Credit Risk Analytics
The Python Companion

Harald Scheule and Daniel Rösch and Bart Baesens

1 Introduction

Welcome to one of the first references for credit risk analytics in Python which aims to take you on a journey to witness the building of credit risk models from start to finish. Accessing real credit data via the accompanying website www.creditriskanalytics.net, you will master a wide range of applications, including building your own PD, LGD and EAD models as well as mastering industry challenges such as reject inference, low default portfolio risk modeling, model validation and stress testing.

This book has been written as a companion to Baesens, B., Roesch, D. and Scheule, H., 2016. Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS. John Wiley & Sons. Following the outline and numbering of the companion book we have used the same underlying data sets and provide the equivalent code in Python. Note that as the exhibits in the original book do not relate to Python these are not included in this book and the numbering of exhibits is not necessarily sequential. Despite its heritage, the book can also be read and used in isolation. Python is preferred by individuals and smaller organizations which value open access, while SAS is preferred in large organizations which value software assurance and scaling. In the process of this project, we realized that Python, being a general-purpose programming language, occasionally requires custom code to get a similar result as in SAS and R. Although basic regression techniques are implemented in a similar fashion resulting in identical parameter estimates (i.e., through the `statsmodels` library), more advanced statistical analysis techniques do not come out of the box. However, the model outputs may be different in terms of the range of additional statistics reported and we do not aspire to provide identical model outputs but outputs that have the same rigor and effect on credit risk analytics.

1.1 Data sets

The book uses the same underlying data sets and structure as its SAS companion by Baesens et al. (2016). You can access these data sets and a large range of additional material on:

www.creditriskanalytics.net

We work with four data sets: HMEQ (chapter 5), Mortgage (chapters 4, 6, 7, 8, 9, 11, 12, 13, and 14), LGD (chapter 10) and Ratings (chapter 5).

1.2 HMEQ

The data set HMEQ includes control and default information about home equity loans. The variables of this data set are as follows:

1. `BAD`: 1 = applicant defaulted on loan or seriously delinquent; 0 = applicant paid loan
2. `LOAN`: Amount of the loan request
3. `MORTDUE`: Amount due on existing mortgage
4. `VALUE`: Value of current property
5. `REASON`: DebtCon = debt consolidation; HomeImp = home improvement
6. `JOB`: Occupational categories
7. `YOJ`: Years at present job
8. `DEROG`: Number of major derogatory reports
9. `DELINQ`: Number of delinquent credit lines
10. `CLAGE`: Age of oldest credit line in months
11. `NINQ`: Number of recent credit inquiries

12. CLNO: Number of credit lines
13. DEBTINC: Debt-to-income ratio

1.3 Mortgage

The dataset mortgage includes origination, performance, default and payoff information for US residential mortgage borrowers over 60 periods. The periods have been de-identified. The data is a randomized selection of mortgage loan-level data collected from the portfolios underlying US RMBS securitization portfolios and provided by International Financial Research.[1] Variables that are measured at origination are labeled _orig_time and variables that are measured in the respective period are labeled _time. The variables of this data set are as follows:

1. id: Borrower id
2. time: Time stamp of observation
3. orig_time: Time stamp for origination
4. first_time: Time stamp for first observation
5. mat_time: Time stamp for maturity
6. balance_time: Outstanding balance at observation time
7. LTV_time: Loan to value ratio at observation time, in %
8. interest_rate_time: Mortgage rate at observation time, in %
9. hpi_time: House price index at observation time, base year=100
10. gdp_time: GDP growth at observation time, in %
11. uer_time: Unemployment rate at observation time, in %
12. REtype_CO_orig_time: Real estate type condominium: 1, otherwise: 0
13. REtype_PU_orig_time: Real estate type planned urban developments: 1, otherwise: 0
14. REtype_SF_orig_time: Single family home: 1, otherwise: 0
15. investor_orig_time: Investor borrower: 1, otherwise: 0
16. balance_orig_time: Outstanding balance at origination time
17. FICO_orig_time: FICO score at origination time
18. LTV_orig_time: Loan to value ratio at origination time, in %
19. Interest_Rate_orig_time: Mortgage rate at origination time, in %
20. hpi_orig_time: House price index at observation time, base year = 100
21. default_time: Default observation at observation time
22. payoff_time: Payoff observation at observation time
23. status_time: Default (1), payoff (2) and non-default/non-payoff (0) observation at observation time

1.4 LGD

The dataset LGD reports the loss rates given default (LGDs) for defaulted loans and has been kindly provided by a European bank. It has been slightly modified and anonymized. The variables of this data set are as follows:

1. LTV: Loan to value ratio, in %
2. Recovery_rate: Recovery rate, in %
3. lgd_time: Loss rate given default (LGD), in %
4. y_logistic: Logistic transformation of the LGD
5. lnrr: Natural logarithm of the recovery rate
6. Y_probit: Probit transformation of the LGD

[1] www.internationalfinancialresearch.org

7. purpose1: Indicator variable for the purpose of the loan; 1: renting purpose, 0: other
8. event: Indicator variable for a default or cure event; 1: event, 0: no event

1.5 Ratings

The dataset ratings reports corporate ratings and control information. The variables of this data set are as follows:

1. COMMEQTA: Common equity to total assets
2. LLPLOANS: Loan loss provision to total loans
3. COSTTOINCOME: Operating costs to operating income
4. ROE: Return on equity
5. LIQASSTA: Liquid assets to total assets
6. SIZE: Natural logarithm of total assets

1.6 Housekeeping

Please feel free to contact us anytime if you would like to share feedback, errata or suggest extensions or topics that you would be interested in seeing covered in the next edition to us:

- Harald Scheule: harald@scheule.com
- Daniel Roesch: daniel.roesch@ur.de
- Bart Baesens: bart.baesens@kuleuven.be

Please also check the book website to stay in touch and current news: www.creditriskanalytics.net.

Our world is full of risk - we hope you stay safe and have fun!

Bart Baesens, Daniel Roesch and Harry Scheule

Month Year

2 Introduction to Python

In this chapter, we discuss some of the basic concepts of Python. Python is an interpreted high-level, general-purpose programming language first released in the 1990s. Python is growing in popularity in the data science community over the years. We use the Anaconda distribution with Python 3.6 in this book. Anaconda is an open source distribution of Python that is freely available at https://www.anaconda.com/. Anaconda not only installs the Python standard libraries but also a large set of other software and libraries useful for large-scale data processing, predictive analytics, and scientific computing. For example, the Anaconda distribution comes with the program Spyder, which is a Scientific PYthon Development EnviRonment. Users can use Spyder to follow along with the code examples in the book.

The advantage of using the Anaconda distribution is that it takes care of the interdependencies between the Python libraries when installing them. As said before, Anaconda comes with many useful libraries such as:

- **NumPy**: The fundamental Python library for scientific computing. Most notably, it provides a powerful n-dimensional array object as well as a lot of sophisticated functions. Many libraries are built on top of NumPy. We use the `1.14.0` version in this book.
- **SciPy**: Another open source Python library for scientific computing that extends NumPy's functionalities in many ways. We will mainly use its submodule for statistics. In this book, we use the `1.0.0` version.
- **Matplotlib**: A plotting library for producing high quality figures with Python. We use the `2.1.2` version in this book.
- **Pandas**: A powerful Python toolkit for data analysis. It provides two primary data structures: `Series` and `DataFrame`. A `Series` is a one-dimensional data construct typically used to represent a variable. A `DataFrame` is a two-dimensional data construct typically used to represent a collection of variables. R users will find it convenient to work with a `DataFrame` object, as it has a strong resemblance to R's `data.frame`. Note, however, that a pandas data frame provides a much richer functionality. These two data structures exploit many of the functionalities provided by the aforementioned libraries, as we will see throughout th book. We make use of the `0.22.0` version.
- **StatsModels**: A Python library for fitting statistical models. It allows using R-style formulas together with pandas data frames to fit statistical models. We use the `0.8.0` version in this book.

We will make heavy use of pandas data frames throughout the book, as it provides the best data representation of our structured data sets and comes with many bells and whistles useful for data analysis.

Contrary to SAS where columns are defined as variables and rows as observations, it is the user's responsibility how rows and columns are defined. For our purposes, a pandas `DataFrame` defines columns as variables and rows as observations. To avoid doubt, the terms "columns"/"variables" and "rows"/"observations" are used interchangeably throughout this book to better align this book with the main textbook "Credit Risk Analytics, Measurement Techniques, Applications and Examples in SAS". Furthermore, we use the terms "fitting" (predominantly used in Python) and "estimation" (predominantly used in SAS) interchangeably.

Software such as SAS easily provides the user with additional output by specifying the relevant options in a procedure, for example model diagnostic plots of a linear regression model. In contrast, analysis in Python frequently builds upon a series of steps in order to receive similar outputs.

Therefore, it is sometimes necessary to use the results produced in one step for another step and so on, until one obtains the desired information. As with many other programming languages, it is also possible to conduct arithmetics with Python which encompasses common operators such as +, -, /, or *. In order to use some of the results for further calculations, they can be stored in an object. This is done by using the assignment symbol =. Next to the two data structures of pandas, the following data types are the most frequently used to store data with regular Python: `list`, `tuple`, `set`, and `dict`.

- A `list`, declared with squared brackets `[ ]`, contains a sequence of values, each can be of a different type such as `int`, `float`, or `str`. More complex data types are also possible, e.g. to make a list of lists.
- A `tuple`, declared with parentheses `( )`, contains also a sequence of values, but, unlike `list`, a tuple is fixed in size and immutable once it is declared.
- A `set`, declared with curly brackets { }, is an unordered collection of unique elements and operates in many ways like a mathematical set.
- A `dict` is a Python dictionary that stores data in form of `key:value` pairs. This is a very powerful data type in Python. Unlike for example `list` that uses index numbers, a dictionary allows users to quickly access data with the associated `key`. It is straightforward to turn a `dict` into a pandas `DataFrame` object.

We start a new Python session and import all libraries needed for this chapter.

```
import pandas as pd
import statsmodels.formula.api as smf
```

Next, import the external CSV file of our mortgage data set into Python.

```
mortgage = pd.read_csv("mortgage.csv")
```

2.1 Data Manipulation

One convenient way to proceed is to convert an object to a pandas data frame. Using pandas data frames provides for an easier way to access and manipulate columns that generally represent variables. This step is not necessary at first sight as the function `pd.read_csv` will generate a data frame object by default. However, this function may be useful if the data set was originally in a different format, such as a Python dictionary.

```
mortgage = pd.DataFrame(mortgage)
```

One can always use the `type` function to check the type of a Python object.

```
type(mortgage)
```

```
pandas.core.frame.DataFrame
```

In order to get an overview of our mortgage data set, we can use the `info` method of the data frame object.

```
mortgage.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 622489 entries, 0 to 622488
Data columns (total 23 columns):
id                          622489 non-null int64
time                        622489 non-null int64
orig_time                   622489 non-null int64
first_time                  622489 non-null int64
mat_time                    622489 non-null int64
balance_time                622489 non-null float64
LTV_time                    622219 non-null float64
interest_rate_time          622489 non-null float64
hpi_time                    622489 non-null float64
gdp_time                    622489 non-null float64
uer_time                    622489 non-null float64
REtype_CO_orig_time         622489 non-null int64
REtype_PU_orig_time         622489 non-null int64
REtype_SF_orig_time         622489 non-null int64
investor_orig_time          622489 non-null int64
balance_orig_time           622489 non-null float64
FICO_orig_time              622489 non-null int64
LTV_orig_time               622489 non-null float64
Interest_Rate_orig_time     622489 non-null float64
hpi_orig_time               622489 non-null float64
default_time                622489 non-null int64
payoff_time                 622489 non-null int64
status_time                 622489 non-null int64
dtypes: float64(10), int64(13)
memory usage: 109.2 MB
```

The `info` method provides a summary of useful information such as *number of rows* and *columns* in the data frame object, *name* and *type of each column*, number of `non-null` values per column, and more. There are several ways to access a column of a pandas data frame. Suppose we are interested in accessing the `LTV_time` column, we could do so as follows.

```
# Access column by name
mortgage.LTV_time
mortgage["LTV_time"]
mortgage.loc[:, "LTV_time"]
```

We could also access the `LTV_time` column via its column index with the `iloc` method. Note that, unlike R, Python uses zero-based indexing. With the `LTV_time` column index being 6, we could access it as such.

```
# Access column by index
mortgage.iloc[:, 6]
```

Note that both the `iloc` and `loc` method allows us to simultaneously make a row selection. Above, we used `:` in order to select all rows.

2.2 Basic Data Manipulation with Pandas

This section shows some useful functions to manipulate data. First, subsamples may be generated from data sets by using the `query` function, for example, take a subsample where the FICO scores are greater than or equal to 500.

```
mortgage_temp = mortgage.query("FICO_orig_time >= 500")
```

There is an alternative method to retrieve the subset of a data set according to certain conditions other than using the `query` function. Here again, we can make use of the `loc` method. The command `data.loc[x, y]` refers to the element of a data frame in the row `x` and column `y`. Similarly, `data.loc[condition, :]` returns a subset with rows that satisfy the condition and all the columns since the `y` argument is set to `:`.

```
mortgage_temp2 = mortgage.loc[mortgage.default_time == 1, :]
```

To create a new simple variable on the basis of an existing one, we can make use of the `map` method in which we specify a `lambda` function, a powerful construct for creating anonymous functions.

```
mortgage["FICO_cat"] = mortgage.FICO_orig_time.map(
    lambda x: 1 if 500 < x <= 700 else 2 if x > 700 else 0
)
```

The `map` method runs over all elements of `FICO_orig_time`. In each step, the value of `FICO_cat` is determined by the value of `FICO_orig_time`, which is represented by the `x` in our `lambda` function. If `x` is in $(500, 700]$, the value 1 is returned; otherwise if `x` is larger than 700, the value 2 is returned; otherwise 0 is returned.

Delete variables from the data set. For example, we can delete the column `status_time` with the `drop` method.

```
mortgage_temp3 = mortgage_temp.drop("status_time", axis="columns")
```

Note that the `drop` method returns a new data frame object with `status_time` removed. The argument `axis="columns"` is necessary in order to specify that the column should be dropped.

Combine variables of interest by columns into a new data frame `select_cols`.

```
select_cols = mortgage[
    ["default_time", "FICO_orig_time", "LTV_orig_time", "gdp_time"]
]
```

Calculate summary statistics of these selected columns by using the `describe` method.

```
select_cols.describe()
```

```
        default_time  FICO_orig_time  LTV_orig_time       gdp_time
count  622489.000000   622489.000000  622489.000000  622489.000000
```

```
mean         0.024351      673.616922       78.975460        1.381032
std          0.154135       71.724558       10.127052        1.964645
min          0.000000      400.000000       50.100000       -4.146711
25%          0.000000      626.000000       75.000000        1.104163
50%          0.000000      678.000000       80.000000        1.850689
75%          0.000000      729.000000       80.000000        2.694111
max          1.000000      840.000000      218.500000        5.132464
```

Fit a linear regression model using R-style formulas with the `statsmodels` library, and execute the `summary` method to display the key statistics of the fitted linear regression.

```
#  OLS: ordinary least squares regression
mortgage_ols = smf.ols(
    formula="default_time ~ FICO_orig_time + LTV_orig_time + gdp_time",
    data=mortgage,
).fit()
mortgage_ols.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:           default_time   R-squared:
0.008
Model:                            OLS   Adj. R-squared:
0.008
Method:                 Least Squares   F-statistic:
1746.
Date:                Sun, 08 Apr 2018   Prob (F-statistic):
0.00
Time:                        14:46:02   Log-Likelihood:
2.8334e+05
No. Observations:              622489   AIC:
-5.667e+05
Df Residuals:                  622485   BIC:
-5.666e+05
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                     coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
Intercept          0.0796      0.003     30.707      0.000       0.074
0.085
FICO_orig_time    -0.0001   2.75e-06    -42.023      0.000      -0.000
-0.000
LTV_orig_time      0.0004   1.94e-05     19.590      0.000       0.000
0.000
gdp_time          -0.0055   9.91e-05    -55.019      0.000      -0.006
-0.005
==============================================================================
Omnibus:                   692979.256   Durbin-Watson:
2.012
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
36528558.470
Skew:                           6.094   Prob(JB):
0.00
```

```
Kurtosis:                       38.494   Cond. No.
9.08e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 9.08e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
"""
```

2.3 Self-defined functions in Python

The following is an example of how to create your own (self-defined) function to fit a linear regression model using the inputted data. The expressions `lhs` and `rhs` are input arguments, and `d` is the concatenation of them in the format of a `DataFrame`. The object `example_lm` is the fitted linear regression, where `lhs` is the dependent variable and rhs is the independent variable. The output prints the summary statistics of the fitted linear model.

```
def example(lhs, rhs):
    """Fit ordinary least squares regression model"""
    d = pd.concat([lhs, rhs], axis="columns")
    cols = d.columns.tolist()  # Get column names as list
    my_formula = "{} ~ {}".format(  # Build R-style formula
        cols[0],  # First variable is the left-hand side
        " + ".join(cols[1:]),  # Remaining variables are the right-hand side
    )
    example_ols = smf.ols(formula=my_formula, data=d).fit()
    return example_ols
```

The merit of writing code in functions is that they can be used multiple times requiring only a call of the function and its arguments, and thus often saving a substantial amount of double coding. The following example output is generated by the self-defined function using the control variable `FICO_orig_time`.

```
model = example(lhs=mortgage["default_time"], rhs=mortgage["FICO_orig_time"])
model.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:           default_time   R-squared:
0.003
Model:                            OLS   Adj. R-squared:
0.003
Method:                 Least Squares   F-statistic:
1838.
Date:                Sun, 08 Apr 2018   Prob (F-statistic):
0.00
Time:                        14:46:03   Log-Likelihood:
2.8165e+05
No. Observations:              622489   AIC:
```

```
-5.633e+05
Df Residuals:                   622487   BIC:
-5.633e+05
Df Model:                            1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
Intercept      0.1029      0.002     55.847      0.000       0.099
0.107
FICO_orig_time    -0.0001   2.72e-06    -42.872      0.000      -0.000
-0.000
==============================================================================
Omnibus:                   696653.038   Durbin-Watson:
2.021
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
37302878.186
Skew:                           6.144   Prob(JB):
0.00
Kurtosis:                      38.877   Cond. No.
6.40e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 6.4e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
"""
```

The following example output is the result of the self-defined function using the control variables `FICO_orig_time` and `LTV_orig_time`.

```
model = example(
    lhs=mortgage["default_time"],
    rhs=mortgage[["FICO_orig_time", "LTV_orig_time"]],
)
model.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:           default_time   R-squared:
0.004
Model:                            OLS   Adj. R-squared:
0.004
Method:                 Least Squares   F-statistic:
1100.
Date:                Sun, 08 Apr 2018   Prob (F-statistic):
0.00
Time:                        14:46:03   Log-Likelihood:
2.8183e+05
No. Observations:              622489   AIC:
-5.637e+05
Df Residuals:                  622486   BIC:
```

```
-5.636e+05
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
Intercept      0.0683      0.003     26.393      0.000       0.063
0.073
FICO_orig_time    -0.0001   2.75e-06    -39.503      0.000      -0.000
-0.000
LTV_orig_time      0.0004   1.95e-05     18.981      0.000       0.000
0.000
==============================================================================
Omnibus:                   696248.054   Durbin-Watson:
2.022
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
37214546.620
Skew:                           6.139   Prob(JB):
0.00
Kurtosis:                      38.834   Cond. No.
9.06e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 9.06e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
"""
```

The last example output uses the self-defined function by using the control variables `FICO_orig_time`, `LTV_orig_time`, and `gdp_time`.

```
model = example(
    lhs=mortgage["default_time"],
    rhs=mortgage[["FICO_orig_time", "LTV_orig_time", "gdp_time"]],
)
model.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:           default_time   R-squared:
0.008
Model:                           OLS   Adj. R-squared:
0.008
Method:                Least Squares   F-statistic:
1746.
Date:               Sun, 08 Apr 2018   Prob (F-statistic):
0.00
Time:                       14:46:04   Log-Likelihood:
2.8334e+05
No. Observations:             622489   AIC:
-5.667e+05
Df Residuals:                 622485   BIC:
```

```
-5.666e+05
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
Intercept      0.0796      0.003     30.707      0.000       0.074
0.085
FICO_orig_time    -0.0001   2.75e-06    -42.023      0.000      -0.000
-0.000
LTV_orig_time      0.0004   1.94e-05     19.590      0.000       0.000
0.000
gdp_time          -0.0055   9.91e-05    -55.019      0.000      -0.006
-0.005
==============================================================================
Omnibus:                   692979.256   Durbin-Watson:
2.012
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
36528558.470
Skew:                           6.094   Prob(JB):
0.00
Kurtosis:                      38.494   Cond. No.
9.08e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 9.08e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
"""
```

2.4 Further Readings

The Python programming language is well-documented. For readers who want to become more familiar with Python, we recommend the official website (https://www.python.org/about/gettingstarted/) as a starting point. There, readers will find plenty of material and references to books, websites, and tutorials. For a general introduction to statistics with Python see, for example, Haslwanter (2016).

3 Exploratory Data Analysis

This chapter provides the Python codes for exploratory data analysis. Start a new Python session and import all libraries needed for this chapter, as well as read in the mortgage data set with pandas.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

# Set floating point output precision
pd.set_option("display.precision", 3)

mortgage = pd.read_csv("mortgage.csv")
```

3.1 One-Dimensional Analysis

3.1.1 Observed Frequencies and Empirical Distributions

We first compute observed frequencies for the defaults and empirical distributions for the FICO score and LTV. For the frequencies, we define the `frequency_table` function that takes as input a pandas series.

```
def frequency_table(x):
    """Compute frequency table for a pandas series"""
    frequency = x.value_counts()
    frequency.index.name = x.name
    frequency.name = "Freq."

    cumulative_frequency = frequency.cumsum()
    cumulative_frequency.name = "Cumulative_Freq."

    percent = x.value_counts(normalize=True) * 100
    percent.name = "Percent"

    cumulative_percent = percent.cumsum()
    cumulative_percent.name = "Cumulative_Percent"

    # Merge the results into a DataFrame
    frequency_table = pd.concat(
        [frequency, percent, cumulative_frequency, cumulative_percent],
        axis=1,
    )
    return frequency_table
```

Now, we can calculate the frequencies of interest for `default_time` using our self-defined function.

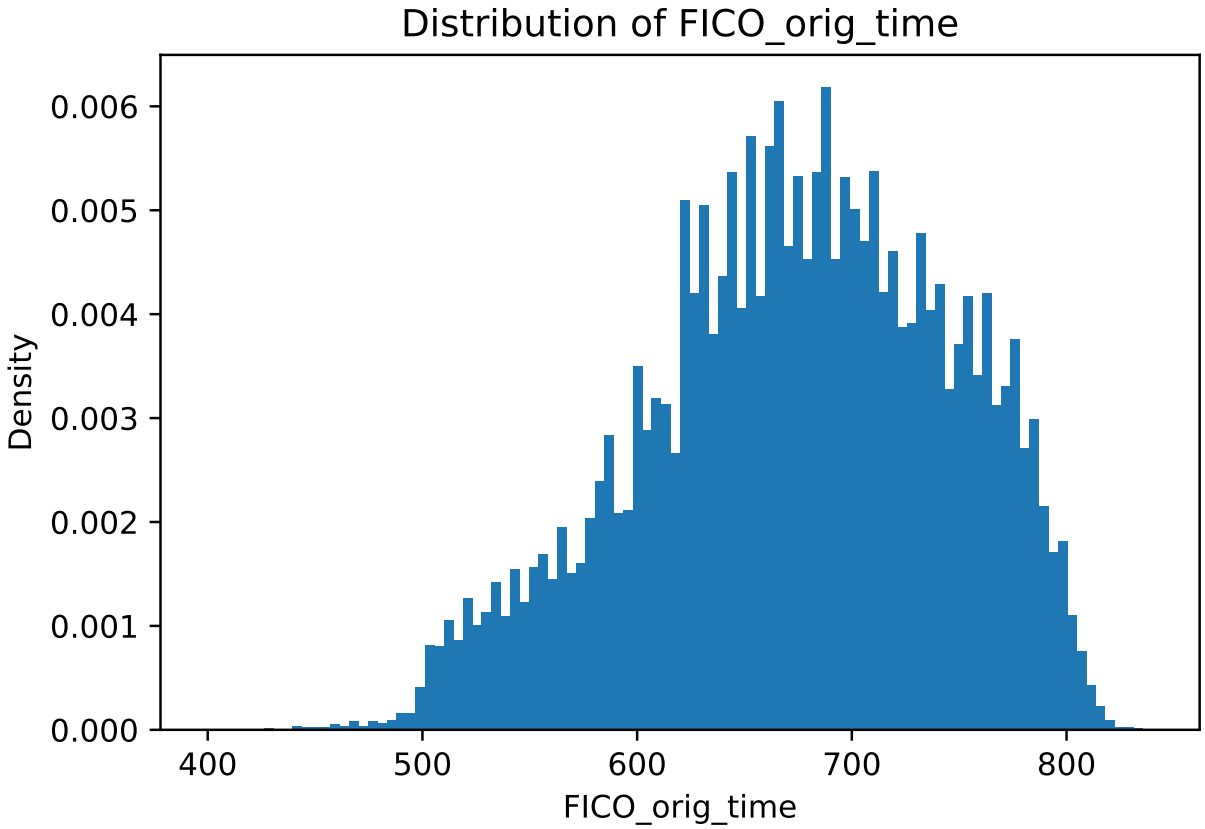
```
frequency_table(mortgage["default_time"])
```

```
              Freq.  Percent  Cumulative_Freq.  Cumulative_Percent
default_time
```

```
|0              607331   97.565            607331             97.565
|1               15158    2.435            622489            100.000
```

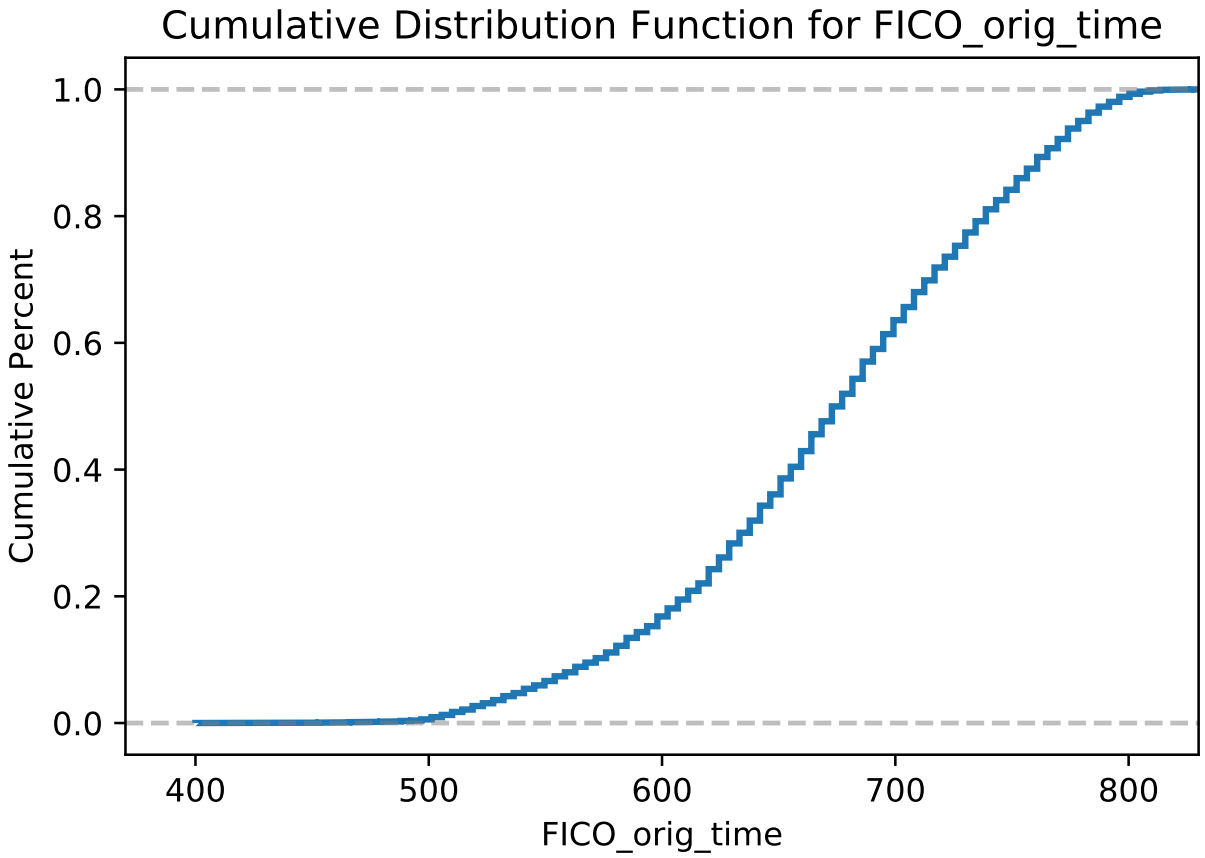
Plot the histogram for `FICO_orig_time` by using the `hist` method.

```
mortgage.hist(column="FICO_orig_time", bins=100, grid=False, normed=True)
plt.title("Distribution of FICO_orig_time")
plt.xlabel("FICO_orig_time")
plt.ylabel("Density")
plt.show()
```



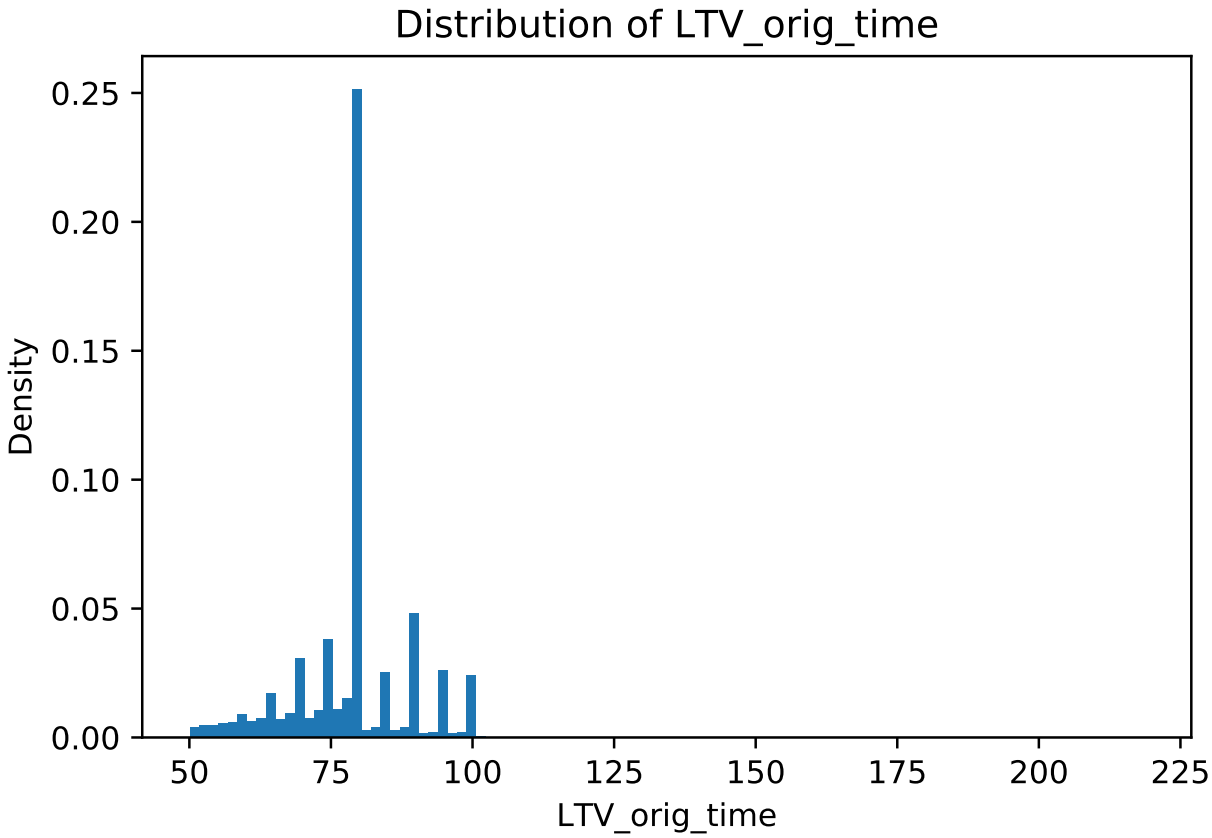
Plot the cumulative distribution function for `FICO_orig_time`.

```
mortgage.hist(
    column="FICO_orig_time",
    cumulative=True,
    histtype='step',
    bins=100,
    grid=False,
    normed=True,
    linewidth=2,
)
plt.title("Cumulative Distribution Function for FICO_orig_time")
plt.xlabel("FICO_orig_time")
plt.ylabel("Cumulative Percent")
plt.xlim([370, 830])
plt.ylim([-.05, 1.05])
plt.axhline(y=0, color='gray', linestyle='--', alpha=.5)
plt.axhline(y=1, color='gray', linestyle='--', alpha=.5)
plt.show()
```

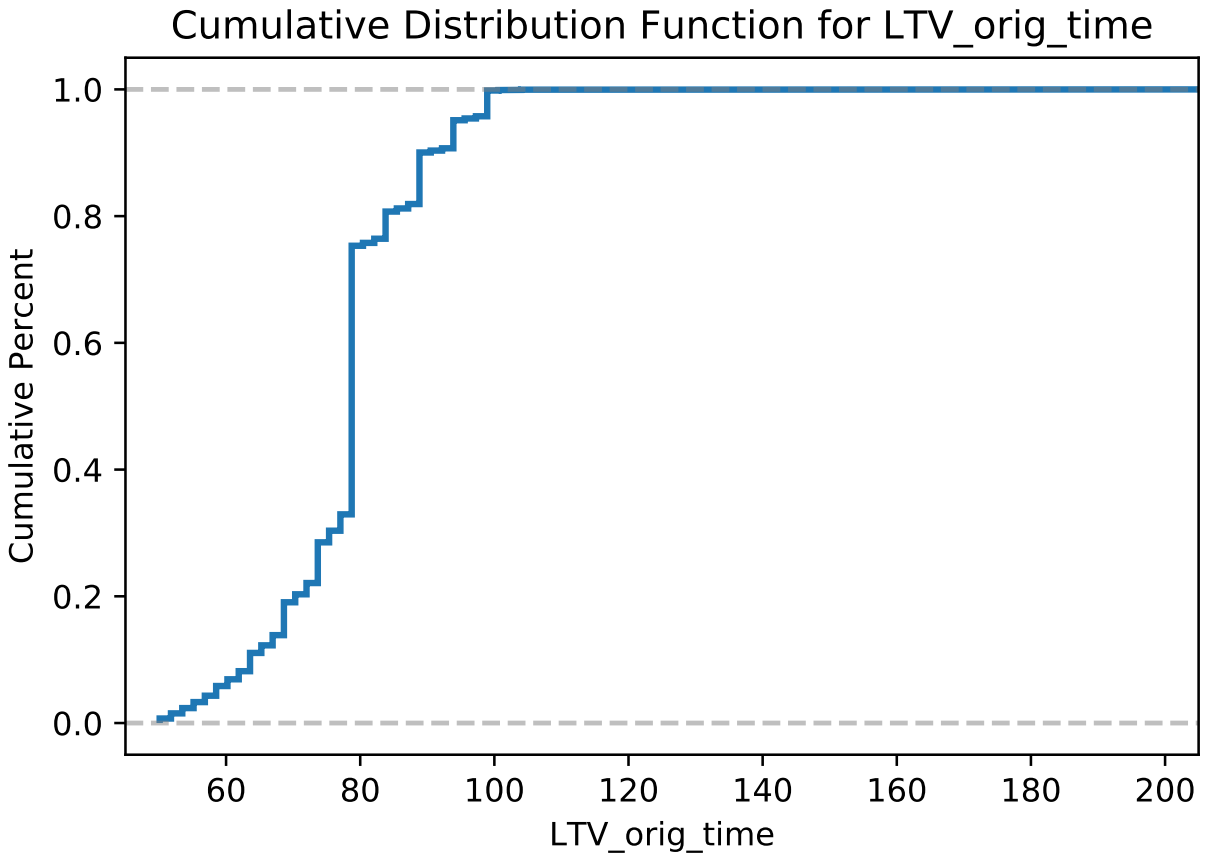


As before, plot the histogram and cumulative distribution function for `LTV_orig_time`.

```
mortgage.hist(column="LTV_orig_time", bins=100, grid=False, normed=True)
plt.title("Distribution of LTV_orig_time")
plt.xlabel("LTV_orig_time")
plt.ylabel("Density")
plt.show()
```




```
mortgage.hist(
    column="LTV_orig_time",
    cumulative=True,
    histtype='step',
    bins=100,
    grid=False,
    normed=True,
    linewidth=2,
)
plt.title("Cumulative Distribution Function for LTV_orig_time")
plt.xlabel("LTV_orig_time")
plt.ylabel("Cumulative Percent")
plt.xlim([45, 205])
plt.ylim([-.05, 1.05])
plt.axhline(y=0, color='gray', linestyle='--', alpha=.5)
plt.axhline(y=1, color='gray', linestyle='--', alpha=.5)
plt.show()
```



3.1.2 Location Measures

Subsequently, we compute location measures (mean, median and mode) for the three variables default, FICO and LTV as well as some percentiles (quantiles). In addition, we create Q-Q plots.

Create a function, which returns a pandas series, that calculates the required descriptive statistics, namely count, mean, median, mode, 1% quantile and 99% quantile.

```
def proc_means(x):
    """Compute location measures for a pandas series"""
    loc_measures = pd.Series(
        data=[
            len(x),
            x.mean(),
            x.median(),
            x.mode().values[0],  # Access the first value
            x.quantile(.01),
            x.quantile(.99),
        ],
        index=[
            "N",
            "Mean",
            "Median",
            "Mode",
            "1st Pctl",
            "99st Pctl",
        ],
    )
    return loc_measures
```

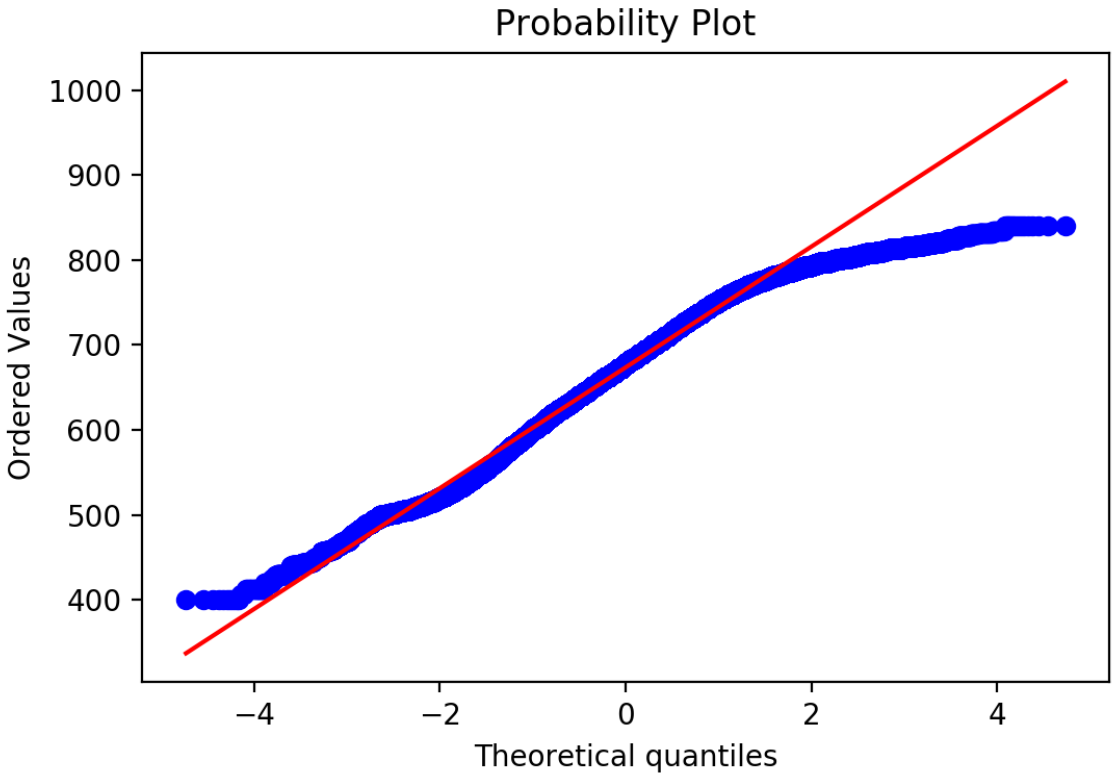
Now, we can use the `apply` method in order to execute the `proc_means` function on one or more columns.

```
# Use \ to continue on the next line
mortgage[["default_time", "FICO_orig_time", "LTV_orig_time"]] \
    .apply(proc_means)
```

```
           default_time  FICO_orig_time  LTV_orig_time
N              622489.000      622489.000     622489.000
Mean                0.024         673.617         78.975
Median              0.000         678.000         80.000
Mode                0.000         660.000         80.000
1st Pctl            0.000         506.000         52.200
99st Pctl           1.000         801.000        100.000
```

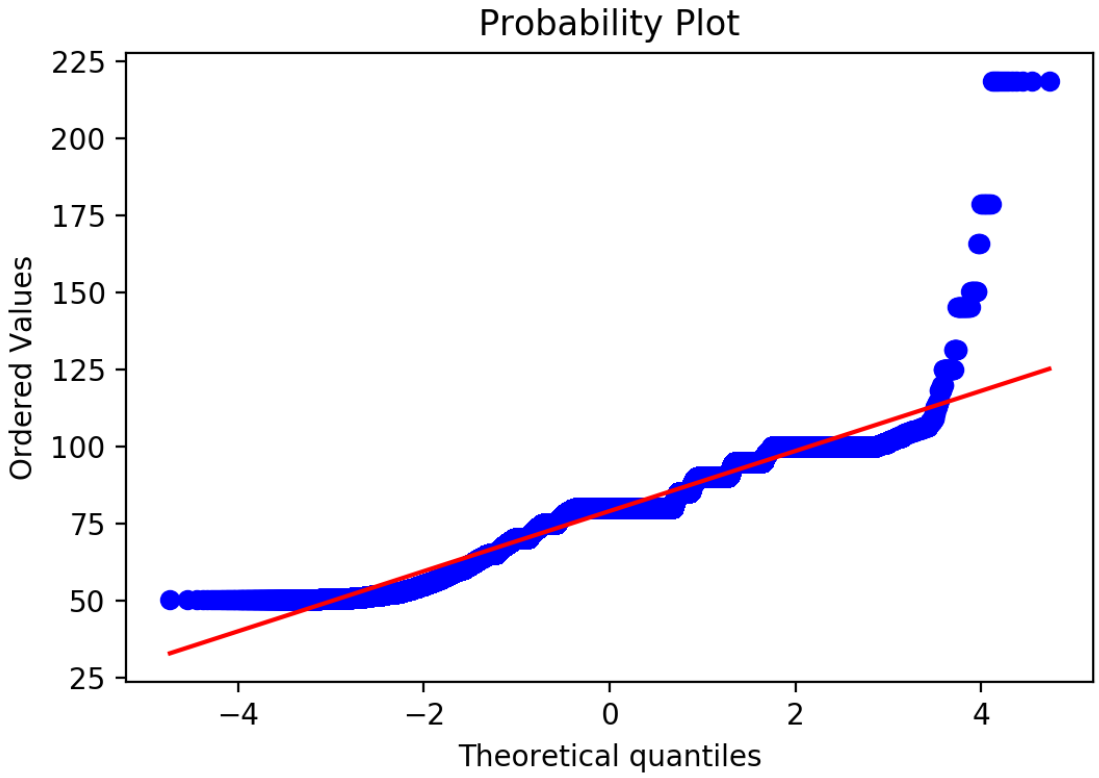
Generate a Q-Q plot using SciPy's `probplot` function for `FICO_orig_time`.

```
stats.probplot(mortgage["FICO_orig_time"], dist="norm", plot=plt)
plt.show()
```



Generate a Q-Q plot using SciPy's `probplot` function for `LTV_orig_time`.

```
stats.probplot(mortgage["LTV_orig_time"], dist="norm", plot=plt)
plt.show()
```



Subsequently, dispersion measures, skewness and kurtosis are computed. As before, create a function to calculate the required descriptive statistics.

```
def proc_means_ext(x):
    """Compute dispersion measures for a pandas series"""
    disp_measures = pd.Series(
        data=[
            len(x),
            x.min(),
            x.max(),
            x.max() - x.min(),
            x.quantile(.75) - x.quantile(.25),
            x.var(),
            x.std(),
            (x.std() / x.mean()) * 100,
        ],
        index=[
            "N",
            "Minimum",
            "Maximum",
            "Range",
            "Quartile Range",
            "Variance",
            "Std. Dev.",
            "Coeff of Variation",
        ],
    )
    return disp_measures
```

Use the `apply` method again in order to execute the `proc_means_ext` function on one or more columns.

```
mortgage[["default_time", "FICO_orig_time", "LTV_orig_time"]] \
    .apply(proc_means_ext)
```

```
                    default_time  FICO_orig_time  LTV_orig_time
N                     622489.000      622489.000     622489.000
Minimum                    0.000         400.000         50.100
Maximum                    1.000         840.000        218.500
Range                      1.000         440.000        168.400
Quartile Range             0.000         103.000          5.000
Variance                   0.024        5144.412        102.557
Std. Dev.                  0.154          71.725         10.127
Coeff of Variation       632.983          10.648         12.823
```

Similarly, define a function to calculate the required descriptive statistics. Use the `skew` and `kurtosis` methods of the pandas data frame object to calculate skewness and Pearson's measure of kurtosis. Subtracting three from Pearson's measure of kurtosis yields the excess kurtosis.

```
def proc_means_skew_kurt(x):
    """Compute skewness and kurtosis for a pandas series"""
    skew_kurt = pd.Series(
        data=[
            len(x),
            x.skew(),
            x.kurtosis() - 3, # Excess kurtosis
        ],
        index=[
            "N",
            "Skewness",
            "Kurtosis",
        ],
    )
    return skew_kurt
```

Again, use the `apply` method in order to execute the `proc_means_skew_kurt` function on one or more columns.

```
mortgage[["default_time", "FICO_orig_time", "LTV_orig_time"]] \
    .apply(proc_means_skew_kurt)
```

```
          default_time  FICO_orig_time  LTV_orig_time
N           622489.000      622489.000     622489.000
Skewness         6.172          -0.321         -0.196
Kurtosis        33.092          -3.468         -1.564
```

3.2 Two-Dimensional Data Analysis

3.2.1 Joint Empirical Distributions

Having explored the empirical data on a one-dimensional basis for each variable, we may also be interested in interrelations between variables. In a first step, we therefore create two-dimensional (or two-way) frequency tables, e.g., for default and FICO classes.

Create a new variable `FICO_orig_time_factor` with the `cut` method. Based on the value of `FICO_orig_time`, assign values 0 to 4 to `FICO_orig_time_factor`.

```
mortgage["FICO_orig_time_factor"] = pd.cut(
    x=mortgage["FICO_orig_time"],
    bins=[
        0,
        mortgage["FICO_orig_time"].quantile(.2),
        mortgage["FICO_orig_time"].quantile(.4),
        mortgage["FICO_orig_time"].quantile(.6),
        mortgage["FICO_orig_time"].quantile(.8),
        float('Inf'),
    ],
    labels=[0, 1, 2, 3, 4],
    right=False,
)
```

Generate the two-dimensional contingency table.

```
contingency_table = pd.crosstab(
    index=mortgage["default_time"],
    columns=mortgage["FICO_orig_time_factor"],
    margins=True,  # Include row and column totals
)
contingency_table
```

FICO_orig_time_factor	0	1	2	3	4	All
default_time						
0	119046	118890	124047	121876	123472	607331
1	4328	3790	3269	2385	1386	15158
All	123374	122680	127316	124261	124858	622489

Divide the table by the grand total to get the total proportion of counts in each cell.

```
contingency_table / contingency_table.loc["All", "All"]
```

FICO_orig_time_factor	0	1	2	3	4	All
default_time						
0	0.191	0.191	0.199	0.196	0.198	0.976
1	0.007	0.006	0.005	0.004	0.002	0.024
All	0.198	0.197	0.205	0.200	0.201	1.000

To get the proportion of counts along each column.

```
contingency_table.div(contingency_table.loc["All", :], axis="columns")
```

FICO_orig_time_factor	0	1	2	3	4	All
default_time						
0	0.965	0.969	0.974	0.981	0.989	0.976
1	0.035	0.031	0.026	0.019	0.011	0.024
All	1.000	1.000	1.000	1.000	1.000	1.000

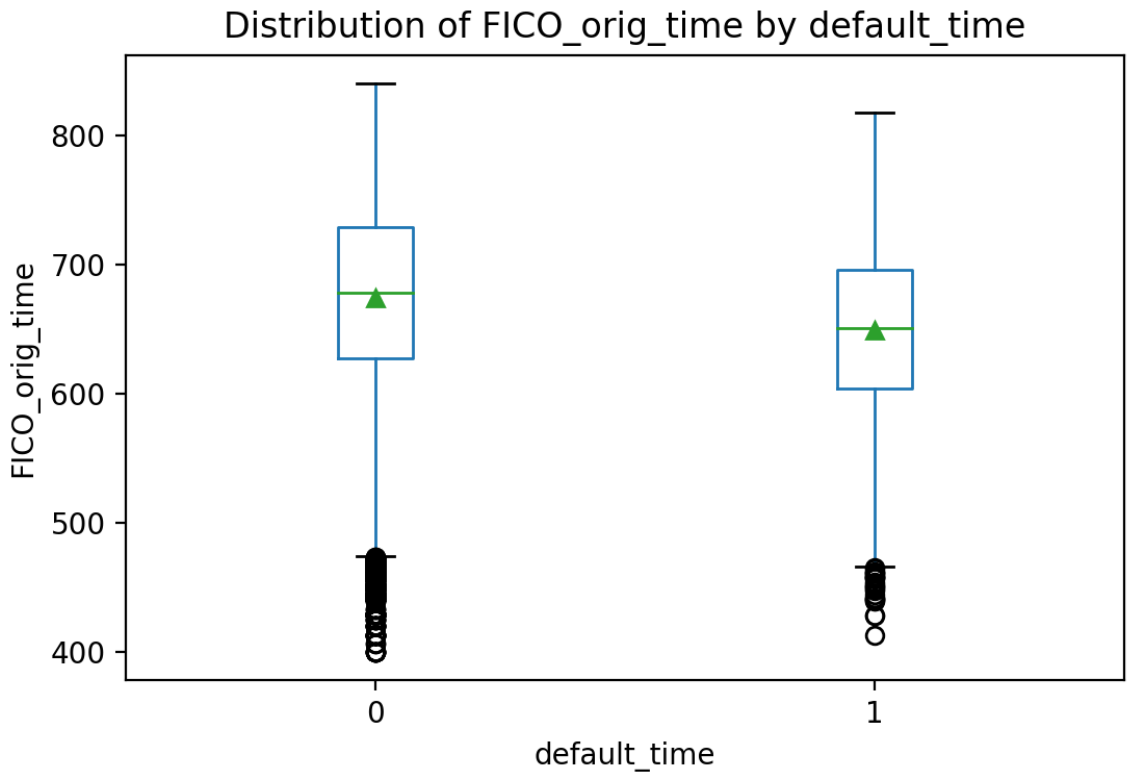
To get the proportion of counts along each row.

```
contingency_table.div(contingency_table.loc[:, "All"], axis="rows")
```

```
FICO_orig_time_factor      0      1      2      3      4  All
default_time
0                      0.196  0.196  0.204  0.201  0.203  1.0
1                      0.286  0.250  0.216  0.157  0.091  1.0
All                    0.198  0.197  0.205  0.200  0.201  1.0
```

Another way of inferring the relation between both variables (without grouping FICO first) is to look at a box plot. Generate the boxplot of `FICO_orig_time` for each value of `default_time` by using the `boxplot` method. Set the `showmeans` argument to true in order to add the means of `FICO_orig_time` by `default_time` to the boxplot.

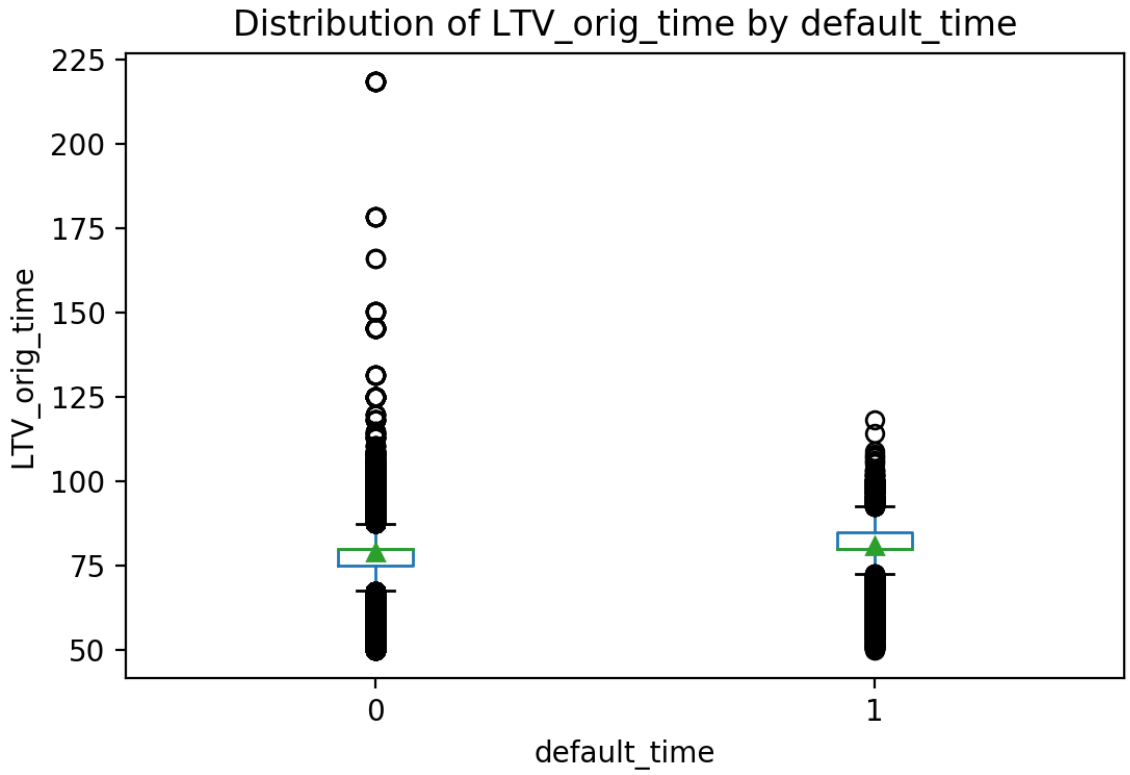
```
mortgage.boxplot(
    column="FICO_orig_time",
    by="default_time",
    grid=False,
    showmeans=True,
)
plt.title("Distribution of FICO_orig_time by default_time")
plt.suptitle("")
plt.xlabel("default_time")
plt.ylabel("FICO_orig_time")
plt.show()
```



As before, generate a boxplot for `LTV_orig_time`.

```
mortgage.boxplot(
    column="LTV_orig_time",
```

```
    by="default_time",
    grid=False,
    showmeans=True,
)
plt.title("Distribution of LTV_orig_time by default_time")
plt.suptitle("")
plt.xlabel("default_time")
plt.ylabel("LTV_orig_time")
plt.show()
```



3.2.2 Correlation Measures

We now compute measures for association and correlation, namely χ^2, ϕ, the contingency coefficient and Cramer's V. Create a simple cross table of `default_time` and `FICO_orig_time_factor` by using the `crosstab` function.

```
tab = pd.crosstab(
    index=mortgage["default_time"],
    columns=mortgage["FICO_orig_time_factor"],
)
```

```
n = tab.sum().sum()  # Sum over columns and rows
chi2, p_value, dof, _ = stats.chi2_contingency(tab)
phi = np.sqrt(chi2 / n)
cont_coeff = np.sqrt(phi ** 2 / (1 + phi ** 2))
cramersV = np.sqrt(phi ** 2 / min(dim - 1 for dim in tab.shape))
s = '<' if p_value < 0.0001 else '='
p = 0.0001 if p_value < 0.0001 else p_value
```

```
phi_coeff = phi if all(dim == 2 for dim in tab.shape) else np.nan
print(
    "X^2 = {:.1f}, df = {:d}, p-value {} {:.4f}".format(chi2, dof, s, p)
)
print("Phi-Coefficient = {:.3f}".format(phi_coeff))
print("Contingency Coeff. = {:.3f}".format(cont_coeff))
print("Cramer's V = {:.3f}".format(cramersV))
```

```
X^2 = 1890.1, df = 4, p-value < 0.0001
Phi-Coefficient = nan
Contingency Coeff. = 0.055
Cramer's V = 0.055
```

Create a sample by setting a seed value with the sample size of 1% without replacement from `FICO_orig_time` and `LTV_orig_time`. Seed values are helpful in order to identify the random draw and being able to repeat the experiment using the exact same values. Note that we sample data points independently for each variable. Combine the two samples into a data frame, where we need to make use of the `reset_index` method with the `drop` argument set to `True`. The index reseting is required, as pandas otherwise concatenates observations with the same index value. And since we sampled randomly, this will create many missing values in `smpl` if the index is not reset.

```
smpl_FICO = mortgage.FICO_orig_time \
    .sample(frac=.01, replace=False, random_state=12345)
smpl_LTV = mortgage.LTV_orig_time \
    .sample(frac=.01, replace=False, random_state=678)
smpl = pd.concat(
    [smpl_FICO.reset_index(drop=True), smpl_LTV.reset_index(drop=True)],
    axis=1,
)
```

We can also compute the Pearson and Spearman correlation coefficients as well as Kendall's Tau and produce a scatter plot. Compute Pearson's correlation and perform a test using `pearsonr`.

```
corr_coeff, p_value = stats.pearsonr(
    smpl["FICO_orig_time"],
    smpl["LTV_orig_time"],
)
s = '<' if p_value < 0.0001 else '='
p = 0.0001 if p_value < 0.0001 else p_value
print(
    "Pearson's correlation coeff. = {:.4f}, p-value {} {:.4f}".format(
        corr_coeff, s, p,
    )
)
```

```
Pearson's correlation coeff. = -0.0210, p-value = 0.0974
```

Compute Spearman's correlation and perform a test.

```
corr_coeff, p_value = stats.spearmanr(
    smpl["FICO_orig_time"],
    smpl["LTV_orig_time"],
```

```
)
s = '<' if p_value < 0.0001 else '='
p = 0.0001 if p_value < 0.0001 else p_value
print(
    "Spearman's correlation coeff. = {:.4f}, p-value {} {:.4f}".format(
        corr_coeff, s, p,
    )
)
```

```
Spearman's correlation coeff. = -0.0223, p-value = 0.0785
```

Compute Kendall's tau and perform a test.

```
corr_coeff, p_value = stats.kendalltau(
    smpl["FICO_orig_time"],
    smpl["LTV_orig_time"],
)
s = '<' if p_value < 0.0001 else '='
p = 0.0001 if p_value < 0.0001 else p_value
print(
    "Kendall's correlation coeff. = {:.4f}, p-value {} {:.4f}".format(
        corr_coeff, s, p,
    )
)
```

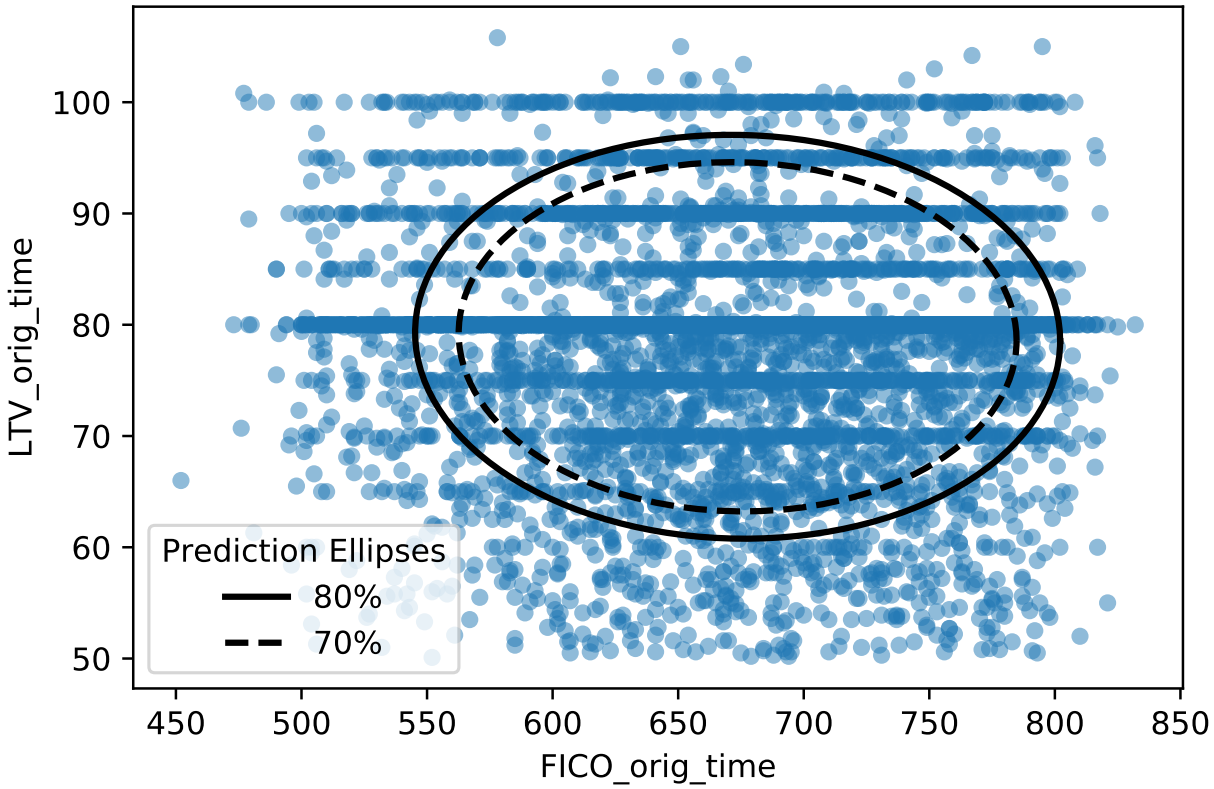
```
Kendall's correlation coeff. = -0.0159, p-value = 0.0766
```

Generate a scatter plot and add prediction ellipses. For the latter, define the `ellipse_coordinates` function which returns a NumPy array with the ellipse coordinates. Finally, add the legend.

```
def ellipse_coordinates(x, y, alpha_level=0.05, n_points=250):
    """Compute coordinates of ellipse for given level of alpha"""
    mu = np.array([np.mean(x), np.mean(y)])
    cov = np.cov(x, y, ddof=1)
    eigvals, eigvecs = np.linalg.eig(cov)
    e1 = -np.dot(eigvecs, np.diag(np.sqrt(eigvals)))
    r1 = np.sqrt(stats.chi2.ppf(1 - alpha_level, 2))
    theta = np.linspace(0, 2 * np.pi, num=n_points)
    v1 = np.c_[[r1 * np.cos(theta), r1 * np.sin(theta)]]
    pts = np.transpose(mu.reshape(2, 1) - np.dot(e1, v1))
    return pts
```

```
elli80 = ellipse_coordinates(
    smpl["FICO_orig_time"],
    smpl["LTV_orig_time"],
    .2,
)
elli70 = ellipse_coordinates(
    smpl["FICO_orig_time"],
    smpl["LTV_orig_time"],
    .3,
)
smpl.plot(kind="scatter", x="FICO_orig_time", y="LTV_orig_time", alpha=.5)
plt.plot(elli80[:, 0], elli80[:, 1], 'k-', lw=2, label="80%")
```

```
plt.plot(elli70[:, 0], elli70[:, 1], 'k--', lw=2, label="70%")
plt.legend(loc="best", title="Prediction Ellipses")
plt.show()
```



3.3 Highlights of Inductive Statistics

3.3.1 Confidence Intervals

Once the parameters have been estimated, they will not match the true (i.e., unknown data generating) parameters in the population exactly, but instead randomly differ from them. Hence, we may want to compute confidence intervals and conduct hypothesis tests. Generate a self-defined function to compute confidence intervals, assuming a normal distribution

```
def proc_univariate(x):
    """
    Compute some univariate statistics and
    a confidence interval for a pandas series
    """
    n = len(x)
    e = stats.norm.ppf(.99) * x.std() / np.sqrt(n)
    uni_stats = pd.Series(
        data=[
            x.mean(),
            x.std(),
            x.var(),
            x.mean() - e,
            x.mean() + e,
        ],
        index=[
```

```
            "Mean",
            "Std. Deviation",
            "Variance",
            "Lower Confidence Limit",
            "Upper Confidence Limit",
        ],
    )
    return uni_stats
```

```
mortgage[["LTV_orig_time"]].apply(proc_univariate)
```

```
                         LTV_orig_time
Mean                            78.975
Std. Deviation                  10.127
Variance                       102.557
Lower Confidence Limit          78.946
Upper Confidence Limit          79.005
```

3.3.2 Hypothesis Testing

Perform a t-distribution test by using the `ttest_1samp` function.

```
# Performs a two-sided test for one sample
t_statistic, p_value = stats.ttest_1samp(
    mortgage["LTV_orig_time"],
    popmean=60,
)
s = '<' if p_value < 0.0001 else '='
p = 0.0001 if p_value < 0.0001 else p_value
print("t = {:.1f}, p-value {} {:.4f}".format(t_statistic, s, p))
```

```
t = 1478.3, p-value < 0.0001
```

4 Data Preprocessing

In this chapter, we illustrate how to preprocess data in Python. Data preprocessing is often a time-consuming but important exercise as subsequent models and model outputs are based on this work.

We start a new Python session and import all libraries needed for this chapter, as well as read in the hmeq data set.

```
import numpy as np
import pandas as pd
import scipy.stats as stats

hmeq = pd.read_csv("hmeq.csv")
```

4.1 Sampling

With a pandas data frame object, the `sample` method can be used to create a sample of size 1,000 without replacement. This returns a new pandas data frame object in which the rows are chosen randomly. We name this new data frame `my_sample`.

```
my_sample = hmeq.sample(n=1000, replace=False, random_state=12345)
```

By specifying a seed it is possible to repeat the experiment with the exact same values. There is no replacement when the sample is drawn, hence no rows can be selected twice.

Next, we are interested in computing the frequency table of the population stratified by the variable `BAD`. For that, we define the `frequency_table` function.

```
def frequency_table(x):
    """Compute frequency table for a pandas series"""
    frequency = x.value_counts(normalize=False)
    frequency.index.name = x.name
    frequency.name = "Frequency"
    percent = x.value_counts(normalize=True)
    percent.name = "Percent"
    frequency_table = pd.concat([frequency, percent], axis="columns")
    return frequency_table
```

This function returns a `DataFrame` object, reporting the frequency and percentage of the observed unique values of the input pandas series `x`.

```
frequency_table(hmeq["BAD"])
```

```
     Frequency   Percent
BAD
0         4771  0.800503
1         1189  0.199497
```

The index of the `DataFrame` object displays the unique values the variable `BAD` can take on, and is named `BAD`. The first column displays the number of rows of hmeq where `BAD` is equal to the

values in the index, which gives the frequency for different values of BAD. The second column divides the result from the first column by the total number of rows, which gives the percentage frequencies. Next, let's compute the frequency table for our sample.

```
frequency_table(my_sample["BAD"])
```

```
     Frequency  Percent
BAD
0          815    0.815
1          185    0.185
```

4.2 Descriptive Statistics

Use the describe method to generate descriptive statistics of my_sample.

```
my_sample["LOAN"].describe()
```

```
count     1000.000000
mean     18350.900000
std      11549.112572
min       2000.000000
25%      10800.000000
50%      16000.000000
75%      22900.000000
max      89000.000000
Name: LOAN, dtype: float64
```

Some values are different to the SAS output as the random sample (my_sample) drawn by Python is not identical to the one drawn by SAS. Setting seed values can bypass this issue within a program environment.

4.3 Missing Values

In a next step, missing values of the numeric variables are replaced by their respective column means with the very handy fillna method.

```
my_sample_na_mean = my_sample.fillna(value=my_sample.mean())
```

Alternatively, we can also replace the missing values with the column medians.

```
my_sample_na_median = my_sample.fillna(value=my_sample.median())
```

4.4 Outlier Detection and Treatment

Standardize our sample and compute the z-scores:

```
# Select all numeric variables
num_variables = my_sample_na_mean \
    .select_dtypes(include=[np.number]).columns.tolist()

# Remove the target variable if selected
if "BAD" in num_variables:
    num_variables.pop(num_variables.index("BAD"))

# Compute z-scores for numeric variables
my_sample_cnt = my_sample_na_mean[num_variables]
zscores = my_sample_cnt.apply(stats.zscore)
```

We select the column names of all numeric variables in the data set, thereby making sure that the target variable `BAD` is not selected, and store the result in `my_sample_cnt`. Next, we apply SciPy's `zscore` function through the `apply` method. The latter is similar to the `map` method but can operate on multiple columns. This computes the mean and standard deviation of each column and respectively applies the z-score transformation. In other words, we compute the z-score for each cell of `my_sample_cnt` and store all z-scores in `zscores`. Now, we can drop rows with z-scores that are greater than three.

```
max_abs = zscores.apply(lambda x: max(abs(x)) < 3, axis="columns")
filtered_sample = zscores.loc[max_abs, :]
```

Compute the maximum absolute value of each row of z-scores, named it `max_abs`. Keep the rows where `max_abs` is less than three.

4.5 Categorization

Generate the data frame `residence` and input the values.

```
residence = pd.DataFrame({
    "default": ["good"] * 6 + ["bad"] * 6,
    "resstatus": [
        "owner", "rentunf", "rentfurn", "withpar", "other", "noanswer"
    ] * 2,
    "count": [6000, 1600, 350, 950, 90, 10, 300, 400, 140, 100, 50, 10],
})
```

Note that a `list` object can easily be extended when multiplied with a positive integer. For example, `["good"] * 3` is equivalent to specifying `["good", "good", "good"]`.

Generate the data frame `coarse1` and input the values.

```
coarse1 = pd.DataFrame({
    "default": ["good"] * 3 + ["bad"] * 3,
    "resstatus": ["owner", "renter", "other"] * 2,
    "count": [6000, 1950, 1050, 300, 540, 160],
})
```

Generate the data frame `coarse2` and input the values.

```
coarse2 = pd.DataFrame({
    "default": ["good"] * 3 + ["bad"] * 3,
    "resstatus": ["owner", "withpar", "other"] * 2,
    "count": [6000, 950, 2050, 300, 100, 600],
})
```

Create a contingency table of the data frame `coarse1` and compute the proportions by row and by column. Finally, perform the chi square test.

```
def contingency_table(df, row, col, val):
    row_values = df[row].unique().tolist()
    column_values = df[col].unique().tolist()
    conti_tbl = pd.DataFrame(index=row_values, columns=column_values)
    for r in row_values:
        for c in column_values:
            row_selected = (df[row] == r) & (df[col] == c)
            conti_tbl.loc[r, c] = df.loc[row_selected, val].values[0]

    print("Contingency table")
    print(conti_tbl)
    print()

    print("Proportions by row")
    print(conti_tbl.T / conti_tbl.sum(axis="columns"))
    print()

    print("Proportions by column")
    print(conti_tbl / conti_tbl.sum(axis="rows"))
    print()

    print("Chi square test")
    chi2, p_value, dof, _ = stats.chi2_contingency(conti_tbl)
    s = '<' if p_value < 0.0001 else '='
    p = 0.0001 if p_value < 0.0001 else p_value
    print(
        "X-squared = {:.1f}, df = {:d}, p-value {} {:.4f}".format(
            chi2, dof, s, p,
        )
    )
```

```
contingency_table(df=coarse1, row="default", col="resstatus", val="count")
```

```
Contingency table
      owner  renter  other
good   6000    1950   1050
bad     300     540    160

Proportions by row
            good   bad
owner   0.666667  0.30
renter  0.216667  0.54
other   0.116667  0.16

Proportions by column
         owner    renter     other
```

```
good  0.952381  0.783133  0.867769
bad   0.047619  0.216867  0.132231

Chi square test
X-squared = 583.9, df = 2, p-value < 0.0001
```

As before, create a contingency table of `coarse2` and compute the proportions by row and by column. Finally, perform a chi square test.

```
contingency_table(df=coarse2, row="default", col="resstatus", val="count")
```

```
Contingency table
      owner  withpar  other
good   6000      950   2050
bad     300      100    600

Proportions by row
             good  bad
owner    0.666667  0.3
withpar  0.105556  0.1
other    0.227778  0.6

Proportions by column
         owner   withpar     other
good  0.952381  0.904762  0.773585
bad   0.047619  0.095238  0.226415

Chi square test
X-squared = 662.9, df = 2, p-value < 0.0001
```