trickster Documentation

Release 0.8.0

Bogdan Kulynych, Nikita Samarin

Apr 18, 2019

Usage

Library and experiments for attacking machine learning in discrete domains using graph search.

CHAPTER 1

Setup

1.1 Library

Install the trickster library as a Python package:

```
pip install -e git+git://github.com/spring-epfl/trickster#egg=trickster
```

Note that trickster requires Python **3.6**.

1.2 Experiments and scripts

1.2.1 Python packages

Install the required Python packages:

```
pip install -r requirements.txt
```

1.2.2 System packages

On Ubuntu, you need these system packages:

```
apt install parallel unzip
```

1.2.3 Datasets

To download the datasets, run this:

```
make data
```

The datasets include:

- UCI German credit dataset
- Zafar Gilani's Twitter bot classification dataset
- Tao Wang's knndata

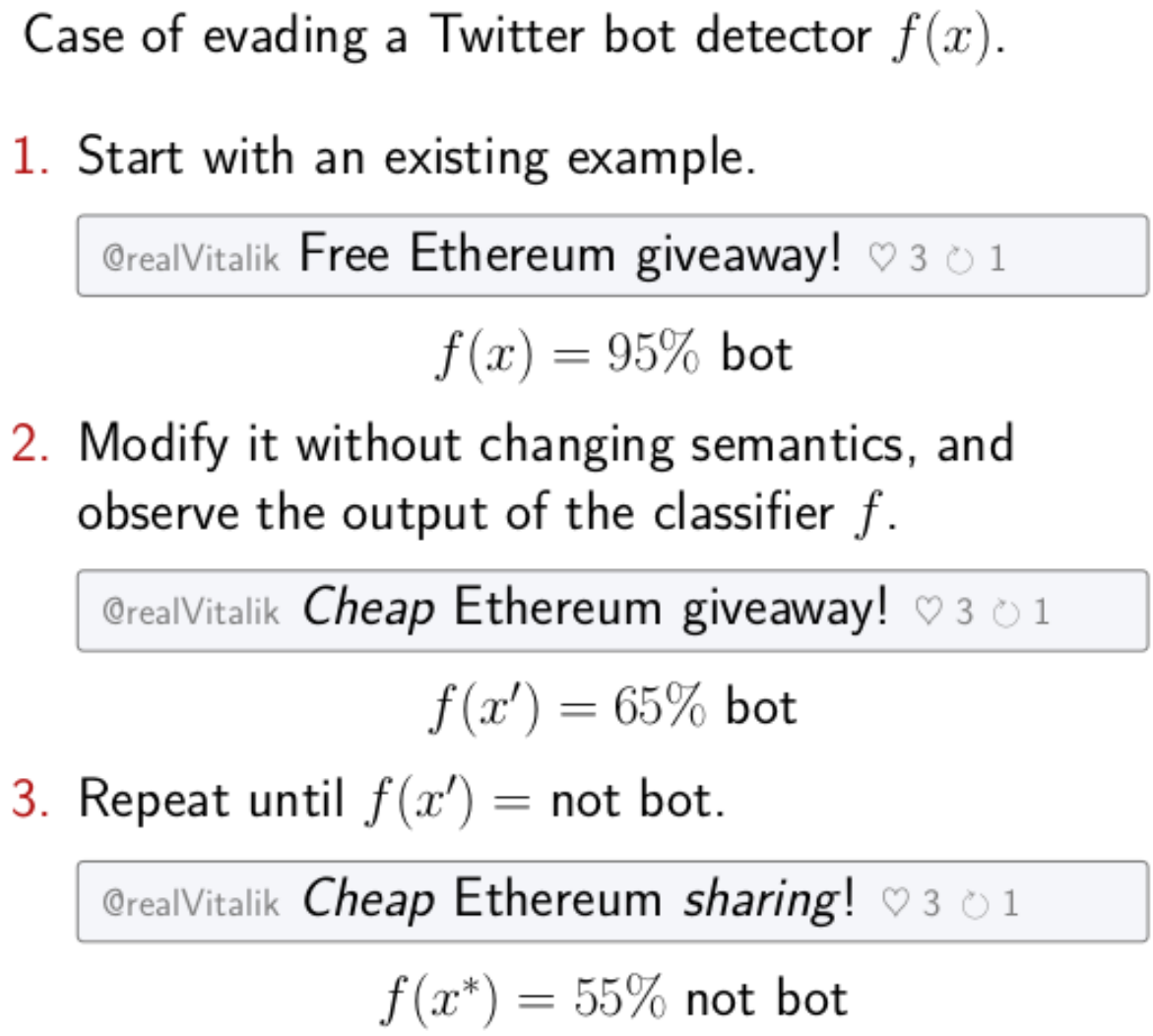
CHAPTER 2

Guide

2.1 Intro

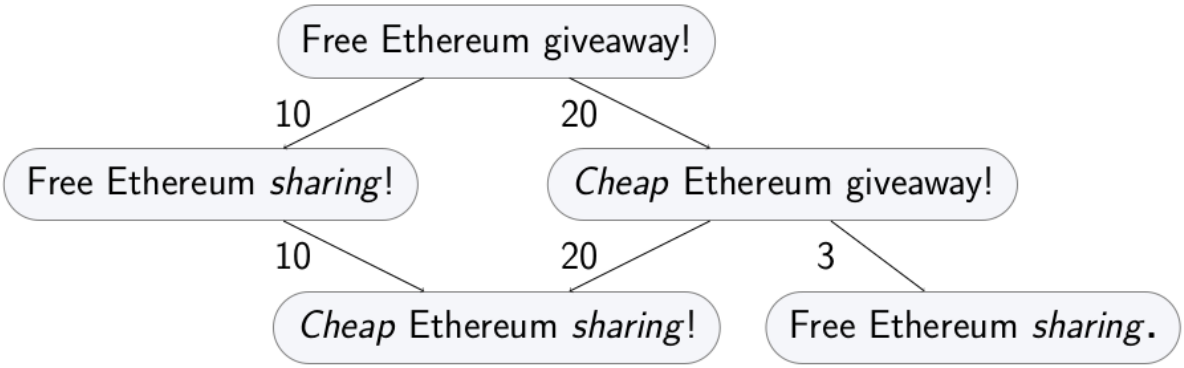
2.1.1 Attacks

trickster allows to run the following kind of attack. Starting with a given correctly classified example, apply a sequence of semantics-preserving transformations to it until the changed version of the example causes a missclassification error. Ideally, the cost of these transformations must be kept to a minimum.



2.1.2 Transformation graph

The *capabilities* of the adversary, including the inherent constraints of the domain, can be formalized as a *transformation graph*. Every example in the domain is a node in the graph, and its direct descendants are some *atomic* transformations. Other transformations are sequences of atomic transformations, so they are paths in the graph.



Each transformation can have an associated cost, representing how hard it is for the adversary to perform it.

Running an attack is then equal to graph search: traversing this transformation graph until a transformed example that flips the classifier decision is found.

2.2 Generic case

This will show how to run a basic evasion attack for a toy Twitter bot detector.

Let's use a different feature space for a Twitter bot detector. Each example will represent an account, and contain two features: number of tweets, and the number of likes of the account's tweets.

```
import attr

@attr.s(auto_attribs=True)
class TwitterAccount:
    num_own_tweets: int
    num_likes: int
```

2.2.1 Defining the transformation graph

The graph is defined using an `expand_fn` function. Given an example (a node in the graph), it returns its atomic transformations (direct descendants in the graph) and their corresponding costs.

One possible way to define atomic transformations for numeric features like the one used in the current example is increments and decrements. Each transformation will either increment or decrement one of the features. It's easy to see that other transformations that change the integer values are compositions of these.

```
def expand_fn(x):
    yield attr.evolve(x, num_own_tweets=x.num_own_tweets + 1), 1
    yield attr.evolve(x, num_own_tweets=x.num_own_tweets - 1), 1
    yield attr.evolve(x, num_likes=x.num_likes + 1), 10
    yield attr.evolve(x, num_likes=x.num_likes - 1), 1

x = TwitterAccount(num_own_tweets=15, num_likes=5)
for transformation, cost in expand_fn(x):
    print('Transformation: %s. Cost: %d' % (transformation, cost))
```

(continues on next page)

(continued from previous page)

```
# Transformation: TwitterAccount(num_own_tweets=16, num_likes=5). Cost: 1
# ...
```

In this instantation of the `expand_fn`, all changes have cost of one, except increasing number of likes to own tweets. That one is more costly, since it requires more work on the adversary's side. Any other cost model is also possible.

2.2.2 Defining the attack goal

Let's first define a toy classifier.

```
def is_bot(x):
    if x.num_likes < x.num_own_tweets:
        return 'Bot!'
    else:
        return 'Not bot.'
```

The goal of the attack is to get a 'not bot' decision. Define it using `goal_fn`:

```
def goal_fn(x):
    return is_bot(x) == 'Not bot.'
```

2.2.3 Hash function

Final technicality: examples need to be hashable. The hash function is defined as `hash_fn`:

```
def hash_fn(x):
    return hash((x.num_likes, x.num_own_tweets)
```

2.2.4 Running the graph search attack

Having an initial example, having defined the transformation graph through `expand_fn` and the adversarial goal through `goal_fn`, you can now run a simple attack that will find an adversarial example that incurs minimal transformation cost to the adversary:

```
from trickster.search import a_star_search

x = TwitterAccount(num_own_tweets=15, num_likes=5)

adv_x, cost = a_star_search(
    start_node=x,
    expand_fn=expand_fn,
    goal_fn=goal_fn,
    hash_fn=hash_fn
)
print('Adversarial account: %s. Cost of the attack: %d' % (adv_x, cost))

# Adversarial account: TwitterAccount(num_own_tweets=5, num_likes=5). Cost of the␣
↪attack: 10
```

The procedure finds a sequence of transformation that flips the decision of the classifier while incurring the minimal possible cost to the adversary. If everything was defined correctly, the adversary can now enact these changes to evade the detection. In this case, the adversary needs to delete some tweets from an existing account.

This adversarial example is provably minimal, at the cost of extensive traversal of the transformation graph. Next sections describe how to do better.

2.3 Using heuristics

The search can be sped up using heuristics. This section is coming up.

2.4 Optimal setup (`trickster.optim`)

For some settings, one can find provably minimal examples even when using heuristics. This section is coming up.

CHAPTER 3

trickster.optim

Everything needed to run the experiments in the optimal setting.

class trickster.optim.**BenchCost**(*problem_ctx*)
Alternative cost function used for analyses and stats.

__call__(*x*, *another*)
Call self as a function.

class trickster.optim.**CategoricalLpProblemContext**(*clf: Any*, *target_class: float*, *target_confidence: float = 0.5*, *epsilon: float = 1.0*, *lp_space=1*, *expansion_specs: List[trickster.domain.categorical.FeatureExpansionSpec] = NOTHING*)
Context for an optimal search instance in Lp space with categorical transformations.

Parameters

- **clf** – Target classifier.
- **target_class** – Target class.
- **target_confidence** – Target class confidence.
- **lp_space** (*LpSpace*) – $$L_p$$ space.
- **epsilon** – Runtime-optimality trade-off parameter. 1 means optimal. More is sub-optimal but faster.
- **expansion_specs** – List of categorical expansion specs (*trickster.domain.categorical.FeatureExpansionSpec*)

```
>>> problem_ctx = CategoricalLpProblemContext(
...     clf="stub", target_class=1, target_confidence=0.5,
...     lp_space="inf", epsilon=10)
```

get_graph_search_problem(*x*)
Generate an instance of graph search problem.

class trickster.optim.**GoalFunc**(*problem_ctx*)
Tells whether an example flips the decision of the classifier.

__call__(*x*)
Call self as a function.

class trickster.optim.**SpecExpandFunc**(*problem_ctx*, *weight_vec=None*)
Expand candidates using given specs with each transformation having L_p cost.

__call__(*x*)
Call self as a function.

trickster.optim.**run_experiment**(*problem_ctx*, *data*, *confidence_margin=1.0*, *reduce_classifier=True*, *transformable_feature_idxs=None*, *make_graph_search_problem=None*, *get_cost_weights_fn=None*, *graph_search_kwargs=None*, *logger=None*)
Experiment runner for optimal setting in categorical feature space.

An experiment finds adversarial examples for a given model and a given dataset.

Parameters

- **problem_ctx** (*trickster.base.ProblemContext*) – Search problem context.
- **data** – Data tuple (X, y)
- **confidence_margin** (*float*) – Pick initial examples that are at most this far away from the target confidence. Use this to get to relax the problem.
- **reduce_classifier** (*bool*) – Whether to use *trickster.linear.create_reduced_linear_classifier()* if possible.
- **transformable_feature_idxs** (*list*) –
- **graph_search_kwargs** (*dict*) – Parameters passed to the search function call.
- **logger** – Logger instance.

CHAPTER 4

trickster.linear

Tools for working with linear and linearized models.

class trickster.linear.**LinearGridHeuristic**(*problem_ctx*, *cache_grad=True*, *weight_vec=None*, *grid_step=1*)

Snaps values of a linear heuristic to values on a regular grid.

This is useful when the transformations in the transformation graph have fixed costs.

Parameters **grid_step** – Regular grid step.

__call__(*x*)

Call self as a function.

class trickster.linear.**LinearHeuristic**(*problem_ctx*, *cache_grad=True*, *weight_vec=None*)

$$L_p$$ distance to the decision boundary of a binary linear classifier.

Parameters

- **cache_grad** – Cache the forward gradient norm.
- **weight_vec** – Feature-wise weights if the $$L_p$$ space is weighted.

__call__(*x*)

Call self as a function.

trickster.linear.**create_reduced_linear_classifier**(*clf*, *x*, *transformable_feature_idxs*)

Construct a reduced-dimension classifier based on the original one for a given example.

The reduced-dimension classifier should behave the same way as the original one, but operate in a smaller feature space. This is done by fixing the score of the classifier on a static part of `x`, and integrating it into the bias parameter of the reduced classifier.

For example, let $$x = [1, 2, 3]$$, weights of the classifier $$w = [1, 1, 1]$$ and bias term $$b = 0$$, and the only transformable feature index is 0. Then the reduced classifier has weights $$w' = [1]$$, and the bias term incorporates the non-transformable part of $$x$$: $$b' = -1 cdot 2 + 1 cdot 3$$.

Parameters

- **clf** – Original logistic regression classifier

- **x** – An example
- **transformable_feature_idxs** – List of features that can be changed in the given example.

trickster.linear.**dist_to_decision_boundary**(*clf*, *x*, *target_class*, *target_confidence*, *lp_space*, *inv_feature_weights=None*, *_grad_norm=None*)

Compute distance to the decision boundary of a binary linear classifier.

trickster.linear.**get_forward_grad**(*clf*, *x*, *target_class=None*)

Get the forward gradient of a classifier.

Parameters

- **clf** – Classifier.
- **x** – Input.
- **target_class** – Currently not supported.

CHAPTER 5

trickster.search

trickster.search.**a_star_search**(*start_node*, *expand_fn*, *goal_fn*, *heuristic_fn=None*, *hash_fn=None*, *iter_lim=None*, *beam_size=None*, *return_path=False*)

Generalized A* search with no beam size limit.

See *generalized_a_star_search()*.

trickster.search.**generalized_a_star_search**(*start_node*, *expand_fn*, *goal_fn*, *heuristic_fn=None*, *hash_fn=None*, *iter_lim=None*, *beam_size=None*, *return_path=False*)

Generalized A* search.

Returns the tuple (cost, target_node) if return_path is set to False. Otherwise, returns the target node, the costs of nodes expanded by the algorithm, and the optimal path from the initial node to the target node.

Parameters

- **start_node** – Initial node.
- **expand_fn** – Returns an iterable of tuples (neighbour, cost).
- **goal_fn** – Returns True if the current node is the target node.
- **heuristic_fn** – Returns an estimate of the cost to the target node. By default, is a constant 0.
- **hash_fn** – Hash function for nodes. By default equals the identity function f(x) = x.
- **iter_lim** – Maximum number of iterations to try.
- **beam_size** – Beam size. Turns the search into beam search.
- **return_path** – Whether to return the optimal path from the initial node to the target node. By default equals False.

trickster.search.**ida_star_search**(*start_node*, *expand_fn*, *goal_fn*, *heuristic_fn=None*, *hash_fn=None*, *iter_lim=None*, *return_path=False*)

IDA* search.

Returns the tuple (cost, target_node) if return_path is set to False. Otherwise, returns the target node, the costs of nodes expanded by the algorithm, and the optimal path from the initial node to the target node.

Parameters

- **start_node** – Initial node.
- **expand_fn** – Returns an iterable of tuples (neighbour, cost).
- **goal_fn** – Returns True if the current node is the target node.
- **heuristic_fn** – Returns an estimate of the cost to the target node. By default, is a constant 0.
- **hash_fn** – Hash function for nodes. By default equals the identity function f(x) = x.
- **iter_lim** – Maximum number of iterations to try.
- **return_path** – Whether to return the optimal path from the initial node to the target node. By default equals False.

CHAPTER 6

trickster.domain

6.1 Categorical feature spaces

Transformations and stats for quantized, categorical, and collection features.

class trickster.domain.categorical.**FeatureExpansionSpec**(*idxs: List[int], expand_fn: Callable, feature_name: str = None, extras: List = None*)

Categorical feature expansion specification.

Parameters

- **idxs** – Indexes in a feature vector that correspond to this feature.
- **expand_fn** – Expansion funciton.
- **feature_name** – Feature name.
- **weights** – Feature-wise weights.

class trickster.domain.categorical.**Node**(*src: List, depth: int = 0, feature_extract_fn: Callable = None*)

Node in a transformation graph.

Parameters

- **x** – *Raw* example.
- **feature_extract_fn** – Feature extraction funcion.
- **depth** – Number of hops from the original example.

expand(*expansion_specs*)

Return the expanded neighbour nodes.

features

Return the feature vector.

trickster.domain.categorical.**expand**(*sample*, *expansion_specs*)
Perform multiple expansions.

Parameters

- **sample** (*Numpy array.*) – Initial node.
- **expansion_specs** – List of *FeatureExpansionSpec* object.

Returns List of numpy arrays.

trickster.domain.categorical.**expand_categorical**(*sample*, *feat_idxs*)
Expand all values of a single categorical feature.

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_collection**(*sample*, *feat_idxs*)
Expand all values of a collection of categorical features (set and unset).

Parameters

- **sample** (*Numpy array.*) – Initial node.
- **feat_idxs** (*Numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns List of numpy arrays.

trickster.domain.categorical.**expand_collection_set**(*sample*, *feat_idxs*)
Expand all values of a collection of categorical features (set [0,1] to [1,0]).

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_collection_unset**(*sample*, *feat_idxs*)
Expand all values of a collection of categorical features (unset [1,0] to [0,1]).

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_quantized**(*sample*, *feat_idxs*)
Get the neighbouring value (shift '1' right and left) in a quantized one-hot feature vector.

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_quantized_decrement**(*sample*, *feat_idxs*)
Get the neighbouring value (shift '1' left) in a quantized one-hot feature vector.

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_quantized_increase**(*sample*, *feat_idxs*)
Expand all higher values of a single categorical feature.

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**expand_quantized_increment**(*sample*, *feat_idxs*)
Get the neighbouring value (shift '1' right) in a quantized one-hot feature vector.

Parameters

- **sample** (*numpy array.*) – Initial node.
- **feat_idxs** (*numpy array or list of ints.*) – Indexes pointing to transformable features in the sample array.

Returns list of numpy arrays.

trickster.domain.categorical.**get_feature_coef_importance**(*X*, *clf*, *transformable_feature_idxs*)
Get the most important features from the transformable feature set based on classifier parameters.

trickster.domain.categorical.**get_feature_diff_importance**(*difference*, *transformable_feature_idxs*)
Get the most important features from the transformable feature set based on the feature difference between the initial and adversarial example.

trickster.domain.categorical.**noop**(*sample*, *feat_idxs*)
Don't expand.

CHAPTER 7

Citing trickster

This is an accompanying code to the paper "Evading classifiers in discrete domains with provable optimality guarantees" by B. Kulynych, J. Hayes, N. Samarin, and C. Troncoso, 2018. Cite as follows:

```
@article{KulynychHST18,
  author    = {Bogdan Kulynych and
               Jamie Hayes and
               Nikita Samarin and
               Carmela Troncoso},
  title     = {Evading classifiers in discrete domains with provable optimality
↪guarantees},
  journal   = {CoRR},
  volume    = {abs/1810.10939},
  year      = {2018},
  url       = {http://arxiv.org/abs/1810.10939},
  archivePrefix = {arXiv},
  eprint    = {1810.10939},
}
```

CHAPTER 8

Acknowledgements

This work is funded by the NEXTLEAP project within the European Union's Horizon 2020 Framework Programme for Research and Innovation (H2020-ICT-2015, ICT-10-2015) under grant agreement 688722.

CHAPTER 9

License

Copyright 2016—2018 Bogdan Kulynych (EPFL SPRING Lab)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

I

L

N

R

S

T