Intelligent Services
Serving Machine Learning

Joseph E. Gonzalez
jegonzal@cs.berkeley.edu; Assistant Professor @ UC Berkeley
joseph@dato.com; Co-Founder @ Dato Inc.
Contemporary Learning Systems

Big Data → Training → Big Models
Contemporary Learning Systems

- scikit-learn
- Dato Create
- MLbase
- TensorFlow
- WEKA
- Spark
- MLlib
- LIBSVM
- VW
- torch
- PredictionIO
- Oryx 2
- mlpack
- theano
- H2O
What happens after we train a model?

Data → Training → Model

- Conference Papers
- Dashboards and Reports
- Drive Actions
What happens after we train a model?

Data → Training → Model

- Conference Papers
- Dashboards and Reports
- Drive Actions
Suggesting Items at Checkout

Fraud Detection

Cognitive Assistance

Internet of Things

Low-Latency

Personalized

Rapidly Changing
Actions

Adapt

Data

Train

Model

Serve
Machine Learning → Intelligent Services
The Life of a Query in an Intelligent Service

Request: Items like x
New Page Images ...
Feedback: Preferred Item

Web Serving Tier

Content Request

Lookup Model
Feature Lookup
Feature Lookup
Product Info
User Data
Model Info

Top-K Query
Top Items
Feedback

math
α
∫
σ
μ
ρ
β
Σ
n
Essential Attributes of Intelligent Services

Responsive
Intelligent applications are interactive

Adaptive
ML models out-of-date the moment learning is done

Manageable
Many models created by multiple people
Responsive: Now and Always

Compute predictions in < 20ms for complex

Models

Queries

Features

SELECT * FROM users JOIN items, click_logs, pages WHERE ...

under heavy query load with system failures.
Experiment: End-to-end Latency in Spark MLlib

1. To JSON
2. HTTP Req.
3. Feature Trans.
4. Evaluate Model

5. HTTP Response
6. Encode Prediction
Figure 2: Distribution of Feature Latency in Milliseconds

Figure 3: Cold Start

Workloads generated from the newsgroups dataset, one with 200 tasks and one with 10,000 tasks. These two workloads have the same amount of training data per-task, but lead to different aggregate training datasets for training the feature functions. Figure 5a demonstrates that when the feature functions have not fully converged, retraining the features can lead to significant improvements in overall prediction accuracy. However, as the features are trained on larger aggregate datasets, they converge and improvements in prediction accuracy come almost exclusively from retraining the per-task ensemble models. We repeated the 200-task experiment with a digits benchmark dataset and find that on the comparatively simpler feature models in digits, even aggregate data from 200 tasks is enough for the feature functions to converge.

We also evaluated the cost of retraining a feature function on the aggregate dataset compared with just training a single merge operator. Figure 6 shows that retraining merge operators can be performed in milliseconds, approximately the same latency as making a prediction. Retraining the feature functions as well increases the model retrain latency by 4 orders of magnitude even for simple models on relatively small aggregate datasets.

Feature functions should be retrained periodically as additional aggregate training data accumulates. However, retraining even simplest feature models with every new training point provides little improvement in accuracy and leads to a huge increase in the training cost. These results provide substantial support for our decision to separate model training into the dual components of shared rich feature models and simple high-resolution ensemble models and treat training of each component separately.

5.4 Feature Caching

To understand the ability of caching to improve prediction latency, we measured the distribution of cache-lookup latencies for comparison to the feature evaluation latencies in Figure 2. We measured the distribution of cache lookup time over 1000 trials. Each feature cache lookup uses the feature's hash function to compute the key then checks for the computed key's presence in the feature hash. We used a Python dict for the cache and the SHA1 hash of the entire input as the hash function. The cache was pre-populated with about 40 MB (2,434,805 keys), including a random subset of digit feature vectors from the digits benchmark. For each trial, we looked up the feature's hash value and counted the latency in milliseconds.

### Count out of 1000

**NOP** (Avg = 5.5, P99 = 20.6)

**Single Logistic Regression** (Avg = 21.8, P99 = 38.6)

**Decision Tree** (Avg = 22.4, P99 = 63.8)

**One-vs-all LR (10-class)** (Avg = 137.7, P99 = 217.7)

**100 Tree Random Forrest** (Avg = 50.5, P99 = 73.4)

**500 Tree Random Forrest** (Avg = 172.56, P99 = 268.7)

**AlexNet CNN** (Avg = 418.7, P99 = 549.8)

End-to-end Latency for Digits Classification
784 dimension input
Served using MLlib and Dato Inc.
<table>
<thead>
<tr>
<th>Model</th>
<th>Latency in Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predict Avg</td>
<td>4.3</td>
</tr>
<tr>
<td>Is &quot;4&quot; LR</td>
<td>21.8</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>22.4</td>
</tr>
<tr>
<td>10-Class LR</td>
<td>137.7</td>
</tr>
<tr>
<td>100 Random Forrest</td>
<td>50.5</td>
</tr>
<tr>
<td>500 Random Forrest</td>
<td>172.6</td>
</tr>
<tr>
<td>C++ AlexNet</td>
<td>418.7</td>
</tr>
</tbody>
</table>
Adaptive to Change at All Scales

- Population
- Granularity of Data
- Session

- Months
- Rate of Change
- Minutes

Shopping for Mom
Shopping for Me
Adaptive to Change at All Scales

Population

Law of Large Numbers ➔ Change Slow
Rely on efficient offline retraining ➔ High-throughput Systems

Months
Adaptive to Change at All Scales

Small Data $\rightarrow$ Rapidly Changing
Low Latency $\rightarrow$ Online Learning
Sensitive to feedback bias
I once looked at cameras on Amazon …

Opportunity for Bandit Algorithms

Bandits present new challenges:
• computation overhead
• complicates caching + indexing
Exploration / Exploitation Tradeoff

Systems that can take actions can adversely bias future data.

Opportunity for Bandits!

Bandits present new challenges:
- Complicates caching + indexing
- tuning + counterfactual reasoning
Management: Collaborative Development

Teams of data-scientists working on similar tasks

➤ “competing” features and models

Complex model dependencies:
Velox Model Serving System

Focuses on the multi-task learning (MTL) domain

Spam Classification

\[ f_1(\text{spam}) \rightarrow \text{SPAM} \]

\[ f_2(\text{spam}) \rightarrow \text{ham} \]

Content Rec. Scoring

Session 1:

\[ f_1(\text{spam}) \rightarrow \star \star \]

Session 2:

\[ f_2(\text{ham}) \rightarrow \star \star \]

Localized Anomaly Detection

\[ f_1(\text{normal}) \rightarrow \text{normal} \]

\[ f_2(\text{anomaly}) \rightarrow \text{anomaly} \]
Velox Model Serving System

Personalized Models (Multi-task Learning)

"Separate" model for each user/context.
Velox Model Serving System

Personalized Models (Multi-task Learning)

[CIDR'15, LearningSys'15]
Hybrid Offline + Online Learning

Update feature functions \textit{offline} using batch solvers
- Leverage high-throughput systems (Apache Spark)
- Exploit slow change in population statistics

\[ f(x; \theta)^T \]

Update the user weights \textit{online}:
- Simple to train + more robust model
- Address rapidly changing user statistics
we analyzed the Wikipedia edit history through 3 January, 2008 [18].

This workload proxy, we compare the cache hit rate for three caching

diction tasks for each editor as identified by their username. With

as a proxy for a live serving workload.

temporal information. As a proxy for an edit fraud prediction task

levels of materialization, we need a realistic access workload with

anywhere from 33-1000x speedup over feature evaluation.

take at least 20 ms to compute, meaning that a cache hit provides

In comparison, even the cheapest feature functions we evaluated

cache implementation the maximum cache lookup time was

mark being found in the cache. Even on this relatively unoptimized

input 10 times to simulate all 10 feature values for the digits bench-

specific ensemble.

orders of magnitude more expensive than retraining only the task-

Figure 6: We treat each article title as a unique hash-key and create a pre-

Figure 5: To better understand the range of cache-hit rates across different

son of cache hit rates for these strategies is in Figure 8.

model-services with separate caches for each service. The compari-

compared the cache hit rate of feature cache shared between two

a new article is edited. Within the context of feature-caching, we

cache for each feature function, so a cache miss only occurs when

history as a prediction-only workload. Feature caching uses a single

time a task model is updated, the prediction cache must be com-

article-editor combination causes a cache miss. Furthermore, every

diction task (each editor in this case) so every previously-unseen

strategies. Prediction caching uses a separate cache for each pre-

Figure 7: Retraining Feature Functions:

Retraining latency: Retraining shared features is several

(a) Newsgroups 200 Tasks

(b) Newsgroups 10,000 Tasks

(c) Digits 200 Tasks

(a) Total Concept Drift

(b) Partial Concept Drift

(c) Stationary Tasks

Hybrid Online + Offline Learning Results

Similar Test Error

Substantially Faster Training

User Pref. Change

Error rate

Hybrid

Offline Full

Offline Full

Hybrid

train points per task

retrain latency (ms)

10^4

10^3

10^2

10^1

10^0

10^{-1}

10^{-2}

10^{-3}

10^{-4}

20

25

30

35

40

45

50

20

25

30

35

40

45

50
Evaluating the Model
Evaluating the Model

- Feature Caching Across Users
- Approximate Feature Hashing
- Anytime Feature Evaluation

Cache Feature Evaluation

Split
Feature Caching

New input: $x$

Compute feature: $f(x; \theta)$

Hash input: $h(x)$

Store result in table

Feature Hash Table

<table>
<thead>
<tr>
<th>$x$</th>
<th>$f(x; \theta)$</th>
<th>$h(x)$</th>
<th>Store result in table</th>
<th>$f(x; \theta)$</th>
<th>$f(x; \theta)$</th>
</tr>
</thead>
</table>
LSH Cache Coarsening

New input \( z \neq x \)
Hash new input: \( h(z) \)
False cache collision
Use Wrong Value! \[ \rightarrow \text{LSH hash fn.} \]

Feature Hash Table

\[ f(x; \theta) \]
LSH Cache Coarsening

Locality-Sensitive Hashing:

\[ x \approx z \quad \Rightarrow \quad h(x) = h(z) \]

Locality-Sensitive Caching:

\[ f(x; \theta) \approx f(z; \theta) \quad \Rightarrow \quad f(h(x)) = h(z) \]
Anytime Predictions

Compute features asynchronously:

\[
\begin{align*}
&\quad w_{u1} + \\
&\quad \text{if a particular element does not arrive use estimator instead}
\end{align*}
\]

Always able to render a prediction by the latency deadline
Coarsening + Anytime Predictions

\[ f_i(x; \theta) \approx f_i(z; \theta) \]

Better

Checkout our poster!
Part of Berkeley Data Analytics Stack

Spark Streaming

BlinkDB
Spark SQL
Graph X
MLbase
ML library

Model Manager
Prediction Service

Velox

Spark
Mesos
Tachyon
HDFS, S3, ...
Dato Predictive Services

Production ready model serving and management system

- Elastic scaling and load balancing of docker.io containers
- AWS Cloudwatch Metrics and Reporting
- Serves Dato Create models, scikit-learn, and custom python
- Distributed shared caching: scale-out to address latency
- REST management API: Demo?
Responsive  Adaptive  Manageable

Key Insights:

Caching, Bandits, & Management  Online/Offline Learning Latency vs. Accuracy
Future of Learning Systems

Actions

Adapt

Data

Train

Model

Serve
Thank You

Joseph E. Gonzalez

jegonzal@cs.berkeley.edu, Assistant Professor @ UC Berkeley

joseph@dato.com, Co-Founder @ Dato