E6895 Advanced Big Data Analytics Lecture 5:

Massive Data Processing

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science

February 18th, 2022
Massive Stream Analysis Challenges
Example IP Packet Stream Instantiation

Dataflow Graph

By IBM Dense Information Gliding Team
Semantic MM Filtering

- Inputs:
  - IP
  - HTTP
  - FTP
  - Audio
  - Session
  - Video
  - NTP

- Dataflow Graph:
  - Packet content analysis
  - Interest Routing
  - Interest Filtering
  - Keywords
  - Advanced content analysis
  - Interested MM streams

- Per PE rates:
  - 200-500MB/s
  - ~100MB/s
  - 10 MB/s
Resource-Accuracy Trade-Offs

Configurable Parameters of Processing Elements to maximize relevant information:

\[ Y''(X \mid q, R) > Y'(X \mid q, R), \]

with resource constraint.

Required resource-efficient algorithms for:

- Classification, routing and filtering of signal-oriented data: (audio, video and, possibly, sensor data)

- **Input data X – Queries q – Resource R**
  - \( Y(X \mid q) \): Relevant information
  - \( Y'(X \mid q, R) \) · \( Y(X \mid q) \): Achievable subset given R
Example: Distributed Video Signal Understanding (Lin et al.)

(Distributed Smart Sensors) Block diagram of the smart sensors

- **Encoding**
  - MPEG-1/2
  - 1.5 Mbps

- **GOP Extraction**
  - 320 Kbps

- **Feature Extraction**
  - 22.4 Kbps

- **Event Extraction**
  - 2.3 Kbps

- **Resource Constraints**
  - User Interests

- **Control Modules**
  - Display and Information Aggregation Modules

- **Server**
  - Concept Detection Processing Elements

- **Meta-data**
  - 600 bps

- **TV broadcast, VCR, DVD discs, Video File Database, Webcam**

- **Sensor 1**
- **Sensor 2**
- **Sensor 3**
- **Sensor N**

- **Control Modules**
  - PE1: 9.2.63.66: 1220
  - PE2: 9.2.63.67
  - PE3: 9.2.63.68
  - PE4: 9.2.63.66:1235
  - PE5: 9.2.63.66: 1240
  - PE6: 9.2.63.66
  - PE7: 9.2.63.66
  - PE100: 9.2.63.66

- **Features**
  - Face
  - Female
  - Outdoors
  - Male
  - Indoors
  - Airplane
  - Chair
  - Clock
Semantic Concept Filters

E.g.:
Complexity Reduction Introduction

- Objective: Real-time classification of instances using Support Vector Machines (SVMs)
- Computationally efficient and reasonably accurate solutions
- Techniques capable of adjusting tradeoff between accuracy and speed based on available computational resources
SVM formulation

- **Given:**
  - Training instances \( \{x_i\} \) with labels \( y_i \)

- **Objective:**
  - Find maximum margin hyperplane separating positive and negative training instances
Decision

- Score of unseen instance \( u_j : w \cdot \phi(u_j) \)
- In terms of Lagrangian multipliers
  \[ \sum_i \alpha_i y_i k(x_i, u_j) \]
- Computational Cost: \( O(n_{sv}d) \)
  - \( n_{sv} \): Number of support vectors
  - \( d \): Dimensionality of each data instance
Problems

- Number of support vectors grows quasi-linearly with size of training set [Tipping 2000]
- Inner product with each support vector of dimensionality $d$ expensive
  - Example TREC2003
    - Human: 19745 support vectors
    - Face: 18090
- High data rates (10 Gbits/sec) means large number of abandoned data
Example

- Processing Power 1 Ghz
- 10000 support vectors
- 1000 / 2 features per instance
- Order of at least $10^7$ operations required per stream per sec
- Translates to less than 100 instances evaluated per sec with only one classifier
Naïve Approach I – Feature Dimension Reduction

- Experimental Results for Weather_News Detector
- Model Selection based on the Model Validation Set
- E.g., for Feature Dimension Ratio 0.22, (the best selection of features are: 3 slices, 1 color, 2 texture selections), the accuracy is decreased by 17%.

---

<table>
<thead>
<tr>
<th>Slice</th>
<th>Color</th>
<th>Texture</th>
<th>Feature Dimension Ratio</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0.7861</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0.666666667</td>
<td>0.7861</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0.666666667</td>
<td>0.7757</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0.666666667</td>
<td>0.5822</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0.444444444</td>
<td>0.7757</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0.444444444</td>
<td>0.5822</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0.444444444</td>
<td>0.5235</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0.333333333</td>
<td>0.4685</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0.333333333</td>
<td>0.6581</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0.333333333</td>
<td>0.1684</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.296296296</td>
<td>0.5235</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0.222222222</td>
<td>0.427</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0.222222222</td>
<td>0.6581</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0.222222222</td>
<td>0.1241</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0.222222222</td>
<td>0.3451</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0.222222222</td>
<td>0.1684</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.222222222</td>
<td>0.1065</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0.148148148</td>
<td>0.0699</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.148148148</td>
<td>0.3457</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.148148148</td>
<td>0.1065</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.111111111</td>
<td>0.3219</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.111111111</td>
<td>0.0314</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0.111111111</td>
<td>0.07</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0.074074074</td>
<td>0.0318</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.074074074</td>
<td>0.0173</td>
</tr>
</tbody>
</table>
Naïve Approach II – Reduction on the Number of Support

Proposed Novel Reduction Methods:
- Ranked Weighting
- P/N Cost Reduction
- Random Selection
- Support Vector Clustering and Centralization

Experimental Results on Weather_News Detectors show that complexity can be at 50% for the cost of 14% decrease on accuracy
Weighted Clustering Approach

- Basic steps
  - Cluster support vectors
  - Use cluster center as representative for all support vectors in cluster
  - Determine scalar weight associated with each cluster center
  - Use only cluster centers to score new instances
Cluster center weight (contd.)

- **Choose** $\gamma_i$ minimizing square of difference in scores over all $\pm_i$ and $d$

- **Sub-cases:**

\[ d \geq \Delta_i \quad \text{and} \quad \Delta_i < d \]

![Diagram showing cluster center weight](image)
Using the weights

- For every support vector in cluster
  - Distance $\Delta_i$ known
  - Two weights computed

- Cumulative effect of all support vectors in clusters additive
  - $\Delta_i$ because of various support vectors added up at center to simulate effect of all support vectors

- $\Delta_i$ sorted, weight arrays rearranged
Experiments

• Datasets
  • TREC video datasets (2003 and 2005)
    • 576 features per instance
    • > 20000 test instances overall
  • MNist handwritten digit dataset (RBF kernel)
    • 576 features
    • 60000 training instances, 10000 test instances

• Performance metrics
  • Speedup achieved over evaluation with all support vectors
  • Average precision achieved
Results (Mnist 0-4)
Results (Mnist 5-9)

Average Precision vs. Speedup Ratio

- Average Precision values: 0.7, 0.775, 0.85, 0.925, 1
- Speedup Ratio values: 1, 10, 100, 1000

Graph shows the relationship between Average Precision and Speedup Ratio for different values.
Results (TREC 2003)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Average Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>0.85</td>
</tr>
<tr>
<td>Outdoors</td>
<td>0.7</td>
</tr>
<tr>
<td>Sport-Event</td>
<td>0.6</td>
</tr>
<tr>
<td>Crowd</td>
<td>0.45</td>
</tr>
<tr>
<td>People-Event</td>
<td>0.25</td>
</tr>
</tbody>
</table>

- **AP_fast**
- **AP_original**

<table>
<thead>
<tr>
<th>Concept</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>10000</td>
</tr>
<tr>
<td>Studio-Setting</td>
<td>1000</td>
</tr>
<tr>
<td>Crowd</td>
<td>100</td>
</tr>
</tbody>
</table>
Summary of Complexity Reduction

- Techniques presented demonstrate reasonable performance in terms of both speedup and average precision over multiple concepts in datasets
- Speedups
  - MNist: All concepts at least 50 times faster with AP within 0.04 of original
  - TREC 2003: Eight out of nine concepts speedup greater than 80 times with AP within 0.05 of original
  - TREC 2005: APs in some cases along with speedup respectable
- APs of most concepts close to original APs
Acceleration of Neural Networks
Neuron Importance Score Propagation (NISP, Yu et al 2018)
Methods for Running CNNs on Mobile Devices

- Sending CNN jobs to cloud
- Compression (pruning) of CNN
- Speeding up CNN
- Sending CNN jobs to cloud
- Compression (pruning) of CNN
- Speeding up CNN
- Sending CNN jobs to cloud
- Compression (pruning) of CNN
- Speeding up CNN
Thinking Differently

- All existing methods can be viewed as approximations of an overly-redundant CNN. But do we really need such a CNN as the starting point?

CNN Sliming!
Slim CNN

- Slim CNN leads to:
  - less storage space
  - less memory usage
  - less computation
  - less power consumption
Feature Selection on CNN

- CNNs can be viewed as a set of "overly-redundant" feature extractors.
A method for Pruning Redundant Neurons and Kernels of

Apply thermal

1. A pre-trained CNN
2. Extract CNN Responses
3. Measure the Importance of Feature Extractors
4. Prune Model
5. Fine-tuning
A method for Pruning Redundant Neurons and Kernels of Deep Convolutional Neural Networks (NISP)

- Intractable → tractable
- Inconsistent → consistent

A pre-trained CNN → Extract Responses of a High-level Layer → Measure the Importance of Feature Extractors → Back-propagate the Importance & Prune Model → Fine-tuning

Forward Propagation

Input layers → Response → ... → Response → ... → Response → Important Score Back Propagation and Pruning
Fine-tuning the Pruned Model

- Our method outperforms the baselines in three aspects
  - Very small accuracy loss at the beginning ==> retains the most important neurons
  - Converges much faster than baselines
  - For LeNet on MIST, our method only decreases 0.02% top-1 accuracy with a running ratio of 50% as compared to the pre-pruned network.
Fine-tuning the Pruned Model

- The pruned model consists of important feature extractors, but will suffer loss of accuracy due to loss of redundant features

- Good starting point on the learning curve due to feature selection

- Fine-tuning the pruned model with a lower learning rate to recover the performance
Stream Analysis using Spark
Spark ML Classification and Regression

**MLlib: Main Guide**
- Basic statistics
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

**Classification**
- Logistic regression
  - Binomial logistic regression
  - Multinomial logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine
- One-vs-Rest classifier (a.k.a. One-vs-All)
- Naive Bayes

**Regression**
- Linear regression
- Generalized linear regression
  - Available families
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression
Spark Streaming
Spark Streaming

input data stream

Spark Streaming

batches of input data

Spark Engine

batches of processed data
Spark Streaming

Streaming Data Sources
- Parquet
- akka
- kafka

Static Data Sources
- mongoDB
- MySQL
- HBase
- Postgres

MLlib
Machine Learning

Spark Streaming

Spark SQL
SQL + DataFrames

Data Storage Systems
- memsql
- kafka
- elasticsearch

Figure: Overview Of Spark Streaming

https://www.edureka.co/blog/spark-streaming/
Spark Streaming

- **Basic Concepts**
  - Linking
  - Initializing StreamingContext
  - Discretized Streams (DStreams)
  - Input DStreams and Receivers
  - Transformations on DStreams
  - Output Operations on DStreams
  - DataFrame and SQL Operations
  - MLlib Operations
  - Caching / Persistence
  - Checkpointing
  - Accumulators, Broadcast Variables, and Checkpoints
  - Deploying Applications
  - Monitoring Applications
- **Performance Tuning**
  - Reducing the Batch Processing Times
  - Setting the Right Batch Interval
  - Memory Tuning
- **Fault-tolerance Semantics**
Spark Streaming Example

First, we import `StreamingContext`, which is the main entry point for all streaming functionality. We create a local `StreamingContext` with two execution threads, and batch interval of 1 second.

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. 9999).

```python
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

This `lines` DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

```python
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

```python
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```

```python
# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```
Spark Streaming Example

```
$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
```

Then, any lines typed in the terminal running the netcat server will be counted and printed on screen every second. It will look something like the following.

```
# TERMINAL 1:
# Running Netcat
$ nc -lk 9999
hello world
...
```

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
------------------------------------------------------------------------
------------------------------------------------------------------------
(hello,1)
(world,1)
...
```
Discretized Streams
Discretized Streams

- **lines DStream**: lines from time 0 to 1 → lines from time 1 to 2 → lines from time 2 to 3 → lines from time 3 to 4
  - Using a `flatMap` operation

- **words DStream**: words from time 0 to 1 → words from time 1 to 2 → words from time 2 to 3 → words from time 3 to 4
Discretized Streams

https://www.edureka.co/blog/spark-streaming/
DStream Transforms

https://www.edureka.co/blog/spark-streaming/
Output DStreams

Transformed DStream → Output Operations → Output DStream → External Systems
- Database
- File System

https://www.edureka.co/blog/spark-streaming/
DStreams Caching

https://www.edureka.co/blog/spark-streaming/
// Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._
import java.io.File

object twitterSentiment {
  def main(args: Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage: TwitterPopularTags <consumer key> <consumer secret> " + "<access token> <access token secret>
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()
    // Passing our Twitter keys and tokens as arguments for authorization
    val Array(consumerKey, consumerSecret, accessToken, accessTokenSecret) = args.take(4)
    val filters = args.takeRight(args.length - 4)

    // Set the system properties so that Twitter4j library used by twitter stream
    // Use them to generate OAuth credentials
    System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
    System.setProperty("twitter4j.oauth.accessTokenSecret", accessTokenSecret)

    val sparkConf = new SparkConf().setAppName("twitterSentiment").setMaster("local[2]")
    val ssc = new StreamingContext
    val stream = TwitterUtils.createStream(ssc, None, filters)

    // Rest of the code...
DStreams Example — Twitter Sentiment Analysis

```scala
//Input DStream transformation using flatMap
val tags = stream.flatMap { status => Get Text From The Hashtags }

//RDD transformation using sortBy and then map function
val now = current time of each Tweet
val rdd = tags.countByValue().sortBy(_._2)
val (x, now) = map(x => (x, now))
//Saving our output at ~/twitter/ directory
saveAsTextFile(s"~/twitter/$now")

//DStream transformation using filter and map functions
val tweets = stream.filter { t =>
  val tags = t.Split On Spaces .filter(_.startsWith("#")).Convert To Lower Case
tags.exists { x => true }
}
val data = tweets.map { status =>
  val sentiment = SentimentAnalysisUtils.detectSentiment(status.getText)
  val tagss = status.getHashtagEntities.map(_.getText.toLowerCase)
  (status.getText, sentiment.toString, tagss.toString())
}
data.print()
//Saving our output at ~/ with filenames starting like twitters
data.saveAsTextFiles("~/twitters","20000")
ssc.start()
ssc.awaitTermination()
```

https://www.edureka.co/blog/spark-streaming/
DStreams Example — Twitter Sentiment Analysis

Results:

The following are the results that are displayed in the Eclipse IDE while running the Twitter Sentiment Streaming program.

All the tweets are categorized into Positive, Neutral and Negative according to the sentiment of the contents of the tweets.

https://www.edureka.co/blog/spark-streaming/
Questions?