Databricks

Machine-Learning-Professional

Databricks Certified Machine Learning Professional

For More Information – Visit link below:

https://www.examsempire.com/

Product Version

1. **Up to Date products, reliable and verified.**
2. **Questions and Answers in PDF Format.**

Latest Version: 6.0

Question: 1

A data science team must train a Gradient Boosted Trees (GBT) model on a feature table containing 150 TB of dense numerical data for daily propensity modeling. Given the massive scale and the need for GPU acceleration on Databricks, which approach is recommended to leverage Spark's distributed capabilities for model training?

A. Use the open source Python xgboost .train function within a single large Databricks ML Runtime GPU instance, ensuring the instance memory far exceeds 150 TB.
B. Employ the built-in sparkdl .xgboost .XGBoostC1assifier estimator, configuring num_workers equal to the cluster core count to manage distributed training efficiently.
C. Collect the entire 150 TB dataset into a single Pandas DataFrame using df.toPandas(), then train the model using a localized GPU library.
D. Utilize xgboost. spark. SparkXGBC1assifier, leveraging Spark's distributed architecture to partition the 150 TB dataset across multiple executor GPUs for concurrent training.
E. Implement an MLflow PyFunc model wrapper around a standard single-node XGBoost instance and log it directly to Unity Catalog.

Answer: D

Explanation:
SparkML, specifically distributed ML libraries like 'xgboost.spark', is highly recommended when the dataset size far exceeds the memory capacity of a single machine (150 TB). Option D, using 'SparkXGBClassifier' (part of the modern 'xgboost.spark' module), correctly identifies the mechanism for utilizing Spark's distributed computing to train a scalable model across multiple executors, potentially leveraging GPUs if configured. Option A is infeasible as 150 TB cannot fit on a single instance. Option B uses the deprecated 'sparkdl.xgboost' module. Option C will result in an immediate out-of-memory error. Option E addresses deployment, not scalable training.

Question: 2

A financial institution needs to execute batch inference against a newly received dataset of 500 million records, logging results to a Delta table within a tight 4-hour SLA. A trained PyFunc model, fraud_clf, is registered in Unity Catalog. Which efficient and scalable implementation snippet ensures distributed scoring across the required volume?

- ○ Load the model using `model = mlflow.pyfunc.load_model(...)` and iterate through the data row-by-row using a Python `for` loop to call `model.predict(row)`.
- ○ Deploy the model to Databricks Model Serving as a REST API endpoint and send 500 million asynchronous requests to it.
- ○ Use a simple Pandas UDF (non-Pandas/Arrow enabled) to apply prediction logic sequentially on the driver node.
- ○ Implement the inference using `loaded_model_udf = mlflow.pyfunc.spark_udf(spark, model_uri)` and apply it as shown: `df_result.withColumn('predictions', loaded_model_udf(struct(features)))`.
- ○ Utilize Databricks AutoML to automatically generate the scoring notebook, regardless of the model flavor used.

A. Option A

B. Option B
C. Option C
D. Option D
E. Option E

Answer: D

Explanation:
For high-throughput, high-volume offline scoring (500 million records, 4-hour SLA), Spark batch inference is mandatory. Option D demonstrates the best practice using 'mlflow.pyfunc.spark_udf' to wrap the MLflow model into a Spark User-Defined Function (UDF). This function automatically leverages Apache Arrow and distributes the model loading and inference computation across all cluster worker nodes, making it highly scalable and performant for batch tasks. Option A is single-threaded and non-scalable. Option B is designed for low-latency/real-time use cases, not massive asynchronous batch jobs. Option C suffers from Python serialization overhead and runs poorly at scale.

Question: 3

A media company implements a Quality of Service (QOS) solution to monitor streaming video performance in near real-time, requiring continuous processing (latency < 5 minutes). The processing involves enriching incoming event data with geo-location metadata stored in a constantly updated Delta table and scoring the augmented data using a fault-tolerant deep learning model. Which configuration element confirms that Apache Spark Structured Streaming is the appropriate component for this use case?

A. The use of the trigger (realTime="5 minutes") option ensures ultra-low latency inference suitable for real-time customer feedback loops.
B. The pipeline structure enables ETL transformations, stream-stream joins (event stream + metadata stream), and embedding deep learning model inference as a transform()operation, all within a unified dataflow.
C. The utilization of dedicated single-node compute clusters configured for the GPU-enabled Databricks Runtime ML.
D. The model is deployed via Model Serving, receiving requests asynchronously from the Structured Streaming job via REST API calls.
E. Data ingress uses RDD APIs to achieve fault tolerance via Kafka connectors.

Answer: B

Explanation:
Apache Spark Structured Streaming is recommended for continuous data processing workloads where the complexity involves incremental ETL, stream-stream joins (e.g., enriching events with Delta table data), and high-throughput scoring, supporting near real-time requirements (latency < 5 minutes). Option B correctly describes how Structured Streaming unifies these capabilities, using transformations and model UDFs applied directly within the data stream. Option A's trigger is valid for low latency but 'realTime' mode is intended for operational workloads where latency is sub-second (ms), not near real-

time (minutes). Option C limits scalability. Option D introduces unnecessary latency overhead compared to using distributed UDFs directly in Spark.

Question: 4

A team developing a large-scale recommendation system requires extracting features from 50 TB of raw user log data (Bronze tables) and materializing rich features (Gold tables). The ETL pipeline involves complex data aggregation, deduplication, and cleaning operations. Why is Apache Spark essential for the feature engineering phase, even if the eventual model training algorithm (e.g., a simple linear model) could run on a small sampled dataset?

A. Spark automatically performs semantic type detection required for metadata extraction before cleaning the data.
B. Spark provides native integration with RDD APIs necessary for efficient data aggregation workflows in the Bronze layer.
C. The use of vectorized Pandas UDFs within Spark allows the transformation and aggregation of the 50 TB dataset to be distributed and scaled efficiently across the cluster before model training, preventing out-of-memory issues.
D. The complexity of deduplication and cleaning at scale requires SQL syntax, which is only executed efficiently via the Spark engine.
E. It is required for compliance that all feature engineering workloads utilize a dedicated MLlib Transformer, regardless of data scale.

Answer: C

Explanation:
Spark is critical for the Data Prep and Feature Engineering phase when dealing with massive volumes of raw data (50 T B). Even if the model input is small, the initial ETL must scale. Option C highlights the ability of Spark (using optimizations like Pandas/Arrow UDFs for efficient parallel row operations) to distribute the computationally intensive aggregation and cleaning across worker nodes, which a single-machine solution could not handle due to memory limits. Option B is outdated; modern ETL relies on Structured APIs (DataFrames/SQL). Option D ignores that complex transformations often require Python/Scala logic.

Question: 5

An e-commerce company needs to build a scalable model for collaborative filtering (product recommendations) using implicit feedback from millions of users and billions of product interaction events. Since high dimensionality and large user/item matrices are expected, which algorithm and platform choice provides optimal distributed processing for this specific task?

A. Use the built-in "learn. cluster. KMeans algorithm on a single node after downsampling the interaction data to 5 GB.
B. Implement a custom deep learning network trained via PyTorch Distributed (TorchDistributor) to handle matrix factorization iteratively.

C. Leverage Spark MLlib's distributed implementation of the Alternating Least Squares (ALS) algorithm to efficiently perform matrix factorization across the distributed user-item dataset.
D. Use MLflow's Prompt Engineering tools to train a Large Language Model (LLM) for personalized recommendations.
E. Use the GraphFrames library to model the user-item interactions and apply a distributed Shortest Path algorithm.

Answer: C

Explanation:
Recommendation systems, particularly those using collaborative filtering with large, sparse user-item interaction matrices (implicit feedback), are classic big data problems where models scale poorly on single machines. Spark MLlib includes the distributed ALS algorithm specifically for this purpose (matrix factorization), providing an optimal, scalable solution for large datasets. Option A is unsuitable due to the massive loss of fidelity from extreme downsampling. Option B is overly complex when a specialized, tested SparkML algorithm exists. Option D (LLMs/Prompt Engineering) is generally reserved for generating content or human-readable summaries, not for high-volume collaborative filtering matrix operations.

Question: 6

A deployment pipeline must ensure that a feature normalization step using org.apache. spark. ml. feature. StandardScAIer is applied identically to the test data as it was to the training data'. The team aims to manage this dependency automatically within the ML framework. Which Spark ML workflow practice must be implemented to guarantee this reproducible transformation chaining?

- ○ The final fitted model should be logged using `mlflow.log_model(..., metadata={'requires_scaling': True})` to instruct the inference routine to manually apply the scaling parameters.
- ○ The `StandardScaler` parameters (mean and standard deviation) must be extracted manually from the training run and hardcoded into the prediction script.
- ○ The pipeline must be defined using the `Pipeline(stages=[StandardScaler, Estimator])` syntax, fitted on the training data to produce a single `PipelineModel` artifact, which implicitly stores all required transformation parameters.
- ○ Utilize a temporary view in Spark SQL to cache the transformed training data before running the estimator.
- ○ The use of the Python `dbutils.jobs.taskValues` utility must be implemented to pass the fitted scaling parameters between notebook tasks.

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: C

Explanation:
The core purpose of the Spark ML 'Pipeline' abstraction is to ensure that a sequence of Transformers' (like 'StandardScaler') and an 'Estimator' are treated as a single entity. When the Pipeline is 'fit()' on the training data, the resulting 'PipelineModel' captures all transformation parameters (e.g., the computed mean/std dev from the scaler) alongside the final learned model weights. This guarantees that the exact transformations are applied downstream, ensuring reproducibility. Option B violates MLOps principles

of code independence. Option E uses external orchestration mechanisms when the ML framework itself provides the solution.

Question: 7

A retail fraud team has developed a highly accurate, single-node XGBoost model (logged via MLflow) but requires predictions to be delivered to a real-time checkout system with an end-to-end latency SLA of 10 milliseconds. Why is deploying this model directly using a Spark Structured Streaming inference pipeline generally NOT the recommended production pattern for meeting this strict low-latency requirement?

A. Spark is specifically optimized for row-centric operations, making single-row lookups inherently fast.
B. The operational overhead of starting or waking a Spark job, even for minimal computation, results in latencies typically exceeding the required millisecond threshold.
C. Structured Streaming pipelines require complex, continuous manual monitoring that violates MLOps automation principles.
D. Spark MLlib algorithms cannot natively integrate with single-node Python frameworks like XGBoost.
E. Structured Streaming only supports JSON and CSV data formats, complicating integration with transactional systems.

Answer: B

Explanation:
Apache Spark's core strength is high-throughput, distributed processing (batch or stream/micro-batch), usually measured in seconds or minutes. It is typically poorly suited for ultra-low latency, row-centric requests where the requirement is consistently below —50ms. The overhead associated with initiating a Spark job or micro-batch execution often makes it impossible to meet SLAs of 10 milliseconds. For such requirements, deploying the model via a specialized low-latency service like Databricks Model Serving (REST API) is recommended.

Question: 8

Which scenarios definitively necessitate the use of Apache Spark's distributed computation engine, specifically supporting ML tasks (MLlib or third-party distributed libraries like xgboost. spark), over utilizing single-machine ML frameworks running on Databricks Runtime ML clusters? (Select ALL correct options.)

A. The required algorithm is natively implemented only within the Spark MLlib library (e.g., distributed KMeans or ALS for collaborative filtering) and no equivalent single-machine implementation exists that supports the scale needed.
B. The cumulative volume of data for model training exceeds the memory constraints of the largest available single-machine cluster instance (e.g., training data > 1
C. The inference requirement demands processing tens of thousands of requests per second (RPS) with a strict latency SLA below 5 milliseconds.
D. The use case involves complex graph processing or running iterative algorithms across billions of vertices and edges.

E. The business objective requires orchestrating a modular ML workflow involving preprocessing, training, and evaluation steps into a single, versioned object for reproducibility.

Answer: A,B,D

Explanation:
SparkML/MLlib is designed for scale and distributed algorithms. Option A: Spark MLlib algorithms (like ALS for recommendation systems or distributed clustering) are optimized to run across a cluster for distributed computation. Option B: When data size surpasses single- machine capacity, necessitating horizontal scaling (the core purpose of Spark). Option D: Complex, iterative graph workloads require distributed memory and processing (e.g., GraphFrames). Option C is incorrect; ultra-low latency requirements favour dedicated serving platforms like Databricks Model Serving, not Spark's general purpose distributed computing model. Option E (using a versioned object like 'Pipeline') is a feature of the Spark ML API, but can also be achieved with single-machine frameworks wrapped in MLflow, so it does not definitively ‘necessitate’ Spark's distributed engine alone.

Question: 9

A data science team is constructing a SparkML Pipeline to prepare high-dimensional sparse data for a distributed clustering algorithm. The pipeline must convert a categorical string column ('customer_segment') to a numeric format and aggregate all relevant inputs into a single feature vector. Given the component instances 'indexer', 'encoder', and 'assembler', which order correctly specifies the required sequence of transformations in the pipeline stages?

- ○ `Pipeline().setStages([assembler, indexer, encoder])`
- ○ `Pipeline().setStages([encoder, assembler, indexer])`
- ○ `Pipeline().setStages([indexer, assembler, encoder])`
- ○ `Pipeline().setStages([indexer, encoder, assembler])`
- ○ `Pipeline().setStages([encoder, indexer, assembler])`

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: D

Explanation:
To correctly process categorical data for machine learning algorithms in a SparkML Pipeline, the string values must first be converted to numerical indices (using 'StringIndexer'), then optionally encoded (using 'OneHotEncoder' or similar techniques) to prevent accidental ordinal interpretation of the indices,

and finally aggregated along with other features into a single input column using 'VectorAssembler'. Therefore, the order is Indexer -> Encoder Assembler.

Question: 10

A machine learning engineer is configuring a hyperparameter search using SparkML's 'CrossValidator'. The goal is to optimize the 'maxDepth' parameter (for depths 5, 10, and 15) of a pre-defined 'DecisionTreeClassifier' instance named 'dt'. Which Python code snippet correctly uses the 'ParamGridBuilder' to define the search space?

- ○ `ParamGridBuilder().addGrid("maxDepth",).build()`
- ○ `ParamGridBuilder().addGrid(dt.maxDepth,).build()`
- ○ `ParamGridBuilder().addGrid(dt, {"maxDepth":}).build()`
- ○ `ParamGridBuilder().addGrid(dt.param("maxDepth"), (5, 10, 15)).build()`
- ○ `ParamGridBuilder().addGrid(dt.param,).build()`

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: B

Explanation:
In SparkML's 'ParamGridBuilder', the 'addGrid' method expects the actual parameter object from the Estimator instance (e.g., 'dt.maxDepth') as the first argument, followed by a list or tuple of values to try for that parameter. The structure is the . addGrid(dt . maxDepth,) canonical and correct way to define the grid for a specific parameter instance.

Thank You for Trying Our Product

Special 16 USD Discount Coupon: NSZUBG3X

Email: support@examsempire.com

Check our Customer Testimonials and ratings available on every product page.

Visit our website.

https://examsempire.com/