2

Machine Learning with Big Data

Summary

In the last lecture we looked at how distributed computation over large amounts of data could be optimally achieved by distributed data and through pushing computation to that data. In this lecture we'll look at how we can practically perform machine learning on *Big Data* by mapping machine learning problems against distributed and parallel processing frameworks. In particular, we'll look at how efficient k-means clustering can be achieved with a MapReduce model, and also look at distributed stochastic gradient descent using the Spark model.

Key points

- Distributing K-Means with MapReduce
  - Recap: **K-Means clustering** is an **iterative** algorithm that attempts to partition data into ***K* groups**.
    - Each item is represented by a **vector**;
    - Each cluster is represented by the **mean vector** (the **centroid**) of the items assigned to it;
    - Each item is assigned to the **"closest"** cluster centroid by some **distance metric**.
    - *Basic algorithm is shown in Appendix 1*.
  - **K-Means with big data** - Example: building codebooks for large-scale visual image search.
    - Cluster **millions** of **128-dimensional** "SIFT" vectors into **1 million** "visual words".
    - How much data?
      - Cluster centroids take up ~1GB (assuming double-precision floating point vectors)
      - Assume 1M images being indexed:
        - ~3000 SIFT features/image -> ~3MB/image
        - ~3TB of vectors!
  - How can K-Means be **parallelized**?
    - Fundamentally, there are two key parts to the algorithm:
      - **Assignment** of vectors to the closest centroid.
      - **Recomputation** of the centroid based on the average of the assigned vectors.
    - Both of these can be parallelized:
      - During assignment the centroids are **static**, so given enough processing resource each vector could be assigned in **parallel**.
        - The process of searching for the closest centroid is also parallelizable – the distance to each centroid can be computed in parallel.
        - This looks a lot like a **Map** operation that takes a `vector` and returns a `<vector, centroid>` pair!
      - Each centroid update is independent of the others, so they can also be performed in **parallel**.
        - Computing the new centroid is a matter of accumulating the corresponding elements of all the assigned vectors, and dividing by the number of assigned vectors.
          - This could (potentially) be parallelized on a per-element basis (although there *might* be practical limitations).
        - This looks a lot like a **Reduce** operation that takes a `<centroid, list(vector)>` and produces a new `centroid`.
        - So, the k-means algorithm maps quite nicely to the MapReduce model, with **repeated iterations of mappers and reducers** (*see appendix 3*).
      - Typically use a **fixed number of iterations**, but could check for convergence in the reducer (centroids not moving far) in the reducer
      - Map: `<null, vector> -> <centroid, vector>`
        - The Map function needs access to the centroids - this must be provided **externally**.

Written by Jonathon Hare. Send any comments/bugs/typos to jsh2@ecs.soton.ac.uk.

                - In Hadoop this can be efficiently **loaded** in the mapper **setup** and **shared** across multiple Map calls.
                - You don't need to emit the centroid vector; an (integer) identifier will work and reduce the amount of data.
                    - But, you're still emitting **every** data vector – implying that you would be **transferring a lot of data**!
                        - Interestingly, this is often not the bottleneck; in the case of very large *K,* the cost of nearest neighbour computation outweighs the cost of data transfer.
                        - It is possible to massively reduce the data transferred however (see below).
        - Reduce: `<centroid, list<vector>> -> <null, centroid>`
            - As noted above, the input `centroid` doesn't actually have to be the vector; it's more efficient if it's just an **identifier**.
                - The new centroid vector needs to be emitted however.
- K-Means needs to be initialised with a set of starting centroids. There are lots of options for choosing these:
    - **Randomly generated centroids**:
        - In some cases, you can just generate some random vectors as a starting point.
            - This only works well if the vectors you generate cover the space of the data vectors.
            - In the case of SIFT features, the space, although bounded, is highly non-uniform and contains many areas with very low density, and others with very high density.
    - **Randomly sampled centroids**:
        - Choosing the centroids by randomly uniformly sampling the actual data is often a good starting point that works well in practice.
        - How can we efficiently sample in our big data scenario?
            - We don't necessarily know exactly how many features we have
                - Finding out is relatively expensive because it means reading through all the data.
            - One possible approach:
                - In the mapper, read in all the vectors of the subset being processed to memory, and then uniformly randomly emit *K* vectors (e.g. with a null key).
                - Using a single reducer, randomly sample from the list of values *K* vectors to form the initial centroids.
                - *See appendix 2 for more details.*
    - **Canopy clustering**:
        - Fast algorithm to compute **approximate** clusters based on a pair of **distance thresholds**.
            - Can be implemented in a single Hadoop MapReduce round:
                - Each Mapper processes a subset of the data and produces a set of clusters ("canopies").
                - The canopy clustering measure is applied to the centroids of all the mapper canopies (in a single Reducer) to produce the final set of initial centroids.
- As already noted, there are some potential **performance problems** with the overall MapReduce K-Means approach.
    - Firstly, **nearest-neighbour computation is expensive** – to the extent that it can even outweigh the costs of data transfer.
        - Note that this does **not** mean that it's better not to distribute the process; we still benefit from having more computational units.
        - **Approximate** nearest-neighbour algorithms, such as through the use of **ensembles of KD-Trees** can massively reduce the computation time, at the expense of accuracy.
    - Reducing the amount of data transferred between the mappers and reducer(s).
        - Hadoop's MapReduce framework has an additional concept called a **Combiner**.
            - Combiners aim to reduce the data sent to the reducer by performing **aggregation** of data before it is sent to the reducer.

        - A combiner behaves like a reducer that runs on the data output from mappers on a **single machine**.
            - Because the combiner and mapper run on the same machine, so there is no network IO.
    - For K-Means, we could modify the MapReduce implementation to use a combiner that outputted **updated centroid vectors** together with a **weight** (the number of vectors contributing to the updated centroid). The reducer could then combine the **weighted centroids** belonging to a single cluster to produce the **updated centroid vector** for that cluster.
        - *See appendix 4 for more details.*
- Remaining disadvantage: we have to load the data from disk multiple times.
    - That's the nature of MapReduce! If we could hold it in (distributed) memory we'd use **Spark**...

- Distributed Gradient Descent with Spark
    - Gradient Descent is a set of common **first order optimisation algorithms** that attempt to find a (local) **minimum** of a **differentiable function** by taking steps proportional to the **negative gradient** of the function at the **current point**.
        - In machine learning we consider the problem of minimising an objective function that has the form of a **sum**:

          $$Q(w) = \sum_{i=1}^{n} Q_i(w),$$

          where *w* is the parameter vector that you're trying to estimate, and each summand function Q_i is associated with the *i*-th observation of the training set.
            - You've already come across this in the **backpropagation algorithm** applied to the training of **neural networks**, but it also has many other applications, such as in large scale learning of **matrix decompositions for recommender systems** (as used in the solution to the Netflix challenge).
        - **Batch Gradient Descent (BGD)** performs the following iterations:

          $$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w),$$

          where α is the **learning rate**.
            - Learning rate is often a **fixed valued scalar**, but it can be provided by a **function** (which is usually dependent on the current iteration number), and it can be a **vector** (implying different rates for each parameter).
        - When the training set is enormous, evaluating the sums of gradients becomes very expensive, because evaluating the gradient requires evaluating all the summand functions' gradients. To reduce the computational cost at each iteration, **Stochastic Gradient Descent (SGD)** samples a subset of summand functions at every step. This is very effective in the case of large-scale machine learning problems.
            - **SGD** uses only a single observation to estimate the gradient in each iteration:

              $$w := w - \alpha \nabla Q_i(w).$$
            - This however has two main disadvantages:
                - It's not obvious how to parallelise it.
                - Convergence is not as smooth as batch gradient descent.
        - **Mini-batch SGD (MBSGD)** is a compromise between BGD and SGD, and computes the gradient over **small subsets of the training data**, called **mini-batches**, at each iteration.
            - The computation of the summation over the gradients of the samples can be performed in parallel.
            - Convergence is smoother than SGD.
    - Assume that we have a **"big" distributed dataset of observations** on which we wish to perform gradient descent learning.
        - We can form mini-batches for use in MBSGD such that each mini-batch only contains observation samples hosted on a single cluster node.
        - We can distribute the computation such that the gradient of each mini-batch is computed on the node that hosts it.
            - The only data transfer that is required is for passing of the parameters (*w*) and computed gradient to a "master" node that keeps track of the algorithm state.
            - **But, there is a big problem**: only a single node will be working at any given time, as the mini-batches have to be **processed sequentially** (although the

processing of those mini-batches can take advantage of multiple cores to run in parallel).

- **Downpour SGD** is a technique developed at Google that parallelises SGD.
  - In Downpour, mini-batches are processed in parallel (on different machines) by "**workers**", and the model parameters are maintained by a "**master**".
    - Before starting to process a batch, the worker gets the **current parameters from the master**.
    - At the end of processing the batch, each worker **returns its gradient to the master**, which then **updates the parameters.**
      - The original downpour paper suggests using **Adagrad** for learning rate decay.
      - Adagrad provides a learning rate vector that updates as a function of the sum-squared gradients of each parameter over time.
    - Obviously, at any given point in time there is no guarantee that the workers have an **up-to-date copy of the parameters**, and will thus return technically **incorrect gradients** to the master.
      - **In practice this doesn't seem to matter, and the scheme actually works very well with real data!**
- This is trivial to implement in Spark
  - The "master" is the node on which you run the program; it sets up a server that provides methods to get the current parameters and update the parameters with new gradients.
    - Communications could use anything: HTTP, RMI, even MPI…
    - The update method takes a gradient vector and uses it to alter the parameters of the model.
  - The master then uses the **mapPartitions** action with a function that:
    - Fetches the current parameters from the master
    - Computes the gradients of each mini-batch (the mini-batch being defined as a RDD partition – that is basically equivalent to a block of the underlying file on HDFS).
    - Sends a message to the master informing it of the computed gradients.

Further reading

- Dean, Jeffrey & Ghemawat, Sanjay (2004). "MapReduce: Simplified Data Processing on Large Clusters". Retrieved November 23, 2011.
- McCallum, A.; Nigam, K.; and Ungar L.H. (2000) "Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching", Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, 169-178 doi:10.1145/347090.347123
- Hare, Jonathon S., Samangooei, Sina & Lewis, Paul H. (2012) Practical scalable image analysis and indexing using Hadoop. *Multimedia Tools and Applications*, 1-34. (doi:10.1007/s11042-012-1256-0).
- Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. NIPS2012. http://research.google.com/archive/large_deep_networks_nips2012.pdf
- Dirk Neumann. Deepdist: Downpour SGD on Spark. https://github.com/dirkneumann/deepdist

Appendix 1: K-Means

Synopsis: group items into K clusters

Input:

A collection of feature vectors; one per item being clustered.

K; an integer representing the number of clusters.

K initial cluster centres; typically chosen randomly or from a sample of the existing data points.

Algorithm:

Then the following process is performed iteratively until the centroids don't move between iterations (or the maximum number of iterations is reached):

Each point is assigned to its closest centroid

The centroid is recomputed as the mean of all the points assigned to it. If the centroid has no points assigned it is randomly re-initialised to a new point.

The final clusters are created by assigning all points to their nearest centroid.

Appendix 2: Efficiently sampling data with MapReduce

Synopsis: sample *K* items from an unknown amount of data

Input: collection of files containing feature-vectors; read one vector at a time.

Mapper:

Map function:

Input: <*Key:* null, *Value:* vector>

Outputs: *nothing*

Read the input vectors into the mapper memory.

Output on shutdown: <*Key:* null, *Value:* vector>

Output K uniformly randomly sampled vectors on shutdown of the mapper.

Reduce function:

Input: <*Key:* centroid_id, *Value:* [vectors of the items assigned to the cluster]>

Outputs: <*Key:* Null, *Value:* centroid vector>

Average the vectors to compute the new centroid, and emit this with a null key.

Appendix 3: A single K-Means iteration with Hadoop MapReduce

Synopsis: perform a k-means clustering iteration

Input: collection of files containing feature-vectors; read one vector at a time.

Mapper:

Additional data: current centroid vectors, read from DFS on instantiation of the mapper

Map function:

Input: <*Key:* null, *Value:* vector>

Outputs: <*Key:* centroid_id, *Value:* vector>

Find the closest centroid to vector and emit a single <key,value> pair with the identifier of the centroid (i.e. its index) and the vector.

Reduce function:

Input: <*Key:* centroid_id, *Value:* [vectors of the items assigned to the cluster]>

Outputs: <*Key:* Null, *Value:* centroid_vector>

Average the vectors to compute the new centroid, and emit this with a null key.

Appendix 4: A single K-Means iteration with Hadoop MapReduce and a Combiner

Synopsis: perform a k-means clustering iteration

Input: collection of files containing feature-vectors; read one vector at a time.

Mapper:

Additional data: current centroid vectors, read from DFS on instantiation of the mapper

Map function:

Input: <*Key:* null, *Value:* vector>

Outputs: <*Key:* centroid_id, *Value:* <vector, 1>>

Find the closest centroid to vector and emit a single <key,value> pair with the identifier of the centroid (i.e. its index) and the vector.

Combiner:

Input: <*Key:* centroid_id, *Value:* [list of <vector, 1>]>

Outputs: <*Key:* centroid_id, *Value:* <local_centroid_vector, count>>

Compute the average of all the vectors and the number of vectors. Emit the average vector and count using the centroid_id as the key.

Reduce function:

Input: <*Key:* centroid_id, *Value:* [list of <vector, count> assigned to cluster]>

Outputs: <*Key:* Null, *Value:* centroid_vector>

Compute the updated centroid vector by sum(vector*count)/sum(count) and emit it.