FAQs

• Term project
  • 5:00PM March 29, 2018

Today's topics

• Evaluation/Validation Techniques

• In-Memory cluster computing
  • Apache Spark

Plain Accuracy

• Classifier accuracy
  • General measure of classifier performance

\[
\text{Accuracy} = \frac{\text{Number of correct decisions made}}{\text{Total number of decisions made}}
\]

• Pros
  • Very easy to measure

• Cons
  • Cannot consider realistic cases
The Confusion Matrix

- A type of contingency table
- \( n \) classes
  - \( n \times n \) matrix
    - The columns labeled with actual classes
    - The rows with predicted classes
- Separates out the decisions made by the classifier
  - How one class is being confused for another
  - Different sorts of errors may be dealt with separately

<table>
<thead>
<tr>
<th>n (predicted)</th>
<th>y (negative)</th>
<th>n (positive)</th>
<th>y (positive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True negative</td>
<td>True negative</td>
<td>False positive</td>
<td>False negative</td>
</tr>
</tbody>
</table>

Problems with Unbalanced Classes

- Consider a classification problem where one class is rare
- Sifting through a large population of normal entities to find a relatively small number of unusual ones
- Looking for defrauded customers, or defective parts
- The class distribution is unbalanced or skewed

<table>
<thead>
<tr>
<th>Confusion Matrix of A, evaluated with 1,000 decisions</th>
<th>Confusion Matrix of B, evaluated with 1,000 decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Y</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
</tr>
</tbody>
</table>

F-measure (\( F1 \) score)

- Summarizes confusion matrix
- True positives \( TP \), False Positives \( FP \), True Negatives \( TN \), and False Negatives \( FN \)
- True positive rate (sensitivity) = \( TP/(TP+FN) \)
- False negative rate (miss rate) = \( FN/(TP+FN) \)
- \( F\)-measure = \( 2 \times \text{precision} \times \text{recall} / \left( \text{precision} + \text{recall} \right) \)
- \( \text{precision} = \frac{TP}{TP+FP} \)
- \( \text{recall} = \frac{TP}{TP+FN} \)
- Accuracy = \( (TP + TN) / (P + N) \)

<table>
<thead>
<tr>
<th>P</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>FP</td>
<td>TN</td>
</tr>
</tbody>
</table>

Large Scale Data Analytics

In-Memory Cluster Computing: Apache Spark
This material is built based on

- Spark Overview, https://spark.apache.org/docs/2.3.0/
- Spark programming guide
  - Job Scheduling
    - https://spark.apache.org/docs/2.0.0-preview/job-scheduling.html

Inefficiencies for emerging applications:

1. Data reuse
   - Data reuse is common in many iterative machine learning and graph algorithms
     - PageRank, K-means clustering, and logistic regression

2. Interactive data analytics
   - User runs multiple ad-hoc queries on the same subset of the data

Existing approaches

- Hadoop
  - Writing output to an external stable storage system
  - e.g. HDFS
  - Substantial overheads due to data replication, disk I/O, and serialization

- Pregel
  - Iterative graph computations

- HaLoop
  - Iterative MapReduce interface

- Pregel/HaLoop support specific computation patterns
  - e.g. looping a series of MapReduce steps

Distributed processing with the Spark framework

- Spark Storage
  - HDFS/file system/
  - HBase/Cassandra, etc.

- Spark standalone
- YARN
- Mesos

Large Scale Data Analytics

In-Memory Cluster Computing: Apache Spark

RDD (Resilient Distributed Dataset)
RDD (Resilient Distributed Dataset)

- Read-only, memory resident partitioned collection of records
  - A fault-tolerant collection of elements that can be operated on in parallel

- RDDs are the core unit of data in Spark
  - Most Spark programming involves performing operations on RDDs

Word Count Example

We use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file.

```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile.flatMap(s -> Arrays.asList(s.split(" "))).iterator()
.mapToPair(word -> new Tuple2<>(word, 1))
.reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

Overview of RDD

- **Lineage**
  - How it was derived from other dataset to compute its partitions from data in stable storage?
  - RDDs do not need to be materialized at all times

- **Persistence**
  - Users can indicate which RDDs they will reuse and the storage strategy

- **Partitioning**
  - Users can specify the partitioning method across machines based on a key in each record

Spark Programming Interface to RDD:

**Transformation** [1/3]

- "transformations"
  - Operations that create RDDs
    - Return pointers to new RDDs
      - e.g. map, filter, and join
  - RDDs can only be created through deterministic operations on either
    - Data in stable storage
    - Other RDDs

**Action** [2/3]

- "actions"
  - Operations that return a value to the application or export data to a storage system
    - e.g. count: returns the number of elements in the dataset
    - e.g. collect: returns the elements themselves
    - e.g. save: outputs the dataset to a storage system

**Persist** [3/3]

- "persist"
  - Indicates which RDDs they want to reuse in future operations
  - Spark keeps persistent RDDs in memory by default
  - If there is not enough RAM
    - It can spill them to disk
  - Users are allowed to,
    - store the RDD only on disk
    - replicate the RDD across machines
    - specify a persistence priority on each RDD
Example: Console Log Mining [1/3]

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop file system (HDFS) to find the cause.

- The user loads the error messages from the logs into the RAM across a set of nodes and query them interactively.

```scala
lines = spark.textFile("hdfs://...");
errors = lines.filter(_.startsWith("ERROR"));
errors.persist();
```

No work has been performed; user can use the RDD in actions.

Example: Console Log Mining [2/3]

- Users can perform further transformations and actions on the RDD.

```scala
//To count number of error messages
errors.count();
//Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count();
//Return the time fields of errors mentioning HDFS as an array (assuming time is field number 3 in a tab-separated format
errors.filter(_.contains("HDFS"))
  .map(_.split('/t')(3))
  .collect();
```

Lazy Evaluation

- Transformations on RDDs are lazily evaluated.
  - Spark will NOT begin to execute until it sees an action.
  - Spark internally records metadata to indicate that this operation has been requested.

- Loading data from files into an RDD is lazily evaluated.

- Reduces the number of passes it has to take over our data by grouping operations together.

Example: Console Log Mining [3/3]

```scala
lines = spark.textFile("hdfs://...");
errors = lines.filter(_.startsWith("ERROR"));
errors.persist();
errors.filter(_.contains("HDFS")
  .map(_.split('/t')(3))
  .collect();
```

Benefits of RDDs as a distributed memory abstraction [1/3]

- RDDs can only be created ("written") through coarse-grained transformations.

- Distributed shared memory (DSM) allows reads and writes to each memory location.

- Reads on RDDs can still be fine-grained.
  - A large read-only lookup table.

- Applications perform bulk writes.

- More efficient fault tolerance.
  - Lineage based fault recovery.

Benefits of RDDs as a distributed memory abstraction [2/3]

- Advantage of using RDDs' immutable data.

- System can mitigate slow nodes (Stragglers).
  - Creates backup copies of slow tasks.
    - without accessing the same memory.
  - Spark distributes the data over different working nodes that run computations in parallel.
  - Orchestrates communicating between nodes to integrate intermediate results and combine them for the final result.
Benefits of RDDs as a distributed memory abstraction [3/3]

- Runtime can schedule tasks based on data locality
  - To improve performance
- RDDs degrade gracefully when there is insufficient memory
  - Partitions that do not fit in the RAM are stored on disk

Applications not suitable for RDDs

- RDDs are best suited for batch applications that apply the same operations to all elements of a dataset
  - Steps are managed by lineage graph efficiently
  - Recovery is managed effectively
- RDDs would not be suitable for applications
  - Making asynchronous fine-grained updates to shared state
    - e.g., a storage system for a web application or an incremental web crawler

Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark

RDD in Spark

RDD in Spark: The Runtime

Representing RDDs

- A set of partitions
  - Atomic pieces of the dataset
- A set of dependencies on parent RDDs
- A function for computing the dataset based on its parents
- Metadata about its partitioning scheme
- Data placement
Dependency between RDDs

- **Narrow dependency**
  - Each partition of the parent RDD is used by at most one partition of the child RDD

- **Wide dependency**
  - Multiple child partitions may depend on a single partition of parent RDD

Interface used to represent RDDs in Spark

- `partitions()`
  - Returns a list of partition objects
- `preferredLocations(p)`
  - List nodes where partition p can be accessed faster due to data locality
- `dependencies()`
  - Returns a list of dependencies
- `iterator (p, parentIters)`
  - Compute the elements of partition p given iterators for its parent partitions
- `partitioner()`
  - Return metadata specifying whether the RDD is hash/range partitioned
Spark cluster and resources

- Driver program
- SparkContext
- Cluster Manager
- Executor
- Task
- Cache

Spark cluster

1/3

- Each application gets its own executor processes
  - Must be up and running for the duration of the entire application
  - Run tasks in multiple threads
- Isolate applications from each other
  - Scheduling side (each driver schedules its own tasks)
  - Executor side (tasks from different applications run in different JVMs)
- Data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system

Spark cluster

2/3

- Spark is agnostic to the underlying cluster manager
  - As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN)

Spark cluster

3/3

- Driver program must listen for and accept incoming connections from its executors throughout its lifetime
  - Driver program must be network addressable from the worker nodes
  - Driver program should run close to the worker nodes
    - On the same local area network

Cluster Manager Types

- Standalone
  - Simple cluster manager included with Spark
- Mesos
  - Fine-grained sharing option
  - Frequently shared objects for interactive applications
  - Mesos master determines the machines that handle the tasks
- Hadoop YARN
  - Resource manager in Hadoop 2

Dynamic Resource Allocation

- Dynamically adjust the resources that the applications occupy
  - Based on the workload
  - Your application may give resources back to the cluster if they are no longer used
- Only available on coarse-grained cluster managers
  - Standalone mode, YARN mode, Mesos coarse grained mode
Large Scale Data Analytics
In-Memory Cluster Computing: Apache Spark
Scheduling

Jobs in Spark application

• "Job"
  • A Spark action (e.g., save, collect) and any tasks that need to run to evaluate that action
  • Within a given Spark application, multiple parallel tasks can run simultaneously
    • If they were submitted from separate threads

Job scheduling

• User runs an action (e.g., count or save) on an RDD
  • Scheduler examines that RDD's lineage graph to build a DAG of stages to execute
  • Each stage contains as many pipelined transformations as possible
    • With narrow dependencies
  • The boundaries of the stages are the shuffle operations
    • For wide dependencies
      • For any already computed partitions that can short circuit the computation of a parent RDD

Example of Spark job stages

Default FIFO scheduler

• By default, Spark's scheduler runs jobs in FIFO fashion
  • First job gets the first priority on all available resources
    • Then the second job gets the priority, etc.
    • As long as the resource is available, jobs in the queue will start right away

Fair Scheduler

• Assigns tasks between jobs in a "round robin" fashion
  • All jobs get a roughly equal share of cluster resources
  • Short jobs that were submitted when a long job is running can start receiving resources right away
    • Good response times, without waiting for the long job to finish
  • Best for multi-user settings
Fair Scheduler Pools

- Supports grouping jobs into pools
  - With different options (e.g., weights)
  - "High-priority" pool for more important jobs

- This approach is modeled after the Hadoop Fair Scheduler

- Default behavior of pools
  - Each pool gets an equal share of the cluster
  - Inside each pool, jobs run in FIFO order
  - If the Spark cluster creates one pool per user
    - Each user will get an equal share of the cluster
    - Each user’s queries will run in order

Questions?