loss(F_{subset} \cup h_{best} \cup \mathcal{A}^{h_{best}}, S, C)$
5: $\quad F_{subset} = F_{subset} \cup h_{best} \cup \mathcal{A}^{h_{best}}$
6: $\quad \mathcal{H}_{left} = \mathcal{H}_{left} \setminus h_{best}$
7: $\quad$ **if** $l < l_{best}$ **then**
8: $\quad\quad l_{best} = l, \quad F_{best} = F_{subset}$
9: $\quad$ **end if**
10: **end while**
11: **return** F_{best}

We define the loss function $loss(Subset, S, C)$ as training the base learning algorithm on the set of demonstrations S using only the features in $Subset$, then calculating the loss of the learned classifier using the set of demonstrations C.

Functor Structured Wrapper (FSW): This is a wrapper method that works directly over the structured representation. It also employs a greedy search, but considers only the set of functors, i.e., it does not select among the set of attributes of each functor. As we can see in Algorithm 2, each time a functor h is selected (added to $F_{selected}$), all of its attributes $\mathcal{A}^{h_{best}}$ are also added to $F_{selected}$.

Attribute Structured Wrapper (ASW): The *Attribute Structured Wrapper* (ASW) operates similarly to Functor Structured Wrapper, but also selects which of the attributes of each selected functor to include in the structured representation (see Algorithm 3). The algorithm also uses a greedy search approach, and each time a functor is selected, the set of its attributes ($\mathcal{A}^{f_{best}} \cap F$) are added to the set of features to consider in future iterations (lines 7-9). Therefore, the algorithm might select a given functor, but only select a subset of its attributes to be added to the representation.

Subtractive from Min Attribute Structured Wrapper (SASW$_{min}$): One issue with the greedy search approach of ASW is when a functor is added to the selected features, none of its attributes are added initially. Thus, if a given functor is only important if a given attribute is included, it might not be added until very late in the feature selection process. To avoid this, we devised this alternative search process, which first calls Functor Structured Wrapper to obtain

Algorithm 3 ASW$(F, \mathcal{H}, S, C)$

1: $F_{best} = F_{subset} = \emptyset, F_{left} = \mathcal{H}, l_{best} = 1,$
2: **while** $F_{left} \neq \emptyset$ **do**
3: $\quad f_{best} = argmin_{f \in F_{left}} loss(F_{subset} \cup f, S, C)$
4: $\quad l = loss(F_{subset} \cup f_{best}, S, C)$
5: $\quad F_{subset} = F_{subset} \cup f_{best}$
6: $\quad F_{left} = F_{left} \setminus f_{best}$
7: $\quad$ **if** $f_{best} \in \mathcal{H}$ **then**
8: $\quad\quad F_{left} = F_{left} \cup (\mathcal{A}^{f_{best}} \cap F)$
9: $\quad$ **end if**
10: $\quad$ **if** $l < l_{best}$ **then**
11: $\quad\quad l_{best} = l, \quad F_{best} = F_{subset}$
12: $\quad$ **end if**
13: **end while**
14: **return** F_{best}

Algorithm 4 SASW$_{min}(\mathcal{H}, S, C)$

1: $F_{best} = F_{subset} = FSW(\mathcal{H}, S, C), l_{best} = 1,$
2: **while** $F_{subset} \neq \emptyset$ **do**
3: $\quad f_{best} = argmin_{f \in F_{subset}} loss(F_{subset} \setminus f, S, C)$
4: $\quad l = loss(F_{subset} \setminus f_{best}, S, C)$
5: $\quad F_{subset} = F_{subset} \setminus f_{best}$
6: $\quad$ **if** $l < l_{best}$ **then**
7: $\quad\quad l_{best} = l, \quad F_{best} = F_{subset}$
8: $\quad$ **end if**
9: **end while**
10: **return** F_{best}

the initial set of functors, and then employs a greedy search that subtracts attributes or functors one by one out of the initially selected functors, until no removal can be done without increasing the loss (see Algorithm 4). The algorithm considers both removing a functor h (along with all its attributes, $\mathcal{A}^h$), and removing just one attribute.

Subtractive from Max Attribute Structured Wrapper (SASW$_{max}$): Many different sets of features often achieve the same loss (since adding irrelevant features sometimes does not reduce the loss). This method operates identically to the previous (SASW$_{min}$), except it uses a modified version of the Functor Structured Wrapper which returns the *largest* set of features that achieves the minimum loss.

Filter Methods

Proportion-Filtered ASW (PF-ASW): this method was created to address the high computational cost of the ASW wrapper. In this method, the features are preprocessed by testing each feature individually and recording the results, then taking the top $X\%$ of them, where X is a user-defined parameter (see Algorithm 5 for the detailed procedure). This selection of top features then becomes the list of selectable features for ASW, the notion being that most of the helpful features will still be maintained while considerably reducing the computational cost of the algorithm. Three variants were tested: PF-ASW$_{10\%}$, PF-ASW$_{20\%}$, and PF-ASW$_{30\%}$, which keep the top 10%, 20% , and 30% of features, respectively.

Algorithm 5 PF-ASW($\mathcal{H}, S, C, X$)

1: $F_{best} = F_{subset} = \emptyset, F_{start} = \mathcal{H}, l_{best} = 1, storeLoss = \{\}$
2: **for** $f \in F_{start}$ **do**
3: storeFeatureAndLossPairs.add(loss(f,S,C))
4: **end for**
5: storeFeatureAndLossPairs.sortByLoss()
6: $F_{left} = storeLoss.getBestFeatures(X)$
7: **return** ASW($F_{left}, \mathcal{H}, S, C$)

Naive Sampling Feature Selection (NSFS)

All the wrapper methods above are based on greedy search (either greedily adding features one by one or removing them one by one). This is since searching over the complete space of possible feature subsets using systematic search is unfeasible since there is an exponential number of feature subsets to consider. In this section we present an alternative search method based on Monte Carlo sampling.

The key insight is that feature selection can be seen as a *Combinatorial Multi-Armed Bandit* (CMAB) problem (Ontanón 2013). A CMAB is an extension of the classic multi-armed bandit (MAB) problem (Kocsis and Szepesvári 2006), where at each iteration an agent picks a value for a set of n variables $X = \{X_1, ..., X_n\}$, where variable X_i can take K_i different values in order to maximize the cumulative reward (where the unknown stochastic reward function depends on the selected values). When solving a CMAB, an agent needs to estimate the potential rewards of each variable combination based on past observations, balancing exploration and exploitation in order to find the combination that maximizes expected reward. There is a combinatorial number of possible values (or macro-arms) which the agent can select at each iteration. Additionally, not all possible value combinations might be legal, a function $L : \mathcal{X} \rightarrow \{true, false\}$ determines which of them are.

Given a set of n features F, we define a CMAB with n binary variables, where X_i determines whether feature $f_i \in F$ will be selected or not. The legality function L prevents selecting an attribute but not its functor. *Naive Sampling Feature Selection* (NSFS) exploits such a formalization and works as follows (see Algorithm 6):

- The algorithm runs for a fixed number of iterations T (the computation budget). At each iteration, a CMAB sampling policy is used to select a set of features (line 4).
- With each set of features, the base learning algorithm is trained using C, its performance is evaluated by a test set made out of a *single* instance (a single observation-action pair picked at random from C, and removed from the training set before training) to obtain a reward (lines 5-6). Since we employ instance-based learning methods, training time is basically zero.
- At the end of the computation budget, the set of features that has been selected most often (i.e., the ones that the sampling policy considers as the one that maximizes the expected reward) is returned as the set of selected features ($mostFrequentMacroarm$ function in the algorithm).

Algorithm 6 NSFS($\mathcal{H}, S, C$)

1: $F = \mathcal{H} \cup \left(\bigcup_{h \in \mathcal{H}} \mathcal{A}^h\right)$
2: $CMAB$ = new with $|F|$ binary variables.
3: **for** $t = 1...T$ **do**
4: F' = Naïve Sampling($CMAB, \epsilon_0, \epsilon_l, \epsilon_g$)
5: (z, a) = randomly selected from any $T \in C$
6: $r = 1 - loss(F', S, \{\{(z, a)\}\})$
7: Update $CMAB$ reward for F' with r
8: **end for**
9: **return** $mostFrequentMacroarm(CMAB)$

Intuitively, evaluating the performance of each feature subset via testing it against a single instance in each iteration, minimizes the amount of time it takes to obtain a reward for a given feature subset, letting the algorithm test a larger number of feature subsets in the same computation time. However, since we are testing against a single instance, the loss computed for each feature has a large degree of randomness. However, CMAB sampling policies are designed to handle such stochastic reward functions.

Specifically, we employed *Naïve Sampling* (Ontanón 2013), a CMAB sampling strategy based on using nested ϵ-greedy sampling strategies. Naïve Sampling has three parameters: ϵ_0, ϵ_l, and ϵ_g, which are set to 0.3, 0.3, and 0.0 in our experiments, based on preliminary experiments. *Naïve Sampling*, initially behaves as if it selects macro-arms at random (when no information about the expected reward of each macro-arm is available), and iteratively converges to selecting only those macro-arms that are promising.

Experimental Evaluation

Experimental Setup

To evaluate our approach, we used Minecraft as our application domain. We collected five different traces each from two different worlds, all taken by one person. The same procedural generation seed was used to create the world for each set of 5, and traces were taken from the moment of world generation, for consistency. The data stored includes the players inventory, nearby clusters of blocks, passive entities, and aggressive entities. The average trace length is 723 observation-action pairs (40 seconds of gameplay), with each pair containing at least one action (pairs where no action is taken are not added to the traces). In all traces, the player's task was to "obtain a piece of cobblestone". The results reported below used 10-fold cross validation (where one trace was held out as the test set, and the other nine were used as the training set). Each algorithm was also tested with time windows of sizes from 1 to 4 where feasible.

We used *nearest neighbor* (Cover and Hart 1967) as our base learning method. For the propositional representations, we used the well known Euclidean distance. For the structured representations, we employed the *Jaccard* similarity, which is defined as the number of predicates shared between two world states, divided by the number of different predicates appearing in both world states: $J(z_1, z_1) = \frac{|z_1 \cap z_2|}{|z_1 \cup z_2|}$,

where $z_1 \cap z_2$ is a clause that contains only those predicates present both in z_1 and in z_2, and $z_1 \cup z_2$ is a clause that contains the union of predicates in z_1 and in z_2. $|\cdot|$ represents the number of predicates in a clause. For two predicates to be considered equal, both their functor and all their attributes must be identical.

The loss method employed was the normalized Levenshtein distance between the actions in the ground truth and the actions predicted by nearest neighbor. To do this we represented predicates as trees and employed Pawlik and Augsten's 2009 tree edit distance measure, as in our previous work (Packard and Ontañón 2015).

Results

Table 1 shows the average loss (lower is better) for each feature selection method used, for various time windows. Since even small time windows (of size k >1) can significantly increase runtime, many of the feature selection methods are not computationally feasible to run. A dash (—) in the table indicates that the test did not terminate after 100 hours, and was therefore considered infeasible. Table 2 shows the loss achieved by NSFS for varying numbers of iterations.

The method that achieved the best results for time windows of size 1 was PF-ASW$_{10\%}$, by a very narrow margin. FSW and NSFS performed the worst of the symbolic methods, but not significantly so. In fact, *none* of the symbolic feature selection methods performed statistically significantly better than any other, according to a 2-tailed t-test with $p = 0.05$. However, all symbolic methods performed significantly better than the three baselines. Moreover, FSW, PF-ASW$_{10\%}$, and PF-ASW$_{20\%}$ performed statistically significantly better than both the baselines and PW.

It can also be seen that a greater time window decreases loss, even without feature selection. In fact, using a time window of size 4 gives statistically significantly less loss that a time window of size 1 ($p < 0.05$). Time windows also helped many of the feature selection methods perform better, with the one exception being NSFS. We hypothesize that this is due to having a larger pool of irrelevant features for NSFS, which would make it take more iterations to converge on a subset of features, and also, the high level of randomness introduced by our reward assignment procedure (which we will revise in future work). However, increasing the number of iterations enough to solve this issue caused the algorithm to take longer than the 100 hour cutoff that was employed.

We assessed computational cost by measuring how many times the similarity measure was called during feature selection, since more calls to the similarity measure will typically result in a longer runtime. The number of similarity calls needed for each method with a time window of size 1 is shown in Table 3. As can be clearly seen, PF-ASW$_{10\%}$ and PF-ASW$_{20\%}$ perform by far the best in this regard, followed by NSFS. The next best are FSW, SASW$_{min}$, and PF-ASW$_{30\%}$, which require more calls to the similarity measure, but still considerably less than the other methods. It is also interesting that all of the structured feature selection methods require less calls to the similarity measure, thanks mainly to the fact that attributes are only considered if the corresponding functors are added.

Feat. Sel. Method	$k=1$	$k=2$	$k=3$	$k=4$
AFP	0.355	0.329	0.308	—
AFS	0.374	0.348	0.336	0.308
R	0.798	0.798	0.798	0.798
PW	0.333	—	—	—
FSW	0.265	0.245	—	—
ASW	0.244	—	—	—
SASW$_{min}$	0.244	—	—	—
SASW$_{max}$	0.244	—	—	—
NSFS	0.265	0.269	0.263	—
PF-ASW$_{10\%}$	0.243	0.210	0.205	—
PF-ASW$_{20\%}$	0.244	0.211	—	—
PF-ASW$_{30\%}$	0.247	0.243	—	—

Table 1: Average loss for various feature selection methods (lower is better) for various time window sizes (k).

NSFS Iterations	*Loss*	*Similarity Calls*
1,000	0.295	4.5 Million
2,000	0.288	9 Million
5,000	0.278	27.5 Million
10,000	0.274	45 Million
100,000	0.246	450 Million
200,000	0.248	900 Million
500,000	0.246	2,250 Million

Table 2: Average loss of NSFS for varying iterations (lower is better), and approximate number of similarity measure callsneeded to perform feature selection for a single test.

Related Work

The problem of learning behavior from demonstration has received significant attention in the literature. For example, Ross, Godron, and Bagnell study LfD in the context of the Super Tux Cart and Super Mario Bros games (Ross, Gordon, and Bagnell 2010). Other approaches to LfD include *Inverse Reinforcement Learning* (Abbeel and Ng 2004; Tastan and Sukthankar 2011), Dynamic Bayesian Networks (such as Hidden Markov Models) (Dereszynski et al. 2011; Ontañón, Montaña, and Gonzalez 2014), and supervised learning techniques (Sammut et al. 2014). The reader is referred to (Argall et al. 2009) and (Ontañón, Montaña, and Gonzalez 2014), for recent surveys of LfD.

Although the literature on feature selection is vast, little work has been done towards feature selection for structured representations of traces. For example, some examples of feature selection methods for propositional data are provided by Guyon and Elisseeff (Guyon and Elisseeff 2003). These methods are split into 4 categories: (1) Wrappers/Embedded Methods: which search the space of feature subsets, evaluating each one by re-training the base learning algorithm. (2) Nested Subset Methods: estimate the changes that occur in the objective function value when moving from node to node in the space of feature subsets and combine that information with a greedy search strategy. (3) Direct Objective Optimization: which try to maximize how good of a fit the feature subset is, at the same time as minimizing the size of

Feat. Sel. Method ($k = 1$)	*Loss*	*Similarity Calls*
AFP	0.355	N/A
AFS	0.374	N/A
PW	0.333	19,876 Million
FSW	0.265	1,058 Million
ASW	0.244	8,698 Million
SASW_{Min}	0.244	1,286 Million
SASW_{Max}	0.244	3,273 Million
NSFS	0.265	450 Million
$\text{PF-ASW}_{10\%}$	0.243	214 Million
$\text{PF-ASW}_{20\%}$	0.244	356 Million
$\text{PF-ASW}_{30\%}$	0.247	623 Million

Table 3: Average loss (lower is better),and required number of calls to the similarity measure during feature selection.

that subset. (4) Filter Methods: which use some evaluation function to directly rank or filter the set of attributes.

Specific examples are the wrapper method proposed by Maldonado and Weber for Support Vector Machines (Maldonado and Weber 2009), the filter based method by Hall (Hall 1999), the hybrid methods presented by Kabir, Shahjahan, and Murase (2012) and a method to complement wrappers by Floyd, Davoust, and Esfandiari (2008).

Conclusions

This paper presented a collection of feature selection strategies for structured, sequential data, including methods inspired by wrappers, filters, Monte Carlo sampling, and time windows. Although the motivating task is in the context of LfD, most methods presented in this paper apply to standard supervised learning. Our results indicate that filter/wrapper hybrid methods such as $\text{PF-ASW}_{10\%}$ achieve the best trade-off of performance and computation time. We did not observe significant gains moving from a greedy search strategy to a Monte Carlo search strategy, although improving our Monte Carlo approach is part of our future work.

As part of our future work, we would like to study how to further extend the proposed methods specifically for problems arising in LfD. For example, some actions in the game take time (such as breaking a block) and others only make sense within the context of a sequence of actions, so including previous actions as part of the world state might improve results. Finally, we would like to evaluate our approach with a range of different Minecraft tasks.

References

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

Argall, B. D.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems* 57(5):469–483.

Cover, T., and Hart, P. 1967. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on* 13(1):21–27.

Dereszynski, E. W.; Hostetler, J.; Fern, A.; Dietterich, T. G.; Hoang, T.-T.; and Udarbe, M. 2011. Learning probabilistic behavior models in real-time strategy games. In *AIIDE*.

Dietterich, T. G. 2002. Machine learning for sequential data: A review. In *Structural, syntactic, and statistical pattern recognition*. Springer. 15–30.

Floyd, M. W.; Davoust, A.; and Esfandiari, B. 2008. Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation. In *European Conference on Case-Based Reasoning*, 195–209. Springer.

Guyon, I., and Elisseeff, A. 2003. An introduction to variable and feature selection. *The Journal of Machine Learning Research* 3:1157–1182.

Hall, M. A. 1999. *Correlation-based feature selection for machine learning*. Ph.D. Dissertation, The University of Waikato.

Kabir, M. M.; Shahjahan, M.; and Murase, K. 2012. A new hybrid ant colony optimization algorithm for feature selection. *Expert Systems with Applications* 39(3):3747–3763.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*. Springer. 282–293.

Kohavi, R., and John, G. H. 1997. Wrappers for feature subset selection. *Artificial intelligence* 97(1):273–324.

Maldonado, S., and Weber, R. 2009. A wrapper method for feature selection using support vector machines. *Information Sciences* 179(13):2208–2217.

Nienhuys-Cheng, S.-H., and Wolf, R. d. 1997. *Foundations of Inductive Logic Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Ontañón, S.; Montaña, J. L.; and Gonzalez, A. J. 2014. A dynamic-bayesian network framework for modeling and evaluating learning from observation. *Expert Systems with Applications* 41(11):5212–5226.

Ontanón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Packard, B., and Ontañón, S. 2015. Learning behavior from demonstration in minecraft via symbolic similarity measures. In *Proceedings of the Foundations of Digital Games conference (FDG) 2015*.

Pawlik, M., and Augsten, N. 2009. RTED: A robust algorithm for the tree edit distance. In *Proceedings of the VLDB Endowment)*, volume 5, 334–345. Instanbul, Turkey: VLDB Endowment.

Ross, S.; Gordon, G. J.; and Bagnell, J. A. 2010. No-regret reductions for imitation learning and structured prediction. *CoRR* abs/1011.0686.

Sammut, C.; Hurst, S.; Kedzier, D.; Michie, D.; et al. 2014. Learning to fly. In *Proceedings of the ninth international workshop on Machine learning*, 385–393.

Tastan, B., and Sukthankar, G. R. 2011. Learning policies for first person shooter games using inverse reinforcement learning. In *Proceedings of AIIDE 2011*.