DUMPSARENA

Databricks Certified Machine Learning Associate Exam

Databricks Databricks-Machine-Learning-Associate

Version Demo

Total Demo Questions: 13

Total Premium Questions: 133

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Databricks Machine Learning	16
Topic 2, ML Workflows	32
Topic 3, Spark ML	70
Topic 4, MLflow	15
Total	133

QUESTION NO: 1

A data scientist is working with a feature set with the following schema:

```
customer_id STRING,
spend DOUBLE,
units INTEGER,
loyalty_tier STRING
```

Thecustomer_idcolumn is the primary key in the feature set. Each of the columns in the feature set has missing values. They want to replace the missing values by imputing a common value for each feature.

Which of the following lists all of the columns in the feature set that need to be imputed using the most common value of the column?

A. customer_id, loyalty_tier

B. loyalty_tier

C. units

D. spend

E. customer_id

ANSWER: B

Explanation:

loyalty_tier is the feature that should be filled with its most common observed value, also called the mode. This is appropriate because loyalty tier is a categorical feature: its values represent distinct membership classes rather than measurements on a numeric scale. Using the most frequent tier preserves a valid category and supplies a reasonable default based on the distribution already present in the training data.

In a feature-engineering workflow, imputation strategy should match the feature's data type and semantics. Categorical variables are commonly filled with a mode or, when appropriate for the business context, with a dedicated "unknown" category. Numeric measures such as spending and unit counts are generally handled with a numeric statistic such as a mean or median, while a primary-key identifier should not be treated as an analytical feature for statistical imputation. This distinction prevents an identifier or a numeric measurement from being processed as though it were a category.

Databricks machine-learning workflows use Spark ML preprocessing components for this type of transformation. Spark's Imputer documentation describes supported numeric imputation strategies, including mean, median, and mode. For broader Databricks guidance on preparing data for ML, see Databricks Machine Learning documentation.

QUESTION NO: 2

A machine learning engineer wants to parallelize the inference of group-specific models using the Pandas Function API. They have developed theapply_modelfunction that will look up and load the correct model for each group, and they want to apply it to each group of DataFramedf.

They have written the following incomplete code block:

Which piece of code can be used to fill in the above blank to complete the task?

A. prediction_df = df.groupby("device_id").applyInPandas(apply_model, schema=apply_return_schema)

ANSWER: A

Explanation:

`prediction_df = df.groupby("device_id").applyInPandas(apply_model, schema=apply_return_schema)` is the appropriate grouped Pandas Function API pattern. Calling `groupby("device_id")` creates a grouped Spark DataFrame in which all rows for an individual device are presented together. `applyInPandas` then invokes `apply_model` once per group, passing the group as a pandas DataFrame. This gives the function the group identity and its associated records, allowing it to look up and load the model specific to that device and produce predictions for that group. Spark can distribute groups across executors, enabling inference to be performed in parallel where the cluster and group partitioning permit it.

The explicit `schema=apply_return_schema` argument is required because Spark must know the structure and data types of the pandas DataFrame returned by the function before it converts the results back to a Spark DataFrame. The function must return pandas DataFrames matching that declared schema for every processed group. Databricks documents grouped-map Pandas UDF processing through `GroupedData.applyInPandas`, while the Apache Spark API specifies its per-group execution and schema contract. See Databricks Pandas function APIs and PySpark GroupedData.applyInPandas.

QUESTION NO: 3

In which of the following situations is it preferable to impute missing feature values with their median value over the mean value?

A. When the features are of the categorical type

B. When the features are of the boolean type

C. When the features contain a lot of extreme outliers

D. When the features contain no outliers

E. When the features contain no missing no values

ANSWER: C

Explanation:

When the features contain a lot of extreme outliers, median imputation is the preferable approach. The arithmetic mean is calculated from every value's magnitude, so a relatively small number of unusually large or small observations can substantially shift it away from the value that represents the center of most observations. Using that shifted mean to fill missing entries may introduce values that are not representative of the typical data point. The median is the middle value after sorting the observed values and is much less affected by extreme observations. Consequently, it provides a robust estimate of central tendency for numeric features with outliers or substantial skew. In practical preprocessing workflows, an imputer estimates its replacement statistic using the available, non-null values in each feature, then uses that statistic consistently to fill missing entries. This helps retain a sensible distribution of feature values before model training. Databricks machine learning workflows commonly use scikit-learn-compatible preprocessing, where `SimpleImputer` supports both mean and median strategies; its documentation specifically notes that these strategies apply to numeric data. See the scikit-learn imputation guide and the SimpleImputer API reference.

QUESTION NO: 4

A data scientist has produced three new models for a single machine learning problem. In the past, the solution used just one model. All four models have nearly the same prediction latency, but a machine learning engineer suggests that the new solution will be less time efficient during inference. In which situation will the machine learning engineer be correct?

A. When the new solution requires if-else logic determining which model to use to compute each prediction

B. When the new solution's models have an average latency that is larger than the size of the original model

C. When the new solution requires the use of fewer feature variables than the original model

D. When the new solution requires that each model computes a prediction for every record

E. When the new solution's models have an average size that is larger than the size of the original model

ANSWER: D

Explanation:

When the new solution requires that each model computes a prediction for every record, every inference request performs multiple model-evaluation operations rather than the single evaluation used by the previous solution. Although the individual models have nearly identical prediction latency, the end-to-end work per record increases because each record must be passed through all of the models. This is the typical pattern for an ensemble, such as one that averages regression predictions, combines probabilities, or selects a class through voting. The system must also aggregate the individual outputs into one final prediction. As a result, sequential execution increases request latency, while parallel execution still consumes substantially more compute capacity and can lower overall throughput under load. In Databricks Model Serving, endpoint performance and capacity planning must account for the resources required to execute the model inference workload for incoming requests. Serving several model computations for each input therefore requires more inference work than serving one computation per input, even where individual model latencies are similar. See the Databricks documentation on Model Serving and MLflow documentation on model deployment and registry workflows.

QUESTION NO: 5

Which of the following tools can be used to distribute large-scale feature engineering without the use of a UDF or pandas Function API for machine learning pipelines?

A. Keras

B. Scikit-learn

C. PyTorch

D. Spark ML

ANSWER: D

Explanation:

Spark ML is designed for distributed machine learning workflows on Spark DataFrames, including scalable feature engineering. Its pipeline framework uses native Transformer and Estimator stages, allowing transformations to be planned and executed in parallel by the Spark engine across a cluster. Common feature-engineering stages include StringIndexer for categorical indexing, OneHotEncoder for categorical representation, VectorAssembler for combining input columns into a feature vector, StandardScaler for normalization, and text-processing transformers such as Tokenizer and HashingTF. These stages can be connected in a Pipeline, then fit and applied consistently to training and inference data.

Because these are Spark-native DataFrame operations, they avoid the serialization and execution overhead commonly associated with Python UDFs and do not require the pandas Function API. This makes Spark ML particularly appropriate when datasets are too large for a single machine and the feature transformations can be expressed through its built-in distributed stages. Databricks recommends Spark ML for scalable preprocessing and pipeline-based model development on distributed data. See the Apache Spark ML programming guide and Databricks Spark ML documentation.

QUESTION NO: 6

A data scientist is using a decision tree classifier that achieves very high training accuracy but substantially lower validation accuracy. The scientist believes the tree is overfitting the training data.

Which of the following hyperparameter changes can help reduce the complexity of the decision tree?

A. Reduce maxDepth

B. Increase maxDepth

C. Increase the number of label classes

D. Replace the validation DataFrame with the training DataFrame

E. Increase the prediction threshold to 1.0

ANSWER: A

Explanation:

Reducing `maxDepth` can help reduce decision-tree complexity. This hyperparameter limits how many levels the tree can grow. A smaller maximum depth restricts the number of sequential splits that can be created, which can reduce the model's tendency to memorize noise or highly specific patterns in the training data.

The appropriate value should be selected using validation data or cross-validation because an excessively shallow tree can underfit. Spark ML exposes `maxDepth` for decision tree classifiers and regressors as a primary control on tree complexity. See the PySpark DecisionTreeClassifier API documentation and the Spark ML decision tree documentation.

QUESTION NO: 7

An organization is developing a feature repository and is electing to one-hot encode all categorical feature variables. A data scientist suggests that the categorical feature variables should not be one-hot encoded within the feature repository.

Which of the following explanations justifies this suggestion?

A. One-hot encoding is not supported by most machine learning libraries.

B. One-hot encoding is dependent on the target variable's values which differ for each application.

C. One-hot encoding is computationally intensive and should only be performed on small samples of training sets for individual machine learning problems.

D. One-hot encoding is not a common strategy for representing categorical feature variables numerically.

E. One-hot encoding is a potentially problematic categorical variable strategy for some machine learning algorithms.

ANSWER: E

Explanation:

One-hot encoding is a potentially problematic categorical variable strategy for some machine learning algorithms because a feature repository should generally provide reusable, broadly applicable feature values rather than impose a representation that is optimal only for particular downstream models. One-hot encoding can greatly expand the number of columns when a categorical feature has many distinct values. This produces sparse data, increases storage and computation requirements, and can make training or serving less efficient. The appropriate categorical-feature treatment is also model dependent. Some model families can use categorical values directly or benefit from alternative encodings, while other models require a numeric representation such as one-hot encoding. Applying the transformation universally in the repository therefore reduces flexibility for teams that train different model types from the same governed feature data. Keeping the canonical categorical feature available allows each training pipeline to apply a compatible encoding consistently to its selected model and then persist that preprocessing logic with the model. Databricks recommends using feature engineering practices that maintain consistency between training and inference, while its AutoML documentation notes that preprocessing choices are made based on data and model requirements. See Databricks feature engineering documentation and Databricks AutoML documentation.

QUESTION NO: 8

A data scientist has created a linear regression model that uses log(price) as a label variable. Using this model, they have performed inference and the predictions and actual label values are in Spark DataFrame preds_df.

They are using the following code block to evaluate the model:
regression_evaluator.setMetricName("rmse").evaluate(preds_df)

Which of the following changes should the data scientist make to evaluate the RMSE in a way that is comparable with price?

A. They should exponentiate the computed RMSE value

B. They should take the log of the predictions before computing the RMSE

C. They should evaluate the MSE of the log predictions to compute the RMSE

D. They should exponentiate the predictions and label values before computing the RMSE

ANSWER: D

Explanation:

They should exponentiate the predictions and label values before computing the RMSE. The regression model was trained with `log(price)` as its target, so its `prediction` output represents predicted log-price. The corresponding `label` values used by Spark ML evaluation are likewise on the transformed log scale. Consequently, evaluating RMSE directly measures the typical prediction error in logarithmic units, rather than in the original price units.

To report an error that can be interpreted relative to price, create columns that apply the inverse of the label transformation—normally `exp()` for a natural-log transformation—to both the prediction and label columns. Configure `RegressionEvaluator` to use those derived columns and evaluate with `metricName="rmse"`. This calculates the square root of the mean squared difference between predicted prices and observed prices, yielding an RMSE in the same units as `price`. For example, a result of 25 can then be read as a typical error magnitude of roughly 25 price units. Spark's evaluator supports selecting the prediction and label columns used for regression metrics; see the PySpark RegressionEvaluator API and the PySpark exp function documentation.

QUESTION NO: 9

Which of the following machine learning algorithms typically uses bagging?

A. Gradient boosted trees

B. K-means

C. Random forest

D. Linear regression

E. Decision tree

ANSWER: C

Explanation:

Random forest typically uses bagging, short for bootstrap aggregating. In a random forest, the training process creates multiple decision trees independently. Each tree is trained using a bootstrap sample: a sample drawn from the original training dataset with replacement. The forest then combines the independently generated trees' predictions, ordinarily using majority voting for classification tasks or averaging for regression tasks. This aggregation reduces the variance and instability that an individual decision tree can exhibit, producing a more robust model.

Random forests also introduce random feature selection at each candidate split. This added randomness decreases correlation among individual trees, so combining their predictions is more effective than it would be if every tree tended to make identical decisions. In Spark ML, random forests are tree ensembles whose training includes subsampling of rows and selection of feature subsets, reflecting the bagging-based ensemble approach. This makes random forest the standard algorithm associated with bagging in machine learning and the appropriate answer for this question. See the Apache Spark ML random forest documentation and scikit-learn's ensemble-method documentation.

QUESTION NO: 10

A data scientist wants to use Spark ML to one-hot encode the categorical features in their PySpark DataFramefeatures_df. A list of the names of the string columns is assigned to theinput_columnsvariable.

They have developed this code block to accomplish this task:

```
ohe = OneHotEncoder(
    inputCols=input_columns,
    outputCols=output_columns
)
ohe_model = ohe.fit(features_df)
ohe_features_df = ohe_model.transform(features_df)
```

The code block is returning an error.

Which of the following adjustments does the data scientist need to make to accomplish this task?

A. They need to specify the method parameter to the OneHotEncoder.

B. They need to remove the line with the fit operation.

C. They need to use StringIndexer prior to one-hot encoding the features.

D. They need to useVectorAssemblerprior to one-hot encoding the features.

ANSWER: C

Explanation:

They need to use StringIndexer prior to one-hot encoding the features. Spark ML's `OneHotEncoder` accepts categorical features represented as numeric indices, not raw string values. `StringIndexer` is the appropriate transformer for this preparation step: it learns a mapping from each distinct string label in a column to a numeric index and produces an indexed output column. The fitted indexer model should then transform `features_df` before those indexed columns are supplied to `OneHotEncoder`.

For multiple categorical columns, a common Spark ML approach is to create one `StringIndexer` for each name in `input_columns`, with output names such as `column_index`. After fitting and applying the indexers, configure `OneHotEncoder` with those index-column names as `inputCols` and the desired encoded-vector names as `outputCols`. This sequence is also suitable for a Pipeline, ensuring the category-to-index mappings learned during training are consistently applied when transforming later data. Spark documents that OneHotEncoder encodes category indices, while StringIndexer handles conversion of string categorical labels into those indices. See the PySpark OneHotEncoder API documentation and the PySpark StringIndexer API documentation.

QUESTION NO: 11

A data scientist wants to efficiently tune the hyperparameters of a scikit-learn model. They elect to use the Hyperopt library'sfminoperation to facilitate this process. Unfortunately, the final model is not very accurate. The data scientist suspects that there is an issue with theobjective_functionbeing passed as an argument tofmin.

They use the following code block to create theobjective_function:

```
def objective_function(params):
    max_depth = params["max_depth"]
    max_features = params["max_features"]
    regressor = RandomForestRegressor(
        max_depth=max_depth,
        max_features=max_features
    )
    r2 = mean(cross_val_score(regressor, X_train, y_train, cv=3))
    return r2
```

Which of the following changes does the data scientist need to make to theirobjective_functionin order to produce a more accurate model?

A. Add test set validation process

B. Add a random_state argument to the RandomForestRegressor operation

C. Remove the mean operation that is wrapping the cross_val_score operation

D. Replace the r2 return value with -r2

E. Replace the fmin operation with the fmax operation

ANSWER: D

Explanation:

Replace the r2 return value with -r2 is correct because Hyperopt's `fmin` function is designed to minimize the numeric loss returned by the objective function. In this case, cross-validation produces an R^2 score, for which larger values indicate better predictive performance: an R^2 value closer to 1 means the model explains more of the variation in the target data. Returning R^2 directly to `fmin` causes the optimization to prefer smaller R^2 values, which is the opposite of the intended model-selection goal.

Negating the mean cross-validated R^2 score converts this maximization metric into a minimization-compatible loss. For example, an R^2 of 0.90 becomes a loss of -0.90, while an R^2 of 0.40 becomes -0.40. Because -0.90 is smaller, `fmin` selects the hyperparameter configuration associated with the stronger model. This preserves the useful aggregation of scores across folds while aligning the objective's direction with Hyperopt's optimization behavior. After tuning, the selected parameters can be used to fit the final estimator on the training data and evaluated on held-out data as appropriate. Hyperopt documents that `fmin` minimizes an objective function, and scikit-learn documents R^2 as a regression score where higher values are better. See Hyperopt documentation and scikit-learn R^2 documentation.

QUESTION NO: 12

A data scientist has a Spark DataFrame spark_df. They want to create a new Spark DataFrame that contains only the rows from spark_df where the value in column price is greater than 0.

Which of the following code blocks will accomplish this task?

A. spark_df[spark_df["price"] > 0]

B. spark_df.filter(col("price") > 0)

C. SELECT * FROM spark_df WHERE price > 0

D. spark_df.loc[spark_df["price"] > 0,:]

E. spark_df.loc[:,spark_df["price"] > 0]

ANSWER: B

Explanation:

`spark_df.filter(col("price") > 0)` is correct because Spark DataFrame transformations use expressions based on Spark `Column` objects to define row-level conditions. Here, `col("price")` identifies the `price` column, and the comparison `> 0` constructs a Boolean column expression that is true only for records whose price is positive. Passing that expression to `filter()` produces a new DataFrame containing the matching rows; DataFrames are immutable, so the original `spark_df` is not modified. The result can be assigned to a variable, such as `positive_price_df = spark_df.filter(col("price") > 0)`, and can then participate in additional distributed Spark transformations or actions. In PySpark, `filter()` is the standard DataFrame API method for retaining rows that satisfy a condition, and `where()` is an equivalent alias. The `col` function must be available in the notebook or script, typically through `from pyspark.sql.functions import col`. See the official PySpark DataFrame.filter documentation and the PySpark col function documentation.

QUESTION NO: 13

Which of the following statements describes a Spark ML estimator?

A. An estimator is a hyperparameter arid that can be used to train a model

B. An estimator chains multiple alqorithms toqether to specify an ML workflow

C. An estimator is a trained ML model which turns a DataFrame with features into a DataFrame with predictions

D. An estimator is an alqorithm which can be fit on a DataFrame to produce a Transformer

E. An estimator is an evaluation tool to assess to the quality of a model

ANSWER: D

Explanation:

An estimator is an algorithm that can be fit on a DataFrame to produce a Transformer. This is the central abstraction used by the Spark ML DataFrame-based API for training stages. An estimator implements a `fit()` method: it consumes a dataset, learns any required parameters from that data, and returns a fitted model. That returned model is a Transformer, meaning it implements `transform()` and can apply its learned behavior to a DataFrame. For supervised learning, the transformed DataFrame commonly includes a prediction column; for feature-processing stages, it may include derived or encoded feature columns instead. For example, `LogisticRegression` is an estimator. Fitting it to labeled training data produces a `LogisticRegressionModel`, which is a transformer that generates predictions for input records. This estimator-to-transformer pattern lets Spark ML pipelines consistently combine data preparation, model training, and inference stages. Spark's official ML pipelines documentation defines an estimator as an algorithm that can be fit on a DataFrame to produce a transformer, and its API documentation specifies that calling `fit()` produces a model or fitted transformer. See the Spark ML Pipelines guide and the PySpark Estimator API reference.