
# Stealing Machine Learning Models via Prediction APIs

**Florian Tramèr,** *École Polytechnique Fédérale de Lausanne (EPFL);* **Fan Zhang,** *Cornell University;* **Ari Juels,** *Cornell Tech;* **Michael K. Reiter,** *The University of North Carolina at Chapel Hill;* **Thomas Ristenpart,** *Cornell Tech*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer

**This paper is included in the Proceedings of the 25th USENIX Security Symposium**

**August 10–12, 2016 • Austin, TX**

ISBN 978-1-931971-32-4

**Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX**

# Stealing Machine Learning Models via Prediction APIs

Florian Tramèr
*EPFL*

Fan Zhang
*Cornell University*

Ari Juels
*Cornell Tech, Jacobs Institute*

Michael K. Reiter
*UNC Chapel Hill*

Thomas Ristenpart
*Cornell Tech*

## Abstract

Machine learning (ML) models may be deemed confidential due to their sensitive training data, commercial value, or use in security applications. Increasingly often, confidential ML models are being deployed with publicly accessible query interfaces. ML-as-a-service ("predictive analytics") systems are an example: Some allow users to train models on potentially sensitive data and charge others for access on a pay-per-query basis.

The tension between model confidentiality and public access motivates our investigation of *model extraction attacks*. In such attacks, an adversary with black-box access, but no prior knowledge of an ML model's parameters or training data, aims to duplicate the functionality of (i.e., "steal") the model. Unlike in classical learning theory settings, ML-as-a-service offerings may accept partial feature vectors as inputs and include confidence values with predictions. Given these practices, we show simple, efficient attacks that extract target ML models with near-perfect fidelity for popular model classes including logistic regression, neural networks, and decision trees. We demonstrate these attacks against the online services of BigML and Amazon Machine Learning. We further show that the natural countermeasure of omitting confidence values from model outputs still admits potentially harmful model extraction attacks. Our results highlight the need for careful ML model deployment and new model extraction countermeasures.

## 1 Introduction

Machine learning (ML) aims to provide automated extraction of insights from data by means of a predictive model. A predictive model is a function that maps feature vectors to a categorical or real-valued output. In a supervised setting, a previously gathered data set consisting of possibly confidential feature-vector inputs (e.g., digitized health records) with corresponding output class labels (e.g., a diagnosis) serves to train a predictive model that can generate labels on future inputs. Popular models include support vector machines (SVMs), logistic regressions, neural networks, and decision trees.

ML algorithms' success in the lab and in practice has led to an explosion in demand. Open-source frameworks such as PredictionIO and cloud-based services offered by Amazon, Google, Microsoft, BigML, and others have arisen to broaden and simplify ML model deployment.

Cloud-based ML services often allow model owners to charge others for queries to their commercially valuable models. This pay-per-query deployment option exemplifies an increasingly common tension: The query interface of an ML model may be widely accessible, yet the model itself and the data on which it was trained may be proprietary and confidential. Models may also be privacy-sensitive because they leak information about training data [4, 23, 24]. For security applications such as spam or fraud detection [9, 29, 36, 55], an ML model's confidentiality is critical to its utility: An adversary that can learn the model can also often evade detection [4,36].

In this paper we explore *model extraction attacks*, which exploit the tension between query access and confidentiality in ML models. We consider an adversary that can query an ML model (a.k.a. a prediction API) to obtain predictions on input feature vectors. The model may be viewed as a black box. The adversary may or may not know the model type (logistic regression, decision tree, etc.) or the distribution over the data used to train the model. The adversary's goal is to extract an equivalent or near-equivalent ML model, i.e., one that achieves (close to) 100% agreement on an input space of interest.

We demonstrate successful model extraction attacks against a wide variety of ML model types, including decision trees, logistic regressions, SVMs, and deep neural networks, and against production ML-as-a-service (MLaaS) providers, including Amazon and BigML.[1] In nearly all cases, our attacks yield models that are func-

[1] We simulated victims by training models in our own accounts. We have disclosed our results to affected services in February 2016.

| Service | Model Type | Data set | Queries | Time (s) |
|---|---|---|---|---|
| Amazon | Logistic Regression | Digits | 650 | 70 |
| | Logistic Regression | Adult | 1,485 | 149 |
| BigML | Decision Tree | German Credit | 1,150 | 631 |
| | Decision Tree | Steak Survey | 4,013 | 2,088 |

Table 1: **Results of model extraction attacks on ML services.** For each target model, we report the number of prediction queries made to the ML API in an attack that extracts a 100% equivalent model. The attack time is primarily influenced by the service's prediction latency ($\approx$ 100ms/query for Amazon and $\approx$ 500ms/query for BigML).

tionally very close to the target. In some cases, our attacks extract the exact parameters of the target (e.g., the coefficients of a linear classifier or the paths of a decision tree). For some targets employing a model type, parameters or features unknown to the attacker, we additionally show a successful preliminary attack step involving reverse-engineering these model characteristics.

Our most successful attacks rely on the information-rich outputs returned by the ML prediction APIs of all cloud-based services we investigated. Those of Google, Amazon, Microsoft, and BigML all return *high-precision confidence values in addition to class labels*. They also respond to partial queries lacking one or more features. Our setting thus differs from traditional learning-theory settings [3, 7, 8, 15, 30, 33, 36, 53] that assume only *membership queries*, outputs consisting of a class label only. For example, for logistic regression, the confidence value is a simple log-linear function $1/(1+e^{-(\mathbf{w}\cdot\mathbf{x}+\beta)})$ of the $d$-dimensional input vector $\mathbf{x}$. By querying $d+1$ random $d$-dimensional inputs, an attacker can with high probability solve for the unknown $d+1$ parameters $\mathbf{w}$ and $\beta$ defining the model. We emphasize that while this model extraction attack is simple and non-adaptive, it affects all of the ML services we have investigated.

Such equation-solving attacks extend to multiclass logistic regressions and neural networks, but do not work for decision trees, a popular model choice. (BigML, for example, initially offered only decision trees.) For decision trees, a confidence value reflects the number of training data points labeled correctly on an input's path in the tree; simple equation-solving is thus inapplicable. We show how confidence values can nonetheless be exploited as pseudo-identifiers for paths in the tree, facilitating discovery of the tree's structure. We demonstrate successful model extraction attacks that use adaptive, iterative search algorithms to discover paths in a tree.

We experimentally evaluate our attacks by training models on an array of public data sets suitable as stand-ins for proprietary ones. We validate the attacks locally using standard ML libraries, and then present case studies on BigML and Amazon. For both services, we show computationally fast attacks that use a small number of queries to extract models matching the targets on 100% of tested inputs. See Table 1 for a quantitative summary.

Having demonstrated the broad applicability of model extraction attacks to existing services, we consider the most obvious potential countermeasure ML services might adopt: Omission of confidence values, i.e., output of class labels only. This approach would place model extraction back in the membership query setting of prior work in learning theory [3, 8, 36, 53]. We demonstrate a generalization of an adaptive algorithm by Lowd and Meek [36] from binary linear classifiers to more complex model types, and also propose an attack inspired by the agnostic learning algorithm of Cohn et al. [18]. Our new attacks extract models matching targets on >99% of the input space for a variety of model classes, but need up to 100× more queries than equation-solving attacks (specifically for multiclass linear regression and neural networks). While less effective than equation-solving, these attacks remain attractive for certain types of adversary. We thus discuss further ideas for countermeasures.

In summary, we explore model extraction attacks, a practical kind of learning task that, in particular, affects emerging cloud-based ML services being built by Amazon, Google, Microsoft, BigML, and others. We show:

- *Simple equation-solving model extraction attacks* that use non-adaptive, random queries to solve for the parameters of a target model. These attacks affect a wide variety of ML models that output confidence values. We show their success against Amazon's service (using our own models as stand-ins for victims'), and also report successful reverse-engineering of the (only partially documented) model type employed by Amazon.
- *A new path-finding algorithm for extracting decision trees* that abuses confidence values as quasi-identifiers for paths. To our knowledge, this is the first example of practical "exact" decision tree learning. We demonstrate the attack's efficacy via experiments on BigML.
- *Model extraction attacks against models that output only class labels*, the obvious countermeasure against extraction attacks that rely on confidence values. We show slower, but still potentially dangerous, attacks in this setting that build on prior work in learning theory.

We additionally make a number of observations about the implications of extraction. For example, attacks against Amazon's system indirectly leak various summary statistics about a private training set, while extraction against kernel logistic regression models [57] recovers significant information about individual training data points.

The source code for our attacks is available online at `https://github.com/ftramer/Steal-ML`.

## 2 Background

For our purposes, a ML model is a function $f\colon \mathcal{X} \to \mathcal{Y}$. An input is a $d$-dimensional vector in the feature space

$\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_d$. Outputs lie in the range $\mathcal{Y}$.

We distinguish between categorical features, which assume one of a finite set of values (whose set size is the arity of the feature), and continuous features, which assume a value in a bounded subset of the real numbers. Without loss of generality, for a categorical feature of arity $k$, we let $\mathcal{X}_i = \mathbb{Z}_k$. For a continuous feature taking values between bounds $a$ and $b$, we let $\mathcal{X}_i = [a,b] \subset \mathbb{R}$.

Inputs to a model may be pre-processed to perform feature extraction. In this case, inputs come from a space $\mathcal{M}$, and feature extraction involves application of a function $\mathsf{ex}\colon \mathcal{M} \to \mathcal{X}$ that maps inputs into a feature space. Model application then proceeds by composition in the natural way, taking the form $f(\mathsf{ex}(M))$. Generally, feature extraction is many-to-one. For example, $M$ may be a piece of English language text and the extracted features counts of individual words (so-called "bag-of-words" feature extraction). Other examples are input scaling and one-hot-encoding of categorical features.

We focus primarily on classification settings in which $f$ predicts a nominal variable ranging over a set of classes. Given $c$ classes, we use as class labels the set $\mathbb{Z}_c$. If $\mathcal{Y} = \mathbb{Z}_c$, the model returns only the predicted class label. In some applications, however, additional information is often helpful, in the form of real-valued measures of confidence on the labels output by the model; these measures are called *confidence values*. The output space is then $\mathcal{Y} = [0,1]^c$. For a given $\mathbf{x} \in \mathcal{X}$ and $i \in \mathbb{Z}_c$, we denote by $f_i(\mathbf{x})$ the $i^{\text{th}}$ component of $f(\mathbf{x}) \in \mathcal{Y}$. The value $f_i(\mathbf{x})$ is a model-assigned probability that $\mathbf{x}$ has associated class label $i$. The model's predicted class is defined by the value $\text{argmax}_i\ f_i(\mathbf{x})$, i.e., the most probable label.

We associate with $\mathcal{Y}$ a distance measure $d_{\mathcal{Y}}$. We drop the subscript $\mathcal{Y}$ when it is clear from context. For $\mathcal{Y} = \mathbb{Z}_c$ we use 0-1 distance, meaning $d(y,y') = 0$ if $y = y'$ and $d(y,y') = 1$ otherwise. For $\mathcal{Y} = [0,1]^c$, we use the 0-1 distance when comparing predicted classes; when comparing class probabilities directly, we instead use the total variation distance, given by $d(\mathbf{y},\mathbf{y}') = \frac{1}{2}\sum |\mathbf{y}[i] - \mathbf{y}'[i]|$. In the rest of this paper, unless explicitly specified otherwise, $d_{\mathcal{Y}}$ refers to the 0-1 distance over class labels.

**Training algorithms.** We consider models obtained via supervised learning. These models are generated by a training algorithm $\mathcal{T}$ that takes as input a training set $\{(\mathbf{x}_i, y_i)\}_i$, where $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ is an input with an associated (presumptively correct) class label. The output of $\mathcal{T}$ is a model $f$ defined by a set of *parameters*, which are model-specific, and *hyper-parameters*, which specify the type of models $\mathcal{T}$ generates. Hyper-parameters may be viewed as distinguished parameters, often taken from a small number of standard values; for example, the kernel-type used in an SVM, of which only a small set are used in practice, may be seen as a hyper-parameter.

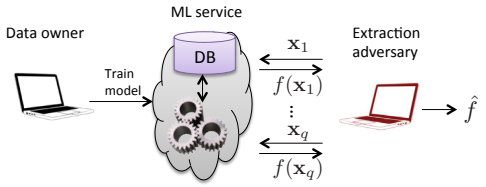


Figure 1: **Diagram of ML model extraction attacks.** A data owner has a model $f$ trained on its data and allows others to make prediction queries. An adversary uses $q$ prediction queries to extract an $\hat{f} \approx f$.

## 3 Model Extraction Attacks

An ML model extraction attack arises when an adversary obtains black-box access to some *target model* $f$ and attempts to learn a model $\hat{f}$ that closely approximates, or even matches, $f$ (see Figure 1).

As mentioned previously, the restricted case in which $f$ outputs class labels only, matches the membership query setting considered in learning theory, e.g., PAC learning [53] and other previous works [3, 7, 8, 15, 30, 33, 36]. Learning theory algorithms have seen only limited study in practice, e.g., in [36], and our investigation may be viewed as a practice-oriented exploration of this branch of research. Our initial focus, however, is on a different setting common in today's MLaaS services, which we now explain in detail. Models trained by these services emit data-rich outputs that often include confidence values, and in which partial feature vectors may be considered valid inputs. As we show later, this setting greatly advantages adversaries.

**Machine learning services.** A number of companies have launched or are planning to launch cloud-based ML services. A common denominator is the ability of users to upload data sets, have the provider run training algorithms on the data, and make the resulting models generally available for prediction queries. Simple-to-use Web APIs handle the entire interaction. This service model lets users capitalize on their data without having to set up their own large-scale ML infrastructure. Details vary greatly across services. We summarize a number of them in Table 2 and now explain some of the salient features.

A model is *white-box* if a user may download a representation suitable for local use. It is *black-box* if accessible only via a prediction query interface. Amazon and Google, for example, provide black-box-only services. Google does not even specify what training algorithm their service uses, while Amazon provides only partial documentation for its feature extraction ex (see Section 5). Some services allow users to monetize trained models by charging others for prediction queries.

To use these services, a user uploads a data set and optionally applies some data pre-processing (e.g., field removal or handling of missing values). She then trains a

| Service | White-box | Monetize | Confidence Scores | Logistic Regression | SVM | Neural Network | Decision Tree |
|---|---|---|---|---|---|---|---|
| Amazon [1] | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Microsoft [38] | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| BigML [11] | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| PredictionIO [43] | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |
| Google [25] | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 2: **Particularities of major MLaaS providers.** 'White-box' refers to the ability to download and use a trained model locally, and 'Monetize' means that a user may charge other users for black-box access to her models. Model support for each service is obtained from available documentation. The models listed for Google's API are a projection based on the announced support of models in standard PMML format [25]. Details on ML models are given in Appendix A.

model by either choosing one of many supported model classes (as in BigML, Microsoft, and PredictionIO) or having the service choose an appropriate model class (as in Amazon and Google). Two services have also announced upcoming support for users to upload their own trained models (Google) and their own custom learning algorithms (PredictionIO). When training a model, users may tune various parameters of the model or training-algorithm (e.g., regularizers, tree size, learning rates) and control feature-extraction and transformation methods.

For black-box models, the service provides users with information needed to create and interpret predictions, such as the list of input features and their types. Some services also supply the model class, chosen training parameters, and training data statistics (e.g., BigML gives the range, mean, and standard deviation of each feature).

To get a prediction from a model, a user sends one or more input queries. The services we reviewed accept both synchronous requests and asynchronous 'batch' requests for multiple predictions. We further found varying degrees of support for 'incomplete' queries, in which some input features are left unspecified [46]. We will show that exploiting incomplete queries can drastically improve the success of some of our attacks. Apart from PredictionIO, all of the services we examined respond to prediction queries with not only class labels, but a variety of additional information, including *confidence scores* (typically class probabilities) for the predicted outputs.

Google and BigML allow model owners to monetize their models by charging other users for predictions. Google sets a minimum price of $0.50 per 1,000 queries. On BigML, 1,000 queries consume at least 100 *credits*, costing $0.10–$5, depending on the user's subscription.

**Attack scenarios.** We now describe possible motivations for adversaries to perform model extraction attacks. We then present a more detailed threat model informed by characteristics of the aforementioned ML services.

*Avoiding query charges.* Successful monetization of prediction queries by the owner of an ML model $f$ requires confidentiality of $f$. A malicious user may seek to launch what we call a *cross-user* model extraction attack, stealing $f$ for subsequent free use. More subtly, in black-box-only settings (e.g., Google and Amazon), a service's business model may involve amortizing up-front training costs by charging users for future predictions. A model extraction attack will undermine the provider's business model if a malicious user pays less for training and extracting than for paying per-query charges.

*Violating training-data privacy.* Model extraction could, in turn, leak information about sensitive training data. Prior attacks such as model inversion [4, 23, 24] have shown that access to a model can be abused to infer information about training set points. Many of these attacks work better in white-box settings; model extraction may thus be a stepping stone to such privacy-abusing attacks. Looking ahead, we will see that in some cases, significant information about training data is leaked trivially by successful model extraction, because the model itself directly incorporates training set points.

*Stepping stone to evasion.* In settings where an ML model serves to detect adversarial behavior, such as identification of spam, malware classification, and network anomaly detection, model extraction can facilitate *evasion attacks*. An adversary may use knowledge of the ML model to avoid detection by it [4, 9, 29, 36, 55].

In all of these settings, there is an inherent assumption of secrecy of the ML model in use. We show that this assumption is broken for all ML APIs that we investigate.

**Threat model in detail.** Two distinct adversarial models arise in practice. An adversary may be able to make *direct* queries, providing an arbitrary input $\mathbf{x}$ to a model $f$ and obtaining the output $f(\mathbf{x})$. Or the adversary may be able to make only *indirect* queries, i.e., queries on points in input space $\mathcal{M}$ yielding outputs $f(\mathsf{ex}(M))$. The feature extraction mechanism $\mathsf{ex}$ may be unknown to the adversary. In Section 5, we show how ML APIs can further be exploited to "learn" feature extraction mechanisms. Both direct and indirect access to $f$ arise in ML services. (Direct query interfaces arise when clients are expected to perform feature extraction locally.) In either case, the output value can be a class label, a confidence value vector, or some data structure revealing various levels of information, depending on the exposed API.

We model the adversary, denoted by $\mathcal{A}$, as a randomized algorithm. The adversary's goal is to use as few queries as possible to $f$ in order to efficiently compute an approximation $\hat{f}$ that closely matches $f$. We formalize "closely matching" using two different error measures:

- *Test error* $R_{\text{test}}$: This is the average error over a test set $D$, given by $R_{\text{test}}(f,\hat{f}) = \sum_{(\mathbf{x},y)\in D} d(f(\mathbf{x}),\hat{f}(\mathbf{x}))/|D|$.

A low test error implies that $\hat{f}$ matches $f$ well for inputs distributed like the training data samples. [2]

- *Uniform error* $R_{\text{unif}}$: For a set $U$ of vectors uniformly chosen in $\mathcal{X}$, let $R_{\text{unif}}(f,\hat{f}) = \sum_{\mathbf{x}\in U} d(f(\mathbf{x}),\hat{f}(\mathbf{x}))/|U|$. Thus $R_{\text{unif}}$ estimates the fraction of the full feature space on which $f$ and $\hat{f}$ disagree. (In our experiments, we found $|U| = 10,000$ was sufficiently large to obtain stable error estimates for the models we analyzed.)

We define the extraction *accuracy* under test and uniform error as $1 - R_{\text{test}}(f,\hat{f})$ and $1 - R_{\text{unif}}(f,\hat{f})$. Here we implicitly refer to accuracy under 0-1 distance. When assessing how close the class probabilities output by $\hat{f}$ are to those of $f$ (with the total-variation distance) we use the notations $R^{\text{TV}}_{\text{test}}(f,\hat{f})$ and $R^{\text{TV}}_{\text{unif}}(f,\hat{f})$.

An adversary may know any of a number of pieces of information about a target $f$: What training algorithm $\mathcal{T}$ generated $f$, the hyper-parameters used with $\mathcal{T}$, the feature extraction function ex, etc. We will investigate a variety of settings in this work corresponding to different APIs seen in practice. We assume that $\mathcal{A}$ has no more information about a model's training data, than what is provided by an ML API (e.g., summary statistics). For simplicity, we focus on *proper* model extraction: If $\mathcal{A}$ believes that $f$ belongs to some model class, then $\mathcal{A}$'s goal is to extract a model $\hat{f}$ from the *same* class. We discuss some intuition in favor of proper extraction in Appendix D, and leave a broader treatment of *improper* extraction strategies as an interesting open problem.

## 4 Extraction with Confidence Values

We begin our study of extraction attacks by focusing on prediction APIs that return confidence values. As per Section 2, the output of a query to $f$ thus falls in a range $[0,1]^c$ where $c$ is the number of classes. To motivate this, we recall that most ML APIs reveal confidence values for models that support them (see Table 2). This includes logistic regressions (LR), neural networks, and decision trees, defined formally in Appendix A. We first introduce a generic *equation-solving* attack that applies to all logistic models (LR and neural networks). In Section 4.2, we present two novel *path-finding* attacks on decision trees.

### 4.1 Equation-Solving Attacks

Many ML models we consider directly compute class probabilities as a continuous function of the input $\mathbf{x}$ and real-valued model parameters. In this case, an API that reveals these class probabilities provides an adversary $\mathcal{A}$ with samples $(\mathbf{x}, f(\mathbf{x}))$ that can be viewed as equations in the unknown model parameters. For a large class of models, these equation systems can be efficiently solved, thus recovering $f$ (or some good approximation of it).

[2] Note that for some $D$, it is possible that $\hat{f}$ predicts true labels better than $f$, yet $R_{\text{test}}(f,\hat{f})$ is large, because $\hat{f}$ does not closely match $f$.

| Data set | Synthetic | # records | # classes | # features |
|---|---|---|---|---|
| Circles | Yes | 5,000 | 2 | 2 |
| Moons | Yes | 5,000 | 2 | 2 |
| Blobs | Yes | 5,000 | 3 | 2 |
| 5-Class | Yes | 1,000 | 5 | 20 |
| Adult (Income) | No | 48,842 | 2 | 108 |
| Adult (Race) | No | 48,842 | 5 | 105 |
| Iris | No | 150 | 3 | 4 |
| Steak Survey | No | 331 | 5 | 40 |
| GSS Survey | No | 16,127 | 3 | 101 |
| Digits | No | 1,797 | 10 | 64 |
| Breast Cancer | No | 683 | 2 | 10 |
| Mushrooms | No | 8,124 | 2 | 112 |
| Diabetes | No | 768 | 2 | 8 |

Table 3: **Data sets used for extraction attacks.** We train two models on the Adult data, with targets 'Income' and 'Race'. SVMs and binary logistic regressions are trained on data sets with 2 classes. Multiclass regressions and neural networks are trained on multiclass data sets. For decision trees, we use a set of public models shown in Table 5.

Our approach for evaluating attacks will primarily be experimental. We use a suite of synthetic or publicly available data sets to serve as stand-ins for proprietary data that might be the target of an extraction attack. Table 3 displays the data sets used in this section, which we obtained from various sources: the synthetic ones we generated; the others are taken from public surveys (*Steak Survey* [26] and *GSS Survey* [49]), from `scikit` [42] (*Digits*) or from the UCI ML library [35]. More details about these data sets are in Appendix B.

Before training, we remove rows with missing values, apply *one-hot-encoding* to categorical features, and scale all numeric features to the range $[-1,1]$. We train our models over a randomly chosen subset of 70% of the data, and keep the rest for evaluation (i.e., to calculate $R_{\text{test}}$). We discuss the impact of different pre-processing and feature extraction steps in Section 5, when we evaluate equation-solving attacks on production ML services.

#### 4.1.1 Binary logistic regression

As a simple starting point, we consider the case of logistic regression (LR). A LR model performs binary classification ($c = 2$), by estimating the probability of a binary response, based on a number of independent features. LR is one of the most popular binary classifiers, due to its simplicity and efficiency. It is widely used in many scientific fields (e.g., medical and social sciences) and is supported by all the ML services we reviewed.

Formally, a LR model is defined by parameters $\mathbf{w} \in \mathbb{R}^d$, $\beta \in \mathbb{R}$, and outputs a probability $f_1(\mathbf{x}) = \sigma(\mathbf{w}\cdot\mathbf{x} + \beta)$, where $\sigma(t) = 1/(1+e^{-t})$. LR is a linear classifier: it defines a *hyperplane* in the feature space $\mathcal{X}$ (defined by $\mathbf{w}\cdot\mathbf{x}+\beta = 0$), that separates the two classes.

Given an oracle sample $(\mathbf{x}, f(\mathbf{x}))$, we get a *linear* equation $\mathbf{w}\cdot\mathbf{x}+\beta = \sigma^{-1}(f_1(\mathbf{x}))$. Thus, $d+1$ samples are both necessary and sufficient (if the queried $\mathbf{x}$ are linearly independent) to recover $\mathbf{w}$ and $\beta$. Note that the required

samples are chosen non-adaptively, and can thus be obtained from a single batch request to the ML service.

We stress that while this extraction attack is rather straightforward, it directly applies, with possibly devastating consequences, to all cloud-based ML services we considered. As an example, recall that some services (e.g., BigML and Google) let model owners monetize black-box access to their models. Any user who wishes to make more than $d+1$ queries to a model would then minimize the prediction cost by first running a cross-user model extraction attack, and then using the extracted model for personal use, free of charge. As mentioned in Section 3, attackers with a final goal of model-inversion or evasion may also have incentives to first extract the model. Moreover, for services with black-box-only access (e.g., Amazon or Google), a user may abuse the service's resources to train a model over a large data set $D$ (i.e., $|D| \gg d$), and extract it after only $d+1$ predictions. Crucially, the extraction cost is independent of $|D|$. This could undermine a service's business model, should prediction fees be used to amortize the high cost of training.

For each binary data set shown in Table 3, we train a LR model and extract it given $d+1$ predictions. In all cases, we achieve $R_{\text{test}} = R_{\text{unif}} = 0$. If we compare the probabilities output by $f$ and $\hat{f}$, $R^{\text{TV}}_{\text{test}}$ and $R^{\text{TV}}_{\text{unif}}$ are lower than $10^{-9}$. For these models, the attack requires only 41 queries on average, and 113 at most. On Google's platform for example, an extraction attack would cost less than \$0.10, and subvert any further model monetization.

#### 4.1.2 Multiclass LRs and Multilayer Perceptrons

We now show that such equation-solving attacks broadly extend to all model classes with a 'logistic' layer, including multiclass ($c > 2$) LR and deeper neural networks. We define these models formally in Appendix A.

A multiclass logistic regression (MLR) combines $c$ binary models, each with parameters $\mathbf{w}_i, \beta_i$, to form a multiclass model. MLRs are available in all ML services we reviewed. We consider two types of MLR models: softmax and one-vs-rest (OvR), that differ in how the $c$ binary models are trained and combined: A softmax model fits a joint multinomial distribution to all training samples, while a OvR model trains a separate binary LR for each class, and then normalizes the class probabilities.

A MLR model $f$ is defined by parameters $\mathbf{w} \in \mathbb{R}^{cd}$, $\boldsymbol{\beta} \in \mathbb{R}^c$. Each sample $(\mathbf{x}, f(\mathbf{x}))$ gives $c$ equations in $\mathbf{w}$ and $\boldsymbol{\beta}$. The equation system is non-linear however, and has no analytic solution. For softmax models for instance, the equations take the form $e^{\mathbf{w}_i \cdot \mathbf{x} + \beta_i} / (\sum_{j=0}^{c-1} e^{\mathbf{w}_j \cdot \mathbf{x} + \beta_j}) = f_i(\mathbf{x})$. A common method for solving such a system is by minimizing an appropriate loss function, such as the logistic loss. With a regularization term, the loss function is *strongly convex*, and the optimization thus converges to a *global minimum* (i.e., a function $\hat{f}$ that predicts the same probabilities as $f$ for all available samples). A similar optimization (over class labels rather than probabilities) is actually used for training logistic models. Any MLR implementation can thus easily be adapted for model extraction with equation-solving.

| Model | Unknowns | Queries | $1 - R_{\text{test}}$ | $1 - R_{\text{unif}}$ | Time (s) |
|---|---|---|---|---|---|
| Softmax | 530 | 265 | 99.96% | 99.75% | 2.6 |
| | | 530 | 100.00% | 100.00% | 3.1 |
| OvR | 530 | 265 | 99.98% | 99.98% | 2.8 |
| | | 530 | 100.00% | 100.00% | 3.5 |
| MLP | 2,225 | 1,112 | 98.17% | 94.32% | 155 |
| | | 2,225 | 98.68% | 97.23% | 168 |
| | | 4,450 | 99.89% | 99.82% | 195 |
| | | 11,125 | 99.96% | 99.99% | 89 |

Table 4: **Success of equation-solving attacks.** Models to extract were trained on the Adult data set with multiclass target 'Race'. For each model, we report the number of unknown model parameters, the number of queries used, and the running time of the equation solver. The attack on the MLP with 11,125 queries converged after 490 epochs.

This approach naturally extends to deeper neural networks. We consider multilayer perceptrons (MLP), that first apply a non-linear transform to all inputs (the hidden layer), followed by a softmax regression in the transformed space. MLPs are becoming increasingly popular due to the continued success of deep learning methods; the advent of cloud-based ML services is likely to further boost their adoption. For our attacks, MLPs and MLRs mainly differ in the number of unknowns in the system to solve. For perceptrons with one hidden layer, we have $\mathbf{w} \in \mathbb{R}^{dh+hc}$, $\boldsymbol{\beta} \in \mathbb{R}^{h+c}$, where $h$ is the number of hidden nodes ($h = 20$ in our experiments). Another difference is that the loss function for MLPs is not strongly convex. The optimization may thus converge to a local minimum, i.e., a model $\hat{f}$ that does not exactly match $f$'s behavior.

To illustrate our attack's success, we train a softmax regression, a OvR regression and a MLP on the Adult data set with target 'Race' ($c = 5$). For the non-linear equation systems we obtain, we do not know a priori how many samples we need to find a solution (in contrast to linear systems where $d+1$ samples are necessary and sufficient). We thus explore various query budgets of the form $\alpha \cdot k$, where $k$ is the number of unknown model parameters, and $\alpha$ is a budget scaling factor. For MLRs, we solve the equation system with BFGS [41] in `scikit` [42]. For MLPs, we use `theano` [51] to run stochastic gradient descent for 1,000 epochs. Our experiments were performed on a commodity laptop (2-core Intel CPU @3.1GHz, 16GB RAM, no GPU acceleration).

Table 4 shows the extraction success for each model, as we vary $\alpha$ from 0.5 to at most 5. For MLR models (softmax and OvR), the attack is extremely efficient, requiring around one query per unknown parameter of $f$ (each query yields $c = 5$ equations). For MLPs, the system to solve is more complex, with about 4 times more

(a) (b)

Figure 2: **Training data leakage in KLR models.** (a) Displays 5 of 20 training samples used as representers in a KLR model (top) and 5 of 20 extracted representers (bottom). (b) For a second model, shows the average of all 1,257 representers that the model classifies as a $3,4,5,6$ or 7 (top) and 5 of 10 extracted representers (bottom).

unknowns. With a sufficiently over-determined system, we converge to a model $\hat{f}$ that very closely approximates $f$. As for LR models, queries are chosen non-adaptively, so $\mathcal{A}$ may submit a single 'batch request' to the API.

We further evaluated our attacks over all multiclass data sets from Table 3. For MLR models with $k = c \cdot (d+1)$ parameters ($c$ is the number of classes), $k$ queries were sufficient to achieve perfect extraction ($R_{\text{test}} = R_{\text{unif}} = 0$, $R^{\text{TV}}_{\text{test}}$ and $R^{\text{TV}}_{\text{unif}}$ below $10^{-7}$). We use 260 samples on average, and 650 for the largest model (Digits). For MLPs with 20 hidden nodes, we achieved >99.9% accuracy with 5,410 samples on average and 11,125 at most (Adult). With 54,100 queries on average, we extracted a $\hat{f}$ with 100% accuracy over tested inputs. As for binary LRs, we thus find that cross-user model extraction attacks for these model classes can be extremely efficient.

#### 4.1.3 Training Data Leakage for Kernel LR

We now move to a less mainstream model class, *kernel logistic regression* [57], that illustrates how extraction attacks can leak private training data, when a model's outputs are directly computed as a function of that data.

Kernel methods are commonly used to efficiently extend support vector machines (SVM) to nonlinear classifiers [14], but similar techniques can be applied to logistic regression [57]. Compared to kernel SVMs, kernel logistic regressions (KLR) have the advantage of computing class probabilities, and of naturally extending to multiclass problems. Yet, KLRs have not reached the popularity of kernel SVMs or standard LRs, and are not provided by any MLaaS provider at the time. We note that KLRs could easily be constructed in any ML library that supports both kernel functions and LR models.

A KLR model is a softmax model, where we replace the linear components $\mathbf{w}_i \cdot \mathbf{x} + \beta_i$ by a mapping $\sum_{r=1}^{s} \alpha_{i,r} K(\mathbf{x}, \mathbf{x}_r) + \beta_i$. Here, $K$ is a kernel function, and the *representers* $\mathbf{x}_1, \ldots, \mathbf{x}_s$ are a chosen subset of the training points [57]. More details are in Appendix A.

Each sample $(\mathbf{x}, f(\mathbf{x}))$ from a KLR model yields $c$ equations over the parameters $\boldsymbol{\alpha} \in \mathbb{R}^{sc}, \boldsymbol{\beta} \in \mathbb{R}^{c}$ and the representers $\mathbf{x}_1, \ldots, \mathbf{x}_s$. Thus, by querying the model, $\mathcal{A}$ obtains a non-linear equation system, the solution of which leaks training data. This assumes that $\mathcal{A}$ knows the exact number $s$ of representers sampled from the data. However, we can relax this assumption: First, note that $f$'s outputs are unchanged by adding 'extra' representers, with weights $\alpha = 0$. Thus, over-estimating $s$ still results in a consistent system of equations, of which a solution is the model $f$, augmented with unused representers. We will also show experimentally that training data may leak even if $\mathcal{A}$ extracts a model $\hat{f}$ with $s' \ll s$ representers.

We build two KLR models with a *radial-basis function* (RBF) kernel for a data set of handwritten digits. We select 20 random digits as representers for the first model, and all 1,257 training points for the second. We extract the first model, assuming knowledge of $s$, by solving a system of 50,000 equations in 1,490 unknowns. We use the same approach as for MLPs, i.e., logistic-loss minimization using gradient descent. We initialize the extracted representers to uniformly random vectors in $\mathcal{X}$, as we assume $\mathcal{A}$ does not know the training data distribution. In Figure 2a, we plot 5 of the model's representers from the training data, and the 5 closest (in $l_1$ norm) extracted representers. The attack clearly leaks information on individual training points. We measure the attack's robustness to uncertainty about $s$, by attacking the second model with only 10 local representers (10,000 equations in 750 unknowns). Figure 2b shows the *average* image of training points classified as a $3,4,5,6$ or 7 by the target model $f$, along with 5 extracted representers of $\hat{f}$. Surprisingly maybe, the attack seems to be leaking the 'average representor' of each class in the training data.

#### 4.1.4 Model Inversion Attacks on Extracted Models

Access to a model may enable inference of privacy-damaging information, particularly about the training set [4, 23, 24]. The *model inversion attack* explored by Fredrikson et al. [23] uses access to a classifier $f$ to find the input $\mathbf{x}_{\text{opt}}$ that maximizes the class probability for class $i$, i.e., $\mathbf{x}_{\text{opt}} = \text{argmax}_{\mathbf{x} \in \mathcal{X}} f_i(\mathbf{x})$. This was shown to allow recovery of recognizable images of training set members' faces when $f$ is a facial recognition model.

Their attacks work best in a *white-box* setting, where the attacker knows $f$ and its parameters. Yet, the authors also note that in a black-box setting, remote queries to a prediction API, combined with numerical approximation techniques, enable successful, albeit much less efficient, attacks. Furthermore, their black-box attacks inherently require $f$ to be queried *adaptively*. They leave as an open question making black-box attacks more efficient.

We explore composing an attack that first attempts to *extract* a model $\hat{f} \approx f$, and then uses it with the [23] white-box inversion attack. Our extraction techniques replace adaptive queries with a non-adaptive "batch" query to $f$, followed by local computation. We show that extraction plus inversion can require fewer queries and less time than performing black-box inversion directly.

As a case study, we use the softmax model from [23], trained over the AT&T Faces data [5]. The data set consists of images of faces ($92 \times 112$ pixels) of 40 people. The black-box attack from [23] needs about 20,600 queries to reconstruct a recognizable face for a single training set individual. Reconstructing the faces of all 40 individuals would require around 800,000 online queries.

The trained softmax model is much larger than those considered in Section 4.1, with 412,160 unknowns ($d =$ 10,304 and $c = 40$). We solve an under-determined system with 41,216 equations (using gradient descent with 200 epochs), and recover a model $\hat{f}$ achieving $R^{\text{TV}}_{\text{test}}, R^{\text{TV}}_{\text{unif}}$ in the order of $10^{-3}$. Note that the number of model parameters to extract is linear in the number of people $c$, whose faces we hope to recover. By using $\hat{f}$ in white-box model inversion attacks, we obtain results that are visually indistinguishable from the ones obtained using the true $f$. Given the extracted model $\hat{f}$, we can recover all 40 faces using white-box attacks, incurring around $20\times$ fewer remote queries to $f$ than with 40 black-box attacks.

For black-box attacks, the authors of [23] estimate a query latency of 70 milliseconds (a little less than in our own measurements of ML services, see Table 1). Thus, it takes 24 minutes to recover a single face (the inversion attack runs in seconds), and 16 hours to recover all 40 images. In contrast, solving the large equation system underlying our model-extraction attack took 10 hours. The 41,216 online queries would take under one hour if executed sequentially and even less with a batch query. The cost of the 40 local white-box attacks is negligible.

Thus, if the goal is to reconstruct faces for all 40 training individuals, performing model inversion over a previously extracted model results in an attack that is both faster and requires $20\times$ fewer online queries.

## 4.2 Decision Tree Path-Finding Attacks

Contrary to logistic models, decision trees do not compute class probabilities as a continuous function of their input. Rather, decision trees partition the input space into discrete regions, each of which is assigned a label and confidence score. We propose a new *path-finding* attack, that exploits API particularities to extract the 'decisions' taken by a tree when classifying an input.

Prior work on decision tree extraction [7, 12, 33] has focused on trees with Boolean features and outputs. While of theoretical importance, such trees have limited practical use. Kushilevitz and Mansour [33] showed that Boolean trees can be extracted using membership queries (arbitrary queries for class labels), but their algorithm does not extend to more general trees. Here, we propose attacks that exploit ML API specificities, and that apply to decision tree models used in MLaaS platforms.

Our tree model, defined formally in Appendix A, allows for binary and multi-ary splits over categorical features, and binary splits over numeric features. Each leaf of the tree is labeled with a class label and a confidence score. We note that our attacks also apply (often with better results) to *regression trees*. In regression trees, each leaf is labeled with a real-valued output and confidence.

The key idea behind our attack is to use the rich information provided by APIs on a prediction query, as a *pseudo-identifier* for the *path* that the input traversed in the tree. By varying the value of each input feature, we then find the predicates to be satisfied, for an input to follow a given path in the tree. We will also exploit the ability to query *incomplete inputs*, in which each feature $x_i$ is chosen from a space $\mathcal{X}_i \cup \{\perp\}$, where $\perp$ encodes the absence of a value. One way of handling such inputs ([11, 46]) is to label each node in the tree with an output value. On an input, we traverse the tree until we reach a leaf or an internal node with a split over a missing feature, and output that value of that leaf or node.

We formalize these notions by defining *oracles* that $\mathcal{A}$ can query to obtain an identifier for the leaf or internal node reached by an input. In practice, we instantiate these oracles using prediction API peculiarities.

**Definition 1** (Identity Oracles)**.** *Let each node $v$ of a tree $T$ be assigned some identifier $\texttt{id}_v$. A leaf-identity oracle $\mathcal{O}$ takes as input a query $\mathbf{x} \in \mathcal{X}$ and returns the identifier of the leaf of the tree $T$ that is reached on input $\mathbf{x}$.*

*A node-identity oracle $\mathcal{O}_\perp$ takes as input a query $\mathbf{x} \in \mathcal{X}_1 \cup \{\perp\} \times \cdots \times \mathcal{X}_d \cup \{\perp\}$ and returns the identifier of the node or leaf of $T$ at which the tree computation halts.*

### 4.2.1 Extraction Algorithms

We now present our path-finding attack (Algorithm 1), that assumes a leaf-identity oracle that returns *unique* identifiers for each leaf. We will relax the uniqueness assumption further on. The attack starts with a random input $\mathbf{x}$ and gets the leaf `id` from the oracle. We then search for all constraints on $\mathbf{x}$ that have to be satisfied to remain in that leaf, using procedures `LINE_SEARCH` (for continuous features) and `CAT_SPLIT` (for categorical features) described below. From this information, we then create new queries for unvisited leaves. Once all leaves have been found, the algorithm returns, for each leaf, the corresponding constraints on $\mathbf{x}$. We analyze the algorithm's correctness and complexity in Appendix C.

We illustrate our algorithm with a toy example of a tree over continuous feature *Size* and categorical feature *Color* (see Figure 3). The current query is $\mathbf{x} = \{Size = 50,\ Color = R\}$ and $\mathcal{O}(\mathbf{x}) = \texttt{id}_2$. Our goal is two-fold: (1) Find the *predicates* that $\mathbf{x}$ has to satisfy to end up in leaf $\texttt{id}_2$ (i.e., $Size \in (40, 60]$, $Color = R$), and (2) create new inputs $\mathbf{x}'$ to explore other paths in the tree.

**Algorithm 1** **The path-finding algorithm.** The notation $\texttt{id} \leftarrow \mathcal{O}(\mathbf{x})$ means querying the leaf-identity oracle $\mathcal{O}$ with an input $\mathbf{x}$ and obtaining a response id. By $\mathbf{x}[i] \Rightarrow v$ we denote the query $\mathbf{x}'$ obtained from $\mathbf{x}$ by replacing the value of $x_i$ by $v$.

```
1: x_init ← {x_1, ..., x_d}                          ▷ random initial query
2: Q ← {x_init}                                      ▷ Set of unprocessed queries
3: P ← {}                                            ▷ Set of explored leaves with their predicates
4: while Q not empty do
5:     x ← Q.POP()
6:     id ← O(x)                                     ▷ Call to the leaf identity oracle
7:     if id ∈ P then                                ▷ Check if leaf already visited
8:         continue
9:     end if
10:    for 1 ≤ i ≤ d do                              ▷ Test all features
11:        if IS_CONTINUOUS(i) then
12:            for (α, β] ∈ LINE_SEARCH(x, i, ε) do
13:                if x_i ∈ (α, β] then
14:                    P[id].ADD('x_i ∈ (α, β]')     ▷ Current interval
15:                else
16:                    Q.PUSH(x[i] ⇒ β)              ▷ New leaf to visit
17:                end if
18:            end for
19:        else
20:            S, V ← CATEGORY_SPLIT(x, i, id)
21:            P[id].ADD('x_i ∈ S')                  ▷ Values for current leaf
22:            for v ∈ V do
23:                Q.PUSH(x[i] ⇒ v)                  ▷ New leaves to visit
24:            end for
25:        end if
26:    end for
27: end while
```

The LINE_SEARCH procedure (line 12) tests *continuous features*. We start from bounds on the range of a feature $\mathcal{X}_i = [a,b]$. In our example, we have $Size \in [0,100]$. We set the value of *Size* in $\mathbf{x}$ to 0 and 100, query $\mathcal{O}$, and obtain $\texttt{id}_1$ and $\texttt{id}_5$. As the ids do not match, a split on *Size* occurs on the path to $\texttt{id}_2$. With a binary search over feature *Size* (and all other features in $\mathbf{x}$ fixed), we find all intervals that lead to different leaves, i.e., $[0,40]$, $(40,60]$, $(60,100]$. From these intervals, we find the predicate for the current leaf (i.e., $Size \in (40,60]$) and build queries to explore new tree paths. To ensure termination of the line search, we specify some *precision* $\varepsilon$. If a split is on a threshold $t$, we find the value $\tilde{t}$ that is the unique multiple of $\varepsilon$ in the range $(t-\varepsilon, t]$. For values $x_i$ with granularity $\varepsilon$, splitting on $\tilde{t}$ is then equivalent to splitting on $t$.

The CATEGORY_SPLIT procedure (line 20) finds splits on *categorical features*. In our example, we vary the value of *Color* in $\mathbf{x}$ and query $\mathcal{O}$ to get a leaf id for each value. We then build a set $S$ of values that lead to the current leaf, i.e., $S = \{\texttt{R}\}$, and a set $V$ of values to set in $\mathbf{x}$ to explore other leaves (one representative per leaf). In our example, we could have $V = \{\texttt{B},\texttt{G},\texttt{Y}\}$ or $V = \{\texttt{B},\texttt{G},\texttt{O}\}$.

Using these two procedures, we thus find the predicates defining the path to leaf $\texttt{id}_2$, and generate new queries $\mathbf{x}'$ for unvisited leaves of the tree.

**A top-down approach.** We propose an empirically more efficient *top-down* algorithm that exploits queries over partial inputs. It extracts the tree 'layer by layer', starting at the root: We start with an empty query (all features set to $\perp$) and get the root's id by querying $\mathcal{O}_\perp$. We then set each feature in turn and query $\mathcal{O}$ again. For exactly one feature (the root's splitting feature), the input will reach a different node. With similar procedures as described previously, we extract the root's splitting criterion, and recursively search lower layers of the tree.

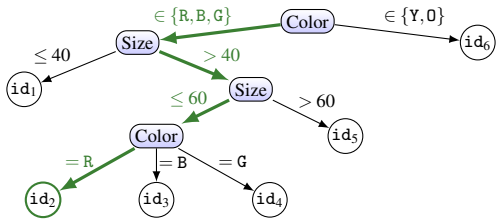


Figure 3: **Decision tree over features Color and Size.** Shows the path (thick green) to leaf $\texttt{id}_2$ on input $\mathbf{x} = \{Size = 50,\ Color = R\}$.

| Data set | # records | # classes | # features |
|---|---|---|---|
| IRS Tax Patterns | 191,283 | 51 | 31 |
| Steak Survey | 430 | 5 | 12 |
| GSS Survey | 51,020 | 3 | 7 |
| Email Importance | 4,709 | 2 | 14 |
| Email Spam | 4,601 | 2 | 46 |
| German Credit | 1,000 | 2 | 11 |
| Medical Cover | 163,065 | $\mathcal{Y} = \mathbb{R}$ | 13 |
| Bitcoin Price | 1,076 | $\mathcal{Y} = \mathbb{R}$ | 7 |

Table 5: **Data sets used for decision tree extraction.** Trained trees for these data sets are available in BigML's public gallery. The last two data sets are used to train regression trees.

**Duplicate identities.** As we verify empirically, our attacks are resilient to some nodes or leaves sharing the same id. We can modify line 7 in Algorithm 1 to detect id duplicates, by checking not only whether a leaf with the current id was already visited, but also whether the current query violates that leaf's predicates. The main issue with duplicate ids comes from the LINE_SEARCH and CATEGORY_SPLIT procedures: if two queries $\mathbf{x}$ and $\mathbf{x}'$ differ in a single feature and reach different leaves with the same id, the split on that feature will be missed.

### 4.2.2 Attack Evaluation

Our tree model (see Appendix A) is the one used by BigML. Other ML services use similar tree models. For our experiments, we downloaded eight public decision trees from BigML (see Table 5), and queried them locally using available API bindings. More details on these models are in Appendix B. We show *online* extraction attacks on black-box models from BigML in Section 5.

To emulate black-box model access, we first issue online queries to BigML, to determine the information contained in the service's responses. We then simulate black-box access locally, by discarding any extra information returned by the local API. Specifically, we make use of the following fields in query responses:

| Model | Leaves | Unique IDs | Depth | Without incomplete queries | | | With incomplete queries | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $1-R_{\text{test}}$ | $1-R_{\text{unif}}$ | Queries | $1-R_{\text{test}}$ | $1-R_{\text{unif}}$ | Queries |
| IRS Tax Patterns | 318 | 318 | 8 | 100.00% | 100.00% | 101,057 | 100.00% | 100.00% | 29,609 |
| Steak Survey | 193 | 28 | 17 | 92.45% | 86.40% | 3,652 | 100.00% | 100.00% | 4,013 |
| GSS Survey | 159 | 113 | 8 | 99.98% | 99.61% | 7,434 | 100.00% | 99.65% | 2,752 |
| Email Importance | 109 | 55 | 17 | 99.13% | 99.90% | 12,888 | 99.81% | 99.99% | 4,081 |
| Email Spam | 219 | 78 | 29 | 87.20% | 100.00% | 42,324 | 99.70% | 100.00% | 21,808 |
| German Credit | 26 | 25 | 11 | 100.00% | 100.00% | 1,722 | 100.00% | 100.00% | 1,150 |
| Medical Cover | 49 | 49 | 11 | 100.00% | 100.00% | 5,966 | 100.00% | 100.00% | 1,788 |
| Bitcoin Price | 155 | 155 | 9 | 100.00% | 100.00% | 31,956 | 100.00% | 100.00% | 7,390 |

Table 6: **Performance of extraction attacks on public models from BigML.** For each model, we report the number of leaves in the tree, the number of unique identifiers for those leaves, and the maximal tree depth. The chosen granularity $\varepsilon$ for continuous features is $10^{-3}$.

- **Prediction.** This entry contains the predicted class label (classification) or real-valued output (regression).
- **Confidence.** For classification and regression trees, BigML computes confidence scores based on a confidence interval for predictions at each node [11]. The prediction and confidence value constitute a node's `id`.
- **Fields.** Responses to black-box queries contain a 'fields' property, that lists all features that appear either in the input query or on the path traversed in the tree. If a partial query $\mathbf{x}$ reaches an internal node $v$, this entry tells us which feature $v$ splits on (the feature is in the 'fields' entry, but not in the input $\mathbf{x}$). We make use of this property for the top-down attack variant.

Table 6 displays the results of our attacks. For each tree, we give its number of leaves, the number of unique leaf `ids`, and the tree depth. We display the success rate for Algorithm 1 and for the "top-down" variant with incomplete queries. Querying partial inputs vastly improves our attack: we require far less queries (except for the Steak Survey model, where Algorithm 1 only visits a fraction of all leaves and thus achieves low success) and achieve higher accuracy for trees with duplicate leaf `ids`. As expected, both attacks achieve perfect extraction when all leaves have unique `ids`. While this is not always the case for classification trees, it is far more likely for regression trees, where both the label and confidence score take real values. Surprisingly maybe, the top-down approach also fully extracts some trees with a large number of duplicate leaf `ids`. The attacks are also efficient: The top-down approach takes less than 10 seconds to extract a tree, and Algorithm 1 takes less than 6 minutes for the largest tree. For online attacks on ML services, discussed next, this cost is trumped by the delay for the inherently adaptive prediction queries that are issued.

# 5 Online Model Extraction Attacks

In this section, we showcase *online* model extraction attacks against two ML services: BigML and Amazon. For BigML, we focus on extracting models set up by a user, who wishes to charge for predictions. For Amazon, our goal is to extract a model trained by ourselves, to which we only get black-box access. Our attacks only use exposed APIs, and do not in any way attempt to bypass the services' authentication or access-control mechanisms. We only attack models trained in our own accounts.

| Model | OHE | Binning | Queries | Time (s) | Price ($) |
|---|---|---|---|---|---|
| Circles | - | Yes | 278 | 28 | 0.03 |
| Digits | - | No | 650 | 70 | 0.07 |
| Iris | - | Yes | 644 | 68 | 0.07 |
| Adult | Yes | Yes | 1,485 | 149 | 0.15 |

Table 7: **Results of model extraction attacks on Amazon.** OHE stands for one-hot-encoding. The reported query count is the number used to find quantile bins (at a granularity of $10^{-3}$), plus those queries used for equation-solving. Amazon charges $0.0001 per prediction [1].

## 5.1 Case Study 1: BigML

BigML currently only allows monetization of decision trees [11]. We train a tree on the *German Credit* data, and set it up as a black-box model. The tree has 26 leaves, two of which share the same label and confidence score. From another account, we extract the model using the two attacks from Section 4.2. We first find the tree's number of features, their type and their range, from BigML's public gallery. Our attacks (Algorithm 1 and the top-down variant) extract an exact description of the tree's paths, using respectively 1,722 and 1,150 queries. Both attacks' duration (1,030 seconds and 631 seconds) is dominated by query latency ($\approx 500\text{ms}/\text{query}$). The monetary cost of the attack depends on the per-prediction-fee set by the model owner. In any case, a user who wishes to make more than 1,150 predictions has economic incentives to run an extraction attack.

## 5.2 Case Study 2: Amazon Web Services

Amazon uses logistic regression for classification, and provides black-box-only access to trained models [1]. By default, Amazon uses two feature extraction techniques: (1) Categorical features are *one-hot-encoded*, i.e., the input space $\mathcal{M}_i = \mathbb{Z}_k$ is mapped to $k$ binary features encoding the input value. (2) *Quantile binning* is used for numeric features. The training data values are split into $k$-quantiles ($k$ equally-sized bins), and the input space $\mathcal{M}_i = [a,b]$ is mapped to $k$ binary features encoding the bin that a value falls into. Note that $|\mathcal{X}| > |\mathcal{M}|$,

i.e., ex increases the number of features. If $\mathcal{A}$ reverse-engineers ex, she can query the service on samples $M$ in input space, compute $\mathbf{x} = \mathsf{ex}(M)$ locally, and extract $f$ in feature-space using equation-solving.

We apply this approach to models trained by Amazon. Our results are summarized in Table 7. We first train a model with no categorical features, and quantile binning disabled (this is a manually tunable parameter), over the Digits data set. The attack is then identical to the one considered in Section 4.1.2: using 650 queries to Amazon, we extract a model that achieves $R_{\text{test}} = R_{\text{unif}} = 0$.

We now consider models with feature extraction enabled. We assume that $\mathcal{A}$ knows the input space $\mathcal{M}$, but not the training data distribution. For one-hot-encoding, knowledge of $\mathcal{M}$ suffices to apply the same encoding locally. For quantile binning however, applying ex locally requires knowledge of the training data quantiles. To reverse-engineer the binning transformation, we use line-searches similar to those we used for decision trees: For each numeric feature, we search the feature's range in input space for thresholds (up to a granularity $\varepsilon$) where $f$'s output changes. This indicates our value landed in an adjacent bin, with a different learned regression coefficient. Note that learning the bin boundaries may be interesting in its own right, as it leaks information about the training data distribution. Having found the bin boundaries, we can apply both one-hot-encoding and binning locally, and extract $f$ over its feature space. As we are restricted to queries over $\mathcal{M}$, we cannot define an arbitrary system of equations over $\mathcal{X}$. Building a well-determined and consistent system can be difficult, as the encoding ex generates sparse inputs over $\mathcal{X}$. However, Amazon facilitates this process with the way it handles queries with *missing features*: if a feature is omitted from a query, all corresponding features in $\mathcal{X}$ are set to 0. For a linear model for instance, we can trivially re-construct the model by issuing queries with a single feature specified, such as to obtain equations with a single unknown in $\mathcal{X}$.

We trained models for the Circles, Iris and Adult data sets, with Amazon's default feature-extraction settings. Table 7 shows the results of our attacks, for the reverse-engineering of ex and extraction of $f$. For binary models (Circles and Adult), we use $d+1$ queries to solve a linear equation-system over $\mathcal{X}$. For models with $c > 2$ classes, we use $c \cdot (d+1)$ queries. In all cases, the extracted model matches $f$ on 100% of tested inputs. To optimize the query complexity, the queries we use to find quantile bins are re-used for equation-solving. As line searches require adaptive queries, we do not use batch predictions. However, even for the Digits model, we resorted to using real-time predictions, because of the service's significant overhead in evaluating batches. For attacks that require a large number of non-adaptive queries, we expect batch predictions to be faster than real-time predictions.

## 5.3 Discussion

**Additional feature extractors.** In some ML services we considered, users may enable further feature extractors. A common transformation is feature scaling or normalization. If $\mathcal{A}$ has access to training data statistics (as provided by BigML for instance), applying the transformation locally is trivial. More generally, for models with a linear input layer (i.e., logistic regressions, linear SVMs, MLPs) the scaling or normalization can be seen as being applied to the learned weights, rather than the input features. We can thus view the composition $f \circ \mathsf{ex}$ as a model $f'$ that operates over the 'un-scaled' input space $\mathcal{M}$ and extract $f'$ directly using equation-solving.

Further extractors include text analysis (e.g., bag-of-words or n-gram models) and Cartesian products (grouping many features into one). We have not analyzed these in this work, but we believe that they could also be easily reverse-engineered, especially given some training data statistics and the ability to make incomplete queries.

**Learning unknown model classes or hyper-parameters.** For our online attacks, we obtained information about the model class of $f$, the enabled feature extraction ex, and other hyper-parameters, directly from the ML service or its documentation. More generally, if $\mathcal{A}$ does not have full certainty about certain model characteristics, it may be able to narrow down a guess to a small range. Model hyper-parameters for instance (such as the free parameter of an RBF kernel) are typically chosen through cross-validation over a default range of values.

Given a set of attack strategies with varying assumptions, $\mathcal{A}$ can use a generic *extract-and-test* approach: each attack is applied in turn, and evaluated by computing $R_{\text{test}}$ or $R_{\text{unif}}$ over a chosen set of points. The adversary succeeds if any of the strategies achieves a low error. Note that $\mathcal{A}$ needs to interact with the model $f$ only once, to obtain responses for a chosen set of extraction samples and test samples, that can be re-used for each strategy.

Our attacks on Amazon's service followed this approach: We first formulated guesses for model characteristics left unspecified by the documentation (e.g., we found no mention of one-hot-encoding, or of how missing inputs are handled). We then evaluated our assumptions with successive extraction attempts. Our results indicate that Amazon uses softmax regression and does not create binary predictors for missing values. Interestingly, BigML takes the 'opposite' approach (i.e., BigML uses OvR regression and adds predictors for missing values).

# 6 Extraction Given Class Labels Only

The successful attacks given in Sections 4 and 5 show the danger of revealing confidence values. While current

ML services have been designed to reveal rich information, our attacks may suggest that returning only labels would be safer. Here we explore model extraction in a setting with no confidence scores. We will discuss further countermeasures in Section 7. We primarily focus on settings where $\mathcal{A}$ can make *direct* queries to an API, i.e., queries for arbitrary inputs $\mathbf{x} \in \mathcal{X}$. We briefly discuss *indirect* queries in the context of linear classifiers.

**The Lowd-Meek attack.** We start with the prior work of Lowd and Meek [36]. They present an attack on any linear classifier, assuming black-box oracle access with membership queries that return just the predicted class label. A linear classifier is defined by a vector $\mathbf{w} \in \mathbb{R}^d$ and a constant $\beta \in \mathbb{R}$, and classifies an instance $\mathbf{x}$ as *positive* if $\mathbf{w} \cdot \mathbf{x} + \beta > 0$ and *negative* otherwise. SVMs with linear kernels and binary LRs are examples of linear classifiers. Their attack uses line searches to find points arbitrarily close to $f$'s decision boundary (points for which $\mathbf{w} \cdot \mathbf{x} + \beta \approx 0$), and extracts $\mathbf{w}$ and $\beta$ from these samples.

This attack only works for linear binary models. We describe a straightforward extension to some non-linear models, such as polynomial kernel SVMs. Extracting a polynomial kernel SVM can be reduced to extracting a linear SVM in the transformed feature space. Indeed, for any kernel $K_{\text{poly}}(\mathbf{x},\mathbf{x}') = (\mathbf{x}^T \cdot \mathbf{x}' + 1)^d$, we can derive a projection function $\phi(\cdot)$, so that $K_{\text{poly}}(\mathbf{x},\mathbf{x}') = \phi(\mathbf{x})^T \cdot \phi(\mathbf{x}')$. This transforms the kernel SVM into a linear one, since the decision boundary now becomes $\mathbf{w}^F \cdot \phi(\mathbf{x}) + \beta = 0$ where $\mathbf{w}^F = \sum_{i=1}^{t} \alpha_i \phi(\mathbf{x}_i)$. We can use the Lowd-Meek attack to extract $\mathbf{w}^F$ and $\beta$ as long as $\phi(\mathbf{x})$ and its inverse are feasible to compute; this is unfortunately not the case for the more common RBF kernels.[3]

**The retraining approach.** In addition to evaluating the Lowd-Meek attack against ML APIs, we introduce a number of other approaches based on the broad strategy of re-training a model locally, given input-output examples. Informally, our hope is that by extracting a model that achieves low *training error* over the queried samples, we would effectively approximate the target model's decision boundaries. We consider three retraining strategies, described below. We apply these to the model classes that we previously extracted using equation-solving attacks, as well as to SVMs.[4]

(1) **Retraining with uniform queries.** This baseline strategy simply consists in sampling $m$ points $\mathbf{x}_i \in \mathcal{X}$ uniformly at random, querying the oracle, and training a model $\hat{f}$ on these samples.

[3] We did explore using approximations of $\phi$, but found that the adaptive re-training techniques discussed in this section perform better.

[4] We do not expect retraining attacks to work well for decision trees, because of the greedy approach taken by learning algorithms. We have not evaluated extraction of trees, given class labels only, in this work.

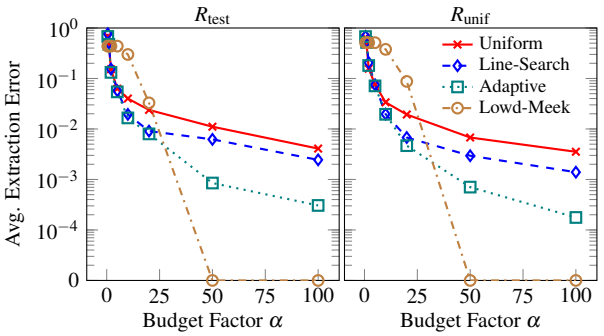


Figure 4: **Average error of extracted linear models.** Results are for different extraction strategies applied to models trained on all binary data sets from Table 3. The left shows $R_{\text{test}}$ and the right shows $R_{\text{unif}}$.

(2) **Line-search retraining.** This strategy can be seen as a model-agnostic generalization of the Lowd-Meek attack. It issues $m$ adaptive queries to the oracle using line search techniques, to find samples close to the decision boundaries of $f$. A model $\hat{f}$ is then trained on the $m$ queried samples.

(3) **Adaptive retraining.** This strategy applies techniques from active learning [18, 47]. For some number $r$ of rounds and a query budget $m$, it first queries the oracle on $\frac{m}{r}$ uniform points, and trains a model $\hat{f}$. Over a total of $r$ rounds, it then selects $\frac{m}{r}$ new points, along the decision boundary of $\hat{f}$ (intuitively, these are points $\hat{f}$ is *least certain* about), and sends those to the oracle before retraining $\hat{f}$.

## 6.1 Linear Binary Models

We first explore how well the various approaches work in settings where the Lowd-Meek attack can be applied. We evaluate their attack and our three retraining strategies for logistic regression models trained over the binary data sets shown in Table 3. These models have $d+1$ parameters, and we vary the query budget as $\alpha \cdot (d+1)$, for $0.5 \leq \alpha \leq 100$. Figure 4 displays the average errors $R_{\text{test}}$ and $R_{\text{unif}}$ over all models, as a function of $\alpha$.

The retraining strategies that search for points near the decision boundary clearly perform better than simple uniform retraining. The adaptive strategy is the most efficient of our three strategies. For relatively low budgets, it even outperforms the Lowd-Meek attack. However, for budgets large enough to run line searches in each dimension, the Lowd-Meek attack is clearly the most efficient.

For the models we trained, about 2,050 queries on average, and 5,650 at most, are needed to run the Lowd-Meek attack effectively. This is 50× more queries than what we needed for equation-solving attacks. With 827 queries on average, adaptive retraining yields a model $\hat{f}$ that matches $f$ on over 99% of tested inputs. Thus, even if an ML API only provides class labels, efficient extrac-

tion attacks on linear models remain possible.

We further consider a setting where feature-extraction (specifically one-hot-encoding of categorical features) is applied by the ML service, rather than by the user. $\mathcal{A}$ is then limited to indirect queries in input space. Lowd and Meek [36] note that their extraction attack does not work in this setting, as $\mathcal{A}$ can not run line searches directly over $\mathcal{X}$. In contrast, for the linear models we trained, we observed no major difference in extraction accuracy for the adaptive-retraining strategy, when limited to queries over $\mathcal{M}$. We leave an in-depth study of model extraction with indirect queries, and class labels only, for future work.

## 6.2 Multiclass LR Models

The Lowd-Meek attack is not applicable in multiclass ($c > 2$) settings, even when the decision boundary is a combination of linear boundaries (as in multiclass regression) [39, 50]. We thus focus on evaluating the three retraining attacks we introduced, for the type of ML models we expect to find in real-world applications.

We focus on softmax models here, as softmax and one-vs-rest models have identical output behaviors when only class labels are provided: in both cases, the class label for an input $\mathbf{x}$ is given by $\text{argmax}_i(\mathbf{w}_i \cdot \mathbf{x} + \beta_i)$. From an extractor's perspective, it is thus irrelevant whether the target was trained using a softmax or OvR approach.

We evaluate our attacks on softmax models trained on the multiclass data sets shown in Table 3. We again vary the query budget as a factor $\alpha$ of the number of model parameters, namely $\alpha \cdot c \cdot (d+1)$. Results are displayed in Figure 5. We observe that the adaptive strategy clearly performs best and that the line-search strategy does not improve over uniform retraining, possibly because the line-searches have to be split across multiple decision-boundaries. We further note that all strategies achieve lower $R_{\text{test}}$ than $R_{\text{unif}}$. It thus appears that for the models we trained, points from the test set are on average 'far' from the decision boundaries of $f$ (i.e., the trained models separate the different classes with large margins).

For all models, $100 \cdot c \cdot (d+1)$ queries resulted in extraction accuracy above 99.9%. This represents 26,000 queries on average, and 65,000 at the most (Digits data set). Our equation-solving attacks achieved similar or better results with 100× less queries. Yet, for scenarios with high monetary incentives (e.g., intrusion detector evasion), extraction attacks on MLR models may be attractive, even if APIs only provide class labels.

## 6.3 Neural Networks

We now turn to attacks on more complex deep neural networks. We expect these to be harder to retrain than multiclass regressions, as deep networks have more parameters and non-linear decision-boundaries. Therefore, we may need to find a large number of points close to a decision boundary in order to extract it accurately.

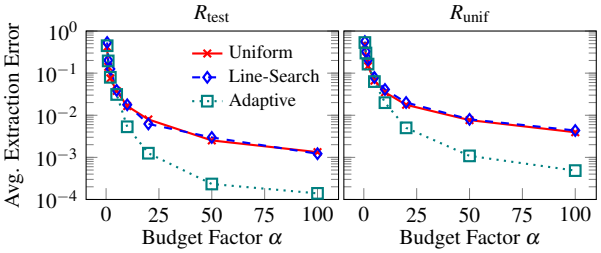


Figure 5: **Average error of extracted softmax models.** Results are for three retraining strategies applied to models trained on all multiclass data sets from Table 3. The left shows $R_{\text{test}}$ and the right shows $R_{\text{unif}}$.

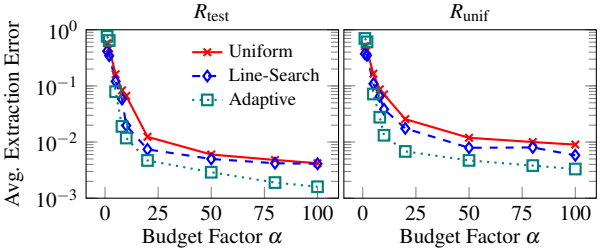


Figure 6: **Average error of extracted RBF kernel SVMs** Results are for three retraining strategies applied to models trained on all binary data sets from Table 3. The left shows $R_{\text{test}}$ and the right shows $R_{\text{unif}}$.

We evaluated our attacks on the multiclass models from Table 3. For the tested query budgets, line-search and adaptive retraining gave little benefit over uniform retraining. For a budget of $100 \cdot k$, where $k$ is the number of model parameters, we get $R_{\text{test}} = 99.16\%$ and $R_{\text{unif}} = 98.24\%$, using 108,200 queries per model on average. Our attacks might improve for higher budgets but it is unclear whether they would then provide any monetary advantage over using ML APIs in an honest way.

## 6.4 RBF Kernel SVMs

Another class of nonlinear models that we consider are support-vector machines (SVMs) with radial-basis function (RBF) kernels. A kernel SVM first maps inputs into a higher-dimensional space, and then finds the hyperplane that maximally separates the two classes. As mentioned in Section 6, SVMs with polynomial kernels can be extracted using the Lowd-Meek attack in the transformed feature space. For RBF kernels, this is not possible because the transformed space has infinite dimension.

SVMs do not provide class probability estimates. Our only applicable attack is thus retraining. As for linear models, we vary the query budget as $\alpha \cdot (d+1)$, where $d$ is the input dimension. We further use the *extract-and-test* approach from Section 5 to find the value of the RBF kernel's *hyper-parameter*. Results of our attacks are in

Figure 6. Again, we see that adaptive retraining performs best, even though the decision boundary to extract is non-linear (in input space) here. Kernel SVMs models are overall harder to retrain than models with linear decision boundaries. Yet, for our largest budgets (2,050 queries on average), we do extract models with over 99% accuracy, which may suffice in certain adversarial settings.

## 7 Extraction Countermeasures

We have shown in Sections 4 and 5 that adversarial clients can effectively extract ML models given access to rich prediction APIs. Given that this undermines the financial models targeted by some ML cloud services, and potentially leaks confidential training data, we believe researchers should seek countermeasures.

In Section 6, we analyzed the most obvious defense against our attacks: prediction API minimization. The constraint here is that the resulting API must still be useful in (honest) applications. For example, it is simple to change APIs to not return confidences and not respond to incomplete queries, assuming applications can get by without it. This will prevent many of our attacks, most notably the ones described in Section 4 as well as the feature discovery techniques used in our Amazon case study (Section 5). Yet, we showed that even if we strip an API to only provide class labels, successful attacks remain possible (Section 6), albeit at a much higher query cost.

We discuss further potential countermeasures below.

**Rounding confidences.** Applications might need confidences, but only at lower granularity. A possible defense is to round confidence scores to some fixed precision [23]. We note that ML APIs already work with some finite precision when answering queries. For instance, BigML reports confidences with 5 decimal places, and Amazon provides values with 16 significant digits.

To understand the effects of limiting precision further, we re-evaluate equation-solving and decision tree path-finding attacks with confidence scores rounded to a fixed decimal place. For equation-solving attacks, rounding the class probabilities means that the solution to the obtained equation-system might not be the target $f$, but some truncated version of it. For decision trees, rounding confidence scores increases the chance of node `id` collisions, and thus decreases our attacks' success rate.

Figure 7 shows the results of experiments on softmax models, with class probabilities rounded to 2–5 decimals. We plot only $R_{\text{test}}$, the results for $R_{\text{unif}}$ being similar. We observe that class probabilities rounded to 4 or 5 decimal places (as done already in BigML) have no effect on the attack's success. When rounding further to 3 and 2 decimal places, the attack is weakened, but still vastly outperforms adaptive retraining using class labels only.

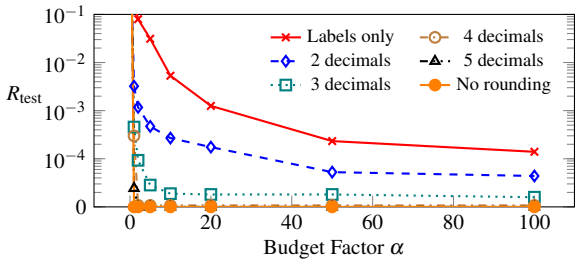


Figure 7: **Effect of rounding on model extraction.** Shows the average test error of equation-solving attacks on softmax models trained on the benchmark suite (Table 3), as we vary the number of significant digits in reported class probabilities. Extraction with no rounding and with class labels only (adaptive retraining) are added for comparison.

For regression trees, rounding has no effect on our attacks. Indeed, for the models we considered, the output itself is unique in each leaf (we could also round outputs, but the impact on utility may be more critical). For classification trees, we re-evaluated our top-down attack, with confidence scores rounded to fewer than 5 decimal places. The attacks on the 'IRS Tax Patterns' and 'Email Importance' models are the most resilient, and suffer no success degradation before scores are rounded to 2 decimal places. For the other models, rounding confidences to 3 or 4 decimal places severely undermines our attack.

**Differential privacy.** Differential privacy (DP) [22] and its variants [34] have been explored as mechanisms for protecting, in particular, the privacy of ML training data [54]. DP learning has been applied to regressions [17,56], SVMs [44], decision trees [31] and neural networks [48]. As some of our extraction attacks leak training data information (Section 4.1.3), one may ask whether DP can prevent extraction, or at least reduce the severity of the privacy violations that extraction enables.

Consider naïve application of DP to protect individual training data elements. This should, in theory, decrease the ability of an adversary $\mathcal{A}$ to learn information about training set elements, when given access to prediction queries. One would not expect, however, that this prevents model extraction, as DP is not defined to do so: consider a trivially useless learning algorithm for binary logistic regression, that discards the training data and sets $\mathbf{w}$ and $\beta$ to 0. This algorithm is differentially private, yet $\mathbf{w}$ and $\beta$ can easily be recovered using equation-solving.

A more appropriate strategy would be to apply DP directly to the model parameters, which would amount to saying that a query should not allow $\mathcal{A}$ to distinguish between closely neighboring model parameters. How exactly this would work and what privacy budgets would be required is left as an open question by our work.

**Ensemble methods.** Ensemble methods such as random forests return as prediction an aggregation of pre-

dictions by a number of individual models. While we have not experimented with ensemble methods as targets, we suspect that they may be more resilient to extraction attacks, in the sense that attackers will only be able to obtain relatively coarse approximations of the target function. Nevertheless, ensemble methods may still be vulnerable to other attacks such as model evasion [55].

## 8 Related Work

Our work is related to the extensive literature on learning theory, such as PAC learning [53] and its variants [3, 8]. Indeed, extraction can be viewed as a type of learning, in which an unknown instance of a known hypothesis class (model type) is providing labels (without error). This is often called learning with membership queries [3]. Our setting differs from these in two ways. The first is conceptual: in PAC learning one builds algorithms to learn a concept — the terminology belies the motivation of formalizing learning from data. In model extraction, an attacker is literally given a function oracle that it seeks to illicitly determine. The second difference is more pragmatic: prediction APIs reveal richer information than assumed in prior learning theory work, and we exploit that.

Algorithms for learning with membership queries have been proposed for Boolean functions [7, 15, 30, 33] and various binary classifiers [36, 39, 50]. The latter line of work, initiated by Lowd and Meek [36], studies strategies for model evasion, in the context of spam or fraud detectors [9, 29, 36, 37, 55]. Intuitively, model extraction seems harder than evasion, and this is corroborated by results from theory [36, 39, 50] and practice [36, 55].

Evasion attacks fall into the larger field of *adversarial machine learning*, that studies machine learning in general adversarial settings [6, 29]. In that context, a number of authors have considered strategies and defenses for *poisoning* attacks, that consist in injecting maliciously crafted samples into a model's train or test data, so as to decrease the learned model's accuracy [10, 21, 32, 40, 45].

In a non-malicious setting, improper model extraction techniques have been applied for interpreting [2, 19, 52] and compressing [16, 27] complex neural networks.

## 9 Conclusion

We demonstrated how the flexible prediction APIs exposed by current ML-as-a-service providers enable new model extraction attacks that could subvert model monetization, violate training-data privacy, and facilitate model evasion. Through local experiments and online attacks on two major providers, BigML and Amazon, we illustrated the efficiency and broad applicability of attacks that exploit common API features, such as the availability of confidence scores or the ability to query arbitrary partial inputs. We presented a generic *equation-solving* attack for models with a logistic output layer and a novel *path-finding* algorithm for decision trees.

We further explored potential countermeasures to these attacks, the most obvious being a restriction on the information provided by ML APIs. Building upon prior work from learning-theory, we showed how an attacker that only obtains class labels for adaptively chosen inputs, may launch less effective, yet potentially harmful, *retraining attacks*. Evaluating these attacks, as well as more refined countermeasures, on production-grade ML services is an interesting avenue for future work.

**Acknowledgments.** We thank Martín Abadi and the anonymous reviewers for their comments. This work was supported by NSF grants 1330599, 1330308, and 1546033, as well as a generous gift from Microsoft.

## References

[1] Amazon Web Services. `https://aws.amazon.com/machine-learning`. Accessed Feb. 10, 2016.

[2] Andrews, R., Diederich, J., and Tickle, A. Survey and critique of techniques for extracting rules from trained artificial neural networks. *KBS 8*, 6 (1995), 373–389.

[3] Angluin, D. Queries and concept learning. *Machine learning 2*, 4 (1988), 319–342.

[4] Ateniese, G., Mancini, L. V., Spognardi, A., Villani, A., Vitali, D., and Felici, G. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *IJSN 10*, 3 (2015), 137–150.

[5] AT&T Laboratories Cambridge. The ORL database of faces. `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`.

[6] Barreno, M., Nelson, B., Sears, R., Joseph, A. D., and Tygar, J. D. Can machine learning be secure? In *ASIACCS* (2006), ACM, pp. 16–25.

[7] Bellare, M. A technique for upper bounding the spectral norm with applications to learning. In *COLT* (1992), ACM, pp. 62–70.

[8] Benedek, G. M., and Itai, A. Learnability with respect to fixed distributions. *TCS 86*, 2 (1991), 377–389.

[9] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *ECML PKDD*. Springer, 2013, pp. 387–402.

[10] Biggio, B., Nelson, B., and Laskov, P. Poisoning attacks against support vector machines. In *ICML* (2012).

[11] BigML. `https://www.bigml.com`. Accessed Feb. 10, 2016.

[12] Blum, A. L., and Langley, P. Selection of relevant features and examples in machine learning. *Artificial intelligence 97*, 1 (1997), 245–271.

[13] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. Occam's razor. *Readings in machine learning* (1990), 201–204.

[14] Boser, B. E., Guyon, I. M., and Vapnik, V. N. A training algorithm for optimal margin classifiers. In *COLT* (1992), ACM, pp. 144–152.

[15] BSHOUTY, N. H. Exact learning boolean functions via the monotone theory. *Inform. Comp. 123*, 1 (1995), 146–153.

[16] BUCILUǍ, C., CARUANA, R., AND NICULESCU-MIZIL, A. Model compression. In *KDD* (2006), ACM, pp. 535–541.

[17] CHAUDHURI, K., AND MONTELEONI, C. Privacy-preserving logistic regression. In *NIPS* (2009), pp. 289–296.

[18] COHN, D., ATLAS, L., AND LADNER, R. Improving generalization with active learning. *Machine learning 15*, 2 (1994), 201–221.

[19] CRAVEN, M. W., AND SHAVLIK, J. W. Extracting tree-structured representations of trained networks. In *NIPS* (1996).

[20] CYBENKO, G. Approximation by superpositions of a sigmoidal function. *MCSS 2*, 4 (1989), 303–314.

[21] DALVI, N., DOMINGOS, P., SANGHAI, S., VERMA, D., ET AL. Adversarial classification. In *KDD* (2004), ACM, pp. 99–108.

[22] DWORK, C. Differential privacy. In *ICALP* (2006), Springer.

[23] FREDRIKSON, M., JHA, S., AND RISTENPART, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS* (2015), ACM, pp. 1322–1333.

[24] FREDRIKSON, M., LANTZ, E., JHA, S., LIN, S., PAGE, D., AND RISTENPART, T. Privacy in pharmacogenetics: An end-to-end case study of personalized Warfarin dosing. In *USENIX Security* (2014), pp. 17–32.

[25] GOOGLE PREDICTION API. `https://cloud.google.com/prediction`. Accessed Feb. 10, 2016.

[26] HICKEY, W. How Americans Like their Steak. `http://fivethirtyeight.com/datalab/how-americans-like-their-steak`, 2014. Accessed Feb. 10, 2016.

[27] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network. *arXiv:1503.02531* (2015).

[28] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural networks 2*, 5 (1989), 359–366.

[29] HUANG, L., JOSEPH, A. D., NELSON, B., RUBINSTEIN, B. I., AND TYGAR, J. Adversarial machine learning. In *AISec* (2011), ACM, pp. 43–58.

[30] JACKSON, J. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. In *FOCS* (1994), IEEE, pp. 42–53.

[31] JAGANNATHAN, G., PILLAIPAKKAMNATT, K., AND WRIGHT, R. N. A practical differentially private random decision tree classifier. In *ICDMW* (2009), IEEE, pp. 114–121.

[32] KLOFT, M., AND LASKOV, P. Online anomaly detection under adversarial impact. In *AISTATS* (2010), pp. 405–412.

[33] KUSHILEVITZ, E., AND MANSOUR, Y. Learning decision trees using the Fourier spectrum. *SICOMP 22*, 6 (1993), 1331–1348.

[34] LI, N., QARDAJI, W., SU, D., WU, Y., AND YANG, W. Membership privacy: A unifying framework for privacy definitions. In *CCS* (2013), ACM.

[35] LICHMAN, M. UCI machine learning repository, 2013.

[36] LOWD, D., AND MEEK, C. Adversarial learning. In *KDD* (2005), ACM, pp. 641–647.

[37] LOWD, D., AND MEEK, C. Good word attacks on statistical spam filters. In *CEAS* (2005).

[38] MICROSOFT AZURE. `https://azure.microsoft.com/services/machine-learning`. Accessed Feb. 10, 2016.

[39] NELSON, B., RUBINSTEIN, B. I., HUANG, L., JOSEPH, A. D., LEE, S. J., RAO, S., AND TYGAR, J. Query strategies for evading convex-inducing classifiers. *JMLR 13*, 1 (2012), 1293–1332.

[40] NEWSOME, J., KARP, B., AND SONG, D. Paragraph: Thwarting signature learning by training maliciously. In *RAID* (2006), Springer, pp. 81–105.

[41] NOCEDAL, J., AND WRIGHT, S. *Numerical optimization*. Springer Science & Business Media, 2006.

[42] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *JMLR 12* (2011), 2825–2830.

[43] PREDICTIONIO. `http://prediction.io`. Accessed Feb. 10, 2016.

[44] RUBINSTEIN, B. I., BARTLETT, P. L., HUANG, L., AND TAFT, N. Learning in a large function space: Privacy-preserving mechanisms for SVM learning. *JPC 4*, 1 (2012), 4.

[45] RUBINSTEIN, B. I., NELSON, B., HUANG, L., JOSEPH, A. D., LAU, S.-H., RAO, S., TAFT, N., AND TYGAR, J. Antidote: understanding and defending against poisoning of anomaly detectors. In *IMC* (2009), ACM, pp. 1–14.

[46] SAAR-TSECHANSKY, M., AND PROVOST, F. Handling missing values when applying classification models. *JMLR* (2007).

[47] SETTLES, B. Active learning literature survey. *University of Wisconsin, Madison 52*, 55-66 (1995), 11.

[48] SHOKRI, R., AND SHMATIKOV, V. Privacy-preserving deep learning. In *CCS* (2015), ACM, pp. 1310–1321.

[49] SMITH, T. W., MARSDEN, P., HOUT, M., AND KIM, J. General social surveys, 1972-2012, 2013.

[50] STEVENS, D., AND LOWD, D. On the hardness of evading combinations of linear classifiers. In *AISec* (2013), ACM, pp. 77–86.

[51] THEANO DEVELOPMENT TEAM. Theano: A Python framework for fast computation of mathematical expressions. *arXiv:1605.02688* (2016).

[52] TOWELL, G. G., AND SHAVLIK, J. W. Extracting refined rules from knowledge-based neural networks. *Machine learning 13*, 1 (1993), 71–101.

[53] VALIANT, L. G. A theory of the learnable. *Communications of the ACM 27*, 11 (1984), 1134–1142.

[54] VINTERBO, S. Differentially private projected histograms: Construction and use for prediction. In *ECML-PKDD* (2012).

[55] ŠRNDIĆ, N., AND LASKOV, P. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP)* (2014), IEEE, pp. 197–211.

[56] ZHANG, J., ZHANG, Z., XIAO, X., YANG, Y., AND WINSLETT, M. Functional mechanism: regression analysis under differential privacy. In *VLDB* (2012).

[57] ZHU, J., AND HASTIE, T. Kernel logistic regression and the import vector machine. In *NIPS* (2001), pp. 1081–1088.

# A Some Details on Models

**SVMs.** Support vector machines (SVMs) perform binary classification ($c = 2$) by defining a maximally separating hyperplane in $d$-dimensional feature space. A linear SVM is a function $f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + \beta)$ where 'sign' outputs 0 for all negative inputs and 1 otherwise. Linear SVMs are not suitable for non-linearly separable data. Here one uses instead kernel techniques [14].

A kernel is a function $K\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Typical kernels include the quadratic kernel $K_{\text{quad}}(\mathbf{x},\mathbf{x}') = (\mathbf{x}^T \cdot \mathbf{x}' + 1)^2$ and the Gaussian radial basis function (RBF) kernel $K_{\text{rbf}}(\mathbf{x},\mathbf{x}') = e^{-\gamma||\mathbf{x}-\mathbf{x}'||^2}$, parameterized by a value $\gamma \in \mathbb{R}$. A kernel's projection function is a map $\phi$ defined by $K(\mathbf{x},\mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}')$. We do not use $\phi$ explicitly, indeed for RBF kernels this produces an infinite-dimension vector. Instead, classification is defined using a "kernel trick": $f(\mathbf{x}) = \text{sign}([\sum_{i=1}^{t} \alpha_i K(\mathbf{x},\mathbf{x}_i)] + \beta)$ where $\beta$ is again a learned threshold, $\alpha_1,\ldots,\alpha_t$ are learned weights, and $\mathbf{x}_1,\ldots,\mathbf{x}_t$ are feature vectors of inputs from a training set. The $\mathbf{x}_i$ for which $\alpha_i \neq 0$ are called support vectors. Note that for non-zero $\alpha_i$, it is the case that $\alpha_i < 0$ if the training-set label of $\mathbf{x}_i$ was zero and $\alpha_i > 0$ otherwise.

**Logistic regression.** SVMs do not directly generalize to multiclass settings $c > 2$, nor do they output class probabilities. Logistic regression (LR) is a popular classifier that does. A binary LR model is defined as $f_1(\mathbf{x}) = \sigma(\mathbf{w}\cdot\mathbf{x}+\beta) = 1/(1+e^{-(\mathbf{w}\cdot\mathbf{x}+\beta)})$ and $f_0(\mathbf{x}) = 1 - f_1(\mathbf{x})$. A class label is chosen as 1 iff $f_1(\mathbf{x}) > 0.5$.

When $c > 2$, one fixes $c$ weight vectors $\mathbf{w}_0,\ldots,\mathbf{w}_{c-1}$ each in $\mathbb{R}^d$, thresholds $\beta_0,\ldots,\beta_{c-1}$ in $\mathbb{R}$ and defines $f_i(\mathbf{x}) = e^{\mathbf{w}_i\cdot\mathbf{x}+\beta_i}/(\sum_{j=0}^{c-1} e^{\mathbf{w}_j\cdot\mathbf{x}+\beta_j})$ for $i \in \mathbb{Z}_c$. The class label is taken to be $\text{argmax}_i\, f_i(\mathbf{x})$. Multiclass regression is referred to as multinomial or softmax regression. An alternative approach to softmax regression is to build a binary model $\sigma(\mathbf{w}_i\cdot\mathbf{x}+\beta_i)$ per class in a *one-vs-rest* fashion and then set $f_i(\mathbf{x}) = \sigma(\mathbf{w}_i\cdot\mathbf{x}+\beta_i)/\sum_j \sigma(\mathbf{w}_j\cdot\mathbf{x}+\beta_j)$.

These are log-linear models, and may not be suitable for data that is not linearly separable in $\mathcal{X}$. Again, one may use kernel techniques to deal with more complex data relationships (c.f., [57]). Then, one replaces $\mathbf{w}_i\cdot\mathbf{x}+\beta_i$ with $\sum_{r=1}^{t} \alpha_{i,r} K(\mathbf{x},\mathbf{x}_r) + \beta_i$. As written, this uses the entire set of training data points $\mathbf{x}_1,\ldots,\mathbf{x}_t$ as so-called representors (here analogous to support vectors). Unlike with SVMs, where most training data set points will never end up as support vectors, here all training set points are potentially representors. In practice one uses a size $s < t$ random subset of training data [57].

**Deep neural networks.** A popular way of extending softmax regression to handle data that is non linearly separable in $\mathcal{X}$ is to first apply one or more non-linear transformations to the input data. The goal of these *hidden layers* is to map the input data into a (typically) lower-dimensional space in which the classes are separable by the softmax layer. We focus here on fully connected networks, also known as multilayer perceptrons, with a single hidden layer. The hidden layer consists of a number $h$ of *hidden nodes*, with associated weight vectors $\mathbf{w}_0^{(1)},\ldots,\mathbf{w}_{h-1}^{(1)}$ in $\mathbb{R}^d$ and thresholds $\beta_0^{(1)},\ldots,\beta_{h-1}^{(1)}$ in $\mathbb{R}$. The $i$-th hidden unit applies a non linear transformation $h_i(\mathbf{x}) = g(\mathbf{w}_i^{(1)}\cdot\mathbf{x}+\beta_i^{(1)})$, where $g$ is an activation function such as tanh or $\sigma$. The vector $h(\mathbf{x}) \in \mathbb{R}^h$ is then input into a softmax output layer with weight vectors $\mathbf{w}_0^{(2)},\ldots,\mathbf{w}_{c-1}^{(2)}$ in $\mathbb{R}^h$ and thresholds $\beta_0^{(2)},\ldots,\beta_{c-1}^{(2)}$ in $\mathbb{R}$.

**Decision trees.** A decision tree $T$ is a labeled tree. Each internal node $v$ is labeled by a feature index $i \in \{1,\ldots,d\}$ and a *splitting function* $\rho : \mathcal{X}_i \to \mathbb{Z}_{k_v}$, where $k_v \geq 2$ denotes the number of outgoing edges of $v$.

On an input $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, a tree $T$ defines a computation as follows, starting at the root. When we reach a node $v$, labeled by $\{i,\rho\}$, we proceed to the child of $v$ indexed by $\rho(x_i)$. We consider three types of splitting functions $\rho$ that are typically used in practice ([11]):

(1) The feature $x_i$ is categorical with $\mathcal{X}_i = \mathbb{Z}_k$. Let $\{S,T\}$ be some partition of $\mathbb{Z}_k$. Then $k_v = 2$ and $\rho(x_i) = 0$ if $x_i \in S$ and $\rho(x_i) = 1$ if $x_i \in T$. This is a binary split on a categorical feature.

(2) The feature $x_i$ is categorical with $\mathcal{X}_i = \mathbb{Z}_k$. We have $k_v = k$ and $\rho(x_i) = x_i$. This corresponds to a $k$-ary split on a categorical feature of arity $k$.

(3) The feature $x_i$ is continuous with $\mathcal{X}_i = [a,b]$. Let $a < t < b$ be a *threshold*. Then $k_v = 2$ and $\rho(x_i) = 0$ if $x_i \leq t$ and $\rho(x_i) = 1$ if $x_i > t$. This is a binary split on a continuous feature with threshold $t$.

When we reach a leaf, we terminate and output that leaf's value. This value can be a class label, or a class label and confidence score. This defines a function $f : \mathcal{X} \to \mathcal{Y}$.

## B Details on Data Sets

Here we give some more information about the data sets we used in this work. Refer back to Table 3 and Table 5.

**Synthetic data sets.** We used 4 synthetic data sets from `scikit` [42]. The first two data sets are classic examples of non-linearly separable data, consisting of two concentric *Circles*, or two interleaving *Moons*. The next two synthetic data sets, *Blobs* and *5-Class*, consist of Gaussian clusters of points assigned to either 3 or 5 classes.

**Public data sets.** We gathered a varied set of data sets representative of the type of data we would expect ML service users to use to train logistic and SVM based models. These include famous data sets used for supervised learning, obtained from the UCI ML repository (*Adult*, *Iris*, *Breast Cancer*, *Mushrooms*, *Diabetes*). We also consider the *Steak* and *GSS* data sets used in prior work on model inversion [23]. Finally, we add a data set of digits available in `scikit`, to visually illustrate training data leakage in kernelized logistic models (c.f. Section 4.1.3).

**Public data sets and models from BigML.** For experiments on decision trees, we chose a varied set of models publicly available on BigML's platform. These models were trained by real MLaaS users and they cover a wide range of application scenarios, thus providing a realistic benchmark for the evaluation of our extraction attacks.

The *IRS* model predicts a US state, based on administrative tax records. The *Steak* and *GSS* models respectively predict a person's preferred steak preparation and happiness level, from survey and demographic data. These two models were also considered in [23]. The *Email Importance* model predicts whether Gmail classifies an email as 'important' or not, given message metadata. The *Email Spam* model classifies emails as spam, given the presence of certain words in its content. The German Credit data set was taken from the UCI library [35] and classifies a user's loan risk. Finally, two regression models respectively predict *Medical Charges* in the US based on state demographics, and the *Bitcoin Market Price* from daily opening and closing values.

## C Analysis of the Path-Finding Algorithm

In this section, we analyze the correctness and complexity of the decision tree extraction algorithm in Algorithm 1. We assume that all leaves are assigned a unique `id` by the oracle $\mathcal{O}$, and that no continuous feature is split into intervals of width smaller than $\varepsilon$. We may use `id` to refer directly to the leaf with identity `id`.

**Correctness.** Termination of the algorithm follows immediately from the fact that new queries are only added to $Q$ when a new leaf is visited. As the number of leaves in the tree is bounded, the algorithm must terminate.

We prove by contradiction that all leaves are eventually visited. Let the *depth* of a node $v$, denote the length of the path from $v$ to the root (the root has depth 0). For two leaves $\texttt{id}, \texttt{id}'$, let $A$ be their deepest common ancestor ($A$ is the deepest node appearing on both the paths of `id` and $\texttt{id}'$). We denote the depth of $A$ as $\Delta(\texttt{id}, \texttt{id}')$.

Suppose Algorithm 1 terminates without visiting all leaves, and let $(\texttt{id}, \texttt{id}')$ be a pair of leaves with maximal $\Delta(\texttt{id}, \texttt{id}')$, such that `id` was visited but $\texttt{id}'$ was not. Let $x_i$ be the feature that their deepest common ancestor $A$ splits on. When `id` is visited, the algorithm calls `LINE_SEARCH` or `CATEGORY_SPLIT` on feature $x_i$. As all leaf `id`s are unique and there are no intervals smaller than $\varepsilon$, we will discover a leaf in each sub-tree rooted at $A$, including the one that contains $\texttt{id}'$. Thus, we visit a leaf $\texttt{id}''$ for which $\Delta(\texttt{id}'', \texttt{id}') > \Delta(\texttt{id}, \texttt{id}')$, a contradiction.

**Complexity.** Let $m$ denote the number of leaves in the tree. Each leaf is visited exactly once, and for each leaf we check all $d$ features. Suppose continuous features have range $[0, b]$, and categorical features have arity $k$. For continuous features, finding one threshold takes at most $\log_2(\frac{b}{\varepsilon})$ queries. As the total number of splits on one feature is at most $m$ (i.e., all nodes split on the same feature), finding all thresholds uses at most $m \cdot \log_2(\frac{b}{\varepsilon})$ queries. Testing a categorical feature uses $k$ queries. The total query complexity is $O(m \cdot (d_{cat} \cdot k + d_{cont} \cdot m \cdot \log(\frac{b}{\varepsilon}))$, where $d_{cat}$ and $d_{cont}$ represent respectively the number of categorical and continuous features.

For the special case of boolean trees, the complexity is $O(m \cdot d)$. In comparison, the algorithm of [33], that uses membership queries only, has a complexity polynomial in $d$ and $2^{\delta}$, where $\delta$ is the tree depth. For degenerate trees, $2^{\delta}$ can be exponential in $m$, implying that the assumption of unique leaf identities (obtained from confidence scores for instance) provides an exponential speedup over the best-known approach with class labels only. The algorithm from [33] can be extended to regression trees, with a complexity polynomial in the size of the output range $\mathcal{Y}$. Again, under the assumption of unique leaf identities (which could be obtained solely from the output values) we obtain a much more efficient algorithm, with a complexity independent of the output range.

**The Top-Down Approach.** The correctness and complexity of the top-down algorithm from Section 4.2 (which uses incomplete queries), follow from a similar analysis. The main difference is that we assume that all nodes have a unique `id`, rather than only the leaves.

## D A Note on Improper Extraction

To extract a model $f$, without knowledge of the model class, a simple strategy is to extract a multilayer perceptron $\hat{f}$ with a large enough hidden layer. Indeed, feed-forward networks with a single hidden layer can, in principle, closely approximate any continuous function over a bounded subset of $\mathbb{R}^d$ [20, 28].

However, this strategy intuitively does not appear to be optimal. Even if we know that we can find a multilayer perceptron $\hat{f}$ that closely matches $f$, $\hat{f}$ might have a far more complex representation (more parameters) than $f$. Thus, tailoring the extraction to the 'simpler' model class of the target $f$ appears more efficient. In learning theory, the problem of finding a succinct representation of some target model $f$ is known as *Occam Learning* [13].

Our experiments indicate that such generic improper extraction indeed appears sub-optimal, in the context of equation-solving attacks. We train a softmax regression over the Adult data set with target "Race". The model $f$ is defined by 530 real-valued parameters. As shown in Section 4.1.2, using only 530 queries, we extract a model $\hat{f}$ from the *same* model class, that closely matches $f$ ($\hat{f}$ and $f$ predict the same labels on 100% of tested inputs, and produce class probabilities that differ by less than $10^{-7}$ in TV distance). We also extracted the same model, assuming a multilayer perceptron target class. Even with 1,000 hidden nodes (this model has 111,005 parameters), and 10× more queries (5,300), the extracted model $\hat{f}$ is a weaker approximation of $f$ (99.5% accuracy for class labels and TV distance of $10^{-2}$ for class probabilities).