FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

AN INTRODUCTION TO MACHINE LEARNING

eMag Issue 50 - April 2017

Introduction to Machine Learning with Python

This series explores various topics and techniques in machine learning, arguably the most talked-about area of technology and computer science over the past several years. In this article, Michael Manapat begins with an extended "case study" in Python: how can we build a machine learning model to detect credit card fraud?

Practicing Machine Learning with Optimism

Using machine learning to solve real-world problems often presents challenges that weren't initially considered during the development of the machine learning method. Alyssa Frazee addresses a few examples of such issues and hopefully provides some suggestions (and inspiration) for how to overcome the challenges using straightforward analyses on the data you already have.

AN INTRODUCTION TO
MACHINE LEARNING
InfoQ
ARTICLE
Introduction to Machine Learning with Python
ARTICLE
Real-World, Man-Machine Algorithms
BOOK REVIEW
Andrew McAfee and Erik Brynjolfsson's "The Second Machine Age"

Anomaly Detection for Time Series Data with Deep Learning

This article introduces neural networks, including brief descriptions of feed-forward neural networks and recurrent neural networks, and describes how to build a recurrent neural network that detects anomalies in time series data. To make the discussion concrete, Tom Hanlon shows how to build a neural network using Deeplearning4j, a popular open-source deep-learning library for the JVM.

Book Review: Andrew McAfee and Erik Brynjolfsson's "The Second Machine Age"

Andrew McAffee and Erik Brynjolfsson begin their book The Second Machine Age with a simple question: what innovation has had the greatest impact on human history?

Real-World, Man-Machine Algorithms

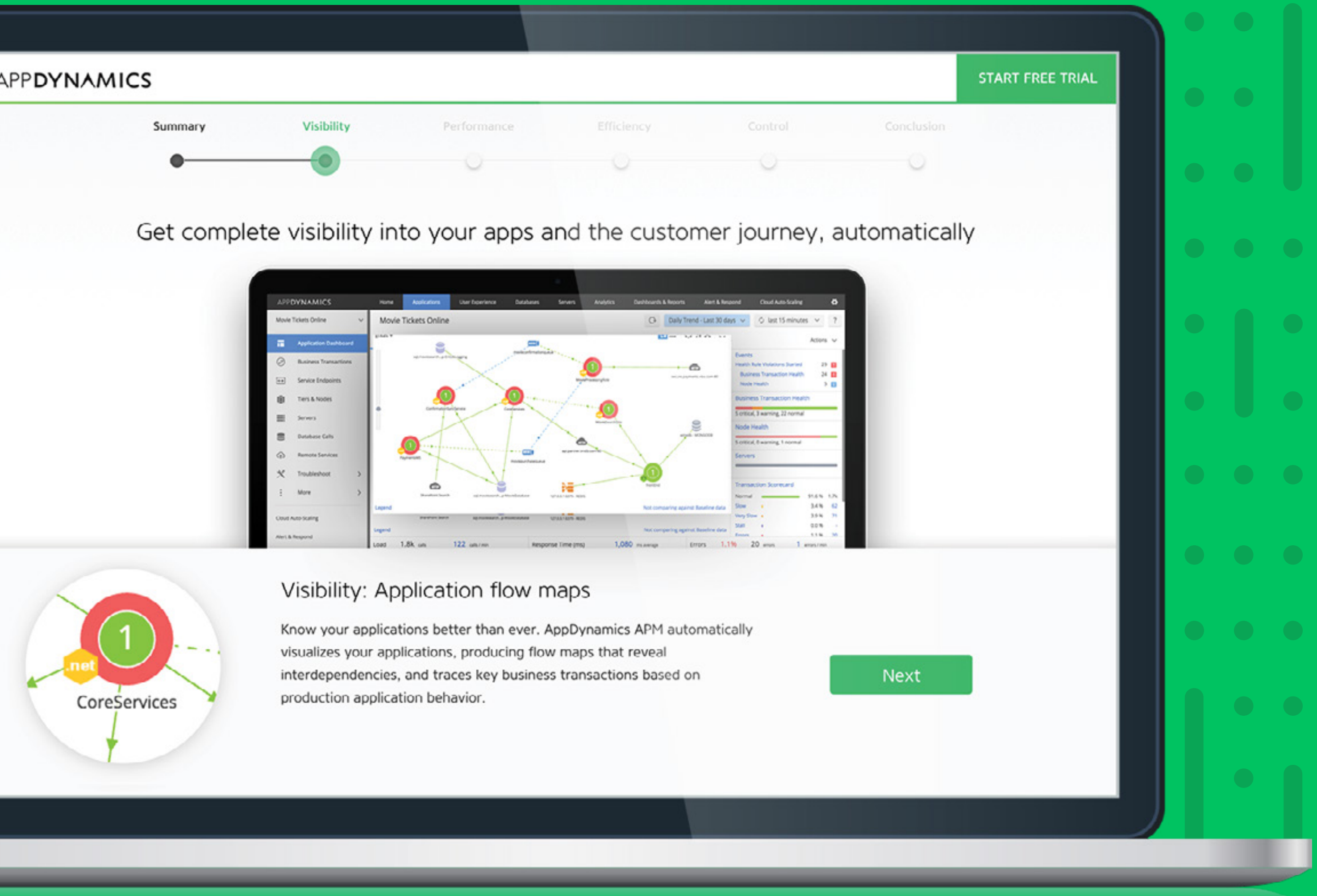
*In this article, Edwin Chen and Justin Palmer talk about the end-to-end flow of developing machine learning models:
where you get training data, how you pick the ML algorithm, what you must address after your model is deployed, and so forth.*

APPDYNAMICS

Get complete visibility into your apps and the customer journey, automatically.

Optimizing customer experience with end-to-end visibility and management of even the most complex and distributed applications with auto-generated flow maps, dynamic baselining, and code-level diagnostics.

Take a Tour at **AppDynamics.com**



Get complete visibility into your apps and the customer journey, automatically

Visibility: Application flow maps

Know your applications better than ever. AppDynamics APM automatically visualizes your applications, producing flow maps that reveal interdependencies, and traces key business transactions based on production application behavior.

MICHAEL MANAPAT
leads work on Stripe's machine learning products, including Stripe Radar. Prior to Stripe, he was an engineer at Google and a postdoctoral fellow in and lecturer on applied mathematics at Harvard. He received a Ph.D. in mathematics from MIT.

A LETTER FROM THE EDITOR

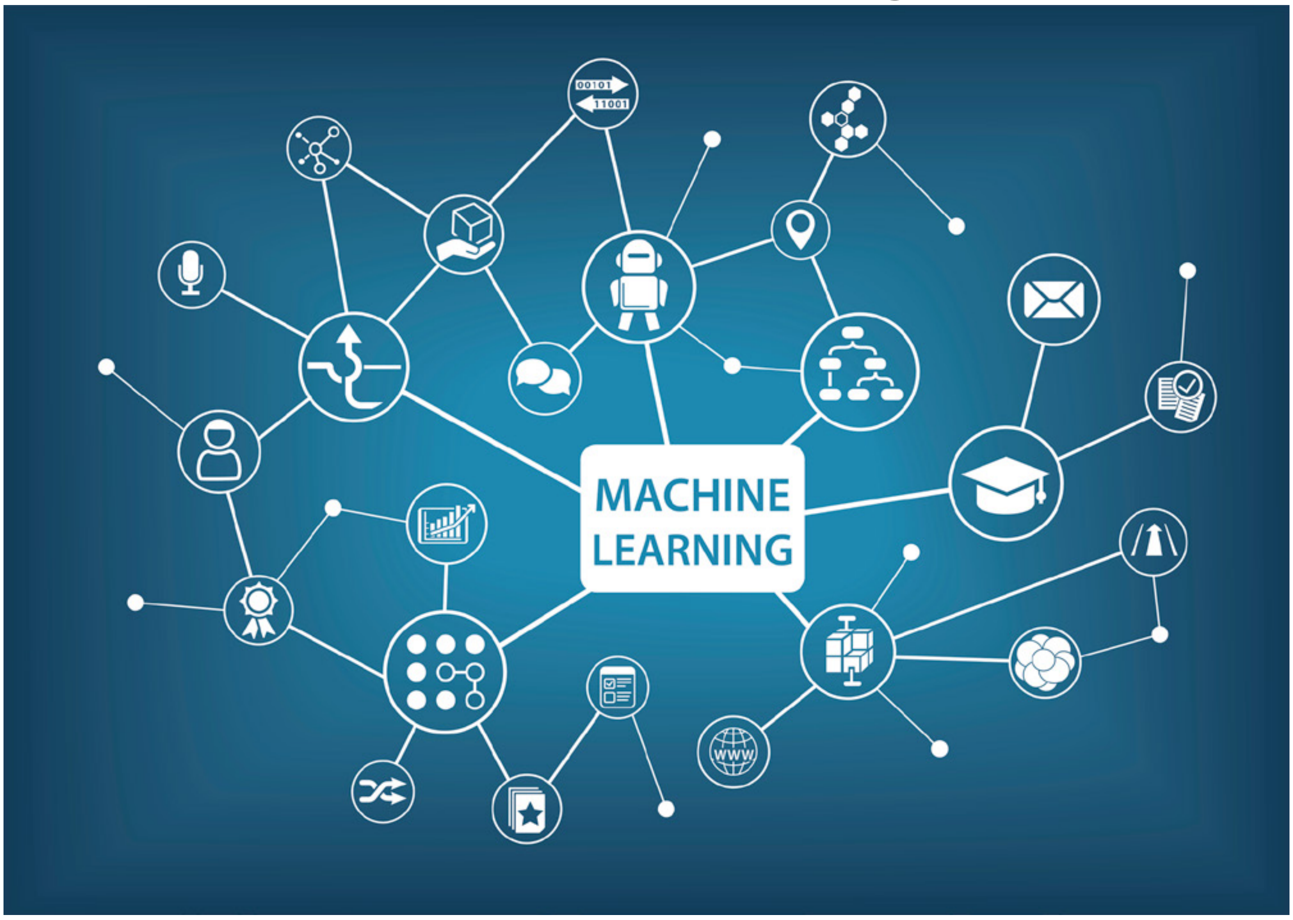
Machine learning has long powered many products we interact with daily—from "intelligent" assistants like Apple's Siri and Google Now, to recommendation engines like Amazon's that suggest new products to buy, to the ad ranking systems used by Google and Facebook.

More recently, machine learning has entered the public consciousness because of advances in "deep learning"—these include AlphaGo's defeat of Go grandmaster Lee Sedol and impressive new products around image recognition and machine translation.

While much of the press around machine learning has focused on achievements that were not previously possible, the full range of machine learning methods—from traditional techniques that have been around for decades to more recent approaches with neural networks—can be deployed to solve many important (but perhaps more prosaic) problems that businesses face. Examples of these applications include, but are by no means limited to, fraud prevention, time-series forecasting, and spam detection.

InfoQ has curated a series of articles for this introduction to machine learning eMagazine covering everything from the very basics of machine learning (what are typical classifiers and how do you measure their performance?), to production considerations (how do you deal with changing patterns in data after you've deployed your model?), to newer techniques in deep learning. After reading through this series, you should be ready to start on a few machine learning experiments of your own.

Introduction to Machine Learning with Python



Michael Manapat

This e-mag will explore various topics and techniques in machine learning, arguably the most-talked-about area of technology and computer science over the past several years.

Machine learning at a high level has been covered in previous InfoQ articles (see, for example, "Getting Started with Machine Learning" in the Getting a Handle on Data Science e-mag and series), and this article and the ones that follow it elaborate on many of the concepts and methods discussed earlier with emphasis on concrete examples and venture into some new areas, including neural networks and deep learning.

We'll begin, in this article, with an extended case study in Python: how can we build a machine-learning model to detect credit-card fraud? (While we'll use the language of fraud detection, much of what we do may apply with little modification to other classification problems — for example, ad-click prediction.) Along the way, we'll encounter many of the key ideas and terms in machine learning.

Target: Credit-card fraud

Businesses that sell products online inevitably have to deal with fraud. In a typical fraudulent transaction, the fraudster will obtain stolen credit-card numbers and use them to purchase goods online. The fraudsters will then sell those goods elsewhere at a discount, pocketing the proceeds, while the business must bear the cost of the chargeback. You can read more about the details of credit-card fraud here.

KEY TAKEAWAYS

Logistic regression is appropriate for binary classification when the relationship between the input variables and the output we're trying to predict is linear or when it's important to be able to interpret the model (by, for example, isolating the impact that any one input variable has on the prediction).

Decision trees and random forests are non-linear models that can capture well more complex relationships but are less amenable to human interpretation.

It's important to assess model performance appropriately to verify that your model will perform well on data it has not seen before.

Putting a machine-learning model into production involves many considerations distinct from those in the model development process: for example, how do you compute model inputs synchronously? What information do you need to log every time you score? And how do you determine the performance of your model in production?

Let's say we're an online business that has been experiencing fraud for some time, and we'd like to use machine learning to help with the problem. More specifically, every time a transaction is made, we'd like to predict whether or not it'll turn out to be fraudulent (i.e., whether or not the authorized cardholder is making the purchase) so that we can take appropriate action. This type of machine-learning problem is known as classification as we are assigning every incoming payment to one of two classes: `fraud` or `not-fraud`.

For every historical payment, we have a Boolean that indicates whether the charge was fraudulent (`fraudulent`) and other attributes that we think might be indicative of fraud — for example, the payment in US dollars (`amount`), the country in which the card was issued (`card_country`), and the number of payments made with the card at our business in the past day (`card_use_24h`). Thus, the data we have to build our predictive model might look like the following CSV:

```
fraudulent,charge_time,amount,card_
country,card_use_24h

False,2015-12-31T23:59:59Z,20484,US,0

False,2015-12-31T23:59:59Z,1211,US,0

False,2015-12-31T23:59:59Z,8396,US,1

False,2015-12-31T23:59:59Z,2359,US,0

False,2015-12-31T23:59:59Z,1480,US,3

False,2015-12-31T23:59:59Z,535,US,3

False,2015-12-31T23:59:59Z,1632,US,0

False,2015-12-31T23:59:59Z,10305,US,1

False,2015-12-31T23:59:59Z,2783,US,0
```

There are two important details we're going to skip over in our discussion but they're worth keeping in mind as they are just as important, if not more so, than the basics of model building we're covering here.

First, there is the problem of **data science** in determining what features we think are indicative of fraud. In our example, we've identified the payment amount, the country in which the card was issued, and the number of times the card was used in the past day as features that we think may be useful in predicting fraud. In general, you'll need to spend a lot of time looking at data to determine what's useful and what's not.

Second, there is the problem of **data infrastructure** in computing the values of features: we need those values for all historical samples to train the model but

. . .every time a transaction is made, we'd like to predict whether or not it'll turn out to be fraudulent (i.e., whether or not the authorized cardholder is making the purchase) so that we can take appropriate action.

we also need their real-time values as payments come in to score new transactions appropriately. It's unlikely that, before we began worrying about fraud, we were already maintaining and recording the number of card uses over 24-hour rolling windows, so if we find that that feature is useful for fraud detection, we'll need to be able to compute it both in production and in batch. Depending on the definition of the feature, this can be highly non-trivial.

These problems together are frequently referred to as **feature engineering** and are often the most involved (and impactful) parts of industrial machine learning.

Logistic regression

Let's start with one of the most basic possible models: a **linear** one. We'll attempt to find coefficients a, b,… Z so that:

$$\text{Probability(fraud)} = a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z$$

For every payment, we'll plug in the values of `amount`, `card_country`, and `card_use_24h` into the formula above, and if the probability is greater than 0.5, we'll "predict" that the payment is fraudulent and otherwise we'll predict that it's legitimate.

Even before we discuss how to compute a, b,… Z, there are two immediate problems to address:

`Probability(fraud)` needs to be a number between 0 and 1, but the quantity on the right side can get arbitrarily large (in absolute value) depending on the values of `amount` and `card_use_24h` (if those feature values are sufficiently large and one of a or b is nonzero).

`card_country` isn't a number: it takes one of a number of values (say US, AU, GB, and so forth). Such so-called "categorical" features need to be encoded appropriately before we can train our model.

Logit function

To address the first problem, instead of directly modeling `p=Probability(fraud)`, we'll model what is known as the log-odds of fraud, so our model becomes:

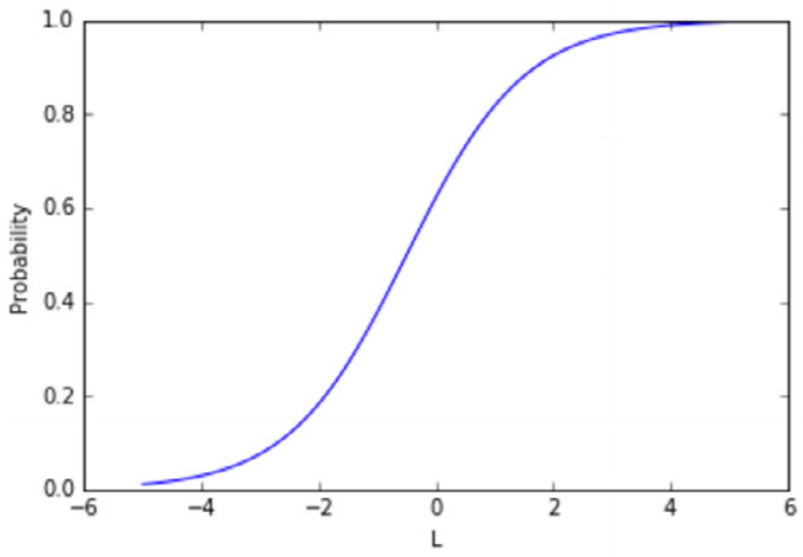
$$\log\left(\frac{p}{1-p}\right) = a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z$$

If an event has probability p, its odds are `p/(1-p)`, which is why the left side is called the "log odds" or "logit."

Given values of a, b,… Z, and the features, we can compute the predicted probability of fraud by inverting the function above to get:

$$p = \frac{\exp(a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z)}{1 + \exp(a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z)}$$

The probability of fraud p is a sigmoidal function of the linear function `L=a*amount+b*card_use_24h+…` and looks like the following:



Regardless of the value of the linear function, the sigmoid maps it to a number between 0 and 1, which is a legitimate probability.

Categorical variables

To address the second problem in our list, we'll take the categorical variable `card_country` (which, say, takes one of N distinct values) and expand it into N-1 dummy variables. These new features will be Booleans of the form `card_country=AU`, `card_country=GB`, etc. We only need N-1 "dummies" because the Nth value is implied when the N-1 dummies are all false. For simplicity, let's say that `card_country` can take just one of three values here: `AU, GB`, or `US`. Then we need two dummy variables to encode it, and the model we would like to fit (i.e., find the coefficient values for) is:

$$\log\left(\frac{p}{1-p}\right) = a \times \text{amount} + b \times \text{card\_use\_24h} + c \times (\text{country} = \text{AU}) + d \times (\text{country} = \text{GB}) + Z$$

This type of model is known as a "logistic regression".

Fitting the model

How do we determine the values of a, b, c, d, and Z? Let's start by picking random guesses for them . We can define the likelihood of this set of guesses as:

$$\ell(a,b,c,d,Z) = \prod_{\text{fraud}} p(x_i) \prod_{\text{not fraud}} (1 - p(x_j))$$

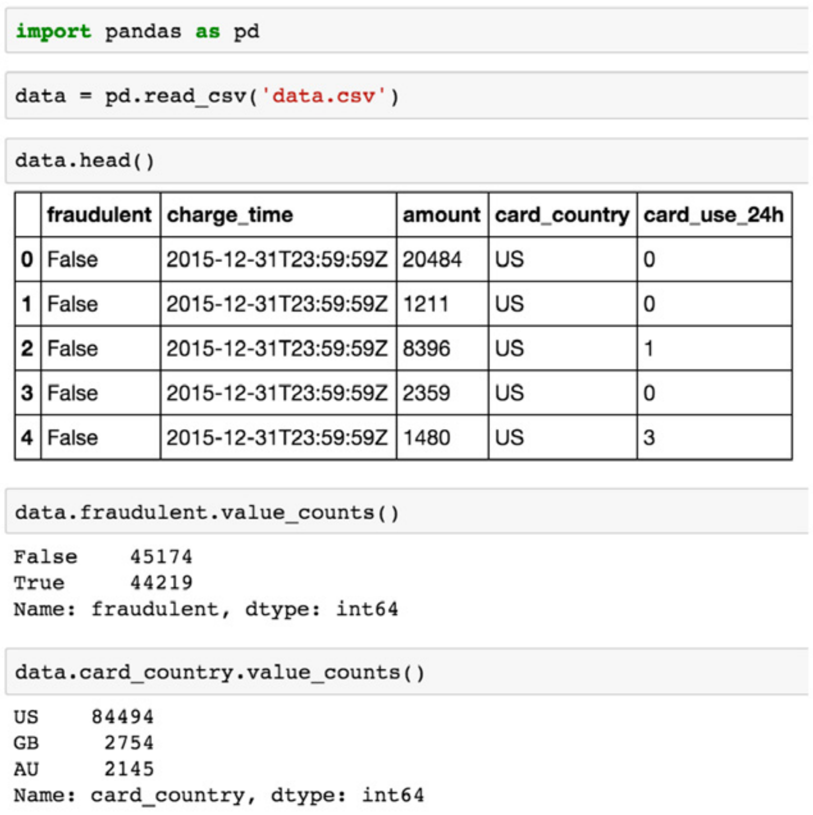
That is, take every sample in our data set and compute the predicted probability of fraud p given our guesses of a, b, c, d, and Z (and the feature values for each sample) using:

$$p = \frac{\exp(a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z)}{1 + \exp(a \times \text{amount} + b \times \text{card\_use\_24h} + \cdots + Z)}$$

For every sample that actually was fraudulent, we'd like p to be close to 1, and for every sample that was not fraudulent, we'd like p to be close to 0 (and so 1-p should be close to 1). Thus, we take the product of p over all fraudulent samples with the product of 1-p over all non-fraudulent samples to assess the accuracy of our guesses for a, b, c, d, and Z. We'd like to make the likelihood function as large as possible (i.e., as close as possible to 1). Starting with our guess, we'll iteratively tweak a, b, c, d, and Z to improve the likelihood until we find that we can no longer increase it by perturbing the coefficients. One common method for this optimization is stochastic gradient descent.

Implementation in Python

Now we'll use some standard open-source tools in Python to put into practice the theory we've just discussed. We'll use pandas, which brings R-like data frames to Python, and scikit-learn, a popular machine-learning package. Let's say the sample data we described above is in a CSV file named "data.csv"; we can load the data and take a peek at it with the following:



```
import pandas as pd
```

```
data = pd.read_csv('data.csv')
```

```
data.head()
```

	fraudulent	charge_time	amount	card_country	card_use_24h
0	False	2015-12-31T23:59:59Z	20484	US	0
1	False	2015-12-31T23:59:59Z	1211	US	0
2	False	2015-12-31T23:59:59Z	8396	US	1
3	False	2015-12-31T23:59:59Z	2359	US	0
4	False	2015-12-31T23:59:59Z	1480	US	3

```
data.fraudulent.value_counts()
```

```
False    45174
True     44219
Name: fraudulent, dtype: int64
```

```
data.card_country.value_counts()
```

```
US    84494
GB     2754
AU     2145
Name: card_country, dtype: int64
```

We can encode `card_country` into the appropriate dummy variables with:

```
encoded_countries = pd.get_dummies(data.card_country, prefix='cc_')
```

```
encoded_countries.head()
```

	cc__AU	cc__GB	cc__US
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1

```
data = data.join(encoded_countries)
```

```
data.head()
```

	fraudulent	charge_time	amount	card_country	card_use_24h	cc__AU	cc__GB	cc__US
0	False	2015-12-31T23:59:59Z	20484	US	0	0	0	1
1	False	2015-12-31T23:59:59Z	1211	US	0	0	0	1
2	False	2015-12-31T23:59:59Z	8396	US	1	0	0	1
3	False	2015-12-31T23:59:59Z	2359	US	0	0	0	1
4	False	2015-12-31T23:59:59Z	1480	US	3	0	0	1

```
y = data.fraudulent
```

```
X = data[['amount', 'card_use_24h', 'cc__AU', 'cc__GB']]
```

Now the data frame has all the data we need, dummy variables and all, to train our model. We've split up the target (the variable we're trying to predict which in this case is `fraudulent`) and the features as scikit-learn takes them as different parameters.

Before proceeding with the model training, there's one more issue to discuss. We'd like our model to generalize well — i.e., it should be accurate when classifying payments that we haven't seen before and it should not just capture the idiosyncratic patterns in the payments we happen to have already seen. To make sure that we don't overfit our models to the noise in the data we have, we'll separate the data into two sets: a training set that we'll use to estimate the model parameters (a, b, c, d, and Z) and a validation set (also called a "test set") that we'll use to compute metrics of model performance (see the next section on what these are). If a model overfits, it will perform well on the training set (as it will have learned the patterns in the set) but poorly on the validation set. There are other approaches to cross-validation (for example, k-fold cross-validation), but a "train-test" split will serve our purposes here.

We can easily split our data into training and testing sets with scikit-learn as follows:

```
from sklearn.cross_validation import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```

In this example, we'll use two thirds of the data to train the model and one third of the data to validate it.

We're now ready to train the model, which at this point is a triviality:

```
# Logistic Regression
from sklearn.linear_model import LogisticRegression
```

```
lr_model = LogisticRegression().fit(X_train, y_train)
```

```
lr_model.coef_
```

```
array([[  4.62586221e-06,   3.53495554e-02,   4.28936114e-03,
          2.49802503e-03]])
```

```
lr_model.intercept_
```

```
array([-0.0157345])
```

The `fit` function runs the fitting procedure (which maximizes the likelihood function described above), and then we can query the returned object for the values of `a`, `b`, `c`, and `d` (in `coef_`) and `Z` (in `intercept_`). Our final model is:

$$\log\left(\frac{p}{1-p}\right) = 4.63 \times 10^{-6} \times \text{amount} + 0.035 \times \text{card\_use\_24h} + \\ 0.0043 \times (\text{cc\_AU} = 1) + 0.0025 \times (\text{cc\_GB} = 1) - 0.016$$

Evaluating model performance

Once we've trained a model, we need to determine how good that model is at predicting the variable of interest (in this case, the Boolean that indicates whether the payment is believed to be fraudulent or not). Recall that we said we'd classify a payment as fraudulent if `Probability(fraud)` is greater than 0.5 and that we'd classify it as legitimate otherwise. Two quantities frequently used to measure performance given a model and a classification policy such as ours are:

- the **false-positive rate**, the fraction of all legitimate charges that are incorrectly classified as fraudulent, and

- the **true-positive rate** (also known as "recall" or "sensitivity"), the fraction of all fraudulent charges that are correctly classified as fraudulent.

While there are many measures of classifier performance, we'll focus on these two.

Ideally, the false-positive rate will be close to 0 and the true-positive rate will be close to 1. As we vary the probability threshold at which we classify a charge as fraudulent (above we said it was 0.5, but we can choose any value between 0 and 1 — low values mean we're more aggressive in labeling payments as fraudulent and high values mean we're more conservative), the false-positive rate and true-positive rate trace out a curve that depends on how good our model is. This is known as the "ROC" curve and can be computed easily with scikit-learn: ▶

```
y_test_predict_lr = lr_model.predict_proba(X_test)
```

```
lr_model.classes_
```

```
array([False,  True], dtype=bool)
```

```
y_test_predict_lr
```

```
array([[ 0.48863551,  0.51136449],
       [ 0.48045414,  0.51954586],
       [ 0.49755085,  0.50244915],
       ...,
       [ 0.48520956,  0.51479044],
       [ 0.49709365,  0.50290635],
       [ 0.48343377,  0.51656623]])
```

```
y_test_scores_lr = [x[1] for x in y_test_predict_lr]
```

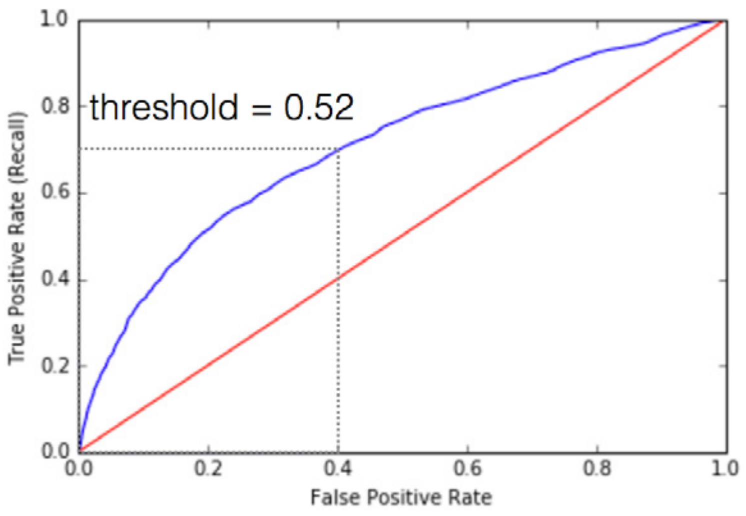
```
from sklearn.metrics import roc_curve, roc_auc_score
```

```
fpr, tpr, thresholds = roc_curve(y_test, y_test_scores_lr)
```

```
fpr[len(fpr)/2], tpr[len(tpr)/2], thresholds[len(thresholds)/2]
```

```
(0.37411749034234715, 0.680933314924754950, 0.51383466612652107)
```

The variables `fpr`, `tpr`, and `thresholds` contain the data for the full ROC curve, but we've picked a sample point here: if we say a charge is fraudulent if `Probability(fraud)` is greater than 0.514, then the false positive rate is 0.374 and the true positive rate is 0.681. The whole ROC curve and the point we picked out are depicted in the graph to the right.



The better a model is, the closer the ROC curve (the blue line above) will hug the left and top borders of the graph (the model would achieve perfect performance in the top left corner). Note that ROC curve tells us how good our model is, and this can be captured with a single number — the area under the curve (AUC). The closer the AUC is to 1, the more accurate the model. The AUC score for our current model is:

```
roc_auc_score(y_test, y_test_scores_lr)
```

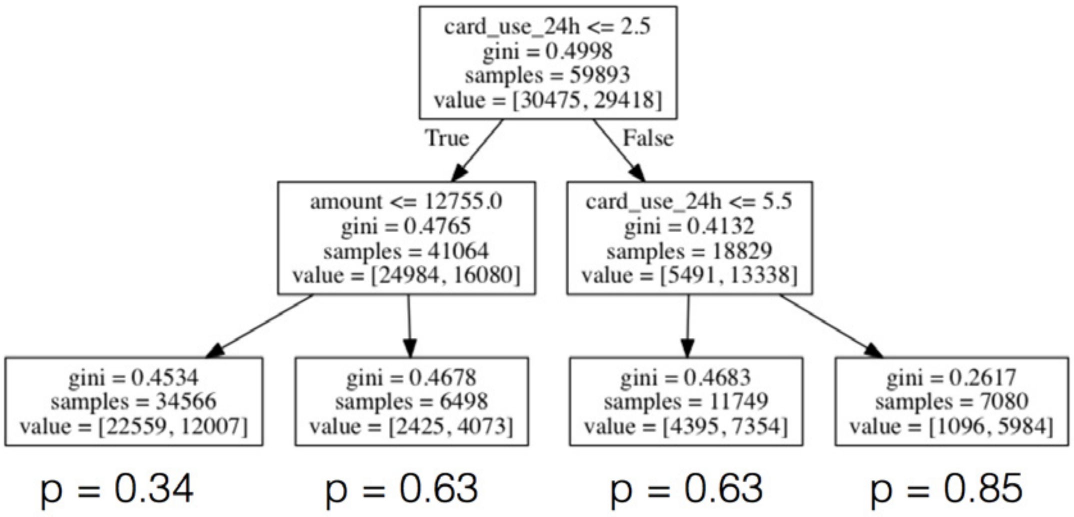
```
0.70995597399167287
```

Of course, when we put a model into production to take an action, we generally need to action the model-outputted probabilities by comparing them to a threshold as we did above, saying that a charge is predicted to be fraudulent if `Probability(fraud)>0.5`. Thus, the performance of our model for a specific application corresponds to a point on the ROC curve — the curve just controls the tradeoff between false-positive rate and true-positive rate, i.e., the policy options we have at our disposal.

Decision trees and random forests

The model above, a logistic regression, is an example of a **linear** machine-learning model. Imagine that every sample payment we have is a point in space whose coordinates are the values of features. If we had just two features, each sample point would be a point in the x-y plane. A linear model like logistic regression will generally perform well if we can separate the fraudulent samples from the non-fraudulent samples with a linear function — in the two-feature case, that just means that almost all the fraudulent samples lie on one side of a line and almost all the non-fraudulent samples lie on the other side of that line.

It's often the case that the relationship between predictive features and the target variable we're trying to predict is nonlinear, in which case we should use a nonlinear model to capture the relationship. One powerful and intuitive type of a nonlinear model is a **decision tree** like the following:



At each node, we compare the value of a specified feature to some threshold and branch either to the left or the right depending on the output of the comparison. We continue in this manner (like a game of 20 Questions, though trees do not need to be 20 levels deep) until we reach a leaf of the tree. The leaf consists of all the samples in our training set for which the comparisons at each node satisfied the path we took down the tree, and the fraction of samples in the leaf that are fraudulent is the predicted probability of fraud that the model reports. When we have a new sample to be classified, we generate its features and play the 20 Questions game until we reach a leaf, which contains a predicted probability of fraud we can assign to that transaction.

In brief, we create a decision tree by selecting a feature and threshold at each node to maximize some notion of information gain or discriminatory power — the "gini" shown in the figure above — and proceed recursively until we hit some pre-specified stopping criterion. While we won't go further into the details of producing the decision tree, training such a model with scikit-learn is as easy as training a logistic regression (or any other model, in fact): ▶

```
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
```

```
dt_model = DecisionTreeClassifier(
    max_depth=3, min_samples_split=20).fit(X_train, y_train)
```

```
y_test_scores_dt = [x[1] for x in dt_model.predict_proba(X_test)]
```

```
roc_auc_score(y_test, y_test_scores_dt)
```

```
0.69289424199670357
```

One issue with decision trees is that they can easily be overfit. A very deep tree in which each leaf has just one sample from the training data will often capture noise pertinent to each sample and not general trends — but random-forest models can help address this. In a random forest, we train a large number of decision trees, but each tree is trained on just a subset of the data we have available, and when building each tree, we only consider a subset of features for splitting. The predicted probability of fraud is then simply the average of the probabilities produced by all the trees in the forest. Training each tree on just a subset of the data, and only considering a subset of the features as split candidates at each node, reduces the correlation between the trees and makes overfitting less likely.

To summarize, linear models like logistic regressions are appropriate when the relationship between the features and the target variable is linear or when we'd like to be able to isolate the impact that any given feature has on the prediction (as this can be directly read off the regression coefficient). On the other hand, nonlinear models like decision trees and random forests are harder to interpret but can capture more complex relationships.

Productionizing machine-learning models

Training a machine-learning model is just one step in the process of using machine learning to solve a business problem. As described above, model training generally must be preceded by the work of feature engineering. And once we have a model, we need to put it into production to take appropriate actions (by blocking payments assessed to be fraudulent, for example). Some call this "productionizing" it.

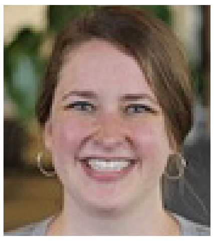
While we won't go into detail here, productionization can involve a number of challenges — for instance, we may use Python for model development while our production stack is in Ruby. If that is the case, we'll either need to port our model to Ruby by serializing it in some format from Python and having our production Ruby code load the serialization or use a service-oriented architecture with service calls from Ruby to Python.

We'll also want to maintain our model's performance metrics in production (as distinct from metrics as computed on the validation data). Depending on how we use our model, this can be difficult because the mere act of using the model to dictate actions can result in not having the data to compute these metrics. Other articles in this series will consider some of these problems.

Supporting materials

A Jupyter notebook with all the code examples above can be found here, and sample data for model training can be found here. ■

Practicing Machine Learning with Optimism



Alyssa Frazee is a machine-learning engineer at Stripe, where she builds models to detect fraud in online credit-card payments. Before Stripe, she did a Ph.D. in biostatistics and fell in love with programming at the Recurse Center. Find her on Twitter at @acfrazee.

Using machine learning to solve real-world problems often presents challenges that weren't initially considered during the development of the machine-learning (ML) method, but encountering challenges from our very own application is part of the joy of being a practitioner!

This article will address a few examples of such issues and hopefully will provide some suggestions (and inspiration) for how to overcome the challenges using straightforward analyses on the data we already have.

Perhaps we'd like to quantify the uncertainty around one of our business metrics. Unfortunately, adding error bars around any metric more complicated than an average can be daunting. Reasonable formulas for the width of the error bars often assume that our data points are independent, which is almost never true in any business — for example, we might have multiple data points per customer or customers connected to each other on a social network. Another common assumption is that our business metric is normally distributed across users, which often fails with super-users or a large number of inactive users. But never fear — simulations and non-parametric methods can almost always be used to create error bars, and all we need is a few lines of code and some computing power.

Or perhaps we're using a binary classifier in production: for example, we may be deciding whether or not to show a website visitor a specific advertisement or whether or not to decline a credit-card transaction due to fraud risk. A classifier that results in action being taken can actually become its own adversary by stopping us ▶

KEY TAKEAWAYS

You can use simulations to determine your confidence around estimates of a given metric.

When you use your machine-learning model to take actions that affect outcomes in the world, you need to have a system for counterfactual evaluation.

You can generate “explanations” for black-box model decisions, and those explanations can help with model interpretation and debugging (even if they’re rudimentary).

from observing the outcome for observations in one of the classes: we never get to see whether a website visitor would have clicked an ad if we don’t show it and we never get to see if a credit-card charge was actually fraudulent unless we process it since we’re missing the data to evaluate. Luckily, there are statistical methods for addressing this.

Finally, we may be using a black-box model: a model that makes accurate, fast predictions that computers easily understand but that aren’t designed to be examined post hoc by a human (random forests are a canonical example). Do our users want understandable explanations for decisions that the model made? Simple modeling techniques can handle that problem too.

One of my favorite things about being a statistician-turned-ML-practitioner is the optimism of the field. It feels strange to highlight optimism in fields concerned with data analysis: statisticians have a bit of a reputation for being party poopers when they point out to collaborators flaws in experimental designs, violations of model assumptions, or issues arising because of missing data. But the optimism I’ve seen derives from the fact that ML practitioners have been doing their very best to develop techniques for overcoming these sorts of problems. We can correct expensive but badly designed biology experiments after the fact. We can build regression models even if our data is correlated in surprising or unquantifiable ways that rule out standard linear regression. We can empirically estimate what could have been if we had missing data.

I mention these examples because they (and countless others like them) have led me to believe that you can solve most of data problems with relatively simple techniques. I’m loath to give up on answering an empirical machine learning question just because, at first glance, our data set isn’t quite textbook. What follows are a few examples of ML problems that at one point seemed insurmountable but that can be tackled with some straightforward solutions.

Problem 1: Your model becomes its own adversary

Adversarial machine learning is a fascinating subfield of ML that deals with model-building within a system whose data changes over time due to an external adversary, i.e., someone trying to exploit weaknesses in the current model or someone who benefits from the model making a mistake. Fraud and security are two huge application areas in adversarial ML.

I work on machine learning at Stripe, a company that builds payments infrastructure for the Internet. Specifically, I build ML models to automatically detect and block fraudulent payments across our platform. My team aims to decline charges being made without the consent of the cardholder. We identify fraud using disputes: cardholders file disputes against businesses where their cards are used without their authorization.

In this scenario, our obvious adversaries are fraudsters: people trying to charge stolen credit-card numbers for financial gain. Intelligent fraudsters are generally aware that banks and

The model is its own adversary, in a loose sense, since it works against model improvements by obscuring performance metrics and depleting the supply of training data

payment processors have models in place to block fraudulent transactions, so they're constantly looking for ways to get around them — so we strive to stay recent with our models in order to get ahead of bad actors.

However, a more subtle adversary is the model itself: once we launch a model in production, standard evaluation metrics for binary classifiers (like precision and recall, described in the first article of this series) can become impossible to calculate. If we block a credit-card charge, the charge never happens and so we can't determine if it would have been fraudulent. This means we can't estimate model performance. Any increase in observed fraud rate could theoretically be chalked up to an increase in inbound fraud rather than degradation in model performance; we can't know without outcome data. The model is its own adversary, in a loose sense, since it works against model improvements by obscuring performance metrics and depleting the supply of training data. This can also be thought of as an unfortunate "missing data" problem: we're missing the outcomes for all of the charges the model blocks. Other ML applications suffer from the same issue: for example, in advertising, it's impossible to see whether a certain visitor to a website would have clicked an ad if it never gets shown to that visitor (based on a model's predicted click probability for that user).

Having labeled training data and model performance metrics is business critical, so we developed a relatively simple approach to work around the issue: we let through a tiny, randomly chosen sample of the charges our models ordinarily would have blocked and see what happens — i.e., observe whether or not the charge is fraudulent and fill in some of our missing data. The probability of reversing a model's decision to block a transaction is dependent on how confident the model is that the charge is fraudulent. Charges the model is less certain about have higher probabilities of being reversed; charges the model gives very high fraud probabilities are approximately never reversed. The reversal probabilities are recorded.

We can then use a statistical technique called inverse probability weighting to reconstruct a fully labeled data set of charges with labeled outcomes. The idea behind inverse probability weighting is that a charge whose outcome we know because a model's block decision was reversed with a probability of 5% represents 20 charges: itself, plus 19 other charges like it whose block decisions the model didn't reverse. So we essentially create a data set containing 20 copies of that charge. From there, we can calculate all the usual binary classifier metrics for our model: precision, recall, false-positive rate, etc. We can also estimate things like incoming fraudulent volume and create weighted training data sets for new, improved models.

Here, we first took advantage of our ability to change the way the underlying system works: we don't control who makes payments on Stripe but we're able to be creative with what happens after the payment in order to get the data we need to improve fraud detection. Our reversal probabilities, which varied with our model's certainty, reflected the business requirements of this solution: we should almost always block charges we know to be fraudulent in the in- ▶

terest of doing what's best for the businesses that depend on us for their payments. And even though the best business solution is not the ideal solution for data analysis, we made use of a classic statistical method to correct for that. Keeping smart system modifications in mind and remembering that we can often adjust our post hoc analyses were both key insights to solving this problem.

Problem 2: Error-bar calculations seem impossible

Determining the margin of error on any estimate is (a) very important, since the certainty in your estimate can very much affect how we act on that information later, and (b) often terrifyingly challenging. Standard error formulas can only get us so far; once we try to put error bars around any quantity that isn't an average, things quickly get complicated. Many standard error formulas also require some estimate of correlation or covariance — a quantification of how the data points going into the calculation are related to each other — or an assumption that those data points are independent.

I'll illustrate this challenge with a real example from the previous section: estimating the recall of our credit-card-fraud model using the inverse-probability-weighted data. We'd like to know what percentage of incoming fraud our existing production model blocks. Let's assume we have a data frame, `df`, with four columns: the charge id; a Boolean `fraud` that indicates whether or not the charge was actually fraudulent; a Boolean `predicted_fraud` that indicates whether or not our model classified the charge as fraudulent; and `weight`, the probability that we observed the charge's outcome. The formula for model recall (in pseudo-code) is then:

```
recall=((df.fraud&df.predicted_fraud).toint*df.weight)/ (df.fraud*df.weight)
```

(Note that & is an element-wise logical on the `df.fraud` and `df.predicted_fraud` vectors, and * is a vector dot product.) There isn't a known closed-form solution for calculating a confidence interval (i.e., calculating the widths of the error bars) around an estimator like that. Luckily, there are straightforward techniques we can use to get around this problem.

Computational methods for error-bar estimation work in virtually any scenario and have almost no assumptions that go along with them. My favorite, and the one I'm going to talk about now, is the bootstrap, invented by Brad Efron in 1979. Efron proved that confidence intervals calculated this way have all the mathematical properties you'd expect from a confidence interval. The main disadvantage to methods like the bootstrap is that they're computationally intensive, but it's 2017 and computing power is cheap, and what made this sometimes unusable in 1979 is basically a non-issue today.

Bootstrapping involves estimating variation in our observed data set using sampling: we take a large number of samples with replacement from the original data set, each with the same number of observations as in the original data set. We then calculate our estimated metric (recall) on each of those bootstrap samples. The 2.5th percentile and the 97.5th percentile of those estimated recalls are then our lower and upper bounds for a 95% con-

```
from numpy import percentile
from numpy.random import randint

def recall(df):
    return ((df.fraud & df.predicted_fraud).toint * df.weight) / (df.fraud *
  df.weight)

n = len(df)
num_bootstrap_samples = 10000
bootstrapped_recalls = []
for _ in xrange(num_bootstrap_samples):
    sampled_data = df.iloc[randint(0, n, size=n)]
    est_recall = recall(sampled_data)
    boostrapped_recalls.append(est_recall)

ci_lower = percentile(bootstrapped_results, 2.5)
ci_upper = percentile(bootstrapped_results, 97.5)
```

fidence interval for the true recall. Here's the algorithm in Python, assuming `df` is the same data frame as in the example below.

With techniques like the bootstrap and the jackknife, error bars can almost always be estimated. They might have to be done in batch rather than in real time, but we can basically always calculate an accurate measure of uncertainty!

Problem 3: A human must interpret a black-box model's decisions

ML models are commonly described as magic or black boxes — the important thing is what goes into them and what comes out, not necessarily how that output is calculated. Sometimes this is what we want: many consumers don't really need to see inside the sausage factory, as they say — they just want a tasty tubular treat. But other times, a prediction from a box full of magic isn't satisfying. For many production ML systems, understanding individual decisions made by the model is crucial: at Stripe, we recently made our machine learning model decisions visible to the online businesses we support, which means business owners can understand what factors led to our models' decisions to decline or allow a charge.

As noted in the introduction, random forests are a canonical example of a black-box model, and they're at the core of Stripe's fraud models. The basic idea behind a random forest is that it is composed of a set of decision trees. The trees are constructed by finding the splits or questions (e.g., "Does the country this credit card was issued in match the country of the IP address it's being used from?") that optimize some classification criterion, e.g., overall classification accuracy. Each item is run through all of the trees, and the individual tree decisions are averaged (i.e., the trees "vote") to get a final prediction. Random forests are flexible, perform well, and quickly evaluate in real-time, so they're a common choice for production ML models. But it's very hard to translate several sets of splits plus tree votes into an intuitive explanation for the final prediction.

It turns out that research has looked at this problem. Even though it likely won't be part of an introductory ML course, the body of knowledge is out there. But perhaps more importantly, this problem exemplifies the lesson that a simple solution can work really well. Again, this is 2017 and raw computing power is abundant and cheap. One way to get a rudimentary explanation for a black-box model is to write a simulation: vary one feature at a time across its domain, and see how the prediction changes — or maybe change the values of two covariates at a time and see how the predictions change. Another approach (one we used for a while here at Stripe) is to recalculate the predicted outcome probability by treating each feature in turn as missing (and non-deterministically traversing both paths whenever there is a split acting on the omitted feature); the features that change the predicted outcome probabilities the most can be considered the most important. This produced some confusing explanations, but worked reasonably well until we were able to implement a more formal solution, which we now use for explanations to our customer businesses.

With each solution we implemented, we anecdotally experienced a marked improvement in overall understanding of why specific decisions were made. ▶

For many production ML systems, understanding individual decisions made by the model is crucial.

Having explanations available was also a great debugging tool for identifying specific areas where our models were making systematic mistakes. Being able to solve this problem again highlights how we have the tools (straightforward math and a few computers) to solve business-critical ML problems, and even simple solutions were better than none. Having a "this is solvable" mindset helped us implement a useful system, and it's given me optimism about our ability to get what we need from the data we have.

Other reasons I'm optimistic

I remain hopeful about being able to solve important problems with data. In addition to the examples in the introduction and the three problems outlined above, several other small, simple, clever ways we can solve problems came to mind:

- Unwieldy data sets — If a data set is too large to fit in memory or computations are unreasonably slow because of the amount of data, down-sample. Many questions can be reasonably answered using a sample of the data set (and most statistical techniques were developed for random samples anyway).

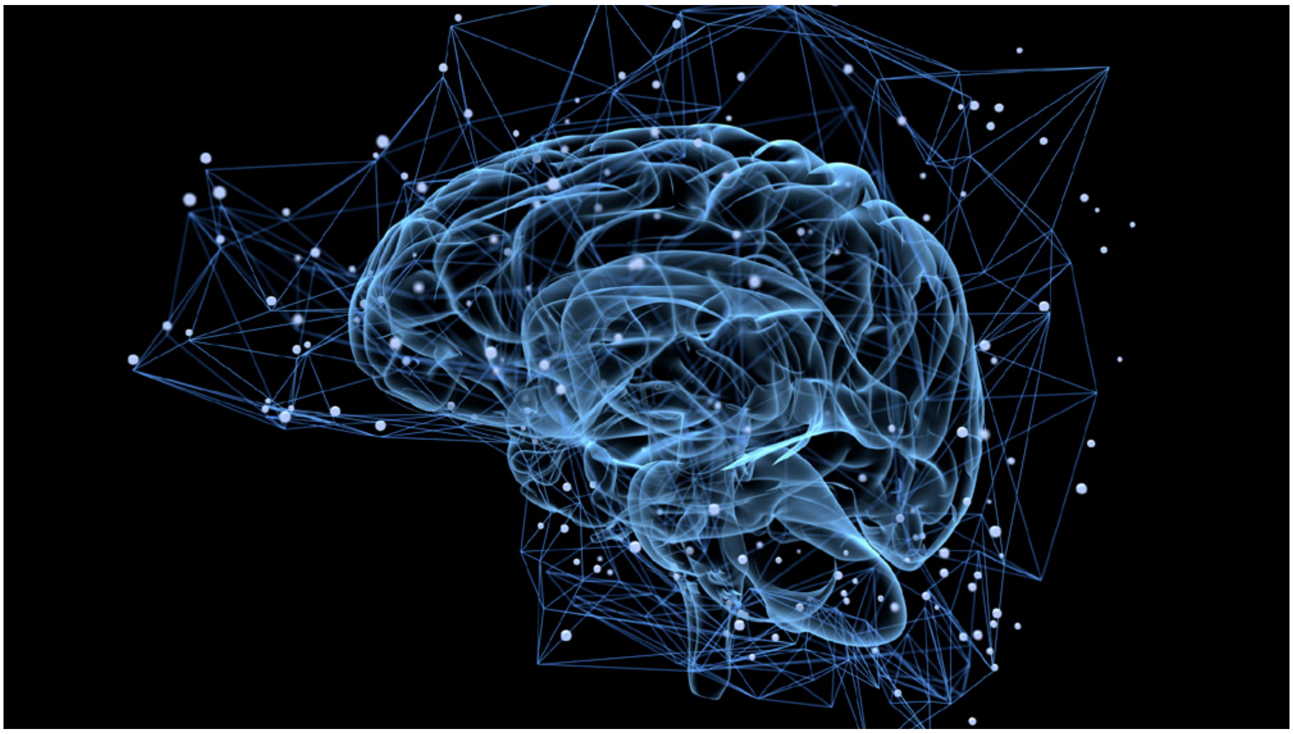
- Lack of rare events in sampled data sets — For example, I often get random samples of charges that don't contain any fraud, since fraud is a rare event. A strategy here is to take all of the fraud in the original data set, down-sample the non-fraud, and use sample weights (similar to the inverse-probability weighting discussed above) in the final analysis.

- Beautiful, clever computational tricks to calculate computationally intensive quantities — Exponentially weighted moving averages are a nice example here. Moving averages are notoriously hard to compute (since we have to keep all of the data points in the window of interest), but exponentially weighted moving averages get at the same idea and use aggregates, so are much faster. HyperLogLogs are a lovely approximation of all-time counts without needing to scan the entire data set, and HLLSeries are their really cool counterpart for counts in a specific window. These strategies are all approximations, but ML is an approximation anyway.

These are just a few data-driven ways to overcome the everyday challenges of practical machine learning.

Read online on InfoQ

Anomaly Detection for Time-Series Data with Deep Learning



Tom Hanlon is currently at Skymind, where he is developing a training program for Deeplearning4J. The consistent thread in Tom's career has been data, from MySQL to Hadoop and now neural networks.

The increasing accuracy of deep neural networks for solving problems such as speech and image recognition has stoked attention and research devoted to deep learning and AI more generally. But widening popularity has also resulted in confusion.

This article introduces neural networks, including brief descriptions of feed-forward neural networks and recurrent neural networks, and describes how to build a recurrent neural network that detects anomalies in time-series data. To make our discussion concrete, we'll show how to build a neural network using Deeplearning4j, a popular open-source deep-learning library for the JVM.

What are neural networks?

Artificial neural networks are algorithms initially conceived to emulate biological neurons. The analogy, however, is a loose one. The features of biological neurons that artificial neural networks mirror include connections between the nodes and an activation threshold, or trigger, for each neuron to fire.

By building a system of connected artificial neurons, we obtain systems we can train to learn higher-level patterns in data and to perform useful functions such as regression, classification, clustering, and prediction.

The comparison to biological neurons only goes so far. An artificial neural network is a collection of compute nodes. We pass data, represented as a numeric ▶

KEY TAKEAWAYS

Neural nets are a type of machine learning model that mimic biological neurons—data comes in through an input layer and flows through nodes with various activation thresholds.

Recurrent neural networks are a type of neural net that maintain internal memory of the inputs they've seen before, so they can learn about time-dependent structures in streams of data.

array, into a network's input layer and the data proceeds through the network's so-called hidden layers until the network generates an output or decision about the data. We then compare the net's resulting output to expected results (ground-truth labels applied to the data, for example) and use the difference between the network's guess and the right answer to incrementally correct the activation thresholds of the net's nodes. As we repeat this process, the net's outputs converge on the expected results.

A whole neural network of many nodes can run on a single machine. It is important to note, for those coming from distributed systems, that a neural network is not necessarily a distributed system of multiple machines. Node, here, means "a place where computation occurs".

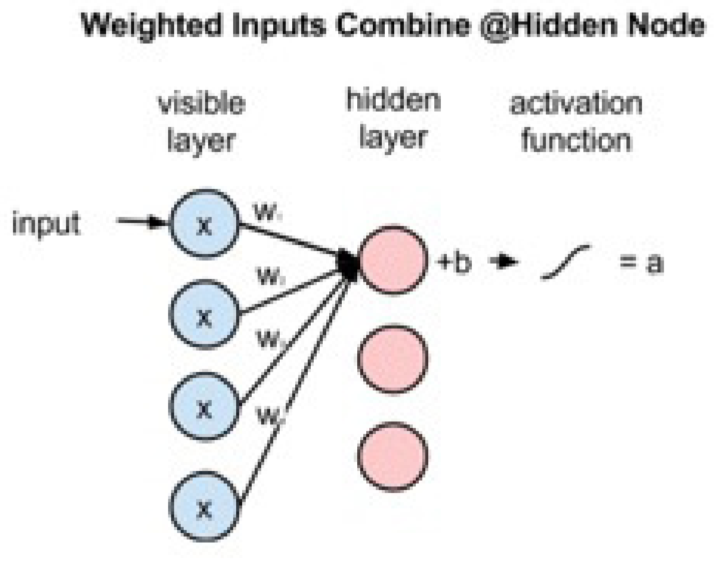
Training process

To build a neural network, we need a basic understanding of the training process and how the net's generates output. While we won't go deep into the equations, a brief description follows.

The net's input nodes receive a numeric array, perhaps a multidimensional array called a tensor, that represents the input data. For example, each pixel in an image may be represented by a scalar that is then fed to a node. That input data passes through the coefficients, or parameters, of the net and through multiplication, those coefficients will amplify or mute the input, depending on its learned importance — i.e., whether or not that pixel should affect the net's decision about the entire input.

Initially, the coefficients are random; i.e., the network is created knowing nothing about the structure of the data. The activation function of each node determines the output of that node given an input or set of inputs. So the node either fires or does not, depending on whether or not the strength of the stimulus it receives, the product of the input and the coefficient, surpasses the threshold of activation.

In a so-called dense or fully connected layer, the output of each node passes to all nodes of the subsequent layer. This continues through all hidden dense layers, ending with the output layer, where the network reaches a decision about the input. At the output layer, the net's decision about the input is evaluated against the expected decision (e.g., do the pixels in this image represent a cat or a dog?). The error is calculated by comparing



While deep learning is a complicated process involving matrix algebra, derivatives, probability, and intensive hardware utilization as large matrices of coefficients are modified, the end user does not need to be exposed to all the complexity.

the net's guess to the true answer contained in a test set and that error is used to update the coefficients of the network in order to change how the net assigns importance to different pixels in the image. The goal is to decrease the error between generated and expected outputs — to correctly label a dog as a dog.

While deep learning is a complicated process involving matrix algebra, derivatives, probability, and intensive hardware utilization as large matrices of coefficients are modified, the end user does not need to be exposed to all the complexity.

There are, however, some basic parameters that we should be aware of to help understand how neural networks function. These are the activation function, optimization algorithm, and objective function (also known as the loss, cost, or error function).

The activation function determines whether and to what extent a signal should be sent to connected nodes. A frequently used activation is just a basic step function that is 0 if its input is less than some threshold and 1 if its input exceeds the threshold. A node with a step-function activation function thus either sends a 0 or 1 to connected nodes. The optimization algorithm determines how the network learns, more accurately how weights are modified after determining the error. The most commonly used optimization algorithm is stochastic gradient descent. Finally, a cost function is a measure of error, which gauges how well the neural network performed when making decisions about a given training sample, compared to the expected output.

Open-source frameworks such as Keras for Python or Deeplearning4j for the JVM make it fairly easy to get started building neural networks. Deciding which network architecture to use often involves matching our data type to a known solved problem and then modifying an existing architecture to suit our use case.

Types of neural networks and their applications

Neural networks have been known and used for many decades. But a number of important technological trends have recently made deep neural nets much more effective.

Computing power has increased with the advent of GPUs to increase the speed of the matrix operations as well as with larger distributed-computing frameworks, making it possible to train neural nets faster and iterate quickly through many combinations of hyperparameters to find the right architecture.

Larger data sets are being generated, and large, high-quality, labeled data sets such as ImageNet already exist. As a rule, the more data a machine-learning algorithm is trained on, the more accurate it will be.

Finally, advances in how we understand and build the neural-network algorithms have resulted in neural networks consistently setting new accuracy records in competitions for computer vision, speech recognition, machine translation, and many other machine-perception and goal-oriented tasks.

Although the universe of neural-network architectures is large, a few main types of networks have seen wide use. ▶

Feed-forward neural networks

A feed-forward neural network has an input layer, an output layer, and one or more hidden layers. Feed-forward neural networks make good universal approximators (functions that map any input to any output) and can be used to build general-purpose models.

This type of neural network can be used for both classification and regression. For example, when using a feed-forward network for classification, the number of neurons on the output layer is equal to the number of classes. Conceptually, the output neuron that fires determines the class that the network has predicted. More accurately, each output neuron returns a probability that the record matches that class, and the class with the highest probability is chosen as the model's output classification.

The benefit of feed-forward neural networks such as multilayer perceptrons is that they are easy to use, less complicated than other types of nets, and available in a wide variety of examples.

Convolutional neural networks

Convolutional neural networks are similar to feed-forward neural nets, at least in the way that data passes through the network. Their form is roughly modeled on the visual cortex. Convolutional nets pass several filters like magnifying glasses over an underlying image. Those filters focus on feature recognition on a subset of the image, a patch or tile, and then repeat that process in a series of tiles across the image field.

Each filter is looking for a different pattern in the visual data. For example, one filter might look for a horizontal line, another might look for a diagonal line, another for a vertical. Those lines are known as "features" and as the filters pass over the image, they construct feature maps that locate each kind of line each time it occurs at a different place in the image. Different objects in images — cats, 747s, masticating juicers — generate different sorts of features maps, which can ultimately be used to classify photos. Convolutional networks have proven very useful in the field of image and video recognition (and because sound can be represented visually in the form of a spectrogram, convolutional networks are widely used for voice recognition and machine-transcription tasks as well).

Both convolutional and feed-forward network types can analyze images, but how they analyze them is different. While a convolutional neural network steps through overlapping sections of the image and trains by learning to recognize features in each section, a feed-forward network trains on the complete image. A feed-forward network trained on images that always depict a feature in a particular position or orientation may not recognize that feature when it shows up in an uncommon position, while a convolutional network would recognize it, if trained well.

Convolutional neural networks are used for tasks such as image, video, voice, and sound recognition as well as in autonomous vehicles.

This article focuses on recurrent neural networks, but convolutional neural networks have performed so well with image recognition that we should acknowledge their utility.

Recurrent neural networks

Unlike feed-forward neural networks, the hidden layer nodes of a recurrent neural network (RNN) maintain an internal state, a memory, that updates with new input fed into the network. Those nodes make decisions based both on the current input and on what has come before. RNNs can use that internal state to process relevant data in arbitrary sequences of inputs, such as time series.

APPDYNAMICS

Secure your app's future.

Make better decisions with deep insights into performance.

Take a Tour

RNNs are used for handwriting recognition, speech recognition, log analysis, fraud detection, and cybersecurity.

They are best for data sets that contain a temporal dimension like logs of web or server activity, sensor data from hardware or medical devices, financial transactions, or call records. Tracking dependencies and correlations within data over many time steps requires that we know the current state and some number of previous states. Although this might be possible with a typical feed-forward network that ingests a window of events and subsequently moves that window through time, such an approach would limit us to dependencies captured by the window, and the solution would not be flexible.

A better approach to track long-term dependencies over time is some sort of "memory" that stores significant events so that later events can be understood and classified in context. The beauty of an RNN is that the "memory" contained in its hidden layers learns the significance of these time-dependent features on its own over very long windows.

In what follows, we will discuss the application of recurrent networks to both character generation and network-anomaly detection. What makes an RNN useful for anomaly detection in time-series data is this ability to detect dependent features across many time steps.

Applying recurrent neural networks

Although our example will be monitoring activity on a computer network, it might be useful to start by discussing a simpler use case for RNNs. The Internet has multiple examples of using RNNs for generating text, one character at a time, after being trained on a corpus of text to predict the next letter given what's gone before. Let's take a look at the features of an RNN by looking more closely at that use case.

RNNs for character generation

RNNs can be trained to treat characters in the English language as a series of time-dependent events. The network will learn that one character frequently follows another ("e" follows "h" in "the", "he," and "she") and as it predicts the next character in a sequence, it will train to reduce error by comparisons with actual English text.

When fed the complete works of Shakespeare, for example, a RNN can then generate impressively Shakespeare-like output: for example, "Why, Salisbury must find his flesh and thought…." When fed a sufficiently large amount of Java code, a RNN will emit something that almost compiles.

Java is an interesting example because its structure includes many nested dependencies. Every parenthesis that opens will eventually close. Every open curly brace pairs with a closed curly brace down the line. These are dependencies not located immediately next to one another — the distance between multiple events can vary. Without being told about these dependent structures, a RNN will learn them.

In anomaly detection, we will be asking our neural net to learn similar, perhaps hidden or non-obvious patterns in data. Just as a character generator understands the structure of data well enough to generate a simulacrum of it, a RNN used for anomaly detection understands the structure of the data well enough to know whether what it is fed looks normal or not….

The example of character generation is useful to show that RNNs are capable of learning temporal dependencies over varying ranges of time. A RNN can use that same capability for anomaly detection in network activity logs. ▶

Applied to text, anomaly detection might reveal grammatical errors, because grammar structures what we write. Likewise, network behavior has a structure: it follows predictable patterns that can be learned. A RNN trained on normal network activity would perceive a network intrusion to be as anomalous as a sentence without punctuation

A sample project in network-anomaly detection

Suppose we wanted to detect network anomalies with the understanding that an anomaly might point to hardware failure, application failure, or an intrusion.

What our model will show us

The RNN will train on a numeric representation of network activity logs, feature vectors that translate the raw mix of text and numerical data in logs.

By feeding a large volume of network activity logs, with each log line a time step, to the RNN, the neural net will learn what normal and expected network activity looks like. When this trained network is fed new activity from the network, it will be able to classify the activity as normal and expected or anomalous.

Training a neural net to recognize expected behavior has an advantage, because it is rare to have a large volume of abnormal data — or to have enough to accurately classify all abnormal behavior. We train our network on the normal data we have so that it alerts us to non-normal activity in the future. We train for the opposite where we have enough data about attacks.

As an aside, the trained network does not necessarily note that certain activities happen at certain times (it does not know that a particular day is Sunday) but it does notice those more obvious temporal patterns we would be aware of, along with other connections between events that might not be apparent.

We'll outline how to approach this problem using Deeplearning4j, a widely used open-source library for deep learning on the JVM. Deeplearning4j comes with a variety of tools that are useful throughout the model development process: its DataVec is a collection of tools to assist with the extract-transform-load (ETL) tasks used to prepare data for model training. Just as Sqoop helps load data into Hadoop, DataVec helps load data into neural nets by cleaning, preprocessing, normalizing, and standardizing data. It's similar to Trifacta's Wrangler but focused a bit more on binary data.

Getting started

The first stage includes typical big-data tasks and ETL. We need to gather, move, store, prepare, normalize, and vectorize the logs. We must decide on the size of the time steps. Data transformation may require significant effort, since JSON logs, text logs, and logs with inconsistent labeling patterns will have to be read and converted into a numeric array. DataVec can help transform and normalize that data. As is the norm when developing machine-learning models, the data must be split into a training set and a test (or evaluation) set.

Training the network

The net's initial training will run on the training split of the input data.

For the first training runs, we may need to adjust some hyperparameters (hyperparameters are parameters that control the configuration of the model and how it trains) so that the model actually learns from the data, and does so in a reasonable amount of time. We discuss a few hyperparameters below. As the model trains, we should look for a steady decrease in error.

There is a risk that a neural network model will overfit on the data. A model that has been trained to the point of overfitting the data set will get good scores on the training data, but will not make accurate decisions about data it has never seen before. It doesn't "generalize" in machine-learning parlance. Deeplearning4J provides regularization tools and "early stopping" that help prevent overfitting while training.

Training the neural net is the step that will take the most time and hardware. Running training on GPUs will lead to a significant decrease in training time, especially for image recognition, but additional hardware comes with additional cost, so it's important that your deep-learning framework uses hardware as efficiently as possible. Cloud services such as Azure and Amazon provide access to GPU-based instances, and neural nets can be trained on heterogeneous clusters with scalable commodity servers as well as purpose-built machines.

Productionizing the model

Deeplearning4J provides a `ModelSerializer` class to save a trained model. A trained model can be saved and used (i.e., deployed to production) or updated later with further training.

When performing network-anomaly detection in production, we need to serialize log files into the same format that the model trained on and, based on the output of the neural network, we would get reports on whether the current activity was in the range of normal expected network behavior.

Sample code

The configuration of a recurrent neural network might look something like this:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
   .seed(123)
   .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT).
  iterations(1)
   .weightInit(WeightInit.XAVIER)
   .updater(Updater.NESTEROVS).momentum(0.9)
   .learningRate(0.005)
   .gradientNormalization(GradientNormalization.ClipElementWiseAbsoluteValue)
   .gradientNormalizationThreshold(0.5)
   .list()
   .layer(0, new GravesLSTM.Builder().activation("tanh").nIn(1).nOut(10).build())
   .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
   .activation("softmax").nIn(10).nOut(numLabelClasses).build())
   .pretrain(false).backprop(true).build();
MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
```

Let's describe a few important lines of this code.

```
.seed(123)
```

This sets a random seed to initialize the neural net's weights, in order to obtain reproducible results. Typically, coefficients are initialized randomly, so to obtain consistent results while adjusting other hyperparameters, we need to set a seed so we can use the same random weights over and over as we tune and test.

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT).iterations(1)
```

This determines which optimization algorithm to use (in this case, stochastic gradient descent) to determine how to modify the weights to improve the error score. We probably won't have to modify this.

```
.learningRate(0.005)
```

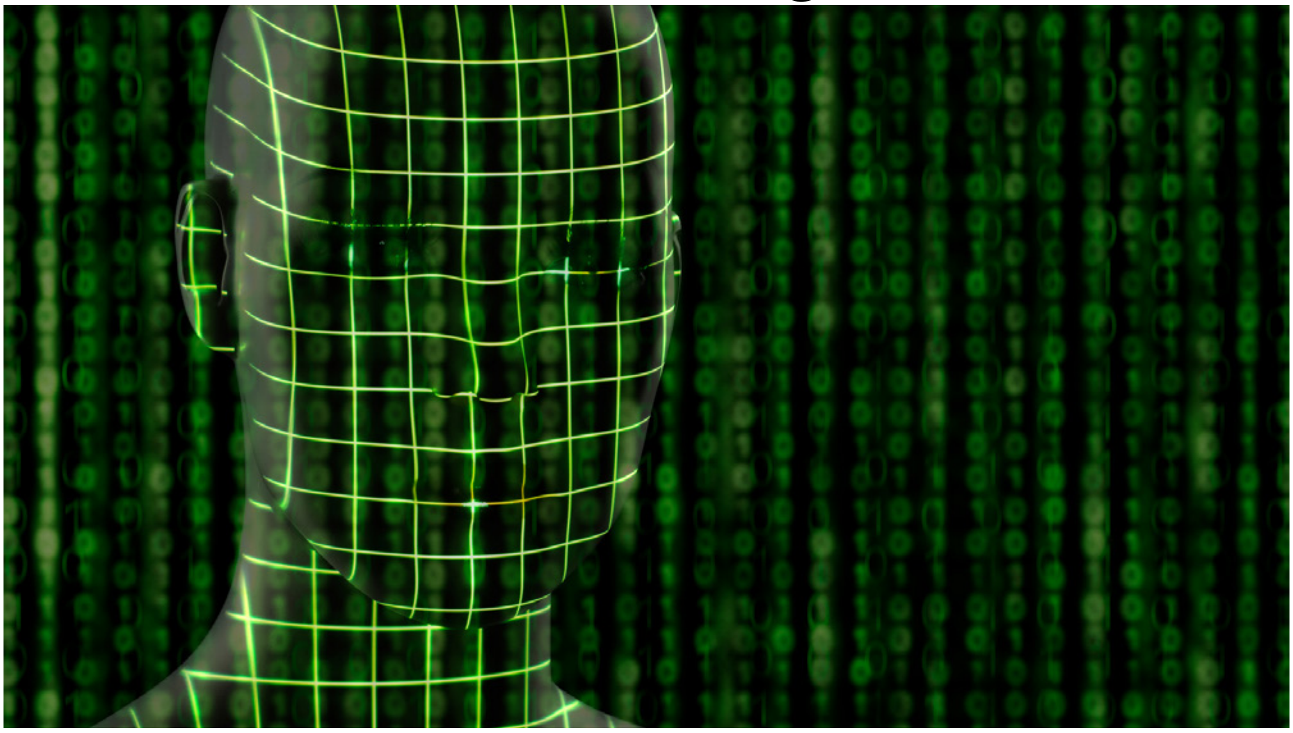
When using stochastic gradient descent, the error gradient (that is, the relation of a change in coefficients to a change in the net's error) is calculated and the weights are moved along this gradient in an attempt to move the error towards a minimum. Stochastic gradient descent gives us the direction of less error, and the learning rate determines how big of a step is taken in that direction. If the learning rate is too high, we may overshoot the error minimum; if it is too low, our training will take forever. This is a hyperparameter that we may need to adjust.

Getting help

There is an active community of Deeplearning4J users who can be found on several support channels on Gitter. ■

Read online on InfoQ

Real-World, Man-Machine Algorithms



Edwin Chen works at Hybrid, a platform for machine learning and human labor. He used to build machine-learning systems for Google, Twitter, Dropbox, and quantitative finance.

Justin Palmer is founder of topfunnel, software for recruiters, and works on Hybrid. He was most recently VP of data at LendingHome and has built ML products for speech recognition and natural language processing at Basis Technology and MITRE.

The previous articles in this eMag focused on the algorithmic part of machine learning (ML): training simple classifiers, pitfalls in classification, and the basics of neural nets. But the algorithmic part of ML is just one small part of the process of deploying a model to solve a real-world problem.

Let's talk about the end-to-end flow of developing ML models: where we get training data, how we pick the ML algorithm, what we must address after our model is deployed, and so forth.

End-to-end model deployment

There are many ML classification problems for which using log data is standard, essentially giving us labels for free. For example, ad-click-prediction models are typically trained on which ads users click on, video-recommendation systems make heavy use of which videos you've watched in the past, etc.

KEY TAKEAWAYS

Real-world machine learning isn't simply about training a model once. Getting training data is often a complicated problem and we will need continuous monitoring and retraining even after the first deployment.

In order to get training data, we often need a large group of human workers to label and annotate data. But this presents a quality-control problem, which we may need statistical monitoring to detect.

Model selection and feature selection are important but are often constrained by the amount of data we have available. Even if a model or feature doesn't work now, it may work later on as we get more data.

Our users and our products will change, and the performance of our machine-learning models will change with them. We'll need to re-gather training data, reevaluate the algorithms and features we chose, and retrain our models so try to automate these steps as much as possible.

However, even these systems need to move beyond simple click data once they reach large enough scale and sophistication; for instance, because they're heavily biased towards clicks, it can be difficult to tune the systems to show new ads and new videos to users, and so explore-exploit algorithms become necessary.

What's more, many of these systems eventually incorporate explicitly human-generated labels as well. For instance, Netflix employs over 40 people to hand-tag movies and TV shows in order to make better recommendations and generate labels like "Foreign movies featuring a strong female lead", YouTube hand-labels every ad to have better features when making ad-click predictions, and Google trains its search algorithm in part on scores that a large, internal team of dedicated raters gives to query-webpage pairs.

Suppose we're an e-commerce site like eBay or Etsy. We've starting to see a lot of spammy profiles selling Viagra, drugs, and other blacklisted products, and we want to fight the problem with machine learning. But how do we do this?

1. First, we're going to need people to label training data. We can't use logs; our users aren't flagging things for us and even if they were, they're surely wildly biased (and spammers themselves would misuse the system). But gathering training data is a generally difficult problem in and of itself. We'll need hundreds of thousands of labels, requiring thousands of hours of work. Where will we get these?

2. Next, we'll need to build and deploy an actual ML algorithm. Even with ML expertise, this is a difficult and time-consuming process: how do we choose an algorithm, how do we choose the features to input into the algorithm, and how do we do this in a repeatable manner, so that we can easily experiment with different models and parameters?

3. We can't rest after we've deployed our first spam classifier. As we get new sources of users, or spammers get more creative, the types of spam appearing on our website will quickly change so we'll need to continually rerun steps 1 and 2 — which is a surprisingly difficult process to automate, especially while maintaining the accuracy levels we need.

4. Even with a working, mature ML pipeline, we're not finished. We don't want to accidentally flag and remove legitimate users so there will always be cases of ML decision boundaries which we need a human to go and look at. But how do we build a scalable human-labor pipeline that seamlessly integrates into our ML and returns results in real time? ▶

At Hybrid, our platform for ML and large-scale human labor, we realized that we were building this complicated pipeline over and over for many of our problems, so we built a way to abstract all the complexity behind a single API call:

```
# Create your classifier
curl https://www.hybridml.com/api/classifiers
    -u API_KEY:
    -d categories="spam, not spam"
    -d accuracy=0.99

# Start classifying
curl https://www.hybridml.com/api/classify
    -u API_KEY:
    -d classifier_id=ABCDEFG
    -d text="Come buy the latest Viagra at 50% off."
```

Behind the scenes, the same call automatically and invisibly decides whether a ML classifier is reliable enough to classify the example on its own or whether it needs human intervention. Models get built automatically, they're continually retrained, and the caller never has to worry whether more data is needed.

In the rest of this article, we'll go into more detail on the problems we described above — problems that are common to all efforts to deploy ML to solve real-world problems.

Labels for training

In order to train any spam classifier, we first need a training set of "spam" and "not spam" labels. One way to provide these is to use our site's visitors and logs. Just add a button that allows visitors to mark profiles as spam and use the results as a training set.

However, this can be a problem for several reasons. Most of our visitors will ignore the button so our training set is likely to be very small.

It's easily gamed: spammers can simply start marking legitimate profiles as spammy.

It's also likely to be biased in unknown ways (after all, plenty of people are fooled by spammy Nigerian e-mail).

Another way to come up with training data is to label a bunch of profiles ourselves. But this is almost certainly a waste of time and resources: spam probably constitutes less than 1-2% of all profiles, so we'd need hundreds of thousands of profile classifications (and thousands of hours) in order to form a reasonable training set.

What we need, then, is a large group of workers to comb through a large set of profiles, and mark them as "spam" or "not spam" according to a set of instructions. Common ways to find workers to perform these types of tasks include hiring off of Craigslist or using online crowdsourcing platforms like Amazon Mechanical Turk, Crowdflower, or Hybrid.

However, the work generated by Craigslist or Mechanical Turk workers is often low quality; at Hybrid, we've often seen spam rates, where workers randomly click on labels, as high as 80-90%. So we'll need to monitor worker output for accuracy.

One common monitoring technique is to label a number of profiles as spam or not spam and randomly send them to your workers in order to see if the workers agree with our labels.

Another potential approach is to use statistical distribution tests to catch outlier workers. For example, imagine a simple image-labeling task: if most workers label 80% of the images as "cat" and 20% as "not cat" then a worker who labels only 45% of images as "cat" should probably be flagged.

One difficulty, though, is that workers deviate from each other in completely legitimate ways. For example, people may tend to upload more cat images during the day, or spammers may tend to operate during the night. In these cases, daytime workers will have higher "cat" and "not spam" labels compared to those who work at night. To account for this kind of natural deviation, a more sophisticated approach is to apply non-parametric Bayesian techniques to cluster worker output, which we then measure for deviations.

Model selection

Once we have enough training labels, we can start building our ML models. Which algorithm should we use? For text classification, for example, three common algorithms are naive Bayes, logistic regression, and deep neural networks.

We won't go deeply into how to choose a ML classifier, but one way to think about the difference between different algorithms is in terms of the bias/variance tradeoff: simpler models tend to perform worse than complex models with large amounts of data (they aren't powerful enough to model the problem so they have high bias), but they can often perform better when the data is limited (complex models can easily be overfit and are sensitive to small changes in the data so they exhibit high variance).

As a result, it's fine — often, actually better — to start with a simpler model or fewer features if we have only a few labels, and to add more sophistication as we get more data.

This also means that we should later re-evaluate a more powerful algorithm that is less accurate early on.

We take this approach at Hybrid. Our goal is to always have the most accurate ML, whether we have 500 data points or 500,000. As a result, we automatically transition between different algorithms: we usually start with the simpler models that perform better with limited amounts of data and switch to more powerful models as more data comes in, depending on how different algorithms perform on an out-of-sample test set.

Feature selection

There are two approaches to choosing which features to use in the ML algorithm.

The more manual approach is to think of feature selection as a pre-processing step: we score each feature independent of the ML model and only keep the top N features or the features that pass some threshold. For example, a common feature-selection algorithm is to score whether the feature has a different distribution under each class (e.g., when considering whether the word "Viagra" should be kept as a feature in an e-mail spam classifier, we can compare whether "Viagra" appears significantly more often in spam vs. non-spam e-mail) and to choose the features with the greatest differences in distributions between classes.

Another, increasingly common approach is to let the ML algorithm select features by itself. For example, logistic regression models can take a regularization parameter that effectively controls whether coefficients in the model are biased towards zero. By experimenting with different values of this parameter and monitoring accuracy on a test set, the model automatically decides which features to zero out (i.e., throw away) and which features to keep.

It's also often useful to add crossed features. For example, suppose teenagers in general and Londoners in general tend to click on ads, but teenagers in London do not. An ad-click-prediction model with a "user is teenager AND user lives in London" feature would likely perform better than a model that only contains separate teenager and Londoner features. This is one of the advantages of deep neural networks: because of their fully connected, hierarchical structure, they can automatically discover feature crosses on their own, whereas models like logistic regression need feature crosses fed into them.

Adapting to changes

The final question we'll look at is how to take care of changes in the data distribution. For example, suppose we've built a spam classifier but suddenly experience a spurt of user growth in a new country or a new spammer has decided to target our website. Because these are new sources of data, our existing classifier is unlikely to be able to accurately handle these cases.

One common practice is to gather new labels on a regular and frequent basis, but this can be inefficient: how do we know how many new labels we need to gather and what if the data distribution hasn't actually changed?

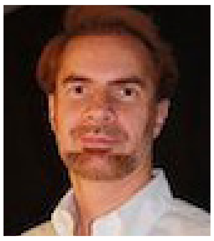
As a result, another common practice is to only gather new labels and retrain models every few months or so. But this is problematic, since quality may severely degrade in the meantime.

One solution is to randomly send examples for human labeling (e.g., less than 1% of the time once a model has reached high-enough accuracy). Doing so, we have an unbiased set of samples we can monitor accuracy against, so we can quickly detect if something has changed. You can also monitor the ML scores returned by the algorithm; if the distribution of these scores change, this is another indication the underlying data has changed and the models need a fresh regime of training. ■

Read online on InfoQ

Book Review: Erik Brynjolfsson and Andrew McAfee's *The Second Machine Age*

by Charles Humble



Erik Brynjolfsson is the director of the MIT Center for Digital Business and one of the most cited scholars in information systems and economics. He is a cofounder of MIT's Initiative on the Digital Economy, along with Andrew McAfee. He and McAfee are the only people named to both the Thinkers 50 list of the world's top management thinkers and the Politico 50 group of people transforming American politics.

Andrew McAfee is a principal research scientist at the MIT Center for Digital Business and the author of Enterprise 2.0. He is a cofounder of MIT's Initiative on the Digital Economy, along with Erik Brynjolfsson. He and Brynjolfsson are the only people named to both the Thinkers 50 list of the world's top management thinkers and the Politico 50 group of people transforming American politics.

Erik Brynjolfsson and Andrew McAfee begin *The Second Machine Age* with a simple question: what innovation has had the greatest impact on human history?

"Innovation" is meant in the broadest sense: agriculture and the domestication of animals were innovations, as were the advent of various religions and forms of government, the printing press, and the cotton gin. But which of these changed the course of humanity the most (and how even is that determined)?

To start, Brynjolfsson and McAfee suggest population and measures of social development as approximate yardsticks. Using either of them, the arc of human history decisively moves "up and to the right" (as Silicon Valley startups would have of all of their metrics) starting around 1765. The authors argue that the trigger for this growth was James Watt's steam engine, a gener-

KEY TAKEAWAYS

A combination of exponential growth in computing power and the increasing digitization of all our data is propelling the recent advances in technology (most of which are advances in machine learning).

There are no clear measures for the impact of the major technological advances of recent years — traditional measures like GDP are inadequate.

The "second machine age" will have increasing economic inequality as a side effect because of the winner-takes-all nature of digital markets.

al-purpose technological innovation more than three times as efficient as its predecessors and one that essentially kicked off the Industrial Revolution.

Brynjolfsson and McAfee, researchers at the MIT Center for Digital Business who have made careers studying the impact of the Internet on business, believe that we're on the precipice of another such revolution — a second "machine age" — and provide some anecdotal evidence for this. These examples all have the same form: a decade ago we were frustratingly far from progress in the area and almost overnight, the problems had been solved (generally by advances in machine learning). The work here progressed in the same way that Ernest Hemingway described how people go bankrupt in *The Sun Also Rises*: "gradually, then suddenly".

Among the examples are self-driving cars, now completely unremarkable on the freeways of Northern California, only a decade ago seemed out of reach. As recently as 2004, the DARPA Grand Challenge to build a car that could autonomously navigate a course in the desert ended disastrously, with all the entrants failing just a few hours in (Popular Science derided the competition as a "Debacle in the Desert"). There was also IBM's Jeopardy-winning Watson, which thoroughly demolished the two most successful human Jeopardy contestants. Watson absorbed massive amounts of information, including the entirety of Wikipedia, and was able to answer instantaneously and correctly even when the clues involved typical-for-Jeopardy puns and indirection (it correctly offered "pentathlon" as the answer to "A 1976 entree in the 'modern' this was kicked out for wiring his epee to score points without touching his foe"). And although it was developed after the book was published, we could add Deepmind's AlphaGo, the first Go program ever to beat a professional player. In October 2015, AlphaGo defeated the reigning three-time European champion Fan Hui 5-0, and in March 2016, it defeated Lee Sedol, the top Go player in the world over the past decade, 4-1. Because Go is so combinatorially complex — on average, the number of possible moves a player can make is almost an order of magnitude more than the equivalent number in chess — it was generally believed that we were still several years away from achievements like those of AlphaGo.

Why has the progress here been so sudden in the past several years? One plausible, specific answer for many of these advances goes unmentioned: developments in neural networks and deep learning. But Brynjolfsson and McAfee focus on three higher-level explanations.

First, there's the exponential growth described by Moore's Law: transistor density doubles every 18 months. Citing Ray Kurzweil's rough rule of thumb that things meaningfully change after 32 doublings (once you're in the "second half of the chess board") and the fact that the Bureau of Economic Analysis first cited "information technology" as a corporate investment category in 1958, the authors peg 2006 as when Moore's Law put us into a new regime of computing.

Second, there's the trend of the digitization of everything: maps, books, speech — they're all being stored digitally in a form that's amenable for processing and analysis. For example, the navigation app Waze uses several streams of information: digitized street maps, location coor- ▶

So, it seems, we're on the brink of a revolution — these mind-boggling technologies being anecdotal evidence of that — but how will that revolution mani- fest?

dinates for cars broadcast by the app, and alerts about traffic jams, among others. It's Waze's ability to bring these streams together and make them useful for its users that causes the service to be so popular.

Digitized information is so powerful because it can be reproduced without cost and therefore can be used in innumerable applications.

Finally, the authors describe innovation as being driven by a recombination of existing technologies.

The Web itself is a pretty straightforward combination: the Internet's much older TCP/IP data-transmission network; a markup language called HTML that specified how text, pictures, and so on should be laid out; and simple software called a "browser" to display the results. None of these elements was particularly novel. Their combination was revolutionary.

As the Internet facilitates the availability of information and other resources, this process of recombination accelerates. Brynjolfsson and McAfee write: "Today, people with connected smartphones or tablets anywhere in the world have access to many (if not most) of the same communications resources and information that we do while sitting in our offices."

So, it seems, we're on the brink of a revolution — these mind-boggling technologies being anecdotal evidence of that — but how will that revolution manifest? A growth in population like the one that attended the industrial revolution is impossible, so is this a revolution just of awe and wonder or is there a measure that captures just how fast and meaningfully these technologies are changing the world? While the authors talk about the inadequacy of traditional economic measures to capture the change (GDP in particular is the bugbear here: "When a business traveler calls home to talk to her children via Skype, that may add zero to GDP, but it's hardly worthless"), they do not offer a clear metric for at least the positive impact (or "bounty") of recent progress.

On the other hand, Brynjolfsson and McAfee do an admirable job of talking concretely about at least one of the negative impacts of all this change: economic inequality (or what they call the "spread"). "Digital technologies can replicate valuable ideas, insights, and innovations at very low cost," they write. "This creates bounty for society and wealth for innovators, but diminishes the demand for previously important types of labor, which can leave many people with reduced incomes."

To those who may argue that tax policy, the influence of the finance industry, or social norms are the source of growing inequality, the authors note that inequality in Sweden, Finland, and Germany has actually increased more rapidly over the past 20 or 30 years than it has in the U.S. Technology is the culprit here, and it has been more disruptive in recent years for two reasons.

The primary reason is that work in digital goods, machine-learning algorithms, Internet software, and so forth is not subject to capacity constraints. The best manual laborer can only sell so many hours of his or her work, leaving opportunities for the second-best laborer (though at an appropriately lower rate). On the other hand, a software programmer who writes a slightly better mapping application — one that loads a little faster, has slightly more complete data, or prettier icons — might completely dominate a market. There would be little, if any, demand for the tenth-best mapping application, even if it got the job done almost as well.

This effect is magnified by globalization, the second reason. Local leaders, who previously could safely serve their users, are now getting disrupted by global leaders: a locally produced mapping application has no advantage over Google Maps whereas a local plumber is not in danger of competition from a better, foreign plumber.

While the book thoroughly discussed inequality as a consequence of recent advances in technology in general and artificial intelligence in particular, I felt the arguments and coverage were weaker in two areas. First, the policy recommendations were mostly quite generic (almost admittedly so as the authors referred to them as "Econ 101" policies). These included suggestions to focus on schooling (emphasizing "ideation, large-frame pattern recognition, and complex communication instead of the three Rs"), to encourage startups, to support science and immigration, and to upgrade infrastructure. While these are all sound policy suggestions, they are generically good and don't specifically address the issues around new artificial intelligence. Their recommendations for the long term do try to be a little more targeted towards the employment impact of new technology, but the authors seem somewhat fatalistic: perhaps a basic income or a negative income tax, they suggest, could help all those who will be displaced. Second, while inequality is a major issue, the authors discuss other difficult problems only in passing in the closing pages of the book. These include issues of privacy, fragility in highly coupled systems, and the possibility of the "singularity" and machine self-awareness.

The Second Machine Age was first published in 2014 (and issued in paperback in 2016), and it feels like it just barely missed deep learning as a framework for understanding why progress has been so significant recently and for anticipating upcoming issues and challenges. In a summary of research that now feels oddly archaic, the authors write that "innovators often take cues from biology as they're working, but it would be a mistake to think that this is always the case, or that major recent AI advances have come about because we're getting better at mimicking human thought.

"Current AI, in short, looks intelligent, but it's an artificial resemblance. That might change in the future."

Indeed it has, and we're beginning to see all the consequences of these changes. ■

APPDYNAMICS

Secure your app's future.

Make better decisions with deep insights into performance.

Take a Tour