DUMPSARENA

Databricks Certified Machine Learning Professional

Databricks Databricks-Machine-Learning-Professional

Version Demo

Total Demo Questions: 11

Total Premium Questions: 112

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Data Processing	16
Topic 2, Model Development	22
Topic 3, Model Deployment	47
Topic 4, Monitoring, Maintenance, and Governance	27
Total	112

QUESTION NO: 1

A machine learning engineer wants to move their model versionmodel_versionfor the MLflow Model Registry modelmodelfrom the Staging stage to the Production stage using MLflow Clientclient. At the same time, they would like to archive any model versions that are already in the Production stage.

Which of the following code blocks can they use to accomplish the task?

A)

B)

C)

D)

A. Option A

B. Option B

C. Option C

D. `client.transition_model_version_stage(name=model, version=model_version, stage="Production", archive_existing_versions=True)`

ANSWER: D

Explanation:

`client.transition_model_version_stage(name=model, version=model_version, stage="Production", archive_existing_versions=True)` is the appropriate MLflow Client call for the legacy Model Registry stage workflow. The `name` argument identifies the registered model, while `version` identifies the specific registered model version to promote. Setting `stage="Production"` performs the requested transition from Staging to Production. Crucially, `archive_existing_versions=True` tells MLflow to archive any other versions that are currently assigned to the target Production stage as part of the transition. This provides a programmatic way to ensure that the promoted version becomes the active Production version while preserving the prior Production versions as archived registry records.

The transition operation is performed through `MlflowClient`, which is the Python API used to manage registered models and their versions. MLflow's API documentation describes `transition_model_version_stage` and its `archive_existing_versions` parameter, while Databricks documents model-version lifecycle management in the Workspace Model Registry. See the MLflow Client API reference and Databricks Workspace Model Registry documentation.

QUESTION NO: 2

A data scientist set up a machine learning pipeline to automatically log a data visualization with each run. They now want to view the visualizations in Databricks.

Which of the following locations in Databricks will show these data visualizations?

A. The MLflow Model Registry Model paqe

B. The Artifacts section of the MLflow Experiment page

C. Logged data visualizations cannot be viewed in Databricks

D. The Artifacts section of the MLflow Run page

E. The Figures section of the MLflow Run page

ANSWER: D

Explanation:

The Artifacts section of the MLflow Run page is correct because visualizations logged during an MLflow run are stored as run artifacts. For example, a pipeline can log a chart with MLflow's figure-logging APIs, or save an image or HTML visualization and log that file as an artifact. Each run has its own artifact collection, preserving the exact outputs produced by that execution alongside its parameters, metrics, tags, and other run metadata. In the Databricks MLflow experiment interface, a user opens the relevant experiment, selects the individual run, and then uses the run's Artifacts section to browse the logged files. Supported visual files can be opened or downloaded from this artifact browser, enabling the data scientist to inspect the visualization associated with that particular pipeline execution. This run-level organization is important for experiment tracking because it keeps visual evidence tied to the specific model, input data, configuration, and metrics from which it was generated. MLflow documents artifacts as output files produced by a run, including images and other files, and Databricks documents viewing and managing run information through MLflow Tracking. See the MLflow Tracking documentation and Databricks MLflow tracking documentation.

QUESTION NO: 3

A machine learning engineer has deployed a model recommender using MLflow Model Serving. They now want to query the version of that model that is in the Production stage of the MLflow Model Registry.

Which of the following model URIs can be used to query the described model version?

A. https:// /model-serving/recommender/Production/invocations

B. The version number of the model version in Production is necessary to complete this task.

C. https:// /model/recommender/stage-production/invocations

D. https:// /model-serving/recommender/stage-production/invocations

E. https:// /model/recommender/Production/invocations

F. models:/recommender/Production

ANSWER: F

Explanation:

`models:/recommender/Production` is the MLflow Model Registry URI that resolves the registered model named `recommender` to whichever version is currently assigned to the `Production` stage. MLflow model URIs use the `models:/` scheme, followed by the registered model name and then a version identifier, a stage name, or (in newer workflows) an alias. Consequently, this URI lets MLflow resolve the stage dynamically rather than requiring a hard-coded numerical version. It can be used anywhere an MLflow model URI is accepted, such as loading a model through MLflow APIs. This is distinct from an HTTP invocation address for a serving endpoint: a registry URI identifies and resolves a model version in the registry. The stage-based URI is applicable to the Workspace Model Registry lifecycle model described in the question. Databricks now recommends aliases rather than stages for new model-management workflows, but `Production` remains the relevant stage identifier for the stated scenario. See the MLflow documentation on Model Registry and the Databricks documentation on model lifecycle management.

QUESTION NO: 4

Which of the following is a simple, low-cost method of monitoring numeric feature drift?

A. Jensen-Shannon test

B. Summary statistics trends

C. Chi-squared test

D. None of these can be used to monitor feature drift

E. Kolmogorov-Smirnov (KS) test

ANSWER: B

Explanation:

Summary statistics trends are a simple, low-cost method for monitoring drift in numeric features because they reduce each batch or time window to a small set of easily computed aggregates. Teams can track measures such as the mean, median, standard deviation, minimum and maximum values, null rate, and selected percentiles, then compare their values with a baseline or observe their movement over time. Meaningful changes in these metrics can reveal shifts in central tendency, spread, range, missingness, or extreme values that may affect model behavior.

This approach is operationally practical because the required aggregates can be calculated directly in batch or streaming data pipelines without retaining all raw observations or performing computationally heavier distribution-comparison procedures. It also supports straightforward alerting: for example, an alert can be raised when a feature's mean or percentile moves outside a defined tolerance from its training or recent-production baseline. Databricks model monitoring supports monitoring input data quality and drift-related metrics over time, helping teams identify changes that require investigation or model maintenance. See the Databricks model monitoring documentation and the MLflow model monitoring documentation.

QUESTION NO: 5

A data scientist wants to examine the feature values that were supplied to a model during a particular MLflow training run. The training DataFrame is available as a pandas DataFrame named `train_df`.

Which of the following MLflow operations can be used to log the training data as an input dataset for the active run?

A. `mlflow.log_input(mlflow.data.from_pandas(train_df), context="training")`

B. `mlflow.log_metric(mlflow.data.from_pandas(train_df), "training")`

C. `mlflow.register_model(train_df, "training")`

D. `mlflow.log_param("training", train_df)`

E. `mlflow.sklearn.log_model(train_df, "training")`

ANSWER: A

Explanation:

`mlflow.log_input(mlflow.data.from_pandas(train_df), context="training")` is correct because MLflow dataset tracking records an input dataset and its context with the active run. The `from_pandas` constructor creates an MLflow dataset representation from the pandas DataFrame, and the context identifies how that dataset was used, such as for training, validation, or testing.

Logging dataset inputs improves run lineage by associating model parameters and metrics with the data used to produce them. Logging the DataFrame as a model artifact is not the dataset-tracking API, and model registration does not record the input dataset for a run. See the MLflow Dataset Tracking documentation.

QUESTION NO: 6

Which of the following tools can assist in real-time deployments by packaging software with its own application, tools, and libraries?

A. Cloud-based compute

B. None of these tools

C. REST APIs

D. Containers

E. Autoscaling clusters

ANSWER: D

Explanation:

Containers are the correct choice because a container packages an application with the runtime, libraries, system tools, and other dependencies required for it to execute consistently. This creates a portable, isolated deployment artifact that can run predictably across development, test, and production environments, provided that a compatible container runtime is available. That consistency is particularly valuable for real-time machine learning deployments, where an inference service must start reliably and serve requests using the exact dependency versions validated during development. Container images also support repeatable build and release processes: teams define the software environment as code, build a versioned image, and deploy that same image to an appropriate serving platform. Databricks supports custom model serving environments through MLflow, enabling model deployments that require dependencies beyond the standard serving environment. Docker similarly defines containers as packages containing the application and everything needed to run it, while sharing the host operating system kernel rather than requiring a complete virtual machine per application. See Databricks documentation on custom model serving environments and Docker's container overview.

QUESTION NO: 7

Which of the following is an advantage of using the python_function(pyfunc) model flavor over the built-in library-specific model flavors?

A. python_function provides no benefits over the built-in library-specific model flavors

B. python_function can be used to deploy models in a parallelizable fashion

C. python_function can be used to deploy models without worrying about which library was used to create the model

D. python_function can be used to store models in an MLmodel file

E. python_function can be used to deploy models without worrying about whether they are deployed in batch, streaming, or real-time environments

ANSWER: C

Explanation:

`python_function can be used to deploy models without worrying about which library was used to create the model` is correct because the MLflow `python_function` flavor, commonly called pyfunc, supplies a standard inference interface for MLflow models. A consumer loads the model through `mlflow.pyfunc.load_model()` and invokes `predict()`, rather than needing to use a framework-specific loading API and prediction object. This creates a consistent integration point for deployment and scoring code when models originate from different supported libraries, such as scikit-learn, Spark ML, XGBoost, or another framework that exposes or includes a pyfunc flavor.

The uniform interface is particularly valuable for platform teams and production pipelines: a shared serving, batch-scoring, or UDF integration can interact with the pyfunc contract instead of branching based on the model's native implementation. The model's environment and required dependencies are still packaged or specified with the MLflow model, so the underlying library remains available where needed; the key benefit is that model consumers use one portable prediction API. MLflow documents pyfunc as its generic Python model flavor and describes how it provides this common loading and inference behavior. See the MLflow Models documentation and the MLflow pyfunc API reference.

QUESTION NO: 8

A machine learning engineer is in the process of implementing a concept drift monitoring solution. They are planning to use the following steps:

1. Deploy a model to production and compute predicted values

2. Obtain the observed (actual) label values

3.

4. Run a statistical test to determine if there are changes over time Which of the following should be completed as Step #3?

A. Obtain the observed values (actual) feature values

B. Measure the latency of the prediction time

C. Retrain the model

D. None of these should be completed as Step #3

E. Compute the evaluation metric using the observed and predicted values

ANSWER: E

Explanation:

Compute the evaluation metric using the observed and predicted values is the required next step because concept drift monitoring depends on measuring whether the model's predictive relationship with the real-world outcome has changed. Once production predictions are joined with subsequently available ground-truth labels, the engineer can calculate an appropriate performance measure for each monitoring window. Examples include accuracy, precision, recall, F1, ROC AUC, or log loss for classification, and RMSE, MAE, or R^2 for regression. The selected metric should align with the model's objective and the business impact of errors.

Those metric values form a time series or set of comparable baseline-versus-current samples. A statistical test in the next step can then determine whether the observed performance change is statistically meaningful rather than routine variation. This workflow is especially important because concept drift concerns a change in the relationship between inputs and the target; evaluating predictions against actual labels directly measures the practical effect of that change. Databricks' MLOps guidance describes monitoring model quality and using production feedback to detect degradation and drive model lifecycle decisions. See Databricks: MLOps at Databricks—Model Monitoring and Databricks documentation on MLflow models and lifecycle management.

QUESTION NO: 9

A machine learning engineer wants to deploy a model for real-time serving using MLflow Model Serving. For the model, the machine learning engineer currently has one model version in each of the stages in the MLflow Model Registry. The engineer wants to know which model versions can be queried once Model Serving is enabled for the model.

Which of the following lists all of the MLflow Model Registry stages whose model versions are automatically deployed with Model Serving?

A. Staging. Production. Archived

B. Production

C. None. Staging. Production. Archived

D. Staging. Production

E. [None. Staging. Production

ANSWER: B

Explanation:

Production is the stage whose model version is automatically deployed and available for query when the legacy MLflow Model Serving integration is enabled for a registered model. The Production stage represents the approved live model in a registry-based lifecycle. Consequently, moving a new version into Production promotes that version as the model intended to receive real-time inference traffic, allowing serving to remain aligned with the version designated for production use.

This behavior supports a controlled deployment workflow: teams can register and evaluate candidate versions, apply their validation and approval process, and then use the Production transition as the deployment signal for the live serving workload. The serving integration therefore treats Production as the operational source of truth for the automatically served model version. Databricks now generally recommends using model aliases and explicitly configured serving endpoints for

newer workflows, but the stage-based automatic-deployment behavior described in this question is specifically associated with the legacy MLflow Model Serving and Model Registry stage workflow.

See the Databricks Model Serving documentation and the MLflow Model Registry documentation for lifecycle and deployment concepts.

QUESTION NO: 10

A data scientist is training a binary classification model using a dataset in which multiple records belong to the same customer. The scientist wants to evaluate the model on customers that were not represented in the training data.

Which of the following data-splitting approaches should be used?

A. Randomly split individual records into training and test datasets

B. Use a group-based split with customer_id as the grouping column

C. Use the same customers in training and test data, but remove duplicate rows

D. Sort by customer_id and use the first 80% of rows for training

E. Use the complete dataset for both training and evaluation

ANSWER: B

Explanation:

A group-based train-test split using `customer_id` is correct because it keeps all records for an individual customer in the same split. This prevents the model from training on one record from a customer and being evaluated on another record from that same customer. Such overlap can inflate offline performance estimates because customer-specific patterns may be learned during training.

By assigning entire customer groups to either training or evaluation data, the resulting evaluation better represents model behavior for previously unseen customers. A random record-level split does not provide this protection when repeated entities exist in the dataset. Scikit-learn documents `GroupShuffleSplit` and related group-aware cross-validation strategies for this purpose. See the scikit-learn documentation for grouped data splits.

QUESTION NO: 11

A machine learning engineer is reviewing an MLflow run that logged a model artifact at the artifact path `model`. The engineer knows the run ID is stored in the variable `run_id` and wants to construct a model URI that refers to this logged model.

Which of the following strings is the correct MLflow model URI?

A. `models:/{run_id}/model`

B. `runs:/{run_id}/model`

C. `run:/{run_id}/models/model`

D. `artifacts:/{run_id}/model`

E. `mlflow:/{run_id}/model`

ANSWER: B

Explanation:

`runs:/{run_id}/model` is correct because the `runs:/` URI scheme references an artifact within a specific MLflow run. The run ID identifies the run that produced the artifact, and `model` is the artifact path supplied when the model was logged. In Python, this can be constructed as `f"runs:/{run_id}/model"`.

This URI can be passed to an appropriate MLflow model-loading API, such as a flavor-specific `load_model` function or the generic pyfunc loader. A `models:/` URI instead references a model registered in Model Registry, while a local file path does not identify a run artifact in MLflow Tracking. See the MLflow documentation for model URIs.