MapReduce Practice Problems: DSC 208 Data Management for Analytics

MapReduce Practice Problems

Approach to casting a data analytics computation onto the MapReduce API

Step 1. Identify the exact data access pattern of the computation over the dataset. Draw it out to see it visually if you like.

Step 2. Identify how to decompose the bulk of the whole computation into a bunch of independent chunk computations on sub-elements (rows/columns/tiles). Typically, scalability along rows is the most preferable because most modern large-scale datasets have large numbers of rows.

Step 3. Identify how to aggregate those decomposed parts to get the final result as if it was computed in a single-threaded in-RAM manner. This aggregation step may not always be needed though.

Step 4. Align the sharding with Step 2. Put the independent chunk computations in the Mapper. Identify what the Mapper's intermediate output (emit) data structure should be. Put the aggregation in Step 3 and any post processing in the Reducer.

Practice Problems

Question 1. Write pseudocode for (or just describe precisely in prose) a MapReduce job to compute the Frobenius norm (aka L2 norm) of a given large matrix. It should be scalable along the number of rows. Make sure to explain your assumption on how the dataset is stored/sharded to begin with.
Answer: Input Split: Shard row-wise so that it is scalable along number of rows.

Need full MR job due to global aggregation for norm, as per 4 steps:

1. Plain total sum of all cells x^2; addition order does not matter.
2. Can chunk whole computation into independent ones over rows/shards.
3. Just add up results of independently computed partial sum.
4. MapReduce
   (a) **Map():** Given the shard with multiple rows, compute x^2 of each cell and add them up into partial sum; emit it with single global dummy key.

```
Map(Key: row_id, Value: row_data)
    partial_sum_of_squares = 0
    For each element x in row_data:
        partial_sum_of_squares = partial_sum_of_squares + (x * x)
    Emit(Key: "dummy_global_key", Value: partial_sum_of_squares)
```

   (b) **Reduce():** Iterator with all partial sums; add them all up, take square root of global sum, emit that as final result: L2 norm.

```
Reduce(Key: "dummy_global_key", Values: list_of_partial_sums)
    global_sum_of_squares = 0
    For each partial_sum in list_of_partial_sums:
        global_sum_of_squares = global_sum_of_squares + partial_sum
    frobenius_norm = SQRT(global_sum_of_squares)
    Emit(Key: "Frobenius_Norm", Value: frobenius_norm)
```

— **Brief Explanation:** This solution is correct because the Frobenius norm requires a global sum of squares. The MapReduce pattern effectively distributes the initial squaring and partial summation across many Mappers, and then a single Reducer aggregates these partial sums to compute the final global result. This ensures scalability by distributing the most computationally intensive part (processing individual cells) and centralizing only the final, lightweight aggregation.

—

Question 2. Suppose you are given a large dataset with 50 numeric and 9 categorical features (domain size of 50 each). The HDFS file size is 3 TB.

a) Write pseudocode for (or just describe precisely in prose) MapReduce job(s) to compute this dataset's correlation matrix. Hint: It is OK to do it as 2 separate MapReduce jobs, although 1 suffices. Answer: Suppose we one-hot encode all categorical features into 50-dimensional 0-1 vectors. Then total number of numerics is $50 + 9 \times 50 = 500$. So, correlation matrix is of size 500×500, which is 250,000 cells. Even with float64, it is only 2MB. So, we will use this as our aggregation state for Mappers to send to Reducer. **Input Split:** Shard row-wise as usual. One approach to compute correlation matrix uses 2 MR jobs: the first to compute the per-feature means and stdevs; and the second to use those to finish the correlation computations.

Job 1: Compute per-feature means and standard deviations

(a) **Map():** Reads tuple, converts each categorical feature to its respective one-hot encoded vector to stitch together full 500-dimensional numeric vector, and computes the "sufficient statistics" needed for means and stdevs, viz., running example count and running (x, x^2) for each feature; emit all those suff. stats as one long vector as value with a single global dummy key.

```
Map(Key: record_id, Value: record_data)
    numeric_vector = []
    For each numeric_feature x_num in record_data:
        Add x_num to numeric_vector
    For each categorical_feature x_cat in record_data:
        one_hot_vector = convert_to_one_hot(x_cat, domain_size=50)
        Append one_hot_vector elements to numeric_vector
    feature_stats = new List of (sum_x, sum_x_squared) for 500 features
    For i from 0 to 499:
        feature_stats[i] = (numeric_vector[i], numeric_vector[i] * numeric_vector[i])
    Emit(Key: "global_dummy_key", Value: (1, feature_stats))
```

(b) **Reduce():** Aggregates all suff. stats vectors to emit total example count and 2 vectors: means of all features, stdevs of all features.

```
Reduce(Key: "global_dummy_key", Values: list_of_partial_stats)
    total_records = 0
    global_feature_sums = new List of 0s (size 500)
    global_feature_sq_sums = new List of 0s (size 500)
    For each partial_stat in list_of_partial_stats:
        total_records += partial_stat.count
        For i from 0 to 499:
            global_feature_sums[i] += partial_stat.feature_stats[i].sum_x
            global_feature_sq_sums[i] += partial_stat.feature_stats[i].sum_x_squared
    means = new List of 0s (size 500)
    stdevs = new List of 0s (size 500)
    For i from 0 to 499:
        means[i] = global_feature_sums[i] / total_records
        variance = (global_feature_sq_sums[i] / total_records) - (means[i] * means[i])
        stdevs[i] = SQRT(variance)
    Emit(Key: "GlobalStats", Value: (total_records, means, stdevs))
```

Job 2: Compute Correlation Matrix

(a) **Map():** Reads tuple, gets the 500-dimensional numeric vector again as before and then emits as suff. stats a 500x500 matrix representing pairwise products for the aggregation needed for the numerator of the Corr matrix formula:

$$Corr(A, B) = \frac{E[(A - \mu_A)(B - \mu_B)]}{\sigma_A \sigma_B}$$

```
Map(Key: record_id, Value: record_data)
    // Assume global_means and global_stdevs are loaded (e.g., Distributed Cache)
    numeric_vector = []
    For each numeric_feature x_num in record_data:
        Add x_num to numeric_vector
    For each categorical_feature x_cat in record_data:
        one_hot_vector = convert_to_one_hot(x_cat, domain_size=50)
        Append one_hot_vector elements to numeric_vector
    pairwise_products_matrix = new 500x500 matrix of 0s
    For i from 0 to 499:
        For j from 0 to 499:
            val_i_centered = numeric_vector[i] - global_means[i]
            val_j_centered = numeric_vector[j] - global_means[j]
            pairwise_products_matrix[i][j] = val_i_centered * val_j_centered
    Emit(Key: "global_dummy_key", Value: pairwise_products_matrix)
```

(b) **Reduce():** Just adds up these individual matrices and divides all cells by total example count and the respective pairs of stdevs obtained from the first MR job.

```
Reduce(Key: "global_dummy_key", Values: list_of_pairwise_product_matrices)
    // Assume total_records, global_means, global_stdevs are loaded
    final_covariance_numerator_sum = new 500x500 matrix of 0s
    For each partial_matrix in list_of_pairwise_product_matrices:
        For i from 0 to 499:
            For j from 0 to 499:
                final_covariance_numerator_sum[i][j] += partial_matrix[i][j]
    correlation_matrix = new 500x500 matrix of 0s
    For i from 0 to 499:
        For j from 0 to 499:
            covariance_AB = final_covariance_numerator_sum[i][j] / total_records
            denominator = global_stdevs[i] * global_stdevs[j]
            If denominator != 0:
                correlation_matrix[i][j] = covariance_AB / denominator
            Else:
                correlation_matrix[i][j] = 0
    Emit(Key: "CorrelationMatrix", Value: correlation_matrix)
```

Alternative approach: A more advanced approach is to get all the suff. stats, as well as the partial matrices with the pairwise products of columns in one go! Input is still sharded row-wise as before.

(a) **Map():** Converts each categorical feature to one-hot encoding, computes running example count, running sum of (x, x^2) per feature, and running partial matrix of pairwise products of feature values.

```
Map(Key: record_id, Value: record_data)
    numeric_vector = []
    For each numeric_feature x_num in record_data:
        Add x_num to numeric_vector
    For each categorical_feature x_cat in record_data:
        one_hot_vector = convert_to_one_hot(x_cat, domain_size=50)
        Append one_hot_vector elements to numeric_vector
    local_count = 1
    local_feature_sums = new List of 0s (size 500)
    local_feature_sq_sums = new List of 0s (size 500)
    local_pairwise_products_matrix = new 500x500 matrix of 0s
    For i from 0 to 499:
        local_feature_sums[i] = numeric_vector[i]
        local_feature_sq_sums[i] = numeric_vector[i] * numeric_vector[i]
        For j from 0 to 499:
            local_pairwise_products_matrix[i][j] = numeric_vector[i] * numeric_vector[j
                ]
    Emit(Key: "global_dummy_key", Value: (local_count, local_feature_sums,
        local_feature_sq_sums, local_pairwise_products_matrix))
```

(b) **Reduce():** Adds up the partial counts, partial matrices, and partial (x, x^2) vectors, respectively, to get all needed global aggregates: count, all means, all stdevs; use formula below to calculate final Corr matrix.

$$Corr(A, B) = \frac{E[AB] - E[A]E[B]}{\sigma_A \sigma_B}$$

```
Reduce(Key: "global_dummy_key", Values: list_of_all_partial_stats)
    total_records = 0
    global_feature_sums = new List of 0s (size 500)
    global_feature_sq_sums = new List of 0s (size 500)
    global_pairwise_products_matrix = new 500x500 matrix of 0s
    For each partial_stat in list_of_all_partial_stats:
        total_records += partial_stat.count
        For i from 0 to 499:
            global_feature_sums[i] += partial_stat.feature_sums[i]
            global_feature_sq_sums[i] += partial_stat.feature_sq_sums[i]
            For j from 0 to 499:
                global_pairwise_products_matrix[i][j] += partial_stat.
                    pairwise_products_matrix[i][j]
    means = new List of 0s (size 500)
    stdevs = new List of 0s (size 500)
    For i from 0 to 499:
        means[i] = global_feature_sums[i] / total_records
        variance = (global_feature_sq_sums[i] / total_records) - (means[i] * means[i])
        stdevs[i] = SQRT(variance)
    correlation_matrix = new 500x500 matrix of 0s
    For i from 0 to 499:
        For j from 0 to 499:
            E_AB = global_pairwise_products_matrix[i][j] / total_records
            numerator = E_AB - (means[i] * means[j])
            denominator = stdevs[i] * stdevs[j]
            If denominator != 0:
                correlation_matrix[i][j] = numerator / denominator
            Else:
                correlation_matrix[i][j] = 0
    Emit(Key: "CorrelationMatrix", Value: correlation_matrix)
```

— **Brief Explanation:** Both the two-job and single-job approaches correctly compute the correlation matrix by leveraging MapReduce for distributed aggregation of sufficient statistics. The two-job approach separates mean/standard deviation calculation from covariance/correlation calculation, which can simplify logic. The single-job approach is more efficient as it reduces data passes and I/O by computing all necessary partial statistics (counts, sums, squared sums, pairwise products) in one Map phase and aggregating them in a single Reduce phase, then deriving the final correlation matrix.

b) Briefly explain how you would scale this computation on an on-premise cluster. Answer: Install a Spark cluster; load and shard data as a Spark DataFrame; write Spark-MR job. Note that Dask is NOT a good fit, since 3 TB file may not fit even on single-node disk. — **Brief Explanation:** For an on-premise cluster, **Apache Spark** is the go-to solution for scaling such a computation. Spark's in-memory processing capabilities and DataFrame API are well-suited for iterative computations like correlation. It provides a robust distributed framework that internally handles data sharding and parallel processing, making it highly effective for large datasets that won't fit on a single machine's disk.

c) Briefly explain how you would scale this computation on AWS. Answer: Option 1: Multi-node EC2+EBS cluster with Q2.C's approach. **Option 2:** Q2.C's approach, except with

EMR. **Option 3:** Single-node EC2 running regular Python and remote reads from S3; this will be very slow due to low parallelism. — **Brief Explanation:** On AWS, the most effective way to scale this is using **AWS Elastic MapReduce (EMR)**, a managed service that provisions and manages Spark/Hadoop clusters. This is superior to manually setting up EC2 instances (Option 1) because EMR handles the operational overhead. Option 3, a single EC2 instance, is impractical for 3TB of data due to its lack of parallelism and I/O bottlenecks with remote S3 reads.

—

Question 3. Write pseudocode (or just describe precisely) using MapReduce/Spark operations to perform the following data science operations at scale:

a) Quadratic (order 2) feature interactions Answer: For Input Split: Assume data is sharded row-wise, as is common. Map-only job suffices. Map() takes feature vector from tuple, performs feature interactions and emits the interacted vector with the same tuple ID.

```
Map(Key: tuple_ID, Value: original_feature_vector)
    interacted_feature_vector = []
    For i from 0 to (length of original_feature_vector - 1):
        For j from i to (length of original_feature_vector - 1):
            new_feature_value = original_feature_vector[i] * original_feature_vector[j]
            Append new_feature_value to interacted_feature_vector
    Emit(Key: tuple_ID, Value: interacted_feature_vector)
```

— **Brief Explanation:** This is correct because **quadratic feature interactions** are a row-wise transformation; the calculation for each record is independent of all other records. Therefore, a **Map-only job** is sufficient, as Mappers can process their assigned data shards completely without needing any global aggregation or shuffle, making it highly efficient.

b) Binning a numeric feature with given bins Answer: For Input Split: Assume data is sharded row-wise, as is common. Also a Map-only job. Map() takes feature value from tuple, performs binning based on given bins and emits the same tuple with same tuple ID, except this feature value is now different.

```
Map(Key: tuple_ID, Value: (original_tuple_data, numeric_feature_value))
    // Assume 'bins' (e.g., [0, 10, 20, 30, infinity]) are known to the Mapper
    binned_value = ""
    If numeric_feature_value < bins[0]:
        binned_value = "bin_less_than_threshold_0"
    Else If numeric_feature_value >= bins[0] AND numeric_feature_value < bins[1]:
        binned_value = "bin_0_to_1"
    Else:
        binned_value = "bin_last_threshold_to_infinity"
    updated_tuple_data = replace_feature_in_tuple(original_tuple_data, binned_value)
    Emit(Key: tuple_ID, Value: updated_tuple_data)
```

— **Brief Explanation:** **Binning** is another **row-wise transformation**. Since the bin boundaries are given and fixed, each Mapper can independently determine which bin a numeric feature falls into for each record in its shard. No information from other records is needed, so a **Map-only job** is the correct and most efficient pattern.

c) One-hot encoding of a categorical feature (assume feature's domain has only 5000 unique values and is given) Answer: For Input Split: Assume data is sharded row-wise, as is common. Also a Map-only job. Map() takes feature value from tuple; performs one-hot encoding based on dictionary to map category value to new feature index; obtains the 0-1 representation for that feature (potentially sparse vector); emits same tuple with same tuple ID, except this feature value is now replaced with the 0-1 vector.

```
Map(Key: tuple_ID, Value: (original_tuple_data, categorical_feature_value))
    // Assume 'domain_map' (e.g., {"red": 0, "blue": 1, ...}) is known to the Mapper
    one_hot_vector_size = 5000
    one_hot_representation = new List of 0s (size one_hot_vector_size)
    category_index = domain_map.get(categorical_feature_value)
    If category_index is not NULL AND category_index is within bounds:
        one_hot_representation[category_index] = 1
```

```
updated_tuple_data = replace_feature_in_tuple(original_tuple_data,
    one_hot_representation)
Emit(Key: tuple_ID, Value: updated_tuple_data)
```

— **Brief Explanation:** **One-hot encoding** is a **row-wise operation** because each categorical value can be converted to its binary vector representation independently. As long as the mapping from category to index is known to all Mappers (e.g., pre-loaded via distributed cache), no cross-record communication is needed, making this a correct application for a **Map-only job**.

d) Whitening a numeric feature Answer: For Input Split: Assume data is sharded row-wise, as is common. 1 MR job + 1 Map-only job: **Job 1: Compute Global Mean and Standard Deviation**

(a) **Map():** Takes feature values from tuple; computes suff. stats for mean and stdev as 3-tuple $(1, x, x^2)$; emits this 3-tuple as value with a single global dummy key.

```
Map(Key: tuple_ID, Value: numeric_feature_value)
    Emit(Key: "global_dummy_key", Value: (1, numeric_feature_value,
        numeric_feature_value * numeric_feature_value))
```

(b) **Reduce():** Iterates over all suff. stats 3-tuples to compute global mean and stdev of this feature; emits that 2-tuple as output.

```
Reduce(Key: "global_dummy_key", Values: list_of_stat_tuples)
    total_count = 0
    total_sum = 0
    total_sum_sq = 0
    For each stat_tuple in list_of_stat_tuples:
        total_count += stat_tuple[0]
        total_sum += stat_tuple[1]
        total_sum_sq += stat_tuple[2]
    mean = total_sum / total_count
    variance = (total_sum_sq / total_count) - (mean * mean)
    stdev = SQRT(variance)
    Emit(Key: "GlobalStats", Value: (mean, stdev))
```

Job 2: Apply Whitening Transformation (Map-only job)

(a) **Map():** Takes feature value from tuple; whitens its based on (mean, stdev) 2-tuple from prior job; emits same tuple with same tuple ID, except this feature value is now different.

```
Map(Key: tuple_ID, Value: (original_tuple_data, numeric_feature_value))
    // Assume global_mean and global_stdev are loaded (e.g., Distributed Cache)
    whitened_value = 0.0
    If global_stdev != 0:
        whitened_value = (numeric_feature_value - global_mean) / global_stdev
    Else:
        whitened_value = 0.0
    updated_tuple_data = replace_feature_in_tuple(original_tuple_data, whitened_value)
    Emit(Key: tuple_ID, Value: updated_tuple_data)
```

— **Brief Explanation:** **Whitening** (Z-score normalization) requires global statistics (mean and standard deviation) of the entire feature, which means it cannot be done in a single Map-only pass. Therefore, a **two-job approach** is correct: the first MapReduce job calculates these global statistics, and the second (Map-only) job then applies these globally computed values to each individual data point.

—

Question 4. Suppose you are performing model selection for a RandomForest model. For hyper-parameter tuning, you do grid search with 3 values of number of trees and 4 values of maximum tree height. To aid your interpretability, you also explore 5 different manually created subsets of features apart from the full feature set. What is the total number of models built in this model selection workload?

Answer: Let Total number of models built $\equiv TNMB$

$$TNMB = (\text{number of trees}) \times (\text{max height values}) \times (\text{number of feature sets} + 1)$$

$$TNMB = (3) \times (4) \times (5+1)$$
$$TNMB = 12 \times 6$$
$$TNMB = 72$$

— **Brief Explanation:** The total number of models is found by multiplying the number of choices for each independent variable: number of trees, maximum tree height, and the available feature sets (which includes the 5 specified subsets plus the full feature set, making 6 options). This is a direct application of the **multiplication principle** in combinatorics.

—

Question 5. You are given a large training dataset of $(Y, X1, X2)$ examples on HDFS for binary classification (i.e., $Y = 0$ or 1) with two categorical features $X1$ and $X2$. The domains of the features are known beforehand as $DX1$ and $DX2$ and have only tens of unique values. Write pseudocode for a single MapReduce job to train a Naive Bayes model. It should be scalable along the number of rows. Make sure to explain your assumption on how the dataset is stored/sharded to begin with. Hint: Naive Bayes training only needs to estimate the distribution $P(Y)$ and all class-conditional probability distributions $P(Xi|Y)$ using frequency counts. **Answer: Input Split:** Shard table row-wise.

Using the 4-step method, identify the access pattern over D: **Step 1:** Computing frequency counts for a probability distribution is akin to a SQL COUNT. So, that just requires a sequential scan over the dataset. Since DX1 and DX2 are small and Y is binary, the prob. distr. stats are small and can all be batched onto one pass over the dataset.

```
Table:
| D | Y | X1 | X2 |
|---|---|----|----|
|   |   |    |    |
|   |   |    |    |
|   |   |    |    |
|   |   |    |    |
```

Step 2: All of these are just counts of tuples with predicates applied on the record/tuple's data. So, they are easily decomposed over the n tuples of D. **Step 3:** The aggregation for each count is just one big sum over the records/tuples of D. All counts can be calculated collectively in one pass. The counts we need:

- # tuples
- # tuples with $Y = 0$
- # tuples with $Y = 1$
- For each $x1$ in $DX1$:
  - # tuples with $(Y = 0 \,\&\, X1 = x1)$
  - # tuples with $(Y = 1 \,\&\, X1 = x1)$
- Likewise for $X2$ as with $X1$.

Step 4:

(a) **Map():** Calculates all counts per shard by iterating over the records in it; emit a value (no/dummy key) a vector of partial counts of length $1 + 2 + 2 \cdot |DX1| + 2 \cdot |DX2|$.

```
Map(Key: record_id, Value: (Y, X1, X2))
    partial_counts = new Map()
    partial_counts["total"] = 1
    If Y == 0:
        partial_counts["Y_0"] = 1
    Else:
        partial_counts["Y_1"] = 1
    partial_counts[("X1", X1, Y)] = 1
    partial_counts[("X2", X2, Y)] = 1
    Emit(Key: "global_counts_key", Value: partial_counts)
```

(b) **Reduce():** Get Iterator of all partial counts from Mappers; add them to get global counts; divide respective counts to get resp. prob. distr. entries, e.g., $P[Y = 1] = \frac{\text{(number of tuples with } Y{=}1)}{\text{number of tuples}}$, etc.

```
Reduce(Key: "global_counts_key", Values: list_of_partial_counts_maps)
    global_counts = new Map()
    For each partial_counts_map in list_of_partial_counts_maps:
        For each (key, count) in partial_counts_map:
            global_counts[key] = global_counts.get(key, 0) + count
    P_Y0 = global_counts["Y_0"] / global_counts["total"]
    P_Y1 = global_counts["Y_1"] / global_counts["total"]
    class_conditional_probs = new Map()
    For each x1_val in DX1: // DX1 is predefined list of values
        count_X1_Y0 = global_counts.get(("X1", x1_val, 0), 0)
        count_X1_Y1 = global_counts.get(("X1", x1_val, 1), 0)
        P_X1_given_Y0 = count_X1_Y0 / global_counts["Y_0"]
        P_X1_given_Y1 = count_X1_Y1 / global_counts["Y_1"]
        class_conditional_probs[("X1", x1_val, 0)] = P_X1_given_Y0
        class_conditional_probs[("X1", x1_val, 1)] = P_X1_given_Y1
    For each x2_val in DX2: // DX2 is predefined list of values
        count_X2_Y0 = global_counts.get(("X2", x2_val, 0), 0)
        count_X2_Y1 = global_counts.get(("X2", x2_val, 1), 0)
        P_X2_given_Y0 = count_X2_Y0 / global_counts["Y_0"]
        P_X2_given_Y1 = count_X2_Y1 / global_counts["Y_1"]
        class_conditional_probs[("X2", x2_val, 0)] = P_X2_given_Y0
        class_conditional_probs[("X2", x2_val, 1)] = P_X2_given_Y1
    Emit(Key: "NaiveBayesModel", Value: (P_Y0, P_Y1, class_conditional_probs))
```

— **Brief Explanation:** This single MapReduce job correctly trains a **Naive Bayes model** by efficiently collecting all necessary frequency counts. Mappers compute local counts for Y, $X1|Y$, and $X2|Y$ within their assigned data records. By emitting these partial counts with a single dummy key, all are directed to a single Reducer (feasible given the small domain sizes). The Reducer then aggregates these counts to calculate the global probabilities, thereby building the Naive Bayes model parameters in one distributed pass.