Test ID: 127520908491063

Test Date: September 14, 2022

Role	Potential fit
Data science	Strong Consider

Automata Data Science

100 /100

Automata Data Science

100 / 100

Data Wrangling and Munging	ML Workflow Implementation	Library Usage and Implementation
100 / 100	**100** / 100	**100** / 100

1 | Introduction

About the Report

This report provides a detailed analysis of the candidate's performance on different assessments. The tests for this job role were decided based on job analysis, O*Net taxonomy mapping and/or criterion validity studies. The candidate’s responses to these tests help construct a profile that reflects her/his likely performance level and achievement potential in the job role

This report has the following sections:

The **Summary** section provides an overall snapshot of the candidate’s performance. It includes a graphical representation of the test scores and the subsection scores.

The **Response** section captures the response provided by the candidate. This section includes only those tests that require a subjective input from the candidate and are scored based on artificial intelligence and machine learning.

The **Proctoring** section captures the output of the different proctoring features used during the test.

Score Interpretation

All the test scores are on a scale of 0-100. All the tests except personality and behavioural evaluation provide absolute scores. The personality and behavioural tests provide a norm-referenced score and hence, are percentile scores. Throughout the report, the colour codes used are as follows:

- Scores between 67 and 100
- Scores between 33 and 67
- Scores between 0 and 33

2 | Response

Automata Data Science

Code Replay 100 / 100

Question 1 (Language: Python3)

You are given a matrix **A**. Transform it into a new matrix **B** such that its i^{th} column has a mean **i** and variance i^4.

Note: **i** equals 1 for the first column and increases sequentially.

The input to the function **editMatrix** shall be a matrix **X**. Return the transformed matrix **B**.

The test cases tab illustrates some examples.

Scores

Final Code Submitted **Compilation Status: Pass**

```
1 def editMatrix(X):
2   import numpy as np
3   ans=np.zeros(X.shape)
4   for j in range(0,X.shape[1]):
5     ans[:,[j]]=((X[:,[j]]-np.mean(X[:,[j]]))/np.std(X[:,[j]]))*(j+1)*(j+1)+(j+
  1)
6   return ans
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution Passed TC: **100%**

Total score 5/5

0%	0%	100%
Basic(**0**/0)	Advance(**0**/0)	Edge(**5**/5)

Test Cases: Deep Dive

Compilation Statistics

0	0	0	0	0	0
Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors

Response time:	**00:15:54**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**100%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 2 (Language: Python3)

Your manager has asked you to build a prediction model using the company's marketing data. The data is a mix of numerical as well as categorical attributes. You only need to one-hot encode (N-1 dummy variables for N categories) the categorical attributes. You can use the numerical attributes as is.

Given an input feature dataframe and a column matrix of true responses, your task is to build a prediction model using linear regression, after applying one-hot encoding on the categorical attributes. Calculate the Mean Squared Error (MSE) between the true responses and the predicted responses.

The inputs to the function **linearRegressionMSE** shall be a dataframe **X** of features and a column matrix **y** of true responses. The function must return the Mean Squared Error (MSE).

The test cases tab illustrates some examples.

Scores

Final Code Submitted — Compilation Status: Pass

```
1  def linearRegressionMSE(X,y):
2    import numpy as np
3    from sklearn.preprocessing import OneHotEncoder
4    from sklearn.linear_model import LinearRegression
5    from sklearn.metrics import mean_squared_error
6
7    enc=OneHotEncoder()
8    ncols=X.shape[1]
9    nrows=X.shape[0]
10
11   for col in X.columns:
12       if X[col].dtype == object:
13           unique, unique_inverse=np.unique(X[col],return_inverse=Tru
     e)
14           unique_inverse=unique_inverse.reshape(len(unique_inverse),
     1)
15           bits=enc.fit_transform(unique_inverse).toarray()
16
17           bits=bits[:,range(0,bits.shape[1]-1)]
18
19           if(col=='c0'):
20               preds=bits
21
22           else:
23               preds=np.concatenate((preds,bits),axis=1)
24       else:
25           if(col=='c0'):
26               preds=X[col].values.reshape(nrows,1)
27           else:
28               preds=np.concatenate((preds,X[col].values.reshape(nrows,
     1)),axis = 1)
29
30   LR=LinearRegression()
31   LR.fit(preds,y)
32   y_pred=LR.predict(preds)
33   ans=mean_squared_error(y,y_pred)
34   return ans
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution — Passed TC: **100%**

Total score

5/5

0%	0%	100%
Basic(**0**/0)	Advance(**0**/0)	Edge(**5**/5)

Test Cases: Deep Dive

Compilation Statistics

Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors
0	0	0	0	0	0

Response time:	**00:11:28**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**100%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

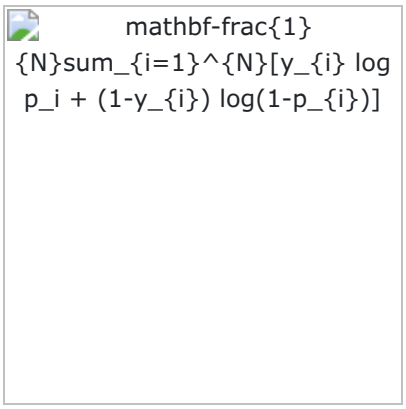
Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

Question 3 (Language: Python3)

Emma has to implement a classification algorithm for a 2-class problem. For each sample, her algorithm outputs the probability of the sample belonging to a particular class. She wants to evaluate the log-loss error of the predictions made by her algorithm.

The Log-loss error is defined as follows:



The input to the function **logarithmic_loss** shall be two column matrices **y** and **y_prob**, storing the true class labels of samples and the probability of the sample belonging to a particular class respectively. The function must return the log-loss error.

The test cases tab illustrates some examples.

Scores

Final Code Submitted **Compilation Status: Pass**

```
1 def logarithmic_loss(y,y_prob):
2   from sklearn.metrics import log_loss
3   ans=log_loss(y,y_prob)
4   return ans
```

Code Analysis

Average-case Time Complexity

Candidate code: Complexity is reported only when the code is correct and it passes all the basic and advanced test cases.

Best case code:

*N represents

Errors/Warnings

There are no errors in the candidate's code.

Structural Vulnerabilites and Errors

There are no errors in the candidate's code.

Test Case Execution Passed TC: **100%**

Total score 5/5

0%	0%	100%
Basic(**0**/0)	Advance(**0**/0)	Edge(**5**/5)

Test Cases: Deep Dive

Compilation Statistics

Total attempts	Successful	Compilation errors	Sample failed	Timed out	Runtime errors
0	0	0	0	0	0

Response time:	**00:01:43**
Average time taken between two compile attempts:	**00:00:00**
Average test case pass percentage per compile:	**100%**

Average-case Time Complexity

Average Case Time Complexity is the order of performance of the algorithm given a random set of inputs. This complexity is measured here using the Big-O asymptotic notation. This is the complexity detected by empirically fitting a curve to the run-time for different input sizes to the given code. It has been benchmarked across problems.

Test Case Execution

There are three types of test-cases for every coding problem:

Basic: The basic test-cases demonstrate the primary logic of the problem. They include the most common and obvious cases that an average candidate would consider while coding. They do not include those cases that need extra checks to be placed in the logic.

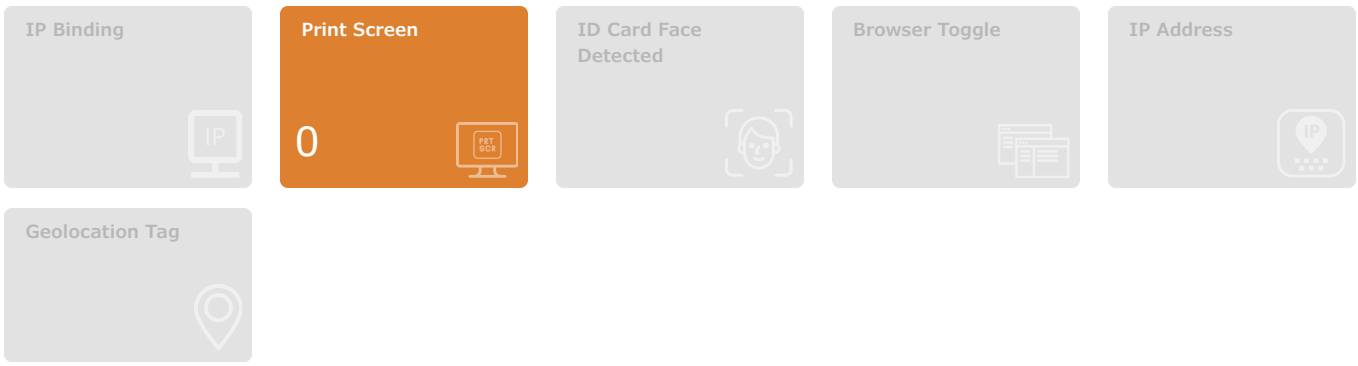
Advanced: The advanced test-cases contain pathological input conditions that would attempt to break the codes which have incorrect/semi-correct implementations of the correct logic or incorrect/semi-correct formulation of the logic.

Edge: The edge test-cases specifically confirm whether the code runs successfully even under extreme conditions of the domain of inputs and that all possible cases are covered by the code

3 | Proctoring

Proctoring Index | Low

The Proctoring Index is a measure of the likelihood a participant was engaging in potentially suspicious behavior. The Index score constitutes a set of pre-determined parameters that are outlined below as a breakdown of the measure. Please hover over the tiles for a brief overview of the respective parameter.



AI Proctoring Information

Print Screen: The number of times the candidate attempted to take a screenshot of the assessment screen using the “print screen” function on their device. Note: This impacts proctoring index.

ID Card Face Detected: Looks at the candidate images captured during the assessment and flags anywhere different people appear to be present. Snapshots are included in the report.

Browser Toggle: Either the proportion of time the candidate spent focused on a tab/window other than that of assessment screen (%), or the number of times the candidate toggled to another tab/window (count). Note: This impacts proctoring index.

IP Address: Confirms that the candidate took the assessment from the specified IP address(s).

Geolocation Tag: Detects whether the candidate attempted the assessment from a location beyond the distance set by the administrator.