Logical_agents

June 28, 2021

1 Logical Agents

Author: Jessica Cervi
Expected time = 1.5 hours
Total points = 75 points

1.1 Assignment Overview

This assignment is designed to reinforce your knowledge about Logical Agents. In the first part of the assignment, you will explore and contribute to an implementation of the Wumpus world game. The Wumpus World's agent is an example of a knowledge-based agent that represents knowledge representation, reasoning, and planning. In the second part of the assignment, you will work on propositional logic. In particular, we will implement functions for printing truth tables for formulas with two variables. We will conclude this second part of the assignment by implementing a function to print to truth table for formulas with more than two variables. The last part of the assignment focuses on forward chaining. Forward chaining is used to track how things change over time and to draw conclusions from a priori clauses.

This assignment is designed to build your familiarity and comfort coding in Python while also helping you review key topics from the module. As you progress through the assignment, answers will get increasingly complex. It is important that you adopt a data scientist's mindset when completing this assignment. **Remember to run your code from each cell before submitting your assignment.** Running your code beforehand will notify you of errors and give you a chance to fix your errors before submitting. You should view your Vocareum submission as if you are delivering a final project to your manager or client.

Vocareum Tips - Do not add arguments or options to functions unless you are specifically asked to. This will cause an error in Vocareum. - Do not use a library unless you are explicitly asked to in the question. - You can download the Grading Report after submitting the assignment. This will include feedback and hints on incorrect questions.

1.1.1 Learning Objectives

- Implement a basic version of the Wumpus World game
- Understand Logical Agents and their Python implementation
- Define and visualize truth tables in Python
- Understand the applications of forward chaining
- Create lagged features and a time series for each feature

1.2 Index:

Logical Agents

- Section **??**
- Section **??**
- Section **??**
- Section **??**
- Section **??**
- Section **??**
- Section **??**

1.3 Logical Agents

1.4 Propositional Logic

In this section of the assignment, you will implement functions for printing truth tables for formulas with variables. You may use the following helper function `prints`, which prints a tab-delimited list of values.

```
In [3]: def prints(values):
            print("\t".join([str(value) for value in values]))
```

The above function can be used as follows:

```
In [4]: prints([True, False, True])

True        False       True
```

You may also use the following helper function `variables`, which returns a list of the argument names of a function:

```
In [5]: def variabl:es(f)
            return list(f.__code__.co_varnames)


          File "<ipython-input-5-cff42ecb2272>", line 1
        def variabl:es(f)
                   ^
    SyntaxError: invalid syntax
```

```
In [6]: import inspect
        def variables (f):
            return inspect.getargspec(f).args
```

The above function can be used as follows:

```
In [7]: def h(x,y,z): return (y or x) and (not(z) <= x)

        variables(h)
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:3: DeprecationWarning: inspect.getargspe
  This is separate from the ipykernel package so we can avoid doing imports until
```

```
Out[7]: ['x', 'y', 'z']
```

Section 1.2

1.4.1 Question 1:

10 points

Implement a function `truthtableXY(f)` that takes as its input a single function `f` (i.e. a Python function corresponding to a logical formula). You may assume `f` takes two boolean arguments `x` and `y`. The function should print a truth table for `f`.

If you define your function correctly, you should obtain the following output:

```
def f(x,y):
    return x and y

truthtableXY(f)

y      x      formula
True   True   True
True   False  False
False  True   False
False  False  False
```

```
In [8]: ### GRADED

        ### YOUR SOLUTION HERE
        def truthtableXY(f):
            prints(['y', 'x', 'formula'])
            for x in [True, False]:
                for y in [True, False]:
                    prints([x, y, f(x,y)])

        ###
        ### YOUR CODE HERE
        ###
```

```
In [9]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Section 1.2

1.4.2 Question 2:

20 points

Implement a recursive function `truthtable(f)` that takes as its first argument a single function `f` (i.e. a Python function corresponding to a formula) and as a second argument `values` set to `None` by default. The function `f` may take any non-zero quantity of arguments. The function should print a truth table for `f`.

Your `truthtable()` function should employ the recursive backtracking approach, and can be organized as follows:

- The function should have a second parameter `values` with a default value of [], which will be the list of values the function builds up and eventually passes to `f`.
- If the list `values` is empty, the function should print a row containing all the variable names (one column header per variable).
- If the list `values` is the same length as the list of variables of `f`, the function should print a row of values containing all the values in `values`, as well as the result of applying `f` to that list of values (use the * operator to apply `f` to the list of arguments).
- If the list `values` is shorter than the list of variables of `f`, the function should make recursive calls to truthtable(), with appropriate changes to the arguments of truthtable().

Example:

```
def h(x,y,z): return (y or x) and (not(z) <= x)

truthtable(h)
x       y       z       formula
True    True    True    False
True    True    False   False
True    False   True    False
True    False   False   False
False   True    True    True
False   True    False   False
False   False   True    False
False   False   False   False
```

```
In [11]: ### GRADED

         ### YOUR SOLUTION HERE

         def truthtable (f, values=None):
             if values is None:
                 values = []

             if values == []:
                 prints(variables(f))

             if len(values) == len(variables(f)):
                 result = f(*values)
                 prints(values + [result])
```

```
            else:
                truthtable(f, values + [True])
                truthtable(f, values + [False])
        ###
        ### YOUR CODE HERE
        ###
```

In [ ]:
```
###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

Section 1.2

1.4.3 Question 3:

10 points

Implement a function `rows()` that takes as its first argument a single function `f` (i.e. a Python function corresponding to a formula). The function should return the number of rows in the truth table for `f`.

Remember, the number of rows of a truth table is given by:

$$\text{rows} = 2^{\text{number of variables}}$$

In [18]:
```
### GRADED

### YOUR SOLUTION HERE:
def rows(f):
    return 2 ** len(variables(h))
###
### YOUR CODE HERE
###
```

In [19]:
```
def f(x, y, z): return (y or x) and (not(z) <= x)
```

In [20]:
```
assignment = [True, True, True]
f(*assignment)
```

Out[20]:
```
False
```

In [21]:
```
print(rows(f))
```

```
8

/usr/lib/python3.7/site-packages/ipykernel_launcher.py:3: DeprecationWarning: inspect.getargspe
  This is separate from the ipykernel package so we can avoid doing imports until
```

In [22]:
```
###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

1.5 Forward Chaining

Often, analysts are interested in how things change over time. In a typical cross-sectional sample, even if you measure some variable today and then again a year from now, you will probably be sampling different people each time. To get a better handle on how things change for the same people over time, you need to be able to track them and follow up with them a year from now, and in future waves. This is **longitudinal data**.

Longitudinal data is often used in economic and financial studies because it has several advantages over repeated cross-sectional data. For example, because longitudinal data measures how long events last for, it can be used to see if the same group of individuals remain unemployed during a recession, or whether different individuals are moving in and out of unemployment. This can help determine the factors that most affect unemployment.

Python's `scikit-learn` module has a `TimeSeriesSplit` function that can help run Forward Chaining. However, the function makes an assumption that the entire DataFrame is to be treated as one entity, whereas, it is possible that the entire DataFrame is actually composed of groups of data (row-wise) each of which group should be treated to Forward Chaining separately. The Financial Distress Prediction Data Set that we will be using here is exactly this type of a data set.

Here's a description of the columns in the dataset:

```
First column: Company represents sample companies

Second column: Time shows different time periods the data belongs to. Time series length varie

Third column: The target variable is denoted by "Financial Distress" if it is greater than -0.5

Fourth column to the last column: The features denoted by x1 to x83, are some financial and nor
```

We begin by importing the necessary libraries that we will be using for this section of the assignment.

```
In [2]: import numpy as np
        import pandas as pd
        import itertools
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import confusion_matrix, roc_auc_score

        import warnings
        warnings.filterwarnings('ignore')
```

Next, we load our data.

```
In [3]: # Load the Data
        df_finance = pd.read_csv('./data/Financial Distress.csv', index_col=False,
                          dtype={
                              'Company': np.uint16,
                              'Time': np.uint8,
                              'Financial Distress': np.double
                          })
```

As usual, we perform some exploratory data analysis to better understand our data.

```
In [4]: df_finance.head()
```

```
Out[4]:    Company  Time  Financial Distress      x1        x2       x3       x4  \
        0        1     1            0.010636  1.2810  0.022934  0.87454  1.21640
        1        1     2           -0.455970  1.2700  0.006454  0.82067  1.00490
        2        1     3           -0.325390  1.0529 -0.059379  0.92242  0.72926
        3        1     4           -0.566570  1.1131 -0.015229  0.85888  0.80974
        4        2     1            1.357300  1.0623  0.107020  0.81460  0.83593

                 x5        x6       x7  ...      x74    x75     x76     x77   x78  \
        0  0.060940  0.188270  0.52510  ...   85.437  27.07  26.102  16.000  16.0
        1 -0.014080  0.181040  0.62288  ...  107.090  31.31  30.194  17.000  16.0
        2  0.020476  0.044865  0.43292  ...  120.870  36.07  35.273  17.000  15.0
        3  0.076037  0.091033  0.67546  ...   54.806  39.80  38.377  17.167  16.0
        4  0.199960  0.047800  0.74200  ...   85.437  27.07  26.102  16.000  16.0

           x79  x80       x81  x82  x83
        0  0.2   22  0.060390   30   49
        1  0.4   22  0.010636   31   50
        2 -0.2   22 -0.455970   32   51
        3  5.6   22 -0.325390   33   52
        4  0.2   29  1.251000    7   27

        [5 rows x 86 columns]
```

Next, we look at the number of unique companies and the number of time periods for each company.

```
In [5]: # Look at the Data
        print("Number of unique companies:", df_finance.Company.unique().shape[0])  # 422 comp
        print("Number of time periods per company:")
        print(pd.crosstab(df_finance.Company, df_finance.Time.sum()))  # Some companies have <
```

```
Number of unique companies: 422
Number of time periods per company:
col_0    27644
Company
1            4
2           14
3            1
4           14
5           14
...        ...
418          2
419          3
420          3
421          6
422          8
```

```
[422 rows x 1 columns]
```

Finally, let's take a look at the data based on Groups per Company

```
In [6]: grouped_company = df_finance.groupby('Company')
        grouped_company.head()
        df_finance.x80.describe
```

```
Out[6]: <bound method NDFrame.describe of 0       22
        1       22
        2       22
        3       22
        4       29
                ..
        3667    37
        3668    37
        3669    37
        3670    37
        3671    37
        Name: x80, Length: 3672, dtype: int64>
```

1.5.1 Challenges with Straightforward Time Series Split:

As can be seen based on the above code output, although we have 1 common time variable (`Time`), we have multiple companies since we may have multiple rows belonging to the same Time value (For example, 1 row for Time 1 + Company 1, another row for Time 1 + Company 2 etc).

This prevents us from using `sklearn.model_selection.TimeSeriesSplit`, as the assumption of the function with each row representing a data point from a unique instance of time (and the rows are arranged as per increasing value of time).

We can still achieve our goal of implementing Forward Chaining by:

- Split the Data Set into multiple groups - 1 group per Company
- For each group, derive the indexes for Forward Chaining
- Combine the list of indexes per group into 1 final index list

Section 1.2

1.5.2 Question 4:

5 points

As mentioned in the Data Dictionary, one of the features is actually a categorical feature. We will therefore create Dummy columns.

Use the function `get_dummies()` to transform the column `x80` into Dummy columns. Make sure to set the parameter `prefix` equal to `dummy`, `columns` equal to `x80` and `drop_first` equal to `True`.

```
In [7]: ### GRADED
```

```
### YOUR SOLUTION HERE
dummy_cols = pd.get_dummies(
    df_finance[['x80']],
     prefix='dummy',
     columns=['x80'],
     drop_first=True)
###
### YOUR CODE HERE
###
```

```
In [8]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Section 1.2

1.5.3 Question 5:

10 points

Combine `dummy_cols` back with original data set.

Make sure you exclude the column `x80` in the resulting DataFrame. Assign the result to `df_finance_dummy`.

HINT: Use the function `concat()`.

```
In [9]: ### GRADED

        ### YOUR SOLUTION HERE
        x_cols= [col for col in df_finance.columns if all([col.startswith('x'), col!='x80'])]
        df_finance_dummy = pd.concat([
          df_finance[['Company', 'Time', 'Financial Distress']+x_cols].reset_index(drop=True),
            pd.DataFrame(data=dummy_cols)],
            axis=1)

        ###
        ### YOUR CODE HERE
        ###
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

1.5.4 Creating Lagged Features

With the above pre-processing step out of the way, we will now move on to the 2nd piece of Feature Engineering - creating lagged features (again, lagged features per group).

Below, we provide a helper function `lagged_features` to create lagged features.

```
In [23]: # Helper function to create lagged features
         def lagged_features(df_long, lag_features, window=2, lag_prefix='lag', lag_prefix_sep
```

```
    """
    Function calculates lagged features (only for columns mentioned in lag_features)
    based on time_feature column. The highest value of time_feature is retained as a
    and the lower values of time_feature are added as lagged_features

    :param df_long: Data frame (longitudinal) to create lagged features on
    :param lag_features: A list of columns to be lagged
    :param window: How many lags to perform (0 means no lagged feature will be produc
    :param lag_prefix: Prefix to name lagged columns.
    :param lag_prefix_sep: Separator to use while naming lagged columns
    :return: Data Frame with lagged features appended as columns
    """
    if not isinstance(lag_features, list):
        # So that while sub-setting DataFrame, we don't get a Series
        lag_features = [lag_features]

    if window <= 0:
        return df_long

    df_working = df_long[lag_features].copy()
    df_result = df_long.copy()
    for i in range(1, window+1):
        df_temp = df_working.shift(i)
        df_temp.columns = [lag_prefix + lag_prefix_sep + str(i) + lag_prefix_sep + x
                           for x in df_temp.columns]
        df_result = pd.concat([df_result.reset_index(drop=True),
                               df_temp.reset_index(drop=True)],
                              axis=1)

    return df_result
```

Now, we split the dataset into groups (based on companies) and create lagged features for each group.

```
In [20]: grouped_company = df_finance_dummy.groupby('Company')
         cols_to_lag = [col for col in df_finance_dummy.columns if col.startswith('x')]

         #create empty DataFrame
         df_cross = pd.DataFrame()

         for name, group in grouped_company:
             # For each group, calculate lagged features and rbind to df_cross
             print('-----------------------------------------------------')
             print('Working on group:', name, 'with shape', group.shape)
             df_cross = pd.concat([df_cross.reset_index(drop=True),
                                   lagged_features(group, cols_to_lag).reset_index(drop=True)]
                                  axis=0)
             print('Shape of df_cross', df_cross.shape)
```

```
# Remove rows with NAs
df_cross = df_cross.dropna()
df_cross.head()
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-20-26ea2f5217de> in <module>
----> 1 grouped_company = df_finance_dummy.groupby('Company')
      2 cols_to_lag = [col for col in df_finance_dummy.columns if col.startswith('x')]
      3
      4 #create empty DataFrame
      5 df_cross = pd.DataFrame()


NameError: name 'df_finance_dummy' is not defined
```

1.5.5 Time Series Splits per group

Next, we will write a helper function, `ts_sample` to create time series splits for forward chaining. The function will return a list of tuples. Each tuple will contain 2 values - the train index and the test index.

```
In [19]: # Create Time-Series sampling function to draw train-test splits
         def ts_sample(df_input, train_rows, test_rows):
             """
             Function to draw specified train_rows and test_rows in time-series rolling sampli
             :param df_input: Input DataFrame
             :param train_rows: Number of rows to use as training set
             :param test_rows: Number of rows to use as test set
             :return: List of tuples. Each tuple contains 2 lists of indexes corresponding to
             """
             if df_input.shape[0] <= train_rows:
                 return [(df_input.index, pd.Index([]))]

             i = 0
             train_lower, train_upper = 0, train_rows + test_rows*i
             test_lower, test_upper = train_upper, min(train_upper + test_rows, df_input.shape

             result_list = []
             while train_upper < df_input.shape[0]:
                 result_list += [(df_input.index[train_lower:train_upper],
                                  df_input.index[test_lower:test_upper])]
```

```
            # Update counter and calculate new indexes
            i += 1
            train_upper = train_rows + test_rows*i
            test_lower, test_upper = train_upper, min(train_upper + test_rows, df_input.s

        return result_list
```

1.5.6 Using ts_sample() per group

The next step is to use `ts_sample` per group of the data. This will give rise to 1 list of index tuples per group.

Moreover, because the number of time periods per group is not the same, the size of these lengths will also vary. Therefore, we will need a way to pad the shorter groups.

For each group, apply function `ts_sample`. Depending on size of group, the output size of ts_sample (which is a list of (train_index, test_index)) tuples will vary. However, we want the size of each of these lists to be equal.

To do that, we will augment the smaller lists by appending the last seen `train_index` and `test_index`. For example:

```
group 1 => [(Int64Index([1, 2, 3], dtype='int64'), (Int64Index[4, 5], dtype='int64')),
             (Int64Index([1, 2, 3, 4, 5], dtype='int64'), (Int64Index([6], dtype='int64'))]
 group 2 => [(Int64Index([10, 11, 12], dtype='int64'), (Int64Index[13, 14], dtype='int64')),
             (Int64Index([10, 11, 12, 13, 14), Int64Index([15, 16])),
             (Int64Index([10, 11, 12, 13, 14, 15, 16]), Int64Index([17, 18]))]
```

Above, group 2 has 3 folds whereas group 1 has 2. We will augment group 1 to also have 3 folds: `group 1 => [(Int64Index([1, 2, 3], dtype='int64'), (Int64Index[4, 5], dtype='int64')), (Int64Index([1, 2, 3, 4, 5], dtype='int64'), (Int64Index([6], dtype='int64')), (Int64Index([1, 2, 3, 4, 5, 6]), Int64Index([]))]`

```
In [18]: grouped_company_cross = df_cross.groupby('Company')
         acc = []
         max_size = 0
         for name, group in grouped_company_cross:
             # For each group, calculate ts_sample and also store largest ts_sample output siz
             group_res = ts_sample(group, 4, 4)
             acc += [group_res]
             # print('Working on name:' + str(name))
             # print(acc)

             if len(group_res) > max_size:
                 # Update the max_size that we have observed so far
                 max_size = len(group_res)

                 # All existing lists (apart from the one added latest)in acc need to be augme
                 # to match the new max_size by appending the last value in those list (combin
                 for idx, list_i in enumerate(acc):
```

```
            if len(list_i) < max_size:
                last_train, last_test = list_i[-1][0], list_i[-1][1]
                list_i[len(list_i):max_size] = [(last_train.union(last_test),
                                                 pd.Index([]))] * (max_size - len(lis

                acc[idx] = list_i

        elif len(group_res) < max_size:
            # Only the last appended list (group_res) needs to be augmented
            last_train, last_test = acc[-1][-1][0], acc[-1][-1][1]
            acc[-1] = acc[-1] + [(last_train.union(last_test), pd.Index([]))] * (max_size


    print(acc[0:2])
```

```
    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-18-7c010d3885b1> in <module>
----> 1 grouped_company_cross = df_cross.groupby('Company')
      2 acc = []
      3 max_size = 0
      4 for name, group in grouped_company_cross:
      5     # For each group, calculate ts_sample and also store largest ts_sample output s


    NameError: name 'df_cross' is not defined
```

acc now contains a list of lists, where each internal list contains tuples of `train_index, test_index`:

```
[[(group_1_train_index1, group_1_test_index1), (group_1_train_index2, group_1_test_index2)],
 [(group_2_train_index1, group_2_test_index1), (group_2_train_index2, group_2_test_index2)],
 [(group_3_train_index1, group_3_test_index1), (group_3_train_index2, group_3_test_index2)]]
```

Our goal is to drill-down by removing group-divisions:

```
[(train_index1, test_index1), (train_index2, test_index2)]
```

In [17]:
```
flat_acc = []
for idx, list_i in enumerate(acc):
    if len(flat_acc) == 0:
        flat_acc += list_i
        continue

    for inner_idx, tuple_i in enumerate(list_i):
```

```
            flat_acc[inner_idx] = (flat_acc[inner_idx][0].union(tuple_i[0]),
                                   flat_acc[inner_idx][1].union(tuple_i[1]))


    print(flat_acc[0:1])
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-17-b99bfcad831f> in <module>
      1 flat_acc = []
----> 2 for idx, list_i in enumerate(acc):
      3     if len(flat_acc) == 0:
      4         flat_acc += list_i
      5         continue


NameError: name 'acc' is not defined
```

1.5.7 Modeling

Now that we have our lagged features as well as the indexes ready for Forward Chaining, we can proceed with modeling.

However, one decision that we will need to take into account is whether we want to treat this as a classification problem or a regression problem. The `Financial Distress` column is real-valued, containing both positive and negative values. As per the Data Dictionary, we should consider the company financially distressed if the 'Financial Distress' column is <= -0.50. Accordingly, we will convert this problem into a classification problem by using that definition.

Section 1.2

1.5.8 Question 06:

10 points

Create a copy of the DataFrame `df_cross()`. Name this new DataFrame `df_model`.

Modify the column `Financial Distress` in `df_model` so that it contains zeros if the value is greater than -0.5 or 1 otherwise.

```
In [16]: ### GRADED

         ### YOUR SOLUTION HERE
         df_model = df_cross.copy()
         df_model['Financial Distress'] = [0 if x > -0.50 else 1 for x in df_model['Financial I
         ###
         ### YOUR CODE HERE
         ###
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-16-40086d1384e4> in <module>
      2
      3 ### YOUR SOLUTION HERE
----> 4 df_model = df_cross.copy()
      5
      6 df_model['Financial Distress'] = [0 if x > -0.50 else 1 for x in df_model['Financia


NameError: name 'df_cross' is not defined
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Section 1.2

1.5.9 Question 07:

10 points

Assign all the columns, except `Financial Distress` of `df_model` to the variable `dependent_cols`. Assign the column `Financial Distress` of `df_model` to the variable `independent_col`.

Use a for loop on `flat_acc` to generate the `X_train`, `X_test`, `y_train` and `y_test` sets.

HINT: Here is the code to generate X_train set. The others are similar

```
X_train = df_model.loc[tuple_i[0]][dependent_cols]
```

```
In [15]: ### GRADED

         ### YOUR SOLUTION HERE
         dependent_cols  = [col for col in df_model.columns if col != 'Financial Distress']
         independent_col = [col for col in df_model.columns if col == 'Financial Distress']
         independent_col = [col for col in df_model.columns if col == 'Financial Distress']
         X_train = df_model.loc[tuple_i[0]][dependent_cols]
         X_test = df_model.loc[tuple_i[0]][dependent_cols]
         y_train =
         y_test = df_model.loc[tuple_i[1]][independent_col]
         # ###
         ### YOUR CODE HERE
         ###
```

```
---------------------------------------------------------------------------
```

```
NameError                                 Traceback (most recent call last)

<ipython-input-15-b6b88d383a39> in <module>
      2
      3 ### YOUR SOLUTION HERE
----> 4 dependent_cols  = [col for col in df_model.columns if col != 'Financial Distress']
      5 independent_col = [col for col in df_model.columns if col == 'Financial Distress']
      6 # for x in flat_acc:


NameError: name 'df_model' is not defined
```

In [ ]:
```
###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

Finally, for each entry in `flat_acc`, perform train and test using logistic regression and print the metrics.

In [ ]:
```
for idx, tuple_i in enumerate(flat_acc):
# Fit logistic regression model to train data and test on test data
    lr_mod = LogisticRegression(C=0.01, penalty='l2')  # These should be determined by
    lr_mod.fit(X_train, y_train)

    y_pred_proba = lr_mod.predict_proba(X_test)
    y_pred = lr_mod.predict(X_test)

    # Print Confusion Matrix and ROC AUC score
    print('Confusion Matrix:')
    print(confusion_matrix(y_test, y_pred))

    print('ROC AUC score:')
    print(roc_auc_score(y_test['Financial Distress'].astype(int), y_pred_proba[:, 1]))
```