Presenting and Preparing Loan Data

Module 2 | Chapter 5 | Notebook 2

In this exercise you will take a quick look at the data for the scenario. You will also clean and prepare it. At the end of this lesson you will have practiced the following important skills:

- Cleaning data.
- Boolean masking
- One-hot encoding

Preparing the data

Scenario: You work for the company *Lemming Loans Limited*. The company provides loans to private individuals. Investors indicate how much money they want to make available and the system pools the money from different investors and forwards this money to people who want to take out a loan. The people who take out a loan often have a low credit rating, which is why they aren't getting the loans from a bank in the traditional way. Particularly risky loans get an interest rate of over 14%. These loans are internally classified as problem loans and require more attention from *Lemming Loans Limited*.

Previously, a service provider calculated the interest rate for each loan. The services of the external provider are now going to be taken over step by step by internal departments. This saves costs and makes the assessment more transparent for *Lemming Loans Limited*. The first task is to automatically divide the loans into problem loans and normal loans. The loans given out so far in the `'loans_data'` table of the *loans.db* database make up the data set for this exercise.

It is therefore a **supervised learning binary classification problem**.

First, let's familiarize ourselves a little with the data and prepare it for analysis. Before we can look at the data, we have to import it first. You will need the modules `pandas` and `matplotlb.pyplot`. Import both these modules with their conventional aliases in the following code cell.

In [10]:
```
import sqlalchemy as sa
import pandas as pd
```

The data is stored in an SQLite database. You saw how to retrieve data from databases in *Module 0, Chapter 1, Databases*. First, you need an engine that specifies the SQL dialect and the database. Store it in a variable called `engine` and use the argument

`'sqlite:///loans.db'`. Create a variable named `connection` to define the connection to the database. This is done with the method `my_engine.connect()`.

In [11]:
```
engine = sa.create_engine('sqlite:///loans.db')
connection = engine.connect()
```

Now we can take a look into the database's tables.

In [3]:
```
inspector = sa.inspect(engine)
inspector.get_table_names()
```

Out[3]:
```
['loans_data']
```

It only contains the table *loans_data*. To get the data, we pass an SQL query and the connection to the `pd.read_sql()` function. Then you should close the connection again. Run the following cell to store the data as a `DataFrame` named `loans`.

In [4]:
```
query = 'SELECT * FROM loans_data'  # define query to select all columns from the tabl
loans = pd.read_sql(query, con=connection)  # execute the query
connection.close()  # close the connection to the database

print('number of rows:', loans.shape[0], 'number of columns', loans.shape[1])
loans.head()  # print the data
```

```
number of rows: 451671 number of columns 19
```

Out[4]:

	id	issue_d	funded_amnt	term	int_rate	grade	sub_grade	emp_title	emp_length
0	36805548	Dec-2014	10400.0	36 months	6.99	A	A3	Truck Driver Delivery Personel	8 years
1	38098114	Dec-2014	15000.0	60 months	12.39	C	C1	MANAGEMENT	10+ years
2	37822187	Dec-2014	9600.0	36 months	13.66	C	C3	Admin Specialist	10+ years
3	37662224	Dec-2014	7650.0	36 months	13.66	C	C3	Technical Specialist	< 1 year
4	37842129	Dec-2014	21425.0	60 months	15.59	D	D1	Programming Analysis Supervisor	6 years

We have 451671 granted loans, each with 19 columns. The following data dictionary explains what the data means.

Column number	Column name	Type	Description
0	`'id'`	categorical	ID number for the loan
1	`'issue_d'`	date (`string`)	year and month when the credit was granted

Column number	Column name	Type	Description
2	`'funded_amnt'`	continuous (`float`)	Amount loaned in USD
3	`'term'`	continuous (`int`)	Number repayment installments in months (36 or 60)
4	`'int_rate'`	continuous (`float`)	Interest rate
5	`'grade'`	categorical	Assigned credit score grade by external service provider
6	`'sub_grade'`	categorical	sub-grade of assigned credit score grade by external service provider
7	`'emp_title'`	categorical	Job title of the borrower when taking out the loan
8	`'emp_length'`	categorical (ordinal)	Length of borrower's current employment in years when taking out the loan
9	`'home_ownership'`	categorical	Housing situation (renting='RENT', mortgage='MORTGAGE', Homeowner='OWN' as well as 'OTHER', 'NONE', and 'ANY')
10	`'annual_inc'`	continuous (`float`)	annual income in USD
11	`'title'`	categorical	Loan purpose category
12	`'zip_code'`	categorical	borrower's zip code (first 3 digits only)
13	`'verification_status'`	categorical	Indicates whether income has been verified or not ('Source Verified', 'Verified', 'Not Verified')
14	`'purpose'`	categorical	Loan purpose category
15	`'total_acc'`	continuous (`int`)	Total number of borrower's loans
16	`'percent_bc_gt_75'`	continuous (`float`)	Proportion of credit cards up to 75% of their limit
17	`'total_bc_limit'`	continuous (`float`)	total credit card limit
18	`'revol_bal'`	continuous (`float`)	Open credit card amounts

Before we start analyzing the data, we should always separate our data into a training, validation and test data set. This time, however, *Lemming Loans Limited* wants to carry out the model tests internally. So we only have to take care of the hyperparameter tuning and to do this we just need to separate the data into training and validation data sets (more data for training). For this we'll use the `train_test_split()` function from `sklearn.model_selection` (see *Module 2, Chapter 4, Vectorizing Text*). So far we've first split the data set into a target variable `target` and features `features` before the split, but `train_test_split()` can also split whole data sets correctly. You then get two data sets `loans_train` , as a training data set, and

`loans_val`, as a validation data set. Now use `train_test_split()` to create the `loans_train`, `loans_val` data sets as the output. Instead of specifying two arguments `features` and `target` as the first two *input* parameters, you just have to enter the data set `loans` as the first argument. After that you only need to use `test_size=0.3` and `random_state=0`.

In [5]:
```
from sklearn.model_selection import train_test_split

loans_train, loans_val = train_test_split(loans,
                                          test_size = 0.3,
                                          random_state = 1)
```

Now we can start analyzing the data. All our decisions regarding data cleaning and feature generation are made exclusively on the basis of the training data set `loans_train`. When we have finished the analysis, we'll apply all the manipulations we have developed to the validation set `loans_val`. Thus simulate the complete independence of the validation data from the training data. Let's get started!

First, we'll create a copy of `loans_train` so that we can process it safely. Use the `.copy()` method to create a copy `train` from `loans_train`.

In [6]:
```
train = loans_train.copy()
```

Now we'll work on `train`. We'll start by creating the target variable. Create a column called `'problem_loan'`. It should contain a `1` if the loan has an interest rate of more than 14%, i.e. if `'int_rate' > 14`. Otherwise it should contain `0`. Then print the number of `1` values. There should be 144189.

In [7]:
```
mask = train.loc[:,'int_rate'] > 14
print(len(train.loc[mask,:]))

train.loc[:,'problem_loan'] = [1 if int_rate > 14 else 0  for int_rate in train.loc[:,

mask = train.loc[:,'problem_loan'] == 1

print(len(train.loc[mask,:]))
print(len(train.loc[:,:]))
```

```
144221
144221
316169
```

This is almost half of all the loans in the training data. So our data set is quite balanced when it comes to the target classes.

What data types do our columns have?

In [8]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 316169 entries, 43200 to 128037
Data columns (total 20 columns):
 #   Column               Non-Null Count   Dtype  
---  ------               --------------   -----  
 0   id                   316169 non-null  int64  
 1   issue_d              316169 non-null  object 
 2   funded_amnt          316169 non-null  float64
 3   term                 316169 non-null  object 
 4   int_rate             316169 non-null  float64
 5   grade                316169 non-null  object 
 6   sub_grade            316169 non-null  object 
 7   emp_title            297214 non-null  object 
 8   emp_length           301725 non-null  object 
 9   home_ownership       316169 non-null  object 
 10  annual_inc           316169 non-null  float64
 11  title                316158 non-null  object 
 12  zip_code             316169 non-null  object 
 13  verification_status  316169 non-null  object 
 14  purpose              316169 non-null  object 
 15  total_acc            316169 non-null  float64
 16  percent_bc_gt_75     280380 non-null  float64
 17  total_bc_limit       283166 non-null  float64
 18  revol_bal            316169 non-null  float64
 19  problem_loan         316169 non-null  int64  
dtypes: float64(7), int64(2), object(11)
memory usage: 50.7+ MB
```

The `object` type appears quite often. The columns `['issue_d', 'term', 'grade', 'sub_grade', 'emp_title', 'emp_length', 'home_ownership', 'title', 'zip_code', 'verification_status', 'purpose']` probably contain *strings*. We'll take a look at these columns and prepare the data if necessary.

Now let's start with `'title'` and `'purpose'`. Both refer to the reason for the loan. Print the first 20 rows of these two columns. Do you notice anything?

In [13]:
```
train.loc[:, ['title','purpose']].head(20)
```

Out[13]:

	title	purpose
43200	Debt consolidation	debt_consolidation
23971	Debt consolidation	debt_consolidation
258958	Life After College	other
358559	Debt Consolidation	debt_consolidation
405277	CC Payoff	debt_consolidation
106910	Debt consolidation	debt_consolidation
147829	Debt consolidation	debt_consolidation
333611	Debt Consolidation	debt_consolidation
358247	Get out of debt and get happy plan	debt_consolidation
99785	Debt consolidation	debt_consolidation
212483	Debt consolidation	debt_consolidation
201885	Debt consolidation	debt_consolidation
361471	Debt Consolidation	debt_consolidation
96610	Debt consolidation	debt_consolidation
196735	Home improvement	home_improvement
370681	Other	other
341420	debt consolidation	debt_consolidation
99624	Credit card refinancing	credit_card
416423	Debt consolidation loan	debt_consolidation
303350	Debt consolidation	debt_consolidation

Apparently both columns are very similar. Get the number of unique values in both columns.

In [19]:
```
len(train.loc[:, 'title'].unique())
len(train.loc[:, 'purpose'].unique())
```

Out[19]:
```
45512
```

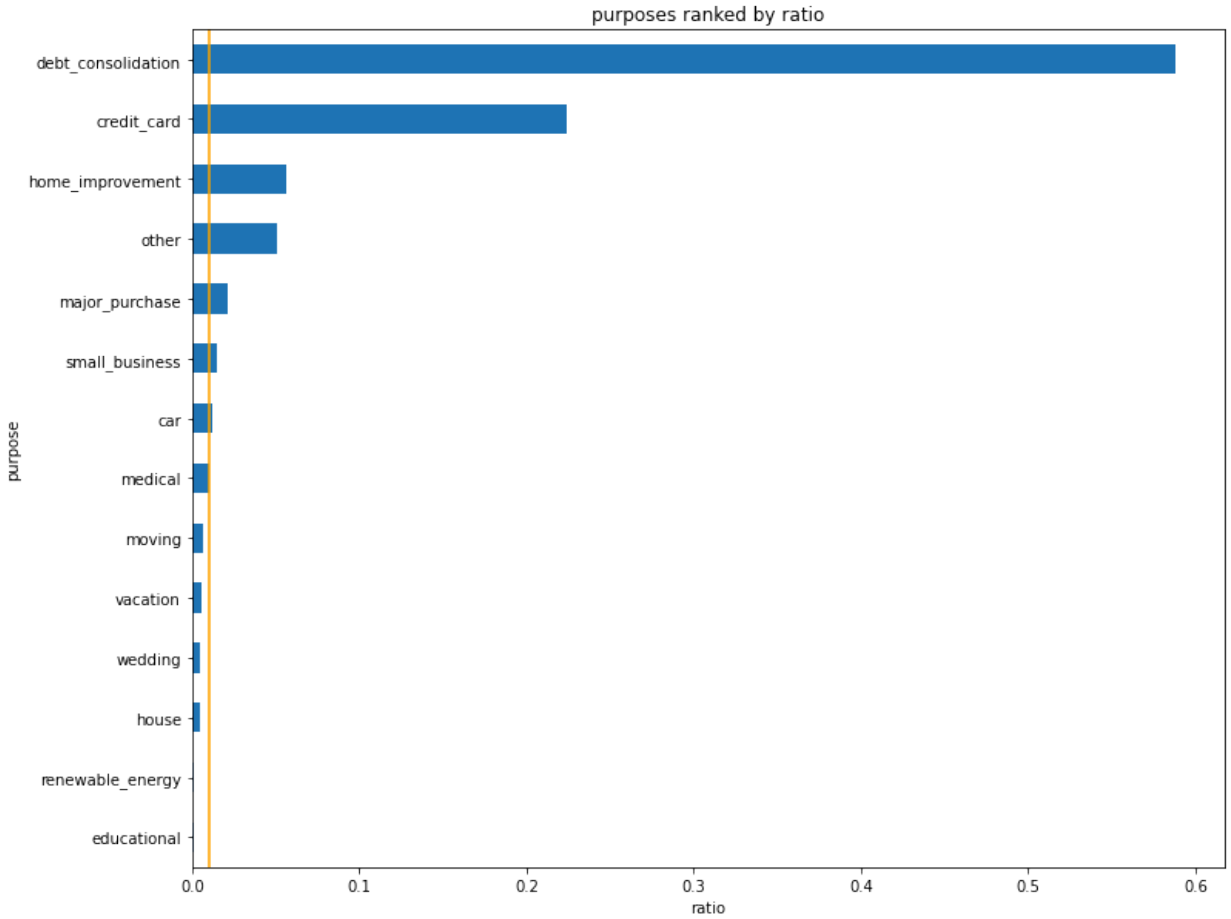
`'Title'` has 45592 unique values, whereas `'purpose'` has only 14. This is because `'title'` is generated from a free text field, Whereas `'purpose'` is limited to a selection of options. Most potential borrowers enter something similar in `'title'`. However, sometimes it may contain completely different information such as `All for one, one for all`. So we should limit ourselves to just using `'purpose'` only, since it is the more concrete version of `'title'`. We'll delete `'title'` later, along with other unnecessary columns.

Now let's return to `'purpose'`. What are the proportions of each purpose in the total data? Run the following cell to get an idea of it.

In [20]:
```
# import matplotlib
import matplotlib.pyplot as plt

# create dataframe with ratios and plot
fig, ax = plt.subplots(1, figsize=(12,10))  # define figure and  axes
train_purpose_ratios = pd.crosstab(index=train.loc[:,'purpose'],columns='ratio', norma
train_purpose_ratios.plot(kind='barh', legend=False, ax=ax)  # plot ratios as horizont

# improve plot
ax.set(xlabel='ratio', title='purposes ranked by ratio')
ax.vlines(x=0.01, ymin=-0.5, ymax=13.5, color ='orange');  # draw 1% line
```



`'dept_consolidation'`, is by far the most frequently cited reason. To make it easier for us, we'll group the categories that make up less than 1% (orange line) into `'other'`. Run the following code cell to do this.

In [21]:
```
# create mask and change values
mask_purpose = train.loc[: , 'purpose'].isin(['educational', 'renewable_energy', 'hous
       'moving', 'medical'])
train.loc[mask_purpose, 'purpose'] = 'other'

train['purpose'].unique()  # check unique values
```

Out[21]:
```
array(['debt_consolidation', 'other', 'home_improvement', 'credit_card',
       'car', 'small_business', 'major_purchase'], dtype=object)
```

The `'zip_code'` column is relatively clear. It contains the borrower's zip code. But only the first three digits. If we want to use them as categories, we end up with quite a lot of new

categories. To make it easier for us to keep track of our categories, we'll simplify the zip code a lot. We'll only use the first digit of the zip code. This only approximately indicates which federal state the code is for, but we'll accept that here.

Create a new column named `'1d_zip'`, which only contains the first digit of each zip code. Then check the number of unique values in this column, it should be 10.

In [23]:
```
train['1d_zip'] = [code[0]  for code in train.loc[:, 'zip_code']]
train.loc[:,['zip_code','1d_zip']]
```

Out[23]:

	zip_code	1d_zip
43200	940xx	9
23971	463xx	4
258958	195xx	1
358559	532xx	5
405277	207xx	2
...	...	...
73349	785xx	7
371403	577xx	5
312201	356xx	3
267336	925xx	9
128037	853xx	8

316169 rows × 2 columns

Now let's look at the `'issue_d'` column. This includes both the month and year that the loan was issued. We'll split up these two pieces of information. The month is in the first 3 characters of each value and the year is in the last 4 characters. Save the months as a new column called `'month'` and the years as `'year'`. Convert the years into a numeric column.

In [28]:
```
train.loc[:,'month'] = train.loc[:,'issue_d'].str[:3]
train.loc[:,'year'] = pd.to_numeric(train.loc[:,'issue_d'].str[4:8])
train.loc[:,['month','year']]
```

Out[28]:

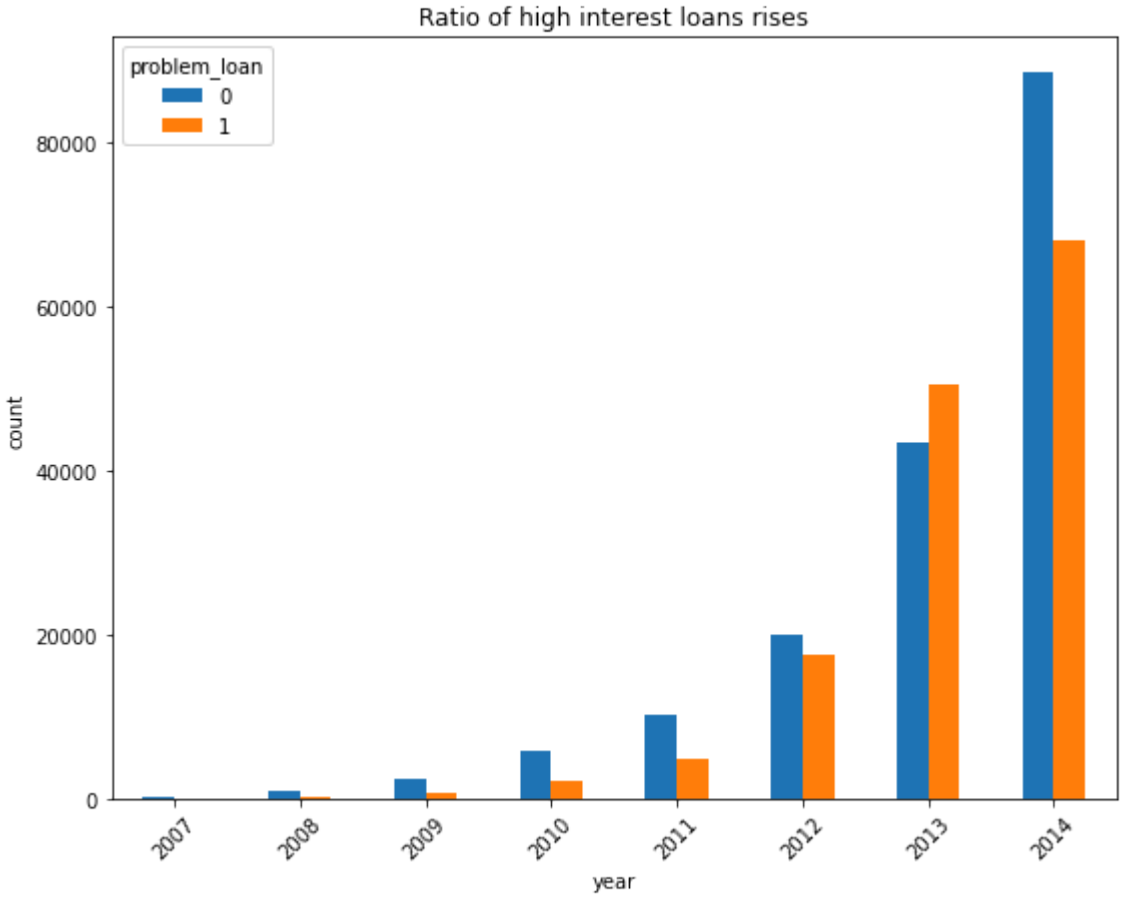
	month	year
43200	Oct	2014
23971	Nov	2014
258958	Sep	2009
358559	May	2013
405277	Nov	2012
...	...	...
73349	Sep	2014
371403	Apr	2013
312201	Sep	2013
267336	Dec	2013
128037	Jun	2014

316169 rows × 2 columns

How many loans were issued in each year? Run the following code cell, to find out.

In [29]:
```
# create dataframe with ratios and plot
fig, ax = plt.subplots(1, figsize=(9,7))
pd.crosstab(index= train['year'], columns=train['problem_loan']).plot(kind='bar', ax=a

# improve plot
ax.xaxis.set_tick_params(rotation=45)  # rotate labels of x axis
ax.set(ylabel='count', title='Ratio of high interest loans rises');
```



As we can see, *Lemming Loans Limited* has grown strongly in recent years and is lending more and more. It's no wonder that the company wants the loan evaluation process to become less dependent on external service providers in the future.

The next column we'll be looking at is `'emp_length'`. Run the following code cell, to see which values this column contains.

In [31]:
```
train.loc[:, 'emp_length'].unique()
```

Out[31]:
```
array(['9 years', '1 year', '< 1 year', '8 years', '7 years', '2 years',
       '3 years', '10+ years', '4 years', '6 years', '5 years', None],
      dtype=object)
```

We see that the length of employment is actually an ordinal variable. However, we still have the values `None`, which are missing values. These missing values could actually have a meaning here. Namely, that the people with a missing value here are unemployed, and so presumably have not indicated either a period of employment or a job title in the `'emp_title'` column.

Create a boolean mask to select rows with a missing value in the `'emp_length'` column. Are the values in the `'emp_title'` column also missing in these rows?

Tip: Use a built-in `DataFrame` or `Series` method that checks for missing values. A direct comparison with `None` does not work.

In [42]:
```
mask = train.loc[:, 'emp_length'].isna()
train.loc[mask, ['emp_length','emp_title']]
```

Out[42]:

	emp_length	emp_title
76571	None	None
126589	None	None
383172	None	None
222207	None	None
108495	None	None
...	...	...
406999	None	None
208239	None	None
338712	None	None
176485	None	None
267336	None	None

14444 rows × 2 columns

In most rows both values are missing. Here we can assume that the borrower was not employed when they applied for the loan. The cases where there is a job title but no employment length are more puzzling. And vice versa. It could be helpful to talk to colleagues who are more familiar with the data to clear this up. But for now we'll remove all these cases. We will convert the employment lengths into an ordinal variable.

Run the following code cell to go through the following steps:

- Replace the `'<1'` value in `'emp_length'` with `'0'`.
- Extract the numerical data from `'emp_length'` and put it in the new column `'emp_length_num'`.
- Create the new column `'unemployed'`. It is populated with `1` if both `'emp_length'` and `'emp_title'` are missing. Otherwise the values should be `0`.
- Delete the lines that have a job title but no employment length - and vice versa.
- Fill the missing values in `'emp_length_num'` with `-1`. This will prevent you from working with missing values or the rows from being deleted later.

In [44]:
```
# extract numeric values of emp_length not emp_length_num
train.loc[:, 'emp_length'] = train.loc[:,'emp_length'].str.replace(pat=r'< 1', repl='0
train.loc[:,'emp_length_num'] = train.loc[:,'emp_length'].str.extract(r'(\d+)', expanc

# create new column unemployed
train.loc[:, 'unemployed'] = 0  # fill with 0 (person is employed)
mask_unemployed = (train.loc[:, 'emp_length'].isna()) & (train.loc[:, 'emp_title'].isr
train.loc[mask_unemployed, 'unemployed'] = 1  # fill selected rows with 0 (person is r

# delete rows with employment length but missing job title
mask_na = (train.loc[:, 'emp_title'].isna()) & (~train.loc[:, 'emp_length'].isna())
train = train.drop(train.index[mask_na])
```

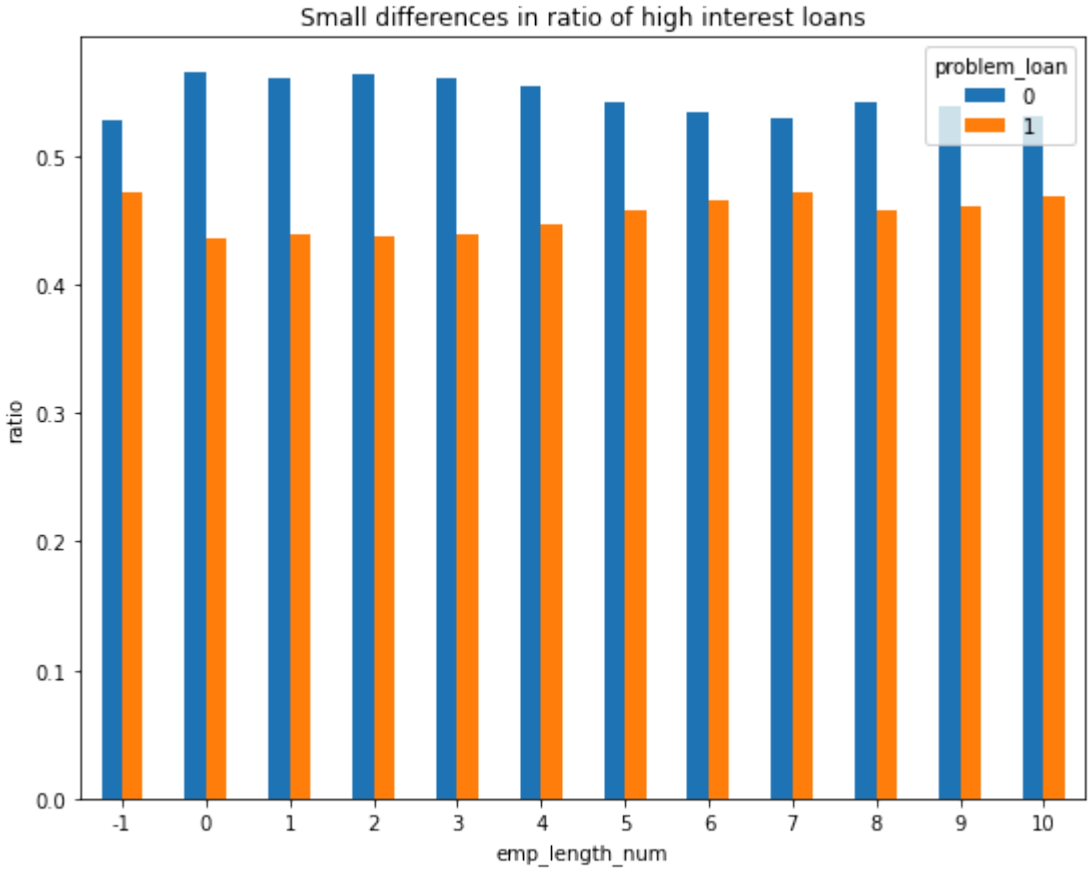
```
# delete rows with job title but missing employment length
mask_na = (~train.loc[:, 'emp_title'].isna()) & (train.loc[:, 'emp_length'].isna())
train = train.drop(train.index[mask_na])

# fill missing values in emp_length_num with -1 to prevent dropping them
train.loc[:, 'emp_length_num'] = train.loc[:, 'emp_length_num'].fillna(-1)
train.loc[:, 'emp_length_num'] = pd.to_numeric(train.loc[:, 'emp_length_num'])
```

Use the following cell to visualize the proportion of high-interest loans as a function of the employment length.

In [45]:
```
# create dataframe with ratios and plot
fig, ax = plt.subplots(1, figsize=(9,7))
pd.crosstab(index= train['emp_length_num'], columns=train['problem_loan'], normalize='

# improve plot
ax.xaxis.set_tick_params(rotation=0)  # rotate labels of xaxis
ax.set(ylabel='ratio', title='Small differences in ratio of high interest loans');
```



You can see that the proportions of normal and problem loans are distributed relatively evenly across all groups of employment length. For people without a job title and without an employment length (`-1`), the share of problem loans is slightly higher. Although normal loans are more common, there are also quite a lot of problem loans, which is not surprising as this is not a traditional bank.

Now let's look at how many unique values the remaining `object` columns have. Run the following cell to do this:

In [46]:
```
train.loc[:, ['term', 'emp_title', 'grade', 'sub_grade', 'home_ownership', 'verificati
```

Out[46]:
```
term                        2
emp_title              148690
grade                       7
sub_grade                  35
home_ownership              6
verification_status         3
dtype: int64
```

You can see that `'emp_title'` has a great deal of unique values. Let's look at the first 20 values in this column.

In [47]:
```
# Print first 20 entries of 'emp_title'
train.loc[:,'emp_title'].head(20)
```

Out[47]:
```
43200                  Program Manager
23971                               RN
258958                     Finish Line
358559                             BDI
405277          University of Maryland
106910                    manufacturing
147829                Business manager
333611                Tuva Interactive
358247                           Oracle
99785     Associate Production Manager
212483                Spectrum manager
201885          Payroll Administration
361471                Constant Contact
96610               Web Services Manager
196735                        Detective
370681               Vortek Instruments
341420                  JP Morgan Chase
99624                   General Manager
416423               City of Hinesville
303350         Production Control Lead
Name: emp_title, dtype: object
```

You can see that many entries have the same meaning and are just written differently, just like in the `'title'` column. This column was probably also created with a free text field. This is an initial indication that `'emp_title'` is irrelevant. It's also obvious that a job title doesn't include much information (especially today). The salary and the length of employment are much more important for the bank. So we'll consider `'emp_title'` as irrelevant and remove it. But we'll get to that later. First let's look at the other columns.

The `'term'` column only contains 2 different values. What are they?

In [50]:
```
train.loc[:,'term'].unique()
```

Out[50]:
```
array([' 36 months', ' 60 months'], dtype=object)
```

The loan duration is either 36 months or 60 months. Replace the values in `'term'` with the number of months, i.e. `36` or `60`. Make sure that the column is recognized as a numeric column afterwards.

In [52]:
```
train.loc[:, 'term'] = train.loc[:, 'term'].replace({' 36 months': 36, ' 60 months': 6
```

We shouldn't use the `'year'` and `'month'` columns in the model for the time being. Your superiors are convinced that seasonal effects play a subordinate role here. The increase in high-interest loans in recent years is mainly due to the marketing strategy. So we no longer need the columns `['id', 'issue_d', 'month', 'year', 'emp_title', 'emp_length', 'title', 'zip_code', 'int_rate']`. Remove them from `train`.

In [55]:
```
train = train.drop(['id', 'issue_d', 'month', 'year', 'emp_title', 'emp_length', 'titl
```

Now you've almost finished cleaning and preparation the data. Then check if there are still values missing. Print the number of missing values for each column.

In [56]:
```
train.isna().sum()
```

Out[56]:
```
funded_amnt                0
term                       0
grade                      0
sub_grade                  0
home_ownership             0
annual_inc                 0
verification_status        0
purpose                    0
total_acc                  0
percent_bc_gt_75       34476
total_bc_limit         31717
revol_bal                  0
problem_loan               0
1d_zip                     0
emp_length_num             0
unemployed                 0
dtype: int64
```

The `'percent_bc_gt_75'` and `'total_bc_limit'` columns still have a lot of missing values. We don't quite know why these values are missing. Both columns are related to credit cards. Perhaps these are mainly people who don't have credit cards. However, the number of missing values in both columns differs by almost 4000. So let's keep things simple by removing the missing values.

In [57]:
```
train = train.dropna()
```

For the variables `['grade', 'sub_grade', 'home_ownership', 'verification_status','1d_zip', 'purpose']` we'll make it easy and use *one-hot encoding* (see *Chapter 3, Data Pipelines with Column Selection*) to create numerical features from them. But before get round to this, let's summarize the steps above in a single function. The function should take the training data `train` as its input and then output the edited version of `train`. Think about how to construct this kind of function, and try it out yourself! Otherwise the function is defined for you in the next code cell and you can use that by executing the cell.

In [58]:
```
#define the function, the argument is a DataFrame `df` which must have the same column
def data_cleaner(df):
```

```
    '''This function performs all above mentioned data manipulation steps.'''

    copy=df.copy()

    # Create target column
    mask=(copy.loc[:,'int_rate'] > 14).astype('int8')
    copy.loc[:,'problem_loan']=mask

    # group 'purpose'
    mask_purpose = copy.loc[: , 'purpose'].isin(['educational', 'renewable_energy', 'h
       'moving', 'medical'])
    copy.loc[mask_purpose, 'purpose'] = 'other'

    # get only first number of zip code
    copy.loc[:,'1d_zip'] = copy.loc[:,'zip_code'].str[0]

    # replace term with numbers
    copy.loc[:, 'term'] = copy.loc[:, 'term'].replace({' 36 months': 36, ' 60 months':

    # extract numeric values of emp_length into emp_length_num
    copy.loc[:, 'emp_length'] = copy.loc[:,'emp_length'].str.replace(pat=r'< 1', repl=
    copy.loc[:,'emp_length_num'] = copy.loc[:,'emp_length'].str.extract(r'(\d+)', expa

    # create new column unemployed
    copy.loc[:, 'unemployed'] = 0  # fill with 0 (person is employed)
    mask_unemployed = (copy.loc[:, 'emp_length'].isna()) & (copy.loc[:, 'emp_title'].i
    copy.loc[mask_unemployed, 'unemployed'] = 1  # fill selected rows with 0 (person i

    # delete rows with employment length but missing job title
    mask_na = (copy.loc[:, 'emp_title'].isna()) & (~copy.loc[:, 'emp_length'].isna())
    copy = copy.drop(copy.index[mask_na])

    # delete rows with job title but missing employment length
    mask_na = (~copy.loc[:, 'emp_title'].isna()) & (copy.loc[:, 'emp_length'].isna())
    copy = copy.drop(copy.index[mask_na])

    # fill missing values in emp_length_num with -1 to prevent dropping them
    copy.loc[:, 'emp_length_num'] = copy.loc[:, 'emp_length_num'].fillna(-1)
    copy.loc[:, 'emp_length_num'] = pd.to_numeric(copy.loc[:, 'emp_length_num'])

    # drop irrelevant columns. We have not introduced `year` and `month`, thus we do n
    copy = copy.drop(['id', 'issue_d', 'emp_title', 'emp_length', 'title', 'zip_code',

    # drop missing values

    copy = copy.dropna()

    return copy
```

Now we can execute the data cleaning in a single line. Run the following code cell to generate the cleaned training data `clean_loans_train` and validation data `clean_loans_val`.

In [60]:
```
clean_loans_train = data_cleaner(loans_train)
clean_loans_val = data_cleaner(loans_val)
```

Now to the *one-hot encoder*. We want to use `OneHotEncoder` from `sklearn.preprocessing`. The next cell contains a way to assign a separate column in the

`DataFrame` for each feature that the `OneHotEncoder` creates. It uses the following steps:

- We import `OneHotEncoder`, from `sklearn.preprocessing`, and `ColumnsTransformer` from `sklearn.compose`.
- We define a list `columns=['grade', 'sub_grade', 'home_ownership', 'verification_status','1d_zip', 'purpose']` with the columns we encode.
- We instantiate `OneHotEncoder` in the variable `ohe` and specify the parameter `sparse=false`. This way the *encoding* is stored in a normal *array* instead of in a sparse matrix.
- We instantiate `ColumnTransformer` in the variable `encoder` and specify the `OneHotEncoder` as a transformer and the columns `columns` which are the columns to be encoded. We also want `ColumnTransformer` to keep all the other columns in the `DataFrame` and not to change them. We can achieve this with the `remainder='passthrough'` parameter.
- We fit `ColumnsTransformer` to the cleaned training data `clean_loans_train` and generate a list with the names of the new features. For this we'll use the `named_transformers_` attribute of `ColumnTransformer` to access the `.get_feature_names()` method of the `OneHotEncoder`. This gives us all the **new** column names. The indices of the **unaffected** columns are located in the third component of the `._remainder` attribute of `ColumnTransformer`. This selects the names of the unaffected columns from the `._df_columns` attribute of `ColumnTransformer`, which contains a list of all names of the DataFrames **before** the transformation, in one list. We store these names in another list.
- We create a new `DataFrame` from the `.transform` method of `ColumnTransformer` and the names of the new columns (including the unaffected ones). We'll do this for the cleaned training and validation data. We store the results in the corresponding variables `final_loans_train` and `final_loans_val`.

The resulting DataFrames should each have 78 columns. Have a look at the resulting DataFrames!

In [61]:
```
# import relevant transformers
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

# define columns for OHE
columns=['grade', 'sub_grade', 'home_ownership', 'verification_status','1d_zip', 'purp

# set up pipeline
ohe = OneHotEncoder(sparse=False)
encoder = ColumnTransformer([('OHE',ohe, columns)], remainder='passthrough')

# fit encoder
encoder.fit(clean_loans_train)

# restore column names for final DataFrames
ohe_names = encoder.named_transformers_['OHE'].get_feature_names(columns)
remaining_names = encoder._df_columns[encoder._remainder[2]]
```

```
#apply encoding and create DataFrames
final_loans_train = pd.DataFrame(encoder.transform(clean_loans_train), columns = list(
final_loans_val = pd.DataFrame(encoder.transform(clean_loans_val), columns = list(ohe_

#check shapes
print(final_loans_train.shape)
print(final_loans_val.shape)
```

```
(276926, 78)
(118561, 78)
```

Now we've finished preparing the data. The only thing left to do is to divide the data into `features` and `target`. Run the following cell to to do just that.

In [62]:
```
# split train and validation data into features and target
target_train = final_loans_train.loc[:, 'problem_loan']
features_train = final_loans_train.drop('problem_loan', axis=1)
target_val = final_loans_val.loc[:, 'problem_loan']
features_val = final_loans_val.drop('problem_loan', axis=1)
```

We'll use this data in the next lesson. So we should save it as a `pickle`. Run the following code cell for this.

In [63]:
```
# save data as pickle
features_train.to_pickle('features_train.p')
target_train.to_pickle('target_train.p')
features_val.to_pickle('features_val.p')
target_val.to_pickle('target_val.p')
```

That's great! Now the data is ready to be used directly with our model. If you accidentally overwrite or delete the data, just run all code cells in this lesson again.

Congratulations: Your superiors at Lemming Loans Limited are very pleased with your data preparation. They are aware that cleaning and preparing data is a large part of the work.

Remember:

- Data cleaning and preparation is often very time consuming but it's also extremely important.
- Boolean masks are almost always used in cleaning or preparation.
- Categories can be prepared for our models with *one-hot encoding*.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
