Machine Learning (2024) 113:4487–4522
https://doi.org/10.1007/s10994-024-06565-4

Classification with costly features in hierarchical deep sets

Jaromír Janisch[1] · **Tomáš Pevný**[1] · **Viliam Lisý**[1]

Received: 16 December 2022 / Revised: 13 February 2024 / Accepted: 2 May 2024 /
Published online: 22 May 2024

Abstract
Classification with costly features (CwCF) is a classification problem that includes the cost of features in the optimization criteria. Individually for each sample, its features are sequentially acquired to maximize accuracy while minimizing the acquired features' cost. However, existing approaches can only process data that can be expressed as vectors of fixed length. In real life, the data often possesses rich and complex structure, which can be more precisely described with formats such as XML or JSON. The data is hierarchical and often contains nested lists of objects. In this work, we extend an existing deep reinforcement learning-based algorithm with hierarchical deep sets and hierarchical softmax, so that it can directly process this data. The extended method has greater control over which features it can acquire and, in experiments with seven datasets, we show that this leads to superior performance. To showcase the real usage of the new method, we apply it to a real-life problem of classifying malicious web domains, using an online service.

Keywords Classification with costly features · Deep reinforcement learning · Deep sets · Hierarchical multiple-instance learning · Hierarchical softmax · Policy decomposition · Application programming interface · Budget · Classification · Structured data

1 Introduction

The online world around us is composed of structured relational data. For example, users of a social network can be described by a set of their friends, posts they published or commented on, likes they received and from whom. This data is often not available as a whole, but rather provided on request by a paid service. Application Programming Interfaces

Editor: Steven Schockaert.

✉ Jaromír Janisch
jaromir.janisch@fel.cvut.cz

Tomáš Pevný
tomas.pevny@fel.cvut.cz

Viliam Lisý
viliam.lisy@fel.cvut.cz

[1] Artificial Intelligence Center, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

(APIs) are specific examples. Google search, maps, Youtube, social networks such as Facebook or Twitter, and more provide rich information that may be free in low volumes but is charged as soon as you consider using it commercially. Even if the complete data is available, one can still save substantial resources by using only its fraction, e.g., when analyzing a large number of users. Recently, we see that sustainability and ecology have started to play an increasingly larger role and the interest could lie in lowering electricity consumption or CO_2 production.

In the social network example, the use of the data may be targeted advertising. As another example, let us consider the field of computer security. One may be interested in whether a particular web domain is legitimate or malicious. Specialized services provide rich sets of features about the requested domain, such as known malware binaries communicating with the domain, WHOIS information, DNS resolutions, subdomains, associated email addresses, and, in some cases, a flag that the domain is known to be malicious. The user can further probe any detail, e.g., after acquiring a list of subdomains, the user can focus on one of them and request more information about it. Again, access to the service may be charged, therefore there is a natural pressure to limit the number of requests.

The problem at hand has multiple names—*classification with costly features* (CwCF) (Janisch et al., 2020), *active feature acquisition and classification* (Shim et al., 2018) or *datum-wise classification* (Dulac-Arnold et al., 2012). In essence, the problem is to sequentially gather features, in a unique order for each sample, and stop optimally when ready to classify. Optimality is usually defined as one of the two: (1) a trade-off between the total cost of the features and the classification accuracy or (2) maximal accuracy with the condition that the total per-sample cost cannot exceed a specified budget. We emphasize that a potentially different feature subset acquired in different order is retrieved for each sample. For example, with some samples, the classification may be made after a single feature is acquired. Other samples may require multiple or all features, and the decision which is made sequentially, based on the values revealed so far. Note that the number of possible ways to process a sample is exponential in its size.

Over the years, many different algorithms have been developed for this problem. Some employ decision trees (Xu et al., 2012; Kusner et al., 2014; Xu et al., 2013, 2014; Nan et al., 2015, 2016; Nan & Saligrama, 2017), recurrent neural networks (Contardo et al., 2016), linear programming (Wang et al., 2014a, b) or partially observable Markov decision processes (Ji & Carin, 2007). There are multiple reinforcement learning (RL) methods based on Dulac-Arnold et al. (2012), e.g., by Janisch et al. (2019, 2020), Shim et al. (2018). The problem itself, or its variations, appears across multiple fields: medicine (Peng et al., 2018; Lee et al., 2020a; Song et al., 2018; Vivar et al., 2020; Lee et al., 2020b; Shpakova & Sokolovska, 2021; Zhu & Zhu, 2020; Goldstein et al., 2020; Erion et al., 2022), meteorology (Banerjee et al., 2020), data analysis (Ali et al., 2020), surveillance (Xu et al., 2021; Liu et al., 2018) or network security (Badr, 2022).

Despite the clear spread of the problem and its applications, we identified a substantial lack on the side of available algorithms. As we have shown in the introductory examples, a data sample is often provided in a complex structure, not a fixed-length vector. Formats such as XML or JSON, to which newly acquired information is sequentially added, are better suited. These formats commonly contain lists of elements with a priori undefined lengths and nested objects. For example, imagine a list of a user's posts (see Fig. 1). However, the common requirement of the available algorithms, which we surveyed above, is a flat structure of the samples. In other words, it is assumed that the samples can be described as fixed-size vectors, with their slices allocated to predefined features.

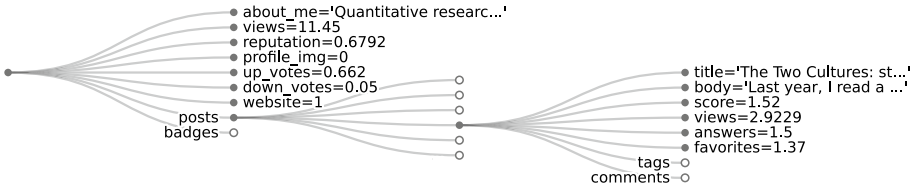


Fig. 1 A pruned data sample from our *stats* dataset, which is extracted from Stats StackExchange online service. The variable number of badges, posts, and their tags and comments means that each sample contains a different number of features. Application of existing techniques (e.g., original CwCF) would require alteration of the data. As a better alternative, we present a modified method that naturally works with the structured data and can select individual features in the hierarchy

If we want to apply the existing algorithms to this structured data, we need to process the samples so they can be described as fixed-size vectors. However, as we show in this article, this approach leads to sub-optimal results. It is much better to provide a means for the algorithm to select individual features anywhere in the structure. Eventually, this is what we expect—the algorithm that can request a few relevant features for one of the user's posts can be much more efficient than an algorithm that uses an aggregated version of all posts with pre-selected features. Note that a pre-selected feature ordering based on their importance is difficult because the number of features differs in every sample. E.g., in the social network example, it is difficult to statically determine the importance of a post's title, because each sample has a different number of posts.

In this article, we extend the original CwCF framework to naturally work with the structured data, which presents two main challenges. First, we need a way to process the data at the input. Deep Sets (Zaheer et al., 2017) is a technique to process variable-sized input and Hierarchical Multiple-Instance Learning (HMIL) is its extension for hierarchically nested data (Pevný & Somol, 2016). It defines a special neural network architecture that accommodates to the specified data and creates its embedding. The second challenge lies in the fact that the original CwCF framework assumes a fixed number of features to select from and that the action space is *static*. However, this assumption does not hold in our case—the data contains lists of (possibly nested) objects, and only a part of the complete sample is visible at any moment. Since we map visible features with unknown values to actions, there is a different number of actions available to the algorithm at any moment. Moreover, there is no a priori known upper bound for the number of actions. Inspired by a technique from natural language processing (Morin & Bengio, 2005), we take advantage of the hierarchical composition of the features and propose to decompose the policy analogically to their structure.

Finally, we demonstrate the extended CwCF framework with a set of experiments. First, we design a synthetic dataset which we use to analyze the algorithm's behavior. Second, we demonstrate the detection of malicious web domains with a real-world service. For this purpose, we created an offline dataset by collecting information about around 1200 domains using the service's API. This dataset enables us to perform the experiments efficiently and credibly imitates real communication with the service. Third, we quantitatively test the methods in five more datasets adapted from public sources.

Let us summarize the contributions of this manuscript:

1. We formalize and bring the community's attention to a novel variant of an important problem (Sect. 4).
2. We extend the existing CwCF framework to work with structured data containing lists and nesting, which was not possible before. This includes processing the data on input and factorizing the dynamic action space to select individual features (Sect. 5). Other minor contributions include:
   - We provide a formula to estimate the gradient of the policy entropy (required for the A2C algorithm) when only the probability of single action is known.
   - We split the classifier and feature selection policy, leading to better sample complexity.
   - We provide an unbiased loss for the classifier, weighted by the terminal action probability.
3. We evaluate our algorithm empirically and compare it to several alternatives, showing its superior performance. We execute the model with data from a real online service, proving its usefulness in a real-life scenario (Sect. 6).
4. We release seven datasets in a unified format to benchmark algorithms for this problem (five datasets are adapted from existing public sources, and two are completely new). We also release the complete code with scripts to reproduce the experiments.

This article is organized as follows. A detailed overview of the related work is presented in Sect. 2. Next, we describe the basic blocks we build upon in Sect. 3. Then we formalize the problem and formal changes to CwCF in Sect. 4. Section 5 focus on the algorithm and which practical changes are required. Experiments are presented in Sect. 6. Finally, Sect. 7 provides answers to a few common questions and 8 concludes the manuscript. Supplementary Material provides auxiliary information that did not fit the main text, such as dataset details, hyperparameters, visualizations, and training graphs.

2 Related work

This work is a direct extension of the Classification with Costly Features (CwCF) framework, originally defined by Dulac-Arnold et al. (2012) and lately advanced by Janisch et al. (2020, 2019). All these algorithms are based on reinforcement learning (RL) but work only with fixed-length vectors. Shim et al. (2018) proposes a method for sets of features, but cannot cope with nesting. We have covered some of the existing approaches (Xu et al., 2012, 2013, 2014; Kusner et al., 2014; Nan et al., 2015, 2016; Nan & Saligrama, 2017; Contardo et al., 2016; Wang et al., 2014a, b; Ji & Carin, 2007; Dulac-Arnold et al., 2012; Janisch et al., 2019, 2020; Shim et al., 2018) and applications (Peng et al., 2018; Lee et al., 2020a; Song et al., 2018; Vivar et al., 2020; Lee et al., 2020b; Shpakova & Sokolovska, 2021; Zhu & Zhu, 2020; Goldstein et al., 2020; Erion et al., 2022; Banerjee et al., 2020; Ali et al., 2020; Xu et al., 2021; Liu et al., 2018; Badr, 2022) for the CwCF problem in Introduction.

Aside from the references mentioned above, multiple papers focus on a similar class of problems or improve the algorithms somehow. Wang et al. (2015) creates macro-features from different disjoint subsets of features. Trapeznikov and Saligrama (2013) and Liyanage et al. (2021) use a fixed order of features, while the latter provides an analytical solution to select them optimally. Tan (1993) analyzes a similar problem but requires memorization

of all training examples. Li and Oliva (2021) uses RL with a generative surrogate model that provides intermediary rewards by assessing the information gain of newly acquired features and other side information. Bayer-Zubek and Dietterich (2005) presents multiple approaches based on the AO* algorithm that searches the policy space, applicable in domains with discrete feature values. A case with a hard budget was explored in Kapoor and Greiner (2005). Deng et al. (2007) approached the problem with multi-armed bandit techniques. Cesa-Bianchi et al. (2011), Zolghadr et al. (2013) analyze the problem theoretically. Kachuee et al. (2019) uses heuristic reward to guide an RL-based algorithm.

A related problem is feature selection (Guyon & Elisseeff, 2003) which pre-selects a fixed set of features for all samples. However, in CwCF and similar approaches, the features are selected dynamically and sequentially. That is, for any particular sample, features are acquired one by one, and each decision is guided by the information gathered so far. This way, a different set of features is acquired for any particular sample. This approach requires more resources to train and execute but can provide higher performance (i.e., higher accuracy with the same average cost). Several approaches extend the feature selection to include costs of the features (Maldonado et al., 2017; Bolón-Canedo et al., 2014). Still, they are designed to find a set of features common for the whole dataset and cannot work with structured data.

In this work, we use Hierarchical Multiple-Instance Learning (HMIL) to process the structured data (Pevný & Somol, 2017, 2016; Pevný & Kovařík, 2019; Mandlík et al., 2022), which is an extension of Deep Sets (Zaheer et al., 2017). In some deep RL problems, the action space is composed of orthogonal dimensions and existing techniques can be used to factorize it (Tang & Agrawal, 2020; Chen et al., 2019; Metz et al., 2017). In our case, the features are arranged in a tree-like structure and we factorize the corresponding action space with hierarchical softmax, a technique similar to the one used in natural language processing (Morin & Bengio, 2005; Goodman, 2001).

We optimize our model with the A2C algorithm derived from (Mnih et al., 2016), which belongs to a class of policy gradient RL algorithms (Sutton & Barto, 2018). It can be replaced with another algorithm from its class that works with discrete actions (e.g., TRPO (Schulman et al., 2015) or PPO (Schulman et al., 2017)). While the use of the A2C algorithm is enough for the purposes of this paper, we note that any recent or future algorithm from the RL community may result in improved performance and better sample complexity.

The problem is distantly related to graph classification algorithms (e.g., (Zhou et al., 2018; Hamilton et al., 2017; Perozzi et al., 2014; Kipf & Welling, 2016)). These algorithms either aim to classify graph nodes or the graph itself as a whole. In our case, we assume that the data is structured in a *tree*, constructed around a point of interest (e.g., a particular web domain). For this kind of data, the HMIL algorithm is better suited and less expensive than the general message-passing. Moreover, the graph classification algorithms do not involve sequential feature acquisition, nor account for the costs of features.

3 Preliminaries

This section describes the methods we build upon in this work. Our method is based on the Classification with Costly Features (CwCF) (Janisch et al., 2019, 2020) framework to set the objective and reformulate the problem as an MDP. However, structured data pose nontrivial challenges due to their variable input size and the variable number of actions. To

create an embedding of the hierarchical input, we use an extension of Deep Sets (Zaheer et al., 2017) called Hierarchical Multiple-Instance Learning (HMIL) (Pevný & Somol, 2016; Mandlík et al., 2022). To select the performed actions, we use hierarchical softmax (Morin & Bengio, 2005; Goodman, 2001). To train our agent, we use Advantage Actor Critic (A2C) (Mnih et al., 2016), a reinforcement learning algorithm from the policy gradient family.

3.1 Classification with costly features

Let us start by explaining the core concept of the Classification with Costly Features (CwCF) (Janisch et al., 2019, 2020). In CwCF, a data sample consists of *features* (e.g., a user's name, reputation, etc.), each of which has a defined cost. Initially, the sample's feature values are unknown. The algorithm proceeds sequentially, and at each step, it decides whether to acquire another feature and which, or classify the sample. Note that the order of features is not fixed, but chosen dynamically. The objective is to optimally balance the total cost of features and classification accuracy, averaged over the dataset. Compared to feature selection (Guyon & Elisseeff, 2003), this approach can achieve higher accuracy with the same cost because it can select a different set of features for each sample. The limitation of the framework is that it assumes that every sample contains exactly the same features and that they can be converted to a fixed-length vector. However, if the sample contains "a list of user's posts", the original CwCF does not provide a way to process it.

The following paragraph defines the problem formally. Let $\mathcal{D}$ be a dataset containing data points (x, y), where x is the sample and y is its label. Let $\mathcal{X}$ be the input space and $\mathcal{Y}$ the set of all labels. We willingly do not define the $\mathcal{X}$ more precisely to allow a wider interpretation of what a feature value is (the CwCF framework defined it as $\mathcal{X} \subseteq \mathbf{R}^n$). Let $\mathcal{F}$ be the set of all possible features. Each feature has a predefined real-valued cost and the cost function $c : \wp(\mathcal{F}) \to \mathbf{R}$ returns their sum, where the $\wp$ symbol denotes a power set. Let the tuple (y_θ, k_θ) denote a model parametrized with θ, where $y_\theta : \mathcal{X} \to \mathcal{Y}$ returns the label and $k_\theta : \mathcal{X} \to \wp(\mathcal{F})$ returns the features used. The objective is:

$$\min_\theta \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[\ell_{rl}(y_\theta(x), y) + \lambda c(k_\theta(x))\right] \tag{1}$$

Here, ℓ_{rl} denotes a classification loss, commonly defined as binary (0 in case of mismatch, -1 otherwise). $\lambda \in \mathbf{R}$ is a trade-off factor between the accuracy and the cost. Minimizing this objective means minimizing the expected classification loss together with the λ-scaled per-sample cost.

Alternatively, CwCF provides (Janisch et al., 2020) two other possible objectives. First, the algorithm can be modified to allow the user to specify directly a per-sample average budget $b \in \mathbf{R}$ and avoid λ. The objective then becomes:

$$\min_\theta \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[\ell_{rl}(y_\theta(x), y)\right], \text{ s.t. } \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[c(k_\theta(x))\right] \leq b \tag{2}$$

Finally, it is possible to set a hard per-sample budget that cannot be exceeded for any sample. The objective is then:

$$\min_\theta \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[\ell_{rl}(y_\theta(x), y)\right], \text{ s.t. } \forall x : c(k_\theta(x)) \leq b \tag{3}$$

We chose to build our extensions with the objective in Eq. (1), as it corresponds to the vanilla algorithm, and the rest of the paper will mention only this one. If the application demands it, the other two objectives are also possible. We included them for completeness and reference. The interested reader can find more details about their implementation in Janisch et al. (2020).

The way to solve Eq. (1) is to construct a special Markov decision process (MDP), in which a single sample (x, y) is analyzed per episode and the total episode reward R is:

$$R = -\left[\ell_{rl}(y_\theta(x), y) + \lambda c(k_\theta(x))\right]$$

Finding an optimal policy parametrized with θ equals to maximizing the expected reward, thus solving Eq. (1). The MDP is constructed as follows. In a particular episode with a sample (x, y), the state space $\mathcal{S}$ consists of states $s = (x, y, \bar{\mathcal{F}})$, where $\bar{\mathcal{F}} \subseteq \mathcal{F}$ is the set of currently observed features. The agent only sees an observation $o(x, \bar{\mathcal{F}})$, which denotes only the parts of x corresponding to features $\bar{\mathcal{F}}$. It also does not know the label y. Each episode starts with an initial state $s_0 = (x, y, \emptyset)$. The action space $\mathcal{A}$ corresponds to features and class labels, $\mathcal{A} = \mathcal{A}_f \cup \mathcal{A}_t$, where $\mathcal{A}_f = \mathcal{F}, \mathcal{A}_t = \mathcal{Y}$ (t in $\mathcal{A}_t$ as *terminal*). Typically, the already acquired features are removed from the selection, hence $\mathcal{A}_f(s) = \mathcal{F} \setminus \bar{\mathcal{F}}$. After performing an action selecting a feature, the reward is proportional to its negative cost, and the feature value is disclosed. After a classifying action, the episode terminates, and the reward is the negative loss of classification. Formally, the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbf{R}$ and transition function $t : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ are defined as follows:

$$r(s, a) = \begin{cases} -\lambda c(a) & \text{if } a \in \mathcal{A}_f \\ -\ell_{rl}(a, y) & \text{if } a \in \mathcal{A}_t \end{cases}$$

$$t(s, a) = \begin{cases} (x, y, \bar{\mathcal{F}} \cup a) & \text{if } a \in \mathcal{A}_f \\ \mathcal{T} & \text{if } a \in \mathcal{A}_t \end{cases}$$

Here, $\mathcal{T}$ denotes the terminal state. When the episode terminates, the final action is a class prediction, and it is used as the model output y_θ. Finally, the set of all acquired features is used as $k_\theta = \bar{\mathcal{F}}$.

The MDP defined above is solved with a deep reinforcement learning algorithm. The result is a policy π_θ that prescribes which actions to take in which states. In the original CwCF implementation, the RL algorithm was DQN (Mnih et al., 2015) with several improvements (Van Hasselt et al., 2016; Wang et al., 2016; Munos et al., 2016). However, the method does not hinge on a particular algorithm, and another one can be easily used.

The Eq. (1) poses a multi-criterial optimization problem that balances the classification accuracy in ℓ_{rl} and the cost of used features in λc, for a fixed λ. The optimal behavior for $\lambda \to \infty$ is to refrain from acquiring any features and immediately classify with the most populous class, given the statistics of the training dataset. With the other extreme, $\lambda = 0$, a classifier that uses all features can be used to estimate a lower bound of the accuracy. Still, it is only a lower bound, since a different model may provide a better accuracy. For the points between, i.e., $\lambda \in (0, \infty)$, the issue is the same—we can only find a lower bound (e.g., with baseline methods). Finally, note that Eq. (1) focuses on the training set performance, but the ultimate goal is to find a model that generalizes to unseen data points.

3.2 A2C algorithm

The method presented in this paper depends on hierarchical policy decomposition (explained in Sect. 5.5), which is possible if the policy is probabilistic. However, the original CwCF uses the DQN algorithm that outputs a deterministic policy that cannot be easily factored. Therefore, we propose to use the Advantage Actor-Critic algorithm (A2C) (Mnih et al., 2016), a basic policy gradient algorithm to find the policy π_θ. However, we note that any other algorithm from the policy gradient family with discrete actions (e.g., (Schulman et al., 2015, 2017)) could be used in its place. This is an advantage of RL-based methods—any recent or future improvement in deep RL algorithms can be immediately used with this method to improve its performance or sample complexity.

A detailed description of the A2C algorithm follows. An MDP is a tuple $(\mathcal{S}, \mathcal{A}, t, r, \gamma)$, where $\mathcal{S}$ represents the state space, $\mathcal{A}$ is a set of actions, $t(s, a)$ is a transition function returning a distribution of states after taking an action a in a state s, $r(s, a, s') \in \mathbf{R}$ is a reward function that returns a reward for a transition from a state s to s' through an action a, and $\gamma \in (0, 1]$ is a discount factor. The A2C algorithm iteratively optimizes a policy $\pi_\theta : \mathcal{S} \to P(\mathcal{A})$, where $P(\mathcal{A})$ denotes a probability distribution over actions, and a value estimate $V_\theta : \mathcal{S} \to \mathbf{R}$ with model parameters θ to achieve the best cumulative reward in a given MDP. Let us define a state-action value function $Q(s, a) = \mathbb{E}_{s' \sim t(s,a)}[r(s, a, s') + \gamma V_\theta(s')]$ and an advantage function $A(s, a) = Q(s, a) - V_\theta(s)$. Then, the policy gradient $\nabla_\theta J$ and the value function loss L_V are:

$$\nabla_\theta J = \mathop{\mathbb{E}}_{s,a \sim \pi_\theta, t} \left[A(s, a) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right] \tag{4}$$

$$L_V = \mathop{\mathbb{E}}_{s,a,s' \sim \pi_\theta, t} \left[q(s, a, s') - V_\theta(s) \right]^2 \tag{5}$$

$$q(s, a, s') = r(s, a, s') + \gamma V_{\theta'}(s') \tag{6}$$

where θ' is a fixed copy of parameters θ and $\pi_\theta(a \mid s)$ denotes the probability of action a under policy π_θ in state s.

To prevent premature convergence, a regularization term L_H in the form of the average policy entropy is used:

$$L_H = \mathop{\mathbb{E}}_{s \sim \pi_\theta, t} \left[H_{\pi_\theta}(s) \right] ; H_\pi(s) = - \mathop{\mathbb{E}}_{a \sim \pi(s)} \left[\log \pi(a \mid s) \right] \tag{7}$$

The total loss is computed as $L_{pg} = -J + \alpha_v L_V - \alpha_h L_H$, with α_v, α_h learning coefficients. The algorithm iteratively gathers sample runs according to a current policy π_θ, and the traces are used as samples for the above expectations. Then, an arbitrary gradient descent method is used with the gradient $\nabla_\theta L_{pg}$. Often, multiple environments are run in parallel to get a better gradient estimate. Note that while (Mnih et al., 2016) used asynchronous gradient updates, A2C performs the updates synchronously.

3.3 Hierarchical multiple-instance learning

In our method, we need a way to process structured data. Our data samples are trees of features and they can contain nested lists of objects, similar to XML and JSON formats. To process this data on input, we use an extension of Deep Sets (Zaheer et al., 2017) for hierarchical data, called Hierarchical Multiple-Instance Learning (HMIL) (Pevný & Somol, 2016; Mandlík et al., 2022). For an illustration of how HMIL works, see Fig. 2.

Let us start with MIL (Pevný & Somol, 2017), which presents a neural network architecture to learn an embedding of an unordered set (called a *bag*) $\mathcal{B}$, composed of m items $v_{\{1..m\}} \in \mathbf{R}^n$. The items are simultaneously processed into their embeddings $z_{v_i} = f_{\vartheta_\mathcal{B}}(v_i)$, where $f_{\vartheta_\mathcal{B}}$ is a non-linear function with parameters $\vartheta_\mathcal{B}$, shared for the bag $\mathcal{B}$. All embeddings are processed by an aggregation function g, commonly defined as an element-wise mean or max operator. The whole process creates a bag's embedding $z_\mathcal{B} = g_{i=1..m}(z_{v_i})$, and is differentiable.

HMIL extends the framework so that it works with nested bags. In MIL, features are real scalars or vectors. In HMIL, a feature can also be a bag of items with the restriction that all the items share the same feature types. Different bags $\mathcal{B}$ have different parameters $\vartheta_\mathcal{B}$ and are recursively processed as in MIL, starting from the hierarchy's leaves and proceeding to the root. The resulting intermediary embeddings $z_\mathcal{B}$ are used as feature values (see Fig. 2). The soundness of the hierarchical approach is theoretically studied by Pevný and Kovařík (2019).

4 Problem

In this paper, we extend the CwCF framework (see Sect. 3.1) to work with the structured data. This kind of data can be naturally processed with the HMIL architecture (Sect. 3.3). In this section, we describe what structured data means and how the problem formulation changes.

4.1 Structured data

Compared to the data usually processed in machine learning, structured data, as we define it, cannot be described by fixed-length vectors. The main difference is that the samples can contain nested sets with a priori unknown cardinality. However, the

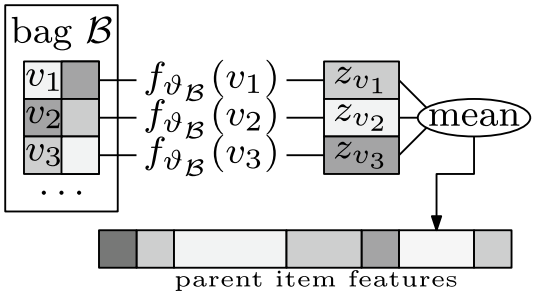
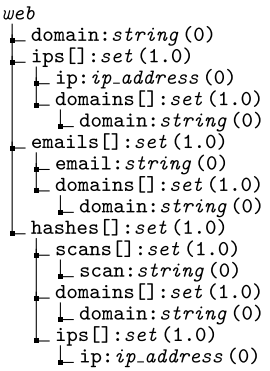


Fig. 2 Illustration of the bag embedding in HMIL. Objects in the bag $\mathcal{B}$ are processed with $f_{\vartheta_\mathcal{B}}$ and aggregated. The result is used as the feature value for the parent object. The process recursively embeds the whole sample

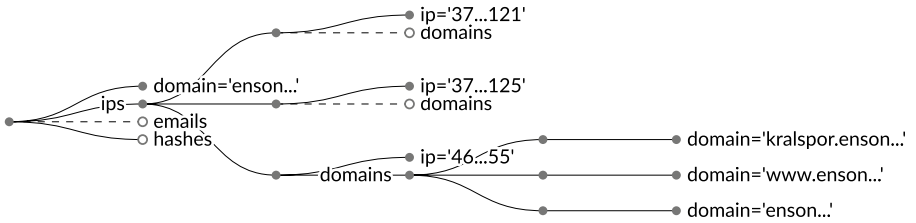
structure of the samples is strictly defined. Below, we define the structured data with terms *schema* and *sample*.

Dataset schema recursively describes the structure, features, their types, and costs. Formally, let an *object schema* be a collection of tuples *(name, type, cost, children_schema)*, where each tuple describes a single feature with its name, data-type, and non-negative real-valued cost. For features with *type=set*, the *children_schema* is an object schema describing the objects in this set. For other features, *children_schema*=∅. A dataset schema $\Sigma_{\mathcal{D}}$ is an object schema describing the whole sample.

Data sample is a collection of feature values, composed in a tree, and its structure strictly follows the schema $\Sigma_{\mathcal{D}}$. Formally, let an *object* be a collection of feature values with types described by the corresponding object schema. We call each feature with *type=set* a *set feature*, and it is a collection of objects whose features are typed by the corresponding *children_schema*. Other features are called *value features*.



(a) schema of the *threatcrowd* dataset



(b) a partial sample

Fig. 3 The schema and a partial sample for the *threatcrowd* dataset. **a** The schema shows the feature names, their types, and their cost in parentheses. A *set* type denotes that this feature contains a set of objects, whose features are described in the level below. **b** A partial sample. The full circles and lines denote features with known feature values. Among other information, the example shows that a list of domains was acquired for one of the IP addresses (`46..55`) with a reverse lookup

Both the schema and sample can be visualized as a tree. Figure 3a shows an example of a schema *threatcrowd* dataset. The schema specifies that each sample contains a free feature *domain* with type `string` and sets of *ips*, *emails*, and *hashes*. Objects in these sets have their own features (e.g., each IP address has a set of reversely translated *domains*). Figure 3b shows an incomplete sample as it would be seen by the augmented CwCF algorithm (only some of the features were acquired). Objects and their features are composed into a tree, according to the schema.

Note that our definition assumes that the cost of a particular feature across all samples is constant. While this assumption decreases the framework's flexibility, we argue that it is reasonable for real-world data where the cost of features can be usually precisely quantified upfront (e.g., the cost of an API request).

Last, it is useful to define a *path* and *prefix* of a feature in a particular sample. Let a path of a feature denote feature names and object positions in sets as a sequence from the root of the sample to the corresponding feature. We use the common programming syntax to denote the path. For example, we can write the path of features from the example in Fig. 3b as *ips[0].ip* (the value of the first IP address), or *ips[1].domains[0].domain* (the first domain of the second IP address). Let a prefix $pre(\kappa)$ of a feature κ be its path without the last item. For example, $pre(ips[0].ip) = ips[0]$.

Note that while we address individual objects in a set by their index, we do this solely for the purposes of definitions and implementation. We assume that the order of objects does not have any predictive value.

4.2 CwCF with structured data

The original CwCF method (see Sect. 3.1) worked with samples $x \in \mathbf{R}^n$. However, the data discussed in this paper cannot be easily converted to this Euclidean space. To accommodate for the issue, we present the following changes.

First, in CwCF, $\mathcal{F}$ denotes a set of all features. However, with structured data, the number of features is no longer constant across samples, as each sample can contain multiple objects in its sets. Therefore, let $\mathcal{F}(x)$ be a sample-dependent set of all features for a particular sample.

Second, a feature can be acquired only if its prefix has been obtained. For example, *ips[0].ip* cannot be acquired before the set *ips* or the object *ips[0]* is obtained. Formally, we modify the available feature-selecting actions to $\mathcal{A}_f(s) = \{\kappa \in (\mathcal{F}(x) \setminus \bar{\mathcal{F}}) \mid pre(\kappa) \in \bar{\mathcal{F}}\}$. These actions correspond to features whose values are unknown, hence we call these features *unobserved*. As a minor optimization that facilitates training, we propose recursively processing the corresponding subtree and acquiring all features with zero cost, whenever a set feature is acquired.

Third, we decouple the classifier y_θ from the policy π_θ. This change is not related to the structured data but results in improved performance and sample complexity. This is because the classifier can now be trained independently in every state and the policy is not burdened by the classification. Formally, we modify the set of terminal actions to include only a single terminal action a_t, $\mathcal{A}_t = \{a_t\}$. The classifier y_θ is now separately trained on observations $o(x, \bar{\mathcal{F}})$ (remember that the observation discloses the parts of x corresponding

to features $\bar{\mathcal{F}}$). To simplify notation, let $\bar{x} = o(x, \bar{\mathcal{F}})$. The final prediction $y_\theta(\bar{x})$ is used when the episode terminates. The reward function needs to reflect this change:

$$r(s,a) = \begin{cases} -\lambda c(a) & \text{if } a \in \mathcal{A}_f \\ -\ell_{rl}(y_\theta(\bar{x}), y) & \text{if } a \in \mathcal{A}_t \end{cases}$$

Note that we use parameters θ for both π_θ and y_θ. Commonly, both of these functions are implemented as a neural network with shared layers and as such, their parameters overlap.

The original CwCF method solved a finite horizon MDP, since, for any dataset, there was a fixed number of features to acquire. To preserve this property in the modified framework, we need to add two assumptions. First, we assume that the dataset schema is finite, i.e., the feature hierarchy is limited in depth. The second assumption is that the number of objects in any set of any data sample is finite. These two assumptions together limit the number of features of any sample, therefore the modified method still operates within a finite horizon MDP.

Given these simple changes, the CwCF framework is *formally* ready to work with structured data. However, the situation is more difficult implementation-wise, which is discussed in the following section.

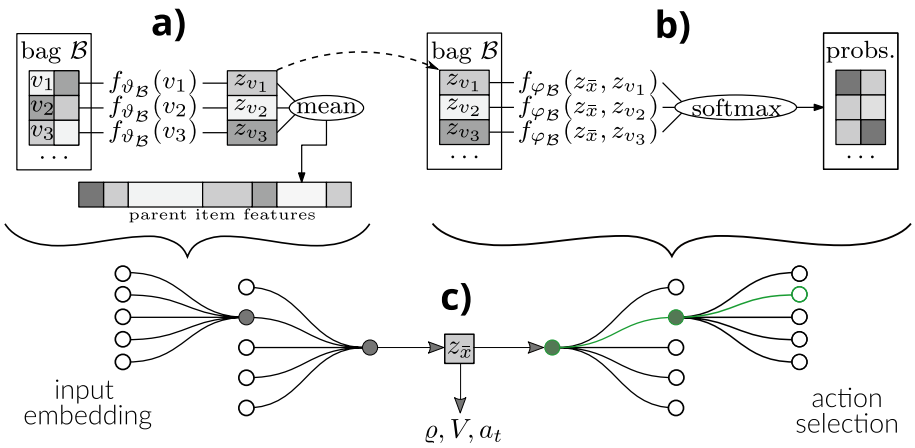


Fig. 4 **a** The input $\bar{x}$ is recursively processed to create embeddings z_v for each object v in the tree and the sample-level embedding $z_{\bar{x}}$. **c** The embedding $z_{\bar{x}}$ is used to compute class probabilities ϱ, value estimate V, and the terminal action potential a_t. **b** An unobserved leaf feature is chosen with a sequence of stochastic decisions. Probabilities are determined by $f_{\varphi_\mathcal{B}}(z_{\bar{x}}, z_v)$. The whole architecture is end-to-end differentiable

5 Method

This section systematically introduces key details of our method to solve CwCF with structured data. The result is a model that is trained to solve Eq. (1). It is composed of several parts, as displayed in Fig. 4 and described by Algorithms 1 and 2. First, the input is processed with HMIL to create item-level and sample-level embeddings. Second, the sample-level embedding is used to create a class prediction, a value function estimate, and a terminal action value. Third, the action space is semantically factored with hierarchical softmax that creates a complete probability distribution over all actions. Our model is a specialized end-to-end differentiable neural network, and we denote it with Θ and its parameters with θ (this includes parameters ϑ in HMIL, φ in action selection and ϱ, V, π output heads). To keep down the overall complexity of the final model, we minimize the number of layers used in each component. For example, we define the classifier ϱ as a single neural network layer. However, this is not a limitation, since it uses the embeddings computed previously in the HMIL phase, and the whole network is updated end-to-end. When using our method, one may try to experiment with the number of layers to tune its performance for a concrete application. To declutter notation in the following text, we avoid using θ when describing gradients in $\nabla_\theta, \varrho_\theta, V_\theta$, and π_θ. For reference, important symbols are summarized in Table 1.

Table 1 Selected important symbols

Symbol	Description
λ	Accuracy versu cost trade-off factor
y_θ, k_θ	Outputs of the model—class and all acquired features
c	Cost function
x, y	Sample and class
$\mathcal{F}(x)$	All features of the sample x
$\bar{\mathcal{F}}$	Set of acquired features
$\bar{x} = o(x, \bar{\mathcal{F}})$	Observation and observation function
$\Sigma_\mathcal{D}$	Schema of a dataset $\mathcal{D}$
a_t	Terminal action
ℓ_{rl}	Classification loss for RL (binary)
ℓ_{cls}	Classifier loss (cross-entropy)
κ	A feature
$pre(\kappa)$	Prefix of the feature κ
$f_{\vartheta_\mathcal{B}}$	HMIL embedding function for the bag $\mathcal{B}$
$f_{\varphi_\mathcal{B}}$	Pre-softmax embedding function for action selection for the bag $\mathcal{B}$
$z_v, z_{\bar{x}}$	Embeddings of an object v and observation $\bar{x}$
π	Action selection policy
ϱ	Classification probabilities
V	Value function
v_{a_t}	Pre-softmax value of terminal action a_t

Algorithm 1 HMIL-CwCF training

$\mathcal{E}$ - list of parallel environments, initialized as $(x, y, \bar{\mathcal{F}} = \emptyset), (x, y) \in \mathcal{D}$
Θ - model and its parameters (neural network)

function TRAIN
  while not converged **do**
    batch $\mathfrak{B} = [\,]$
    for all $env \in \mathcal{E}$ **do** ▷ separate trace per environment
      $s = (x, y, \bar{\mathcal{F}}), \bar{x} = o(x, \bar{\mathcal{F}})$; from $env.s$ ▷ state and observation
      $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \varrho = \Theta(\bar{x})$ ▷ process with the model
        ▷ a, a_t denote the selected and terminal actions
      $r, s' = \text{STEP}(env, a, \text{argmax}\, \varrho)$ ▷ argmax ϱ needed only when $a = a_t$
      append $s, a, r, s', \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \varrho$ to batch $\mathfrak{B}$
    end for
    $L_{pg} = \text{A2C}(\mathfrak{B})$ ▷ policy gradient loss
    $L_{cls} = \mathbb{E}_{\mathfrak{B}}\left[\pi(a_t \mid \bar{x}) \cdot \ell_{cls}(\varrho(\bar{x}), y)\right]$ ▷ classifier loss (cross-entropy), eq. (8)
    update Θ with $\nabla(L_{pg} + L_{cls})$
  end while
end function

function STEP(environment env, action a, prediction $\hat{y}$)
  $(x, y, \bar{\mathcal{F}}) = env.s$
  if $a = a_t$ **then**
    $r = -\ell_{rl}(\hat{y}, y)$ ▷ RL loss (binary)
    sample new (x', y') from $\mathcal{D}$, $\bar{\mathcal{F}} = \emptyset$; $env.s = (x', y', \bar{\mathcal{F}})$ ▷ reset env
    $s' = \mathcal{T}$
  else ▷ $a \in \mathcal{A}_f \subseteq \mathcal{F}$
    $r = -\lambda c(a)$ ▷ cost of the feature
    $\bar{\mathcal{F}} = \bar{\mathcal{F}} \cup a \cup \text{GETFREEFEATURES}(a)$
    $env.s = (x, y, \bar{\mathcal{F}}), s' = env.s$
  end if
  return r, s'
end function

function A2C(batch $\mathfrak{B}$) ▷ with target clipping and sampled entropy
  $\nabla J = \mathbb{E}_{\mathfrak{B}}\left[A(\bar{x}, a) \cdot \nabla \log \pi(a \mid \bar{x})\right]$ ▷ eq. (4)
  $L_V = \mathbb{E}_{\mathfrak{B}}\left[\text{clip}(r + \gamma V'(\bar{x}'), -\infty, 1.0) - V(\bar{x})\right]^2$ ▷ eqs. (5),(11)
    ▷ V' is not updated; $V'(\bar{x}') = 0$ if $s' = \mathcal{T}$
  $\nabla L_H = \mathbb{E}_{\mathfrak{B}}\left[\log \pi(a \mid \bar{x}) \cdot \nabla \log \pi(a \mid \bar{x})\right]$ ▷ eqs. (7),(12)
  return $L_{pg} = -J + \alpha_v L_V - \alpha_h L_H$ ▷ using auto-differentiation
end function

function GETFREEFEATURES(κ_0) ▷ recursively find free features
  $\bar{\mathcal{F}} = \{\}$
  for all $\kappa \mid pre(\kappa) = \kappa_0 \wedge c(\kappa) = 0$ **do**
    $\bar{\mathcal{F}} = \bar{\mathcal{F}} \cup \kappa \cup \text{GETFREEFEATURES}(\kappa)$
  end for
  return $\bar{\mathcal{F}}$
end function

5.1 Input pre-processing

The features in an observation $\bar{x}$ can be of different data types. Before processing with a neural network, they have to be converted into real vectors (only the features holding a value, not *set* features). For strings, we observed good performance with character tri-gram histograms (Damashek, 1995). This hashing mechanism is simple, fast, and conserves similarities between strings. We used it for its simplicity and acknowledge

that any other string processing mechanism is possible. One-hot encoding is used with categorical features.

Algorithm 2 HMIL-CwCF model

function Θ(observation $\bar{x}$)
  $z_{\bar{x}} = \text{HMIL}(\bar{x})$ ▷ embed the observation
  compute $V(z_{\bar{x}})$; $\nu_{a_t}(z_{\bar{x}})$; $\varrho(z_{\bar{x}})$ ▷ separate heads in neural network
  $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}) = \text{SelectAction}(\bar{x}, z_{\bar{x}}, \nu_{a_t})$ ▷ differentiable hierarchical softmax
  return $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \varrho$
end function

function HMIL(bag $\mathcal{B}$)
  for all objects $v \in \mathcal{B}$ **do**
    for all features $\kappa \in v \mid \text{type}(\kappa) = \text{set}$ **do**
      $v_\kappa.\text{value} = \text{HMIL}(v_\kappa.\text{items})$ ▷ recursively process set features
      $v_\kappa.\text{mask} = \text{mean}_{v_i \in v_\kappa.\text{items}}(v_i.\text{mask})$ ▷ % of acquired features in sub-tree
    end for
    $v.\text{value} = \left[\forall \kappa \in v : v_\kappa.\text{value if } \kappa \in \bar{\mathcal{F}} \text{ else } 0\right]$ ▷ concat the features' values
    $v.\text{mask} = \left[\forall \kappa \in v : (1 \text{ or } v_\kappa.\text{mask}) \text{ if } \kappa \in \bar{\mathcal{F}} \text{ else } 0\right]$ ▷ use v_κ.mask if type(κ) = set
    $z_v = f_{\vartheta_\mathcal{B}}(v.\text{value}, v.\text{mask})$ ▷ embed the object v
  end for
  return $\text{mean}_{v \in \mathcal{B}}(z_v)$ ▷ average the vectors
end function

function SelectAction(bag $\mathcal{B}$, $z_{\bar{x}}$, ν_{a_t})
  for $i = 1..n$ **do**
    if i = 1 **then** ▷ at first level, append ν_{a_t} to softmax
      $\mathbb{P}(v, \kappa \text{ or } a_t \mid \bar{x}) = \text{softmax}_{a_t, v, \kappa}\left(\nu_{a_t}, f_{\varphi_\mathcal{B}}(z_{\bar{x}}, z_v) : v \in \mathcal{B}\right)^\dagger$ ▷ z_v from HMIL
      sample $a_1 = (v, \kappa)$ or a_t from $\mathbb{P}$; $\varpi_1 = \mathbb{P}(v, \kappa \text{ or } a_t \mid \bar{x})$
      store $\pi(a_t \mid \bar{x}) = \mathbb{P}(a_t \mid \bar{x})$
      if $a_1 = a_t$ **then** break
    else
      $\mathbb{P}(v, \kappa \mid \bar{x}) = \text{softmax}_{v, \kappa}\left(f_{\varphi_\mathcal{B}}(z_{\bar{x}}, z_v) : v \in \mathcal{B}\right)^\dagger$
      sample $a_i = (v, \kappa)$ from $\mathbb{P}$; $\varpi_i = \mathbb{P}(v, \kappa \mid \bar{x})$
    end if
    if type(κ) = set **then**
      $\mathcal{B} = v_\kappa.\text{items}$ ▷ continue down the tree
    else
      break ▷ v_κ is a leaf unobserved feature
    end if
  end for
  $a = [a_1, ..., a_n]$; $\pi(a \mid \bar{x}) = \prod_{i=1}^n \varpi_i$ ▷ final action and its probability
  return $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x})$

† to avoid choosing observed features, $f^{(\kappa)}_{\varphi_\mathcal{B}}(z_{\bar{x}}, z_v) = -\infty$ if $(v, \kappa) \in \bar{\mathcal{F}}$
end function

For effectivity, the pre-processing step can take place before the training for the whole dataset. When the complete dataset is unavailable and the features are directly streamed upon request (e.g., during real-world inference), the values are converted on the fly.

During inference, the feature values can be unknown. In this case, a zero vector of the appropriate size is used. To help the model differentiate between observed and unobserved features, each feature in x is augmented with a *mask*. It is a single real value, either 1 if the feature is observed or 0 if not. In sets, the mask is the fraction of the corresponding branch that is observed, computed recursively.

5.2 Input embedding

(Figure 4a, Algorithm 2 HMIL) To process and embed the input, the first part of our fully differentiable model is HMIL (see Sect. 3.3). Its structure is determined by the dataset schema $\Sigma_{\mathcal{D}}$. Each set feature corresponds to a bag and the set of all such bags is $\{\mathcal{B}_\kappa : \forall \kappa \in \Sigma_{\mathcal{D}} \mid \text{type}(\kappa) = \text{set}\}$. Before training, parameters $\vartheta_{\mathcal{B}_\kappa}$ are initialized for each bag $\mathcal{B}_\kappa$, which are later used for embedding items with the function $f_{\vartheta_{\mathcal{B}_\kappa}}$. We implement this function as one fully connected layer with LeakyReLU activation.

Let us clarify how HMIL is applied in our particular case to process an observation $\bar{x}$. The process starts with the leaves of the feature hierarchy and recursively proceeds toward the root. Each feature κ with *type=set* consists of a set of unordered objects v, collected in the bag $\mathcal{B}_\kappa$. All of these objects share the same type (enforced by the schema), i.e., they have the same features (however, not their values). The feature values of each object can be concatenated to $\mathbf{R}^n$, where n is the size of the vector for the particular set κ. This is possible because the feature values are pre-processed, unknown features are replaced with zero vectors of the appropriate size, and the value of the set features is taken from the HMIL embedding of their contents. Each object $v \in \mathcal{B}_\kappa$ is processed by the embedding function $f_{\vartheta_{\mathcal{B}_\kappa}}(v) = z_v$, and the embeddings are saved to be used later. All items in the bag are mean-aggregated, and this value is used as the feature value of the parent object. Finally, when the whole tree is processed, the result is the root-level embedding $z_{\bar{x}}$.

5.3 Classifier

(Figure 4c, Algorithm 1 lines 8, 9, 13, 20) The sample-level embedding $z_{\bar{x}}$ encodes the necessary information about the whole observation $\bar{x}$, and it is enough to compute the class probability distribution $\varrho(z_{\bar{x}})$ and the final decision $y_\theta(\bar{x}) = \text{argmax}\, \varrho(z_{\bar{x}})$. We implement ϱ as a single linear layer followed by softmax that converts the output to probabilities, and the classifier is trained parallelly to the policy π.

However, if we simply used every encountered state during training with the same weight, it would result in a biased classifier. This is because the classification is required only in terminal states and their reach probabilities need to be respected. Let $P_\pi(\bar{x})$ denote a probability that the agent reaches $\bar{x}$ and terminates under policy π. The unbiased classification loss is then:

$$L_{cls} = \mathop{\mathbb{E}}_{\bar{x} \sim P_\pi} \left[\ell_{cls}(\varrho(\bar{x}), y) \right] \tag{8}$$

To estimate the expectation in Eq. (8), we can either train the classifier only when the agent terminates, or we can use every encountered state weighted by the terminal action probability $\pi(a_t \mid \bar{x})$. We use the latter because it provides an estimate with a lower variance. For ℓ_{cls}, we use cross-entropy loss.

5.4 Value function and terminal action

(Figure 4c, Algorithm 2 line 3) The embedding $z_{\bar{x}}$ is also used to compute the value function estimate $V(z_{\bar{x}})$ (required by the A2C algorithm) and pre-softmax value of the terminal action $\nu_{a_t}(z_{\bar{x}})$. Both functions are implemented as a single linear layer without any

activation. The activation is not used in the value function, because its output should be unbounded, and it is commonly implemented in deep RL algorithms this way (Mnih et al., 2015). The output of $\nu_{a_t}(z_{\bar{x}})$ is converted to probability during the action selection.

5.5 Action selection

(Figures 4b and 5, Algorithm 2 SelectAction) Let us describe the process of selecting an action. Remember that the observation $\bar{x}$ can be viewed as a tree, where value features are leaves and set features branch further. Note that this hierarchy is semantical, i.e., each set feature groups similar objects related to their parent. Therefore, it makes sense to use this semantical hierarchy for feature selection. We call the method below *hierarchical softmax* and note that a similar technique was used in natural language processing (Morin & Bengio, 2005; Goodman, 2001).

For visualization, see Figs. 4b and 5. Oppositely to the input embedding procedure, the action selection starts at the root of $\bar{x}$ and a series of stochastic decisions are made at each node, continuing down the tree. The root node is regarded as a set with a single object. For each bag $\mathcal{B} \in \mathfrak{B}$, let the probability of selecting a feature κ of an object v be:

$$\mathbb{P}(v, \kappa \mid \bar{x}) = \underset{v,\kappa}{\operatorname{softmax}} \left(f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v) : v \in \mathcal{B} \right) \tag{9}$$

Here, $f_{\varphi_{\mathcal{B}}} : \mathbf{R}^n \to \mathbf{R}^m$ is a function that transforms the embeddings $z_{\bar{x}}$ and z_v into a vector $\mathbf{R}^m$, where $n = |z_{\bar{x}}| + |z_v|$ and m is the number of features for the object v. The bag-specific parameters $\varphi_{\mathcal{B}}$ are initialized prior training with the knowledge of the dataset schema for every possible bag $\mathcal{B} \in \mathfrak{B}$. In plain words, Eq. (9) means that all items in the bag $\mathcal{B}$ are processed with $f_{\varphi_{\mathcal{B}}}$, the outputs are concatenated are passed through the softmax function. This results in a single probability value for each feature in every object of $\mathcal{B}$, which are resolved at once.

Note that the function $f_{\varphi_{\mathcal{B}}}$ is a different function from $f_{\vartheta_{\mathcal{B}}}$. Its parameters are bag-specific, and it is implemented as a single fully connected layer with no activation function, since the output is later passed through softmax. Observed features and parts of the tree that are fully expanded (the mask of the corresponding features is 1) are excluded from the softmax. We enforce this by setting the corresponding outputs of $f_{\varphi_{\mathcal{B}}}$ to $-\infty$, so the softmax returns 0. At the root level, the terminal action potential $\nu_{a_t}(\bar{x})$ is added to the softmax.

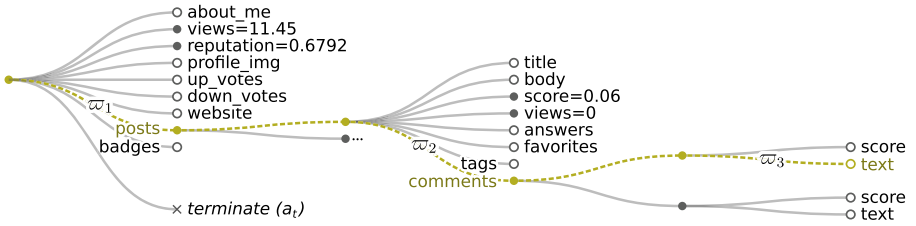


Fig. 5 Visualization of how an action is selected. Sequentially, a path is created from the root to a leaf unobserved feature (or the terminal action) by a series of stochastic decisions. In set features, all items and their features are resolved at once. The probability of the performed action is a product of the partial probabilities on the path. In this example, the chosen action a selects the *posts[0].comments[0].text* feature with probability $\pi(a \mid \bar{x}) = \prod_{i=1}^{3} \varpi_i$

Now, remember that the action selection starts at the root of $\bar{x}$, iteratively samples from $\mathbb{P}(v, \kappa \mid \bar{x})$ and proceeds down the tree, until it reaches a leaf feature (also, see Algorithm 2 SelectAction). Let us define an action $a = [a_1, ..., a_n]$ as a list of the specific choices, $a_1 = (v_1, \kappa_1)$ or $a_t, a_2 = (v_2, \kappa_2), ..., a_n = (v_n, \kappa_n)$, where n is the length of the path. We can write the probability of selecting the action a, given the observation $\bar{x}$, as a product of choice probabilities made on its path:

$$\pi(a \mid \bar{x}) = \prod_{i=1}^{n} \mathbb{P}(a_i \mid \bar{x}) \tag{10}$$

Hence, any action $a \in \mathcal{A}_f \cup a_t$ (i.e., any currently unobserved leaf feature, or the terminal action) can be sequentially sampled from Eq. (10).

The π is a probability distribution of actions, hence it is a *policy*. The decomposition according to Eq. (10) has several benefits. First, it was shown that a sensible policy decomposition introduces inductive biases to the model and speeds up the learning (Tang & Agrawal, 2020). Our decomposition is logical because the decision on each level is made for objects that are semantically related. Second, it is interpretable, because it reveals which objects and features contributed to the decision. Third, it saves computational resources as only the probabilities on the selected path need to be computed. A drawback of the hierarchical softmax is that the decisions are made sequentially for each sample, which limits the parallel computation capabilities of modern GPUs. In our implementation, most of the time is spent on simulating the environment, and hence this drawback is negligible.

5.6 Training

(Algorithm 1 Train and A2C) We use the A2C algorithm (see Sect. 3.2) to optimize the policy π with its parameters θ, with the following changes. Note that we cannot train the model with value-based methods that were used with the original CwCF (e.g., DQN (Mnih et al., 2015)), because they cannot optimize the policy itself.

First, we use the fact that the maximal Q value is 1.0 (the reward for correct prediction is 1.0 and every other step has a negative reward) and clip the target q in Eq. (6) into $(-\infty, 1.0)$:

$$q(s, a, s') = \text{clip}(r(s, a, s') + \gamma V_{\theta'}(s'), -\infty, 1.0) \tag{11}$$

This reduces a maximization bias that occurs when learning a value function with neural networks (Van Hasselt et al., 2016).

Second, the computation of the policy entropy L_H in Eq. (7) requires knowledge of all action probabilities. However, the sequential nature of the hierarchical softmax means that only the $\pi(a \mid \bar{x})$ for the actually performed action a is computed. As the computation and gathering of probabilities for all actions are troublesome and unnecessary, we propose to estimate the entropy as follows. In the A2C algorithm, only the gradient ∇L_H is needed, and basic algebra shows that the correct way to estimate it is (Zhang et al., 2018):

$$\nabla_\theta H_{\pi_\theta}(s) = - \mathop{\mathbb{E}}_{a \sim \pi_\theta(s)} \left[\log \pi_\theta(a \mid s) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right] \tag{12}$$

Here, we use only the performed action to sample the expectation with zero bias, and the variance is decreased through large batches. For completeness, the derivation of Eq. (12) is in the Supplementary Material A.

The A2C algorithm returns the loss L_{pg} at each step. Simultaneously, the classification loss L_{cls} is computed. Multiple parallel samples are processed at once to create a larger batch (see Supplementary Material C for further details). After each step, the model's parameters are updated in the direction of $-\nabla(L_{pg} + L_{cls})$. We believe that the A2C algorithm sufficiently demonstrates the method but note that any recent or future RL enhancement is likely to improve its performance.

5.7 Pretraining classifier

The RL part of the algorithm optimizes Eq. (1), which assumes a trained classifier. However, the classifier is trained simultaneously by minimizing Eq. (8). As the classifier output appears in (1) and Eq. (8) is based on the probability P_π, this introduces nonstationarity in both problems. To mitigate the issue and speed up convergence, we pretrain the classifier ϱ with random observations (pruned samples). We cannot target a specific budget, since it is unknown before the training (only a tradeoff parameter λ is specified). Hence, we cover the whole state space by generating observations $\bar{x}$ ranging from almost empty to complete. The exact details are in Supplementary Material C.

6 Experiments

In this section, we describe several experiments that show the behavior of our algorithm and other tested methods. First, we describe the tested algorithms and the experiment setup. Then, we continue with a synthetic dataset designed to demonstrate the differences in algorithms' behaviors. Next, we apply the algorithm to a real-world problem of identifying malicious web domains. Finally, we gathered five more datasets for a quantitative evaluation. The complete code for all described algorithms and all datasets is shared publicly at https://github.com/jaromiru/rcwcf. For the reproducibility of our results, we also include the scripts to run the experiments and produce the plots.

6.1 Tested algorithms

To our knowledge, there is no other method dealing specifically with costly hierarchical data. We constructed the following algorithms for comparison. Each of them represents certain class of algorithms and they can also be perceived as ablations of the main algorithm presented in this manuscript.

HMIL represents algorithms that disregard the costs and always use all available features. Alternatively, it can be seen as an ablation of the main algorithm, where we leave only the input embedding and classification parts. This method uses the complete information available, processes it directly with the HMIL algorithm and is trained in a supervised manner. This approach provides an estimate of achieveable accuracy, but also with the highest cost. In practice, using all features at once makes the algorithm prone to overfitting, which we mitigated by using aggressive weight decay regularization (Loshchilov & Hutter, 2018).

RandFeats represents a naive approach to the hierarchical composition of features, which are now selected randomly. With this, we can estimate the influence of the informed feature selection. It is an ablation of the full algorithm, implemented by replacing the policy with a random sampling. The algorithm acquires features randomly until a specified

budget is exceeded. All other parts of the algorithm are kept the same. Since this algorithm is uninformed, we expect it to underperform the complete algorithm and give a lower bound estimate for accuracy.

Flat-CwCF: In this case, we demonstrate the original CwCF algorithm, which requires a fixed number of features. We achieve this by flattening the data—only the root-level features are selectable, and the algorithm observes the complete sub-tree (embedded with HMIL) whenever such a feature is selected. This algorithm behaves the same as the full algorithm on the root level but lacks fine control over which features it requests deeper in the structure. Because of that, we expect the method to underperform the full algorithm with lower budgets, but to reach the performance of *HMIL* gradually.

One could argue that we could also engineer a fixed set of features for each dataset and apply the original CwCF or a similar algorithm. For example, the engineered features for the *threatcrowd* dataset (see Fig. 10-right for its schema) could include its domain and aggregated hashes of five random IP addresses, emails, and malware hashes. However, there can be more or fewer of these objects in the actual data sample. Given the variability of individual samples, the automatic selection of a static set of features is difficult, and the standard approaches to feature selection do not work with structured data.

In the original CwCF paper (Janisch et al., 2020), the authors proposed a heuristic baseline method that acquired features in a precomputed order sorted by their importance. For each subset, a specific classifier was trained to estimate the accuracy at this point, resulting in a point in the accuracy vs cost plane. The original CwCF method was shown to outperform this baseline, due to its ability to select per-sample specific features in a unique order. In our case, it is unclear how to apply this baseline to the hierarchical data where each sample has a different number of objects in its sets and a different number of features overall.

Finally, we refer to the full method described in this paper as **HMIL-CwCF**. We searched for the optimal set of hyperparameters for each algorithm and dataset using validation data, and the complete table with all settings is in Supplementary Material C.

6.2 Experiment setup

For each dataset, we ran *HMIL* with ten different seeds, *RandFeats* with 30 different budgets linearly covering either [0, 10], [0, 20] or [0, 40] range (depending on the dataset) and *Flat-CwCF* and *HMIL-CwCF* with 30 different values of λ, logarithmically spaced in $[10^{-4}, 1.0]$ range. For each run, we selected the best epoch based on the validation data (for more details, see convergence graphs in Supplementary Material F).

To visualize the results, we select the best runs that are on the Pareto front of the validation dataset, using the cost and accuracy criteria. We plot the best runs as a scatter plot with the average cost on the x-axis and accuracy on the y-axis and also visualize *their* Pareto front with the testing set. To estimate variance, all other runs are visualized with faint color. For better comparison, we show the mean performance ($\pm$ one standard deviation) of *HMIL* across the whole x-axis.

Apart from the graph form, the results are also reported as normalized Area Under the Trade-off Curve (AUTC). The AUTC metric describes the overall performance across the whole range of budgets. It is computed as the area under the visualized Pareto front, normalized by the total area of the graph, and the area below the prior of the most populous class is subtracted. The AUTC would return 0 for an algorithm that always predicts the

most populous class and 1 for an algorithm with perfect classification. See Supplementary Material D for more details.

6.3 Experiment A: synthetic dataset

This experiment is aimed to demonstrate the behavior of our and other tested algorithms on purposefully crafted data. Note that this synthetic dataset is *designed* to demonstrate the differences between the algorithms and therefore our method (*HMIL-CwCF*) performs the best.

Let us first explain the dataset's structure (follow its schema in Fig. 6). A sample contains two sets (*set_a* and *set_b*), each with ten items. Each item has two features—free feature *item_key* with a value *0* and *item_value* containing a random label. Randomly, a single item in one of the sets is chosen, and its *item_key* is changed to *1* and its *item_value* to the correct sample label. Further, the feature *which_set* contains the information about which set contains the indicative item. The idea is that the algorithm can learn a correct label by retrieving the *which_set* feature, opening the correct set, and retrieving the value for the item with *item_key=1*. Uniquely for this dataset, we test the algorithms directly on the training data.

Figure 7-right shows the performance of the tested algorithms in this dataset and Table 3 shows the AUTC metric. *HMIL* (the ablation with complete data) reaches 100% accuracy with a total cost of 31 (cost of all features). The *Flat-HMIL* is able to reduce the cost by acquiring only the correct set, but it has to retrieve all of its objects. Hence, it also reaches 100% accuracy, but with a cost of 16 (1 for *which_set* feature, 5 for one of the sets, and 10 for all values inside). Contrarily, the complete *HMIL-CwCF* method reaches 100% accuracy with only the cost of 7, since it can retrieve only the single indicative value from the correct set. Moreover, it is able to reduce the cost even further by sacrificing accuracy, as seen in the clustering around the cost of 6 and 0.75 accuracy, something that *Flat-HMIL* cannot do. This is one of the strengths of the proposed method—because it has greater control over which features it acquires, the user can *choose* to sacrifice the accuracy for a lower cost. Lastly, the *RandFeats* method selects the features randomly, and hence, its accuracy is well below *HMIL-CwCF* for corresponding budgets. The accuracy is influenced by the probability of getting the indicative item, which raises with the allocated budget and would reach 100% with the cost of 31 (we run the method with budgets from [0, 20]).

We selected one of the *HMIL-CwCF* models that was trained to reach 100% accuracy and examined how it behaves (see Fig. 7-left). We see that it indeed learned to acquire

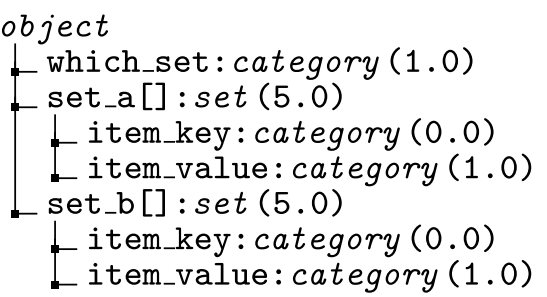


Fig. 6 The schema of the *synthetic* dataset. The numbers in parentheses denote the costs of the corresponding features

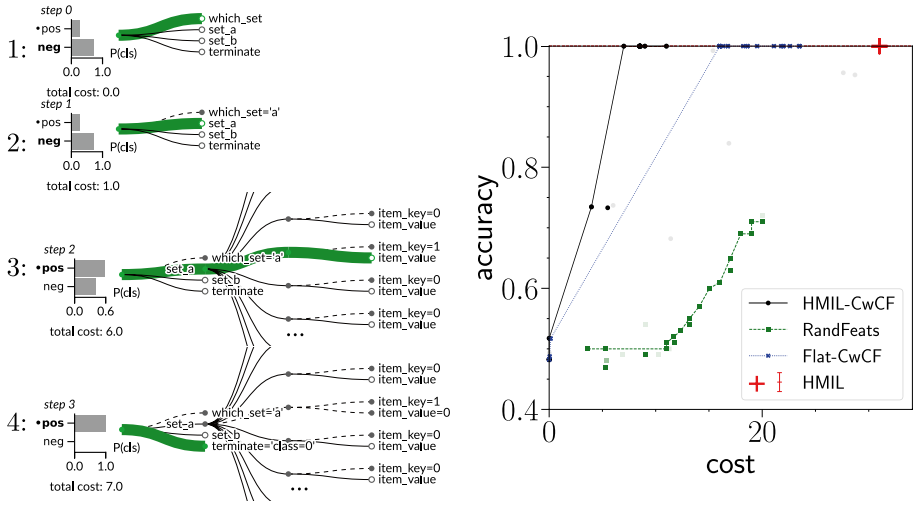


Fig. 7 Results in the *synthetic* dataset. **(left)** The process of feature selection. In this example, the algorithm optimally requests the *which_set* feature, opens *set_a*, and learns the label in the indicative item. **(right)** Performance of all algorithms across different budget settings (*x*-axis). We show our method (*HMIL-CwCF*), its ablation with a random policy (*RandFeats*), ablation with flattened data (*Flat-CwCF*), and the *HMIL* algorithm trained with complete information. We train 30 instances per each algorithm (*HMIL-CwCF*, *RandFeats*, and *Flat-CwCF*), each targeting a different budget. We plot the best runs and their Pareto front. We also show the results of all runs as faint points for information about variance. Uniquely for this dataset, the train, validation and test sets are the same

which_set feature, open the corresponding *set_a* or *set_b* and select the *item_value* of the item with *item_key=1* to learn the right label.

This experiment validates the correct behavior of our method and demonstrates the need for all its parts. Compared to *HMIL* and *Flat-CwCF*, the complete method reaches comparable accuracy with lower cost. Moreover, compared to *Flat-CwCF*, it has better control over which features it requests, achieving better accuracy even in the low-cost region. Finally, the order in which the features are acquired matters, as shown in comparison with *RandFeats*.

6.4 Experiment B: threatcrowd

Let us focus on one of the real-world cases that motivated this work. Threatcrowd is a service providing rich security-oriented information about domains, such as known malware binaries communicating with the domain (identified by their hashes), WHOIS information, DNS resolutions, subdomains, associated email addresses, and, in some cases, a flag that the domain is known to be malicious (see an example of its interface in Fig. 8). This information is stored in a graph structure, but only a part around the current query is visible to the user. However, the user can easily request more information about the connected objects. For example, after probing the main domain `google.com`, the user can focus on one of its multiple IP addresses to analyze its reverse DNS lookups, or which other domains are involved with a particular malware. To make the queries, Threatcrowd provides an API with a limited number of requests per unit of time, which makes it a scarce

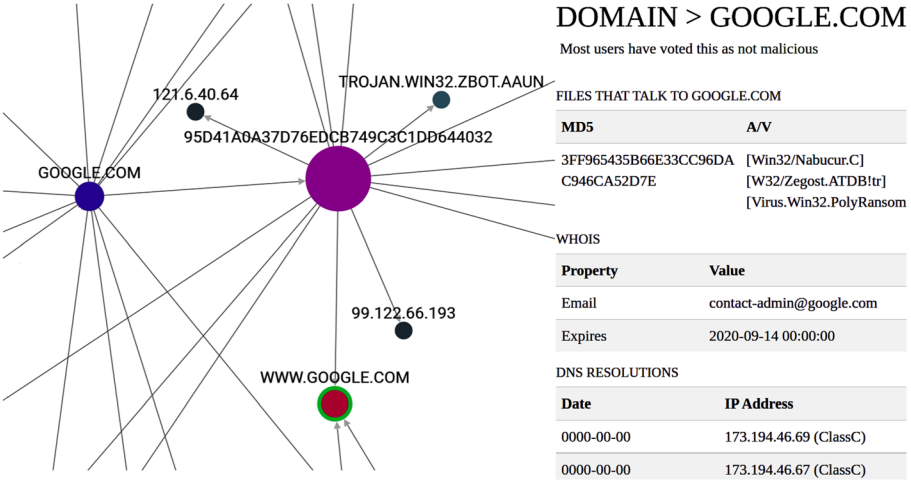


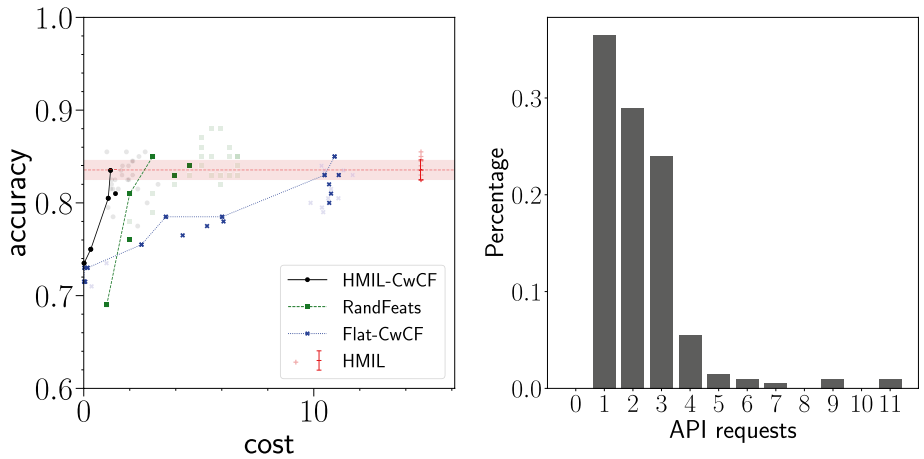
Fig. 8 Threatcrowd interface. The left side shows a part of the information graph, unfolded to a limited depth. Various information is available for each node, and the right side displays the information about the currently focused node

resource. We are interested in the following task: *Classify a specified domain using the information provided through the API, minimizing the number of requests.*

To make the experimentation easier and reproducible, we sourced an offline dataset directly from the Threatcrowd service through their API, with their permission. Programmatically, we gathered information about 1171 domains within a depth of three API requests (including one request for the domain itself) around the original domain and split them into training, validation, and test sets. We chose three API requests because we assume that most of the indicative information is located in the close neighborhood of the root object. Each domain contains its URL as a free feature and a list of associated IP addresses, emails, and malware hashes. These objects can be further reverse-looked up for other domains. This offline dataset perfectly simulates real-life communication with the original service but in a swift and error-free manner. The dataset's schema can be viewed in Fig. 10-right.

We ran all of the algorithms with the sourced data, and the results of the experiment are shown in Fig. 9a and Table 3. The *HMIL* reaches the mean accuracy of 0.83 with a cost of 15 (on average, one needs to make 15 requests to gather all information within the depth of three). Other algorithms reach the same accuracy with a lower cost—*Flat-CwCF* with 11, *RandFeats* with 5, and *HMIL-CwCF* with only 2 (results are rounded). That means that **our method needs only two API requests on average** to reach the same accuracy as *HMIL* (which uses complete information), resulting in 7.5× savings. To better understand what these two requests on average mean, we analyzed a single trained model and plotted a histogram of API requests across the whole test set in Fig. 9b. For example, with a single request, the algorithm can learn a list of all IP addresses (without further details) or a list of associated malware hashes. The histogram shows that in about 36% of samples, a single request is enough for classification, 29% requires two, 23% three, and 12% four requests or more.

Surprisingly, *RandFeats* performs better than *Flat-CwCF*, indicating that only a fraction of information is required, even if randomly sampled. The *Flat-CwCF* algorithm always



(a) results (b) histogram of used API requests

Fig. 9 **a** Results in *threatcrowd* dataset. The shaded area shows ± one standard deviation around the mean performance of *HMIL* (10 runs), across the whole *x*-axis for comparison. **b** Histogram of used API requests for a trained model that uses two requests on average

acquires a complete sub-tree for a specific feature (e.g., a complete list of IP addresses with their reverse lookups, up to the defined depth), resulting in unnecessarily high cost.

To get better insight into our algorithm's behavior and to showcase its explainability, we visualize how a trained model works with a single sample in Fig. 10-left. Initially, only the domain name itself is known, without any additional details and the classification would be *malware* if the model decided to terminate at this point. However, the terminal action probability is low, and the model requests a list of malware hashes (there are not any) and a list of IP addresses instead (steps 0 and 1). The prediction changes to *benign*, likely because no malware communicates with the domain nor any malicious IP address is in the list. Still, the model performs a reverse DNS lookup for two IP addresses, which does not change the prediction (steps 2 and 3). Finally, the algorithm finishes with a correct classification *benign*. With four requests, the method was able to probe and classify an unknown domain.

To conclude, this experiment shows that the complete method leads to substantial savings while achieving the same accuracy. When deployed to production, this could mean that the method can classify much more samples with the same budget, or that the budget can be lowered, leading to monetary savings. To apply the model in a real-life scenario, the only thing required is an interface connecting the model's input and decisions with the Threatcrowd API. After that, the model would be able to perform the classification online. The experiment also verifies that all parts of the algorithm are required. Specifically, the comparison with the *Flat-CwCF* and *RandFeats* baselines showed that flattening the features results in degraded performance and that selecting features based on the knowledge gathered so far is crucial.

6.5 Experiment C: other datasets

To further evaluate our method, how it scales with small and large datasets and how it performs in binary and multi-class settings, we sourced five more datasets from various domains. Because our method targets a novel problem, we did not find datasets

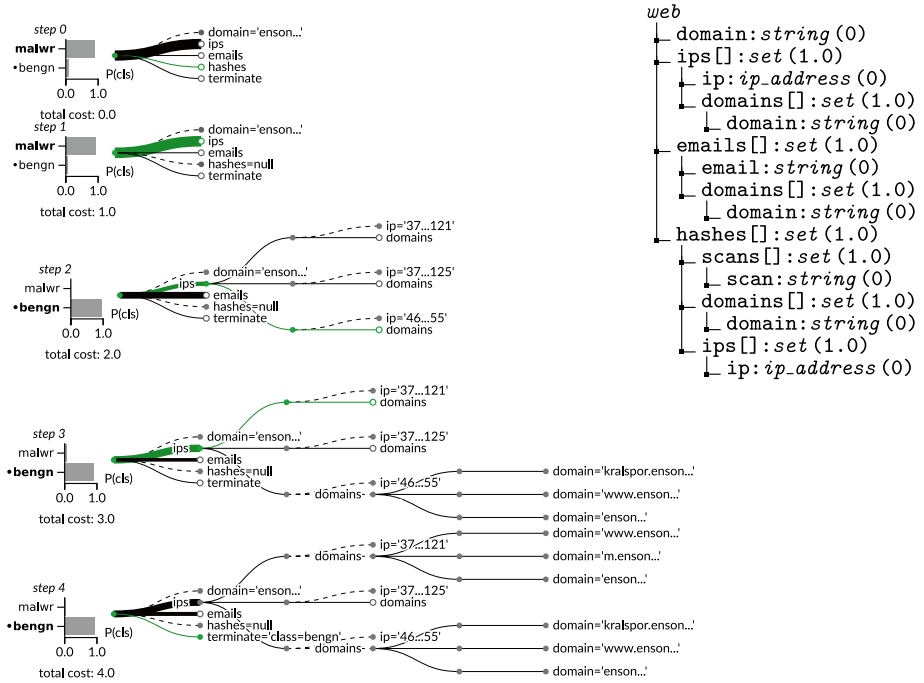


Fig. 10 **(left)** Classification of a potentially malicious domain (*threatcrowd* dataset). At each step, acquired features (full circles) and possible actions (empty circles; unobserved features and terminal action) are shown. The policy is visualized as line thickness and the selection with a green line. The method sequentially requests features: First, it retrieves (step 0) a list of known malware hashes communicating with the domain, then (step 1) a list of associated IP addresses, and finally (steps 2 and 3) performs reverse IP lookups. The correct class is highlighted with a dot. Note that the number of actions differs at each step and the size of sets (IPs, hashes, and emails) differs between samples. **(right)** The dataset's schema in *feature:type(cost)* format. In this dataset, the costs represent API requests

in appropriate format—i.e., datasets with hierarchical structure and cost information. Therefore, we transformed existing public relational datasets into hierarchical forms by fixing the root object (different for each sample) and expanding its neighborhood into a defined depth. We also manually added costs to the features in a non-uniform way, respecting that in reality, some features are more costly than others (e.g., getting a patient's age is easier than doing a blood test). In practice, the costs would be assigned to the real value of the required resources. The depth of the datasets was chosen so that they completely fit into the memory.

6.5.1 Dataset descriptions

We provide brief descriptions of the used datasets below. The statistics are summarized in Table 2. For reproducibility, we published the processed versions, along with a library to load them. More details on how we obtained and processed the datasets, their splits, structure, and feature costs are in Supplementary Material B.

Table 2 Statistics of the used datasets. The *features* column shows the number of features (tree leaves) across all completely observed samples in the corresponding dataset

Dataset	Samples (all splits)	Class distribution	Features (min/mean/max)	Depth
Synthetic	12	0.5/0.5	43/43.0/43	2
Threatcrowd	1171	0.27/0.73	4/701.7/3706	3
Hepatitis	500	0.41/0.59	7/121.7/1065	2
Mutagenesis	188	0.34/0.66	173/332.2/517	3
Ingredients	39774	0.01–0.20	2/11.8/66	2
SAP	35602	0.5/0.5	16/31.8/52	2
Stats	8318	0.49/0.38/0.13	9/52.5/21979	3

Hepatitis: A relatively small medical dataset containing patients infected with hepatitis, types B or C. Each patient has various features (e.g., sex, age, etc.) and three sets of indications. The task is to determine the type of disease.

Mutagenesis: Extremely small dataset (188 samples) consisting of molecules that were tested on a particular bacteria for mutagenicity. The molecules themselves have several features and consist of atoms with features and bonds.

Ingredients: Large dataset containing recipes with a single list of ingredients. The task is to determine the type of cuisine of the recipe. The main challenge is to decide when to stop analyzing the ingredients optimally.

SAP: In this large artificial dataset, the task is to determine whether a particular customer will buy a new product based on a list of past sales. A customer is defined by various features and a list of sales.

Stats: An anonymized content dump from a real website Stats StackExchange. We extracted a list of users to become samples and set an artificial goal of predicting their age category. Each user has several features, a list of posts, and a list of achievements. The posts also contain their own features and a list of tags and comments.

6.5.2 Results

The results are shown in Fig. 11 and in Table 3. Let us select interesting facts and describe them below. The *HMIL* algorithm shows what accuracy is possible to achieve when using all features at once. The variance of its results indicates what should be considered normal in the corresponding dataset. Especially in *hepatitis* and *mutagenesis* (Fig. 11ae), the variance of the results is high, which is given by the datasets' small sizes.

The results in *sap* (Fig. 11c) are noteworthy. Here, the top accuracy of *HMIL* is exceeded by *HMIL-CwCF* and *Flat-CwCF*. We investigated what is happening and concluded that *HMIL* overfits the training data, despite aggressive regularization—we tuned the weight decay to maximize the validation accuracy. Surprisingly, *HMIL-CwCF* and *Flat-CwCF* do not suffer from this issue, with fewer features. We hypothesize that the *sap* dataset contains some features deep in the hierarchy that are very informative on the training set, but do not translate well to the test set. The well-performing methods are able to circumvent the issue by selecting fewer features, which results in less overfitting.

Generally, the *HMIL-CwCF* is among the best-performing algorithms in all datasets, i.e., it reaches the same accuracy with lower cost (in *sap* and *mutagenesis*, it performs

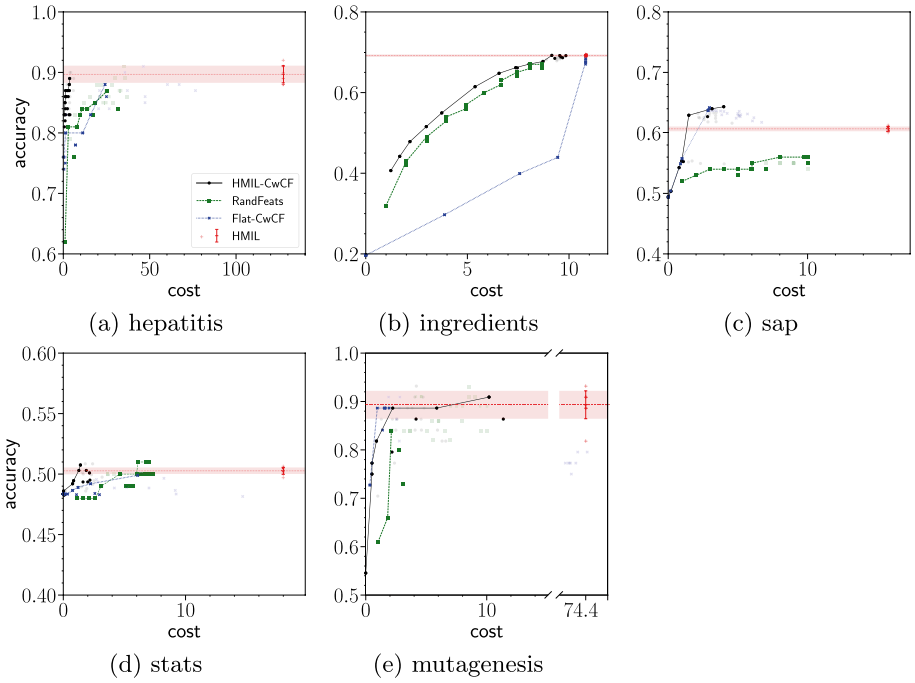


Fig. 11 The performance of the algorithms in five datasets, shown in the cost versus accuracy plane. We show our method (*HMIL-CwCF*), its ablation with a random policy (*RandFeats*), ablation with flattened data (*Flat-CwCF*) and the *HMIL* algorithm trained with complete information. We train 30 instances per each algorithm (*HMIL-CwCF*, *RandFeats* and *Flat-CwCF*), each targeting a different budget. We plot the best runs, selected using validation sets and their Pareto front. For information about variance, we also show the results of all runs as faint points. The *HMIL* is run 10 times, and we plot the mean ± one standard deviation (the bar visualizes the metrics across the whole range of budgets for comparison)

Table 3 Normalized area under the trade-off curve (AUTC; see Sect. 6.2 for description)

Dataset	HMIL-CwCF	Flat-CwCF	RandFeats	HMIL
Synthetic	**0.88**	0.75	0.32	0.50
Hepatitis	**0.74**	0.70	0.69	0.38
Mutagenesis	**0.71**	0.68	0.60	0.36
Ingredients	**0.47**	0.19	0.44	0.31
SAP	**0.24**	0.23	0.11	0.11
Stats	**0.03**	0.02	**0.03**	0.02
Threatcrowd	**0.36**	0.25	**0.36**	0.18

The highest values in the corresponding rows are given in bold

comparatively to *Flat-CwCF*). Compared to *HMIL*, the cost is reduced about 26× in *hepatitis*, 1.2× in *ingredients*, 8× in *sap*, 6× in *stats* and 15× in *mutagenesis*, which are significant savings. *Flat-CwCF* generally exhibits low performance in the low-cost region, due to its limited control over which features it gathers.

Lastly, let us point out the result of *HMIL-CwCF* compared to *RandFeats* in *ingredients* (Fig. 11b). This dataset contains a single set of ingredients, which are objects with a single feature. The best any algorithm can do is to randomly sample the ingredients and stop optimally. While *RandFeats* always uses the given budget, *HMIL-CwCF* can acquire more features in some cases and compensate for that with other samples. Hence, it can reach higher accuracy with the same *average* cost as *RandFeats*.

The *Flat-CwCF* algorithm can either acquire the whole set of ingredients, or nothing. It achieves different points in Fig. 11b by randomization, i.e., it discloses the list of ingredients for some samples, or not for others. Note that the number of ingredients in each recipe varies and ranges from 1 to 65. One could argue that we could use a different encoding of the ingredients—e.g., one-hot encoding of the ingredients that are in a recipe. However, there are 6707 unique ingredients, while the mean number of ingredients in a recipe is around 11. Flattening the data this way would result in a very sparse and long binary feature vector. Applying the original CwCF method with such data would not work very well, since most of the features would encode a *missing ingredient*. This was already exemplified in Janisch et al. (2020), where training in a dataset with categorical values encoded to multiple one-hot encoded features (with a length of 40, compared to the required 6707 in case of *ingredient*) took an order of magnitude longer time to train, compared to similarly-sized dataset without such features.

To conclude, the results in Fig. 11 show that our method consistently performs better or comparatively to other methods—i.e., achieves a similar accuracy with much fewer features. The AUTC metric in Table 3 aggregates the performance for the whole range of costs and confirms the conclusion.

6.6 Remarks

6.6.1 Explainability

Unlike the standard classification algorithms (e.g., *HMIL*), the sequential nature of *HMIL-CwCF* enables easier analysis of its behavior. Figures 10 and 7 present two examples of the feature acquisition process and give insight into the agent's decisions. The weights the model assigns to different features in different samples and steps can be used to assess the agent's rationality or learn more about the dataset. We present more visualizations in Supplementary Material E.

6.6.2 Classifier pretraining

The positive role of pretraining was already established in the original CwCF paper (Janisch et al., 2019). However, as we separate the classifier from the RL algorithm, it is worth to assess how the situation changes. We performed an ablation experiment with the *sap* dataset and a fixed λ, where we ran the experiment 10 times with and without pretraining. The results in Fig. 12 show that the pretraining improves the speed of convergence and the performance on validation data.

6.6.3 Computational requirements

We measured the training times using a single core of Intel Xeon Gold 6146 3.2 GHz and 4 GB of memory. We used only CPU because the most time-consuming part of the

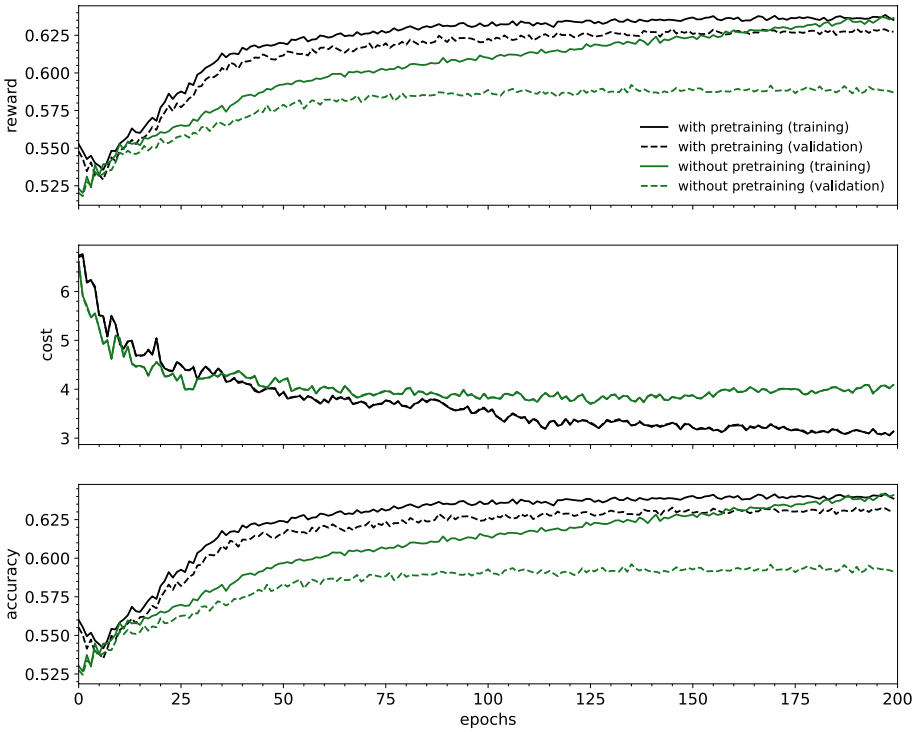


Fig. 12 Training of a model, with and without the classifier pretraining. Performed on the *sap* dataset with $\lambda = 0.00108264$; an average of 10 runs

Table 4 Training times for a single instance (i.e., single setting of λ in *HMIL-CwCF*). Note that most of the time is spent on simulating the environment

Dataset	HMIL-CwCF	RandFeats	Flat-CwCF	HMIL
Synthetic	1 h	30 min	1 h	1 min
Other (average)	19 h	14 h	9 h	1 h

training was the environment's simulation and it cannot benefit from the use of GPU. The measured times are displayed in Table 4. We show the *synthetic* dataset separately because it was much faster to learn. Note that the training times are for a single run (i.e., a single point in Fig. 11), but the runs are independent and are easily parallelized. After training, the inference time is negligible for all methods.

Note that while the training time of *HMIL-CwCF* is much longer than in the case of *HMIL*, it is easily compensated by the fact that our method can save a large amount of resources if correctly deployed. Moreover, computational power rises exponentially

every year (resulting in faster training), while resources like CO_2 production, patients' discomfort, or response time of an antivirus software only gain importance.

7 Discussion

Comparison with graph neural networks (GNNs)

Instead of HMIL, we could use a GNN to perform the input embedding. However, note that the data we work with are hierarchical and constructed around a central root. Hence it makes sense to model the data as *trees*, not as general graphs, and use a method tailored to work with trees. In our case, generic message passing is unnecessary, and a single pass from leaves to the tree's root is sufficient to embed all information correctly. Mandlík (2020) provides a deeper discussion about using HMIL and GNNs in sample-centric applications.

In some special cases, the same object could be located in multiple places (e.g., the same IP address accessible by multiple paths). In our method, we still handle the sample as a tree. If such a situation occurs, the data have to be *unrolled*, i.e., different places of the same object are considered to be different objects.

Is the depth of the tested datasets sufficient?

We argue that most of the relevant information is within the near neighborhood of the central object of interest. Increasing the depth exponentially increases the available feature space and space requirements and slows down training. As the experiments showed that there are substantial differences between the methods, we conclude that the used depth is sufficient.

How to obtain credible cost assignment?

In a real-life application, it should be possible to measure the costs of features up front. For example, the time required to perform an experiment, electricity consumed to retrieve a piece of data, or, as in the Threatcrowd experiment, every feature can represent a single API request.

Advantages and disadvantages of the proposed method

Our solution provides the following advantages, some of which are inherited from the original CwCF framework:

- It directly optimizes the objective in Eq. (1) and although the deep RL has not the same theoretical guarantees as tabular RL, it searches for the optimal solution. In contrast, some related work used heuristics (e.g., proxy rewards (Kachuee et al., 2019) in the flat CwCF case)—such algorithms are not guaranteed to aim for the optimal solution.
- The used HMIL algorithm used to process the hierarchical input is theoretically sound—Pevný and Kovařík (2019) generalizes the universal approximation theorem (Hornik, 1991; Leshno et al., 1993) to HMIL networks.
- As our method is based on a standard deep RL technique, its performance is likely to be improved with advancements in the RL field itself, since it is an actively developed area.
- The novel method can directly utilize many of the extensions developed for CwCF. This includes (1) problems with hard budget, (2) specifying the budget directly and automatic search for an optimal λ, (3) missing features (e.g., features of some objects may be inaccessible, possibly because the training data is incomplete), and (4) using an

external high-performance classifier as one of the features. Points (1–3) are discussed in Janisch et al. (2020), (4) is explored in Janisch et al. (2019).
- The original CwCF paper (Janisch et al., 2020) has already established the competitive performance of the method in the flat data case. Therefore, we believe that the novel algorithm serves as a highly competitive baseline as well.

Below, we state the drawbacks of our algorithm we are aware of:

- Being RL-based, the algorithm is sample inefficient, i.e., it requires a long training. As mentioned, training in the more complicated datasets took about 19 h on average.
- Data must be hierarchical, e.g., it must not contain references to the same object in different places in the hierarchy, nor cycles. As mentioned in the discussion about GNNs, if such structures appear in the data, it must be *unrolled* (e.g., the same object would have to be copied to different places) so that the result is hierarchical.
- With some datasets, there could be non-negligible variance in the performance of trained models. The user is advised to repeat training several times and select the best-performing model, based on validation data.

Alternative approaches

Generally, there are two ways to make the existing algorithms work with the hierarchical data: (a) modifying the data, (b) modifying the algorithm. Below, we suggest several different approaches to these options. Keep in mind that each of these suggestions would require substantial research to implement, and might not be possible at all.

(a) Modifying the data can be done in the way we did in the case of *Flat-CwCF*, but there could be other ways, for example:

- It may be possible to decrease the granularity of choice to the set level by considering each path in the *schema* as a separate feature. While this approach would result in a fixed number of features for all samples, it brings several issues. For example, since sets can contain multiple objects, it is unclear how to choose one of them. An algorithm selecting the objects randomly would have inherently lesser control over which objects to select, and would not be able to utilize possible conditional dependencies between objects' features. In the *RandFeats* baseline, we have already shown that such loss of control results in degraded performance. Second, if it is allowed to get the same feature multiple times (to cover different objects in a set), it is unclear how to aggregate and process these multiple values.
- Another way could be to treat all features in the tree as a set of tuples *(path, type, value)*, each encoded into a $\mathbf{R}^n$ space, and use algorithms designed to process sets (Shim et al., 2018). While this approach would preserve all information, it is unclear how to efficiently encode paths of various lengths that can branch in sets, or values of different types.
- Also, one could manually engineer features based on the known data structure. However, this step is laborious, suboptimal, and may be difficult to apply, because the individual samples vary in size of their sets. Note that the standard approaches to feature selection do not work with hierarchical data.

(b) Let us also discuss the possible modification of the existing algorithms, where the problem is twofold. First, the algorithm needs to be modified to accept hierarchical data with varying size. In some cases, it could be solved by embedding the data sample into a smaller, fixed space, e.g., with the HMIL algorithm, as we did in our case. However, many algorithms for the CwCF problem depend on access to the actual feature values, such as decision trees (Maliah & Shani, 2018), random forests (Nan et al., 2015, 2016; Nan & Saligrama, 2017) or cascade classifiers (Xu et al., 2014) and may not work with such transformations. Second, the modified algorithm needs to be able to select features within the hierarchy. This could be done through direct selection of the corresponding output (as we do in our method, or as the (Shim et al., 2018) would do with the formerly proposed modification), or through some other way of identifying the specific feature (possibly by returning its encoded path).

Again, while believe that many of these problems are solvable, they would require non-trivial further research.

8 Conclusion

We presented an augmented Classification with Costly Features framework that can process hierarchically structured data. Contrarily to existing algorithms, our method can process this kind of data in its natural form and select features directly in the hierarchy. In several experiments, we demonstrated that our method substantially outperforms an algorithm that uses complete information, in terms of the cost of used features. We also showed how the original CwCF would work if the data was flattened so the method could process it. As our augmented HMIL-CwCF model has the ability to choose features with greater precision, it leads to superior performance. In a separate experiment, we applied our method to a real-life problem of classification of malicious web domains, where it also outperformed the other algorithms. The sequential nature of our algorithm and its hierarchical action selection contribute to its explainability, as the features are semantically grouped, and the user can view which of them are considered important at different time steps.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10994-024-06565-4.

Acknowledgements The GPU used for this research was donated by the NVIDIA Corporation. Some computational resources were supplied by the Project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Author contributions JJ designed and implemented the method, performed the experiments and wrote the manuscript. TP and VL supervised and consulted the work.

Funding Open access publishing supported by the National Technical Library in Prague. The authors acknowledge the support of the OP VVV funded Project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics". This research was supported by The Czech Science Foundation (Grant Nos. 22-32620S and 22-26655S).

Data availability All prepared datasets are published at https://github.com/jaromiru/rcwcf.

Code availability The complete code for the presented algorithm and baselines is published at https://github.com/jaromiru/rcwcf.

Declarations

Conflict of interest The authors have no conflict of interest to declare that are relevant to the content of this article.

Consent to participate Not applicable.

Consent for publication Not applicable.

Ethical approval Not applicable.

References

Ali, B., Moriyama, K., Kalintha, W., Numao, M., & Fukui, K.-I. (2020). Reinforcement learning based metric filtering for evolutionary distance metric learning. *Intelligent Data Analysis, 24*(6), 1345–1364.

Badr, Y. (2022). Enabling intrusion detection systems with dueling double deep Q-learning. *Digital Transformation and Society* (ahead-of-print).

Banerjee, S., Pratiher, S., Chattoraj, S., Gupta, R., Patra, P., Saikia, B., Thakur, S., Mondal, S., & Mukherjee, A. (2020) Deep reinforcement learning for variability prediction in latent heat flux from low-cost meteorological parameters. In *Optics and photonics for advanced dimensional metrology, 11352* (pp. 305–311). SPIE.

Bayer-Zubek, V., & Dietterich, T. G. (2005). Integrating learning from examples into the search for diagnostic policies. *Journal of Artificial Intelligence Research, 24*, 263–303.

Bolón-Canedo, V., Porto-Díaz, I., Sánchez-Maroño, N., & Alonso-Betanzos, A. (2014). A framework for cost-based feature selection. *Pattern Recognition, 47*(7), 2481–2489.

Cesa-Bianchi, N., Shalev-Shwartz, S., & Shamir, O. (2011). Efficient learning with partially observed attributes. *Journal of Machine Learning Research, 12*(Oct), 2857–2878.

Chen, Y.-E., Tang, K.-F., Peng, Y.-S., & Chang, E. Y. (2019). Effective medical test suggestions using deep reinforcement learning. arXiv preprint arXiv:1905.12916.

Contardo, G., Denoyer, L., & Artieres, T. (2016). Recurrent neural networks for adaptive feature acquisition. In *International conference on neural information processing* (pp. 591–599). Springer.

Damashek, M. (1995). Gauging similarity with n-grams: Language-independent categorization of text. *Science, 267*(5199), 843–848.

Deng, K., Bourke, C., Scott, S., Sunderman, J., & Zheng, Y. (2007). Bandit-based algorithms for budgeted learning. In *Seventh IEEE international conference on data mining* (ICDM 2007) (pp. 463–468). IEEE.

Dulac-Arnold, G., Denoyer, L., Preux, P., & Gallinari, P. (2012). Sequential approaches for learning datum-wise sparse representations. *Machine Learning, 89*(1–2), 87–122.

Erion, G., Janizek, J. D., Hudelson, C., Utarnachitt, R. B., McCoy, A. M., Sayre, M. R., White, N. J., & Lee, S.-I. (2022). A cost-aware framework for the development of AI models for healthcare applications. *Nature Biomedical Engineering*, 6, 1384–1398.

Goldstein, O., Kachuee, M., Karkkainen, K., & Sarrafzadeh, M. (2020). Target-focused feature selection using uncertainty measurements in healthcare data. *ACM Transactions on Computing for Healthcare, 1*(3), 1–17.

Goodman, J. (2001) Classes for fast maximum entropy training. In *2001 IEEE international conference on acoustics, speech, and signal processing*. Proceedings (Cat. No. 01CH37221) (Vol. 1, pp. 561–564). IEEE.

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

Hamilton, W., Ying, Z., & Leskovec, J. (2017) Inductive representation learning on large graphs. In *Advances in neural information processing systems* (pp. 1024–1034).
Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks, 4*(2), 251–257.
Janisch, J., Pevný, T., & Lisý, V. (2019). Classification with costly features using deep reinforcement learning. In *Proceedings of 33rd AAAI conference on artificial intelligence*.
Janisch, J., Pevný, T., & Lisý, V. (2020). Classification with costly features as a sequential decision-making problem. *Machine Learning, 109*(8), 1587–1615.
Ji, S., & Carin, L. (2007). Cost-sensitive feature acquisition and classification. *Pattern Recognition, 40*(5), 1474–1485.
Kachuee, M., Goldstein, O., Karkkainen, K., Darabi, S., & Sarrafzadeh, M. (2019). Opportunistic learning: Budgeted cost-sensitive learning from data streams. In *International conference on learning representations*.
Kapoor, A., & Greiner, R. (2005). Learning and classifying under hard budgets. In *European conference on machine learning* (pp. 170–181). Springer.
Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.
Kusner, M., Chen, W., Zhou, Q., Xu, Z., Weinberger, K., & Chen, Y. (2014). Feature-cost sensitive learning with submodular trees of classifiers. In *AAAI conference on artificial intelligence* (pp. 1939–1945).
Lee, M. H., Siewiorek, D. P., Smailagic, A., Bernardino, A., & Bermúdez i Badia, S. (2020a). Interactive hybrid approach to combine machine and human intelligence for personalized rehabilitation assessment. In *Proceedings of the ACM conference on health, inference, and learning* (pp. 160–169).
Lee, M. H., Siewiorek, D. P., Smailagic, A., Bernardino, A., & Bermúdez i Badia, S. (2020b). Co-design and evaluation of an intelligent decision support system for stroke rehabilitation assessment. *Proceedings of the ACM on Human-Computer Interaction, 4*(CSCW2), 1–27.
Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks, 6*(6), 861–867.
Li, Y., & Oliva, J. (2021). Active feature acquisition with generative surrogate models. In *International conference on machine learning* (pp. 6450–6459). PMLR.
Liu, X., Kumar, B., Yang, C., Tang, Q., & You, J. (2018). Dependency-aware attention control for unconstrained face recognition with image sets. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 548–565).
Liyanage, Y. W., Zois, D.-S., & Chelmis, C. (2021). Dynamic instance-wise joint feature selection and classification. *IEEE Transactions on Artificial Intelligence*.
Loshchilov, I., & Hutter, F. (2018). Decoupled weight decay regularization. In *International conference on learning representations*.
Maldonado, S., Pérez, J., & Bravo, C. (2017). Cost-based feature selection for support vector machines: An application in credit scoring. *European Journal of Operational Research, 261*(2), 656–665.
Maliah, S., & Shani, G. (2018). Mdp-based cost sensitive classification using decision trees. In *AAAI conference on artificial intelligence* (pp. 3746–3753).
Mandlík, Š. (2020) Mapping the internet—Modelling entity interactions in complex heterogeneous networks. Master's thesis, Czech Technical University in Prague.
Mandlík, Š, Račinský, M., Lisý, V., & Pevný, T. (2022). JsonGrinder.jl: Automated differentiable neural architecture for embedding arbitrary JSON data. *Journal of Machine Learning Research, 23*(298), 1–5.
Metz, L., Ibarz, J., Jaitly, N., & Davidson, J. (2017) Discrete sequential prediction of continuous actions for deep rl. arXiv preprint arXiv:1705.05035.
Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).
Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.
Morin, F., & Bengio, Y. (2005). Hierarchical probabilistic neural network language model. *Aistats, 5*, 246–252.
Munos, R., Stepleton, T., Harutyunyan, A., & Bellemare, M. (2016). Safe and efficient off-policy reinforcement learning. In *Advances in neural information processing systems* (pp. 1054–1062).
Nan, F., & Saligrama, V. (2017). Adaptive classification for prediction under a budget. In *Advances in neural information processing systems* (pp. 4730–4740).
Nan, F., Wang, J., & Saligrama, V. (2015). Feature-budgeted random forest. In *International conference on machine learning* (pp. 1983–1991).
Nan, F., Wang, J., & Saligrama, V. (2016). Pruning random forests for prediction on a budget. In *Advances in neural information processing systems* (pp. 2334–2342).

Peng, Y.-S., Tang, K.-F., Lin, H.-T., & Chang, E. (2018). Refuel: Exploring sparse features in deep reinforcement learning for fast disease diagnosis. In *Advances in neural information processing systems* (pp. 7322–7331).

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014) Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 701–710). ACM.

Pevný, T., & Kovařík, V. (2019). Approximation capability of neural networks on spaces of probability measures and tree-structured domains. arXiv preprint arXiv:1906.00764.

Pevný, T., & Somol, P. (2016). Discriminative models for multi-instance problems with tree structure. In *Proceedings of the 2016 ACM workshop on artificial intelligence and security* (pp. 83–91). ACM.

Pevný, T., & Somol, P. (2017). Using neural network formalism to solve multiple-instance problems. In *International symposium on neural networks* (pp. 135–142). Springer.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning* (pp. 1889–1897). PMLR.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Shim, H., Hwang, S. J., & Yang, E. (2018). Joint active feature acquisition and classification with variable-size set encoding. In *Advances in neural information processing systems* (pp. 1375–1385).

Shpakova, T., & Sokolovska, N. (2021). Probabilistic personalised cascade with abstention. *Pattern Recognition Letters, 147*, 8–15.

Song, C., Chen, C., Li, Y., & Wu, X. (2018). Deep reinforcement learning apply in electromyography data classification. In *2018 IEEE international conference on cyborg and bionic systems (CBS)* (pp. 505–510). IEEE.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). Cambridge, MA: MIT Press.

Tan, M. (1993). Cost-sensitive learning of classification knowledge and its applications in robotics. *Machine Learning, 13*(1), 7–33.

Tang, Y., & Agrawal, S. (2020). Discretizing continuous action space for on-policy optimization. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 34, pp. 5981–5988).

Trapeznikov, K., & Saligrama, V. (2013). Supervised sequential classification under budget constraints. In *Artificial intelligence and statistics* (pp. 581–589).

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double Q-learning. In *AAAI conference on artificial intelligence* (pp. 2094–2100).

Vivar, G., Mullakaeva, K., Zwergal, A., Navab, N., & Ahmadi, S.-A. (2020). Peri-diagnostic decision support through cost-efficient feature acquisition at test-time. In *International conference on medical image computing and computer-assisted intervention* (pp. 572–581). Springer.

Wang, J., Bolukbasi, T., Trapeznikov, K., & Saligrama, V. (2014). Model selection by linear programming. In *European conference on computer vision* (pp. 647–662). Springer.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995–2003).

Wang, J., Trapeznikov, K., & Saligrama, V. (2014). An lp for sequential learning under budgets. In *Artificial intelligence and statistics* (pp. 987–995).

Wang, J., Trapeznikov, K., & Saligrama, V. (2015). Efficient learning by directed acyclic graph for resource constrained prediction. In *Advances in neural information processing systems* (pp. 2152–2160).

Xu, J., Sun, Z., & Ma, C. (2021). Crowd aware summarization of surveillance videos by deep reinforcement learning. *Multimedia Tools and Applications, 80*(4), 6121–6141.

Xu, Z., Kusner, M., Weinberger, K., & Chen, M. (2013). Cost-sensitive tree of classifiers. In *International conference on machine learning* (pp. 133–141).

Xu, Z., Kusner, M., Weinberger, K., Chen, M., & Chapelle, O. (2014). Classifier cascades and trees for minimizing feature evaluation cost. *Journal of Machine Learning Research, 15*(1), 2113–2144.

Xu, Z., Weinberger, K., & Chapelle, O. (2012). The greedy miser: Learning under test-time budgets. In *Proceedings of the 29th international conference on international conference on machine learning* (pp. 1299–1306). Omnipress.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., & Smola, A. J. (2017). Deep sets. In *Advances in neural information processing systems* (pp. 3391–3401).

Zhang, Y., Vuong, Q. H., Song, K., Gong, X.-Y., & Ross, K. W. (2018). Efficient entropy for policy gradient with multidimensional action space. arXiv preprint arXiv:1806.00589.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., & Sun, M. (2018). Graph neural networks: A review of methods and applications. arXiv preprint arXiv:1812.08434.

Zhu, M., & Zhu, H. (2020) Learning a cost-effective strategy on incomplete medical data. In *International conference on database systems for advanced applications* (pp. 175–191). Springer.

Zolghadr, N., Bartók, G., Greiner, R., György, A., & Szepesvári, C. (2013). Online learning with costly features and labels. In *Advances in neural information processing systems* (pp. 1241–1249).