

Before we go ahead and start our discussion on Machine Learning , we will discuss how to get our data ready to be fed into an ML algorithm . We will learn what all transformations different parts of the data need to go through in this chapter . There are two basic goals of the data transformation here :

- All the information needs to be represented numerically, or in other words all the columns of your transformed data need to be numeric
- No missing values in the data [Although some algorithms might support missing values, but for our discussions sake we will focus on developing pipelines which explicitly take care of missing values and give us the data which works with all the algorithms]

We need significant work to achieve this because raw data comes in various forms and has all sorts of fundamentals issues . Not all information is inherently numeric. Information can be non-numeric and still stored as numbers, and list goes on . We will try to cover majority of such scenarios in our subsequent discussion, and will consider what plan of action to adopt for each such scenario. Each of these scenarios will have two things :

- What form the data is in [data type of the column after its been read in by python]
- What transformations it needs to go through in order to meet the two goals mentioned in the beginning .

Information in the column : Numeric, Data Type of the Column: Numeric

Data type of the column being numeric is a simple thing to figure out by looking at `pandas.DataFrame.info()` Or `pandas.DataFrame.dtypes` . However its not necessary that the information stored as numbers is numeric. For example a column might contain integer coding for city names , or it might contain phone numbers or pin codes for addresses. None of this is numeric information. They are arbitrary IDs given to categorical information. A quick way to determine if the information stored as numbers is actually numeric is to check if difference between any two values from the data is meaningful . For example if you have column containing Ages of people, difference between 38 years of age and 42 years of age is meaningful in the sense that the latter person is 4 years older than the former. Now consider two phone numbers say `4152739164` and `3108749015` , the difference between them has no meaning at all.

Once you have determined that the data in the column is stored as numbers and the information is also numeric , there is only one thing that we need to ensure, that is, it shouldn't have any missing values. Missing values in the data will usually show up as `NaN` or `NULL` or something of the kind, but sometimes you might also come across placeholders like `-99` or some other values being used as indicators of missing values , study the data dictionary carefully to handle these properly .

Once you have identified records with missing values across some columns , there are many ways we can handle them

- Fill with business context. For example a missing sensor reading might simply mean `0` . A missing account balance might simply mean either last updated value or `0` .
- When the business context is not available , replace with global median of the column
- Sometimes global median might not make sense , for example if you have missing values of body weight for patients in a medical dataset, instead of replacing missing values with global median of patient weight, it would make much more sense to take group-by median at age brackets and genders and use that to replace.

Information in the column : Numeric , Data Type of the Column: Object [needs just type-casting to convert to number]

Sometimes a column containing numeric information might get read as `object` [categorical] type because some strings in the data for example a column shown below :

Age
32
26
.
'missing'
46
67

in this case we simply typecast them to number like this

```
1 data[col]=pd.to_numeric(data[col],errors='coerce')
```

option `errors='coerce'` is used to force the non-numeric values to `NaN` and then we can treat them with missing value strategies we discussed in the earlier section.

Information in the column : Numeric , Data Type of the Column: Object [needs custom functions to extract numeric info]

Sometimes columns with numeric information are read as object they might be stored as ranges or simply have descriptive strings attached to the data which need to be processed case by case basis . Here is such an example

Job Experience
3 years
>1 years
10+ years
2 years
1 year
7 years

Here we can write a custom function which removes the strings and converts the data into a number . Missing values if any arising from the conversion; get treated as earlier . Note that just removing the strings will not convert the column into a number type, we will have to do type-casting.

```
1 def custom_func_jobex(col_name,data):
2     col_mod=data[col_name].str.replace('years','')
3     col_mod=col_mod.str.replace('year','')
4     col_mod=col_mod.str.replace('10+', '10', regex=False)
5     col_mod=col_mod.str.replace('>1', '1', regex=False)
6     return pd.to_numeric(col_mod,errors='coerce')
```

option `regex=False` is used to make sure that we process string as raw strings instead of regular expressions which can have different meaning than that intended for the data.

Note that you will need different custom functions for each such column depending on the context.

Information in the column : Categorical , Data Type of the Column: Can be Object or Numeric

For categorical information we convert them to one hot encoded representation , here is the process

- replace missing values with a placeholder like `__missing__` which is not very likely to appear in the data otherwise. If that is the case then chose another placeholder .
- Club rare categories in under one category like `__low_freq_cats__` , you will have to chose some frequency cutoff to decide what you consider rare [say `1% of the total obs in the data`]
- Lets say after applying first two steps , the data column looks like this

Cat_feat
'A'
'B'
'__missing__'
'__low_freq_cat__'
'A'
'B'
'C'
⋮

And lets say frequency counts for various categories looks like this in the data

	Frequency
'A'	4000
'__low_freq_cat__'	3500
'B'	1400
'__missing__'	200
'C'	50

Note that its possible that after clubbing low frequency categories together , their combined count is high [not that it will always happen but its possible]. In the remaining n categories we will pick top $n-1$ categories and create one hot encoded columns for them [also called `dummies`]

Cat_feat	Cat_feat_A	Cat_feat_lfc	Cat_feat_B	Cat_feat_miss
'A'	1	0	0	0
'B'	0	0	1	0
'__missing__'	0	0	0	1
'__low_freq_cat__'	0	1	0	0
'A'	1	0	0	0
'B'	0	0	1	0
'C'	0	0	0	0
⋮	⋮	⋮	⋮	⋮

I have kept the original column here for your reference. It gets removed in the processing. Note that the dummy columns take value `1` when the original column takes the value as corresponding category and `0` otherwise . The last category which we are not creating dummy for [known as base category], when that occurs in the data, all dummy columns are `0` as you can see in the example.

Mathematically it doesn't matter which category is considered to be the base category, its just a general convention to treat highest frequency or lowest frequency category to treat as base category.

Information in the column : Date , Data Type of the Column: Can be Object or DateTime

Dates are a special kind of data type. Here are some interesting quirks

- Difference between two dates make sense but any other mathematical operation is meaningless
- Dates themselves are not numeric but their individual components are numeric like `year` , `month` , `weekday` , `hour` etc .
- Components except year are however are not traditionally numeric they are cyclic in nature. For example if you represent months numerically like so `{1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May', 6:'Jun', 7:'Jul', 8:'Aug', 9:'Sep', 10:'Oct', 11:'Nov', 12:'Dec'}` , its not true representation of how months behave for real. For example `1-12=11` , but `January` and `December` are not actually 11 months apart due to cyclicity of months. Same is true for other cyclic components of date as well

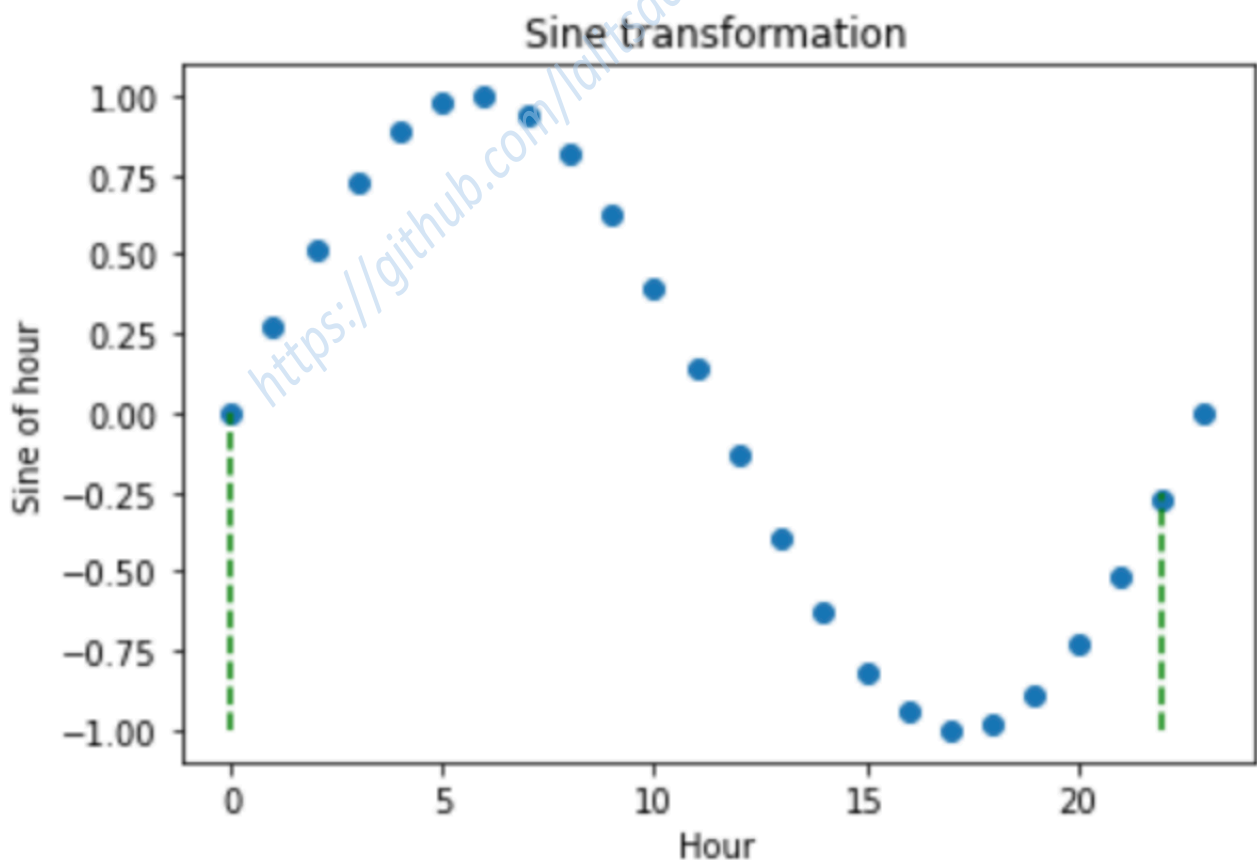
Here is how we will convert dates to numeric representation

- If there are multiple dates in the data, it might be meaningful to create their difference as a new column [in `days` or `hours` or other units depending on the precision available and necessary in the context]
- Cyclicity of the components like `month`, `weekdays` etc can be captured by using cyclic functions [`sin`, `cos`], however a single cyclic function doesn't give unique representation, so pair is used. Since `sin` and `cos` complete their cycles in interval of width 2π , we will need to scale the range of the input components to the same.
- Simple numeric component like `year` can simply be extracted and used if it makes sense in the context.

```
1 component_sin=sin(2*pi*num_component/cycle_freq)
2 component_cos=cos(2*pi*num_component/cycle_freq)
```

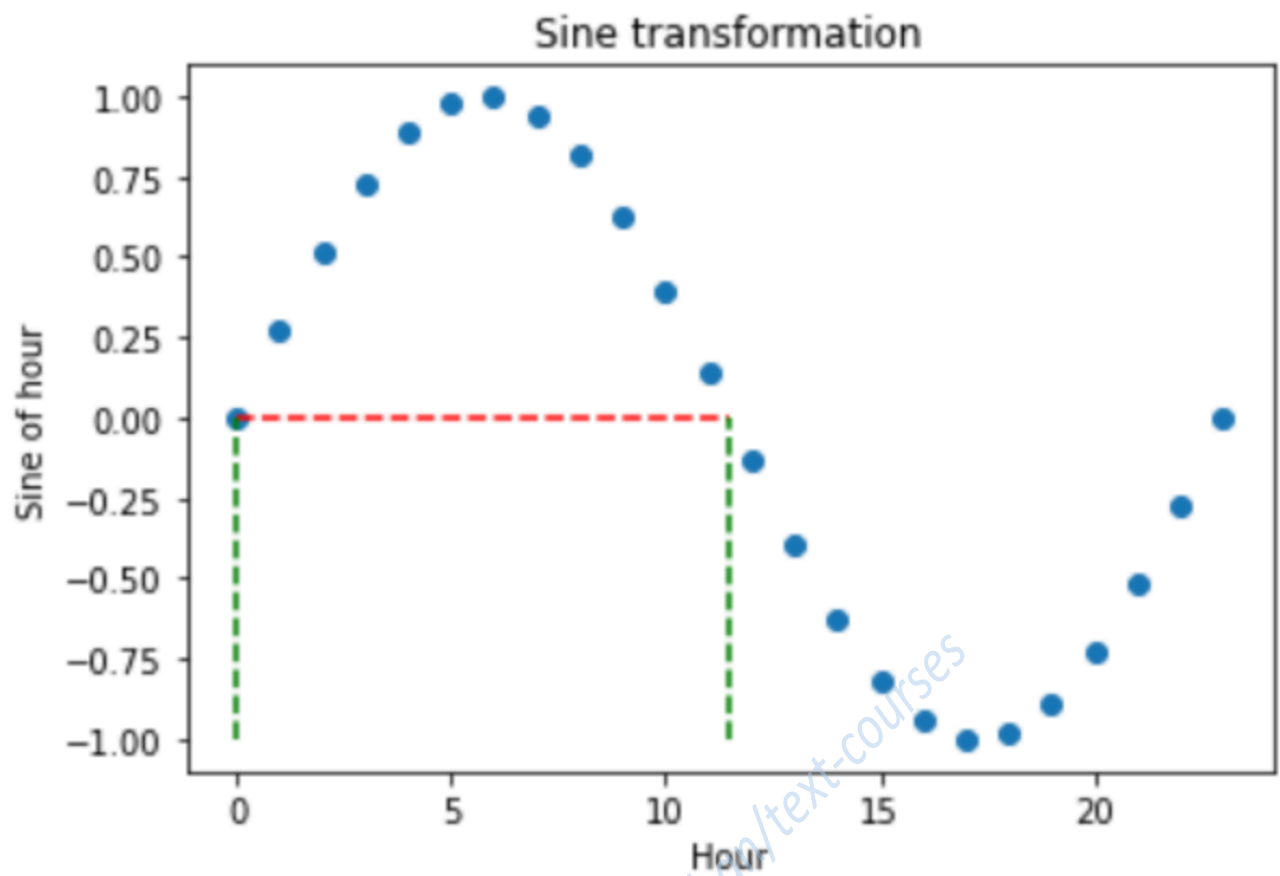
Here the `cycle_freq` is essentially the range of values that the components cycle over. For example, for months it is `12`, for hours in a day it's `24`, for minutes in an hour it's `60` and so on. `num_component` is simply the traditional numeric values assigned to the component values. For example for months we use number `1` to `12`, for weekdays we use numbers `1` to `7` [it doesn't really matter whether we start the week on Monday or Sunday as long as it's consistent in the data].

Here is a visual look at why we need pair of sin/cos for each cyclic component for ensuring uniqueness and capturing cyclicity at the same time.



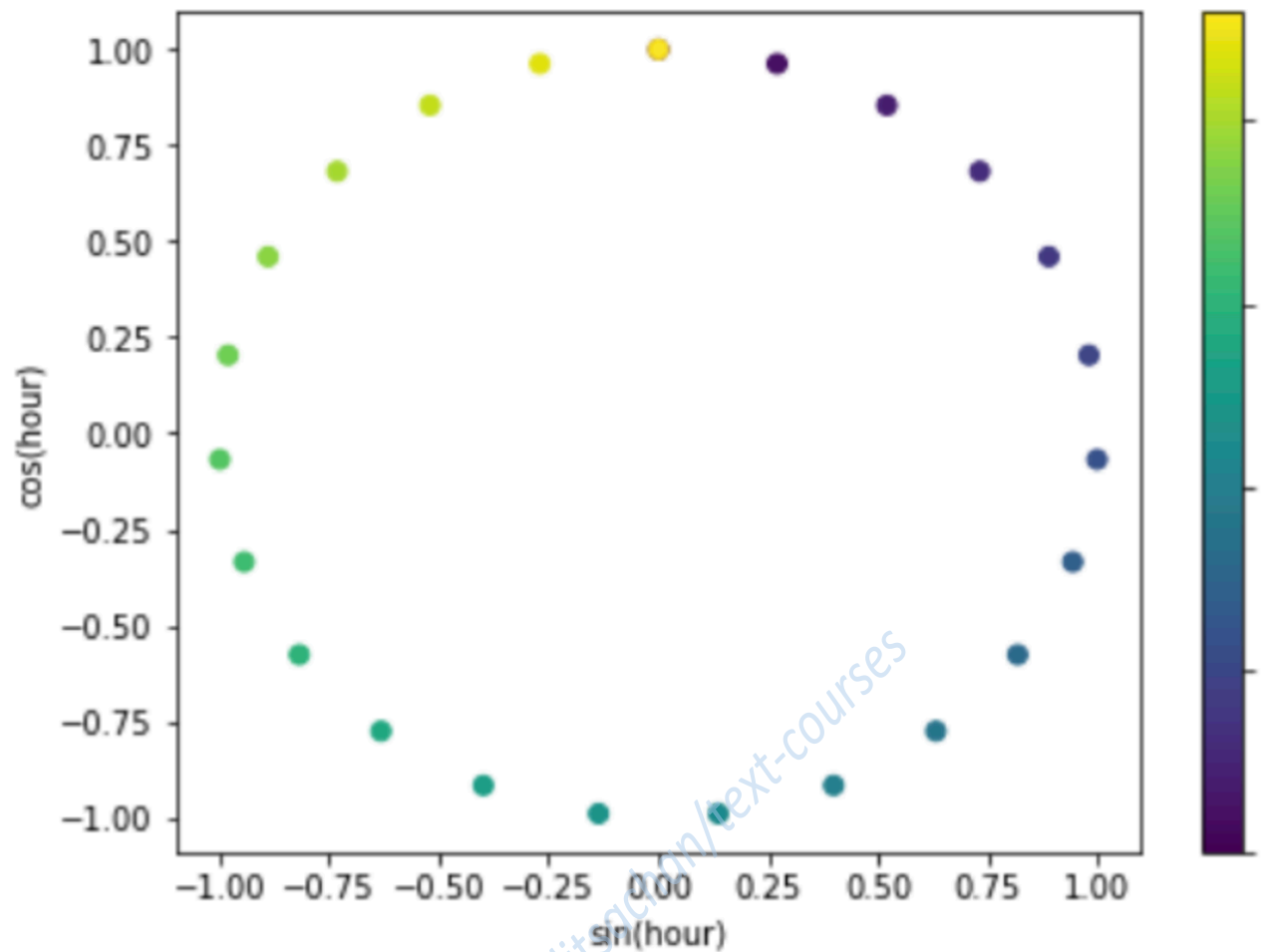
source: [cyclic features](https://github.com/lalitsachan/text-courses)

you can see that cyclicity of hours is captured and hour 0 and 22 are not `22` units apart anymore numerically. However this fails to give unique representation, you can see many hour values having same numeric value on the y-axis in the figure.



source: [cyclic features](#)

However if you consider pair of sin/cos both , it not only captures cyclicity but gives a unique representation for each occurrence .



source: [cyclic features](https://github.com/lalitsachan/text-courses)

Information in the column : Text , Data Type of the Column: Object

We call a column text data when its more verbose kind such as email, sms, tweet, news articles, reviews etc . Where it kinda stops to make sense to call it categorical data. Each observation is unique in itself. As far as tabular numeric representation is concerned, this data is represented as frequency of the words which appear in the data, here is an example :

Document ID	Review
1	The movie was great.
2	The movie was not great, not great at all.
3	I did not like the movie.
4	The plot was great, and the movie was great too.
5	Great plot but the movie was not for me.

Unique words in the data [known as `vocabulary` of the data] are as follows :
 "the", "movie", "was", "great", "not", "i", "did", "like", "plot", "and", "but", "for", "me". Simple `CountVectorizer` from sklearn makes word frequency table like this :

Document	the	movie	was	great	not	i	did	like	plot	and	but	for	me
The movie was great.	1	1	1	1	0	0	0	0	0	0	0	0	0
The movie was not great, not great at all.	1	1	1	2	2	0	0	0	0	0	0	0	0
I did not like the movie.	1	1	0	0	1	1	1	1	0	0	0	0	0
The plot was great, and the movie was great too.	1	1	2	2	0	0	0	0	1	1	0	0	0
Great plot but the movie was not for me.	1	1	1	1	1	0	0	0	1	0	1	1	1

I have kept the original column for your reference. Now issue with this approach is that, it ends up emphasizing words which commonly appear everywhere in the data. Words which appear everywhere are not good differentiators between observations and hence not very useful from the perspective of ML models. Another approach known as TF-IDF tries to solve this problem by penalizing values for the words which appear too often through out the data across observations. TF-IDF is made up of two terms :

Term Frequency :

$$TF(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Inverse Document Frequency

$$IDF(t, D) = \log \left(\frac{N}{|\{d \in D : t \in d\}|} \right)$$

combining them we get

$$TF-IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

Lets look at these calculations for our small dataset. Starting with TF:

Document	the	movie	was	great	not	i	did	like	plot	and	but	for	me
The movie was great.	1/4	1/4	1/4	1/4	0	0	0	0	0	0	0	0	0
The movie was not great, not great at all.	1/9	1/9	1/9	2/9	2/9	0	0	0	0	0	0	0	0
I did not like the movie.	1/6	1/6	0	0	1/6	1/6	1/6	1/6	0	0	0	0	0
The plot was great, and the movie was great too.	1/10	1/10	2/10	2/10	0	0	0	0	1/10	1/10	0	0	0
Great plot but the movie was not for me.	1/9	1/9	1/9	1/9	1/9	0	0	0	1/9	0	1/9	1/9	1/9

and IDF :

- "the" appears in 5 documents: $IDF(\text{the}) = \log\left(\frac{5}{5}\right) = 0$
- "movie" appears in 5 documents: $IDF(\text{movie}) = \log\left(\frac{5}{5}\right) = 0$
- "was" appears in 4 documents: $IDF(\text{was}) = \log\left(\frac{5}{4}\right) = 0.223$
- "great" appears in 4 documents: $IDF(\text{great}) = \log\left(\frac{5}{4}\right) = 0.223$
- "not" appears in 3 documents: $IDF(\text{not}) = \log\left(\frac{5}{3}\right) = 0.510$
- "i", "did", "like", "plot", "and", "but", "for", "me" each appear in 1 or 2 documents with $IDF = \log\left(\frac{5}{1}\right) = 0.698$

Combining them we get TF-IDF values for the data :

Document	the	movie	was	great	not	i	did	like	plot	and	but	for	me
The movie was great.	0	0	0.0558	0.0558	0	0	0	0	0	0	0	0	0
The movie was not great, not great at all.	0	0	0.0247	0.0496	0.1133	0	0	0	0	0	0	0	0
I did not like the movie.	0	0	0	0	0.085	0.1163	0.1163	0.1163	0	0	0	0	0
The plot was great, and the movie was great too.	0	0	0.0446	0.0446	0	0	0	0	0.0698	0.0698	0	0	0
Great plot but the movie was not for me.	0	0	0.0247	0.0247	0.0567	0	0	0	0.0775	0	0.0775	0.0775	0.0775

Usually this can make the data size explode, a single text column can end up creating hundreds or thousands of columns. This can be controlled in some ways:

- Set some cutoff on number of columns being created[`max_features=some_number`]. This cutoff removes the columns for the words which appear lower in the descending frequency count.
- Set cutoff on min appearance across document and maximum appearance across document. For example , if you set `min_df=0.1` , it means that only those words are eligible for getting a feature column for themselves which appear in at least `10%` of the observations. Similarly if you set `max_df=0.8` , words which appear in more than `80%` of the documents , they are removed from the vocabulary being considered .
- Instead of waiting for TF-IDF to correct for the commonly occurring words, explicitly remove such words from the data before even starting the process. A common name for such set of words is `stop_words` . sklearn comes with its own set of english stop words which you can use or you can define your own set of `stop_words` and use that in the function call.

Pipelines

So far we have been looking at ideas of processing different categories of columns. We need some framework which binds all of these processing together and outputs a dataframe where all the columns have been taken care of. There are multiple distinct steps which bound together will make pipelines for these different categories of processing. `sklearn` has a specific class structure for these classes . We will pick an example data frame, see what kind of processing it needs across different categories of columns and we will code the required classes and further combine them into proper pipelines .

Lets have a look at our example data first. you can also follow this along in the shared accompanying notebook and py file. We are going to present the final results of our decisions , you can explore the individual columns in the notebook to understand rationale for these decisions

```
1 ld_train=pd.read_csv("./loan_data_train.csv")
2 ld_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 15 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   ID                                     2199 non-null   float64
1   Amount.Requested                     2199 non-null   object
2   Amount.Funded.By.Investors           2199 non-null   object
3   Interest.Rate                        2200 non-null   object
4   Loan.Length                          2199 non-null   object
5   Loan.Purpose                           2199 non-null   object
6   Debt.To.Income.Ratio                 2199 non-null   object
7   State                                2199 non-null   object
8   Home.Ownership                       2199 non-null   object
9   Monthly.Income                       2197 non-null   float64
10  FICO.Range                           2200 non-null   object
11  Open.CREDIT.Lines                    2196 non-null   object
12  Revolving.CREDIT.Balance             2197 non-null   object
13  Inquiries.in.the.Last.6.Months       2197 non-null   float64
14  Employment.Length                   2131 non-null   object
dtypes: float64(3), object(12)
memory usage: 257.9+ KB
```

The dataset that we are going to work on comes from a peer to peer lending firm . The usual process at the firm is that people apply for a loan and then firm presents their application to the peers who collectively decide what fraction of the loan will be funded and at what interest rate. Firm is looking to streamline this process by proposing an indicative interest rate to the peers for each application, they want this proposed interest rate to be similar to what peers would have come up with on their own. To do that they want to model interest rate as function of the loan application received . We have been given historical data for the same. One caveat there , which should be a lesson for you for other business problems too. Usually the historical data that you receive for modeling , also contains features which might not be actually available at the process point where the eventual model will be deployed . For example in our data we have been given `Amount Funded By Investors` as a column, clearly this information will not be available before the peers have had a chance to look at the application, this certainly will not be part of the loan application. You need to be cautious and make sure that you do not use such information in the modeling process.

These are the decisions that we have taken for processing different columns [explore them individually in the notebook to understand rationale for these decisions]

```
1 Ignore : 'Interest.Rate', 'Amount.Funded.By.Investors','ID'
```

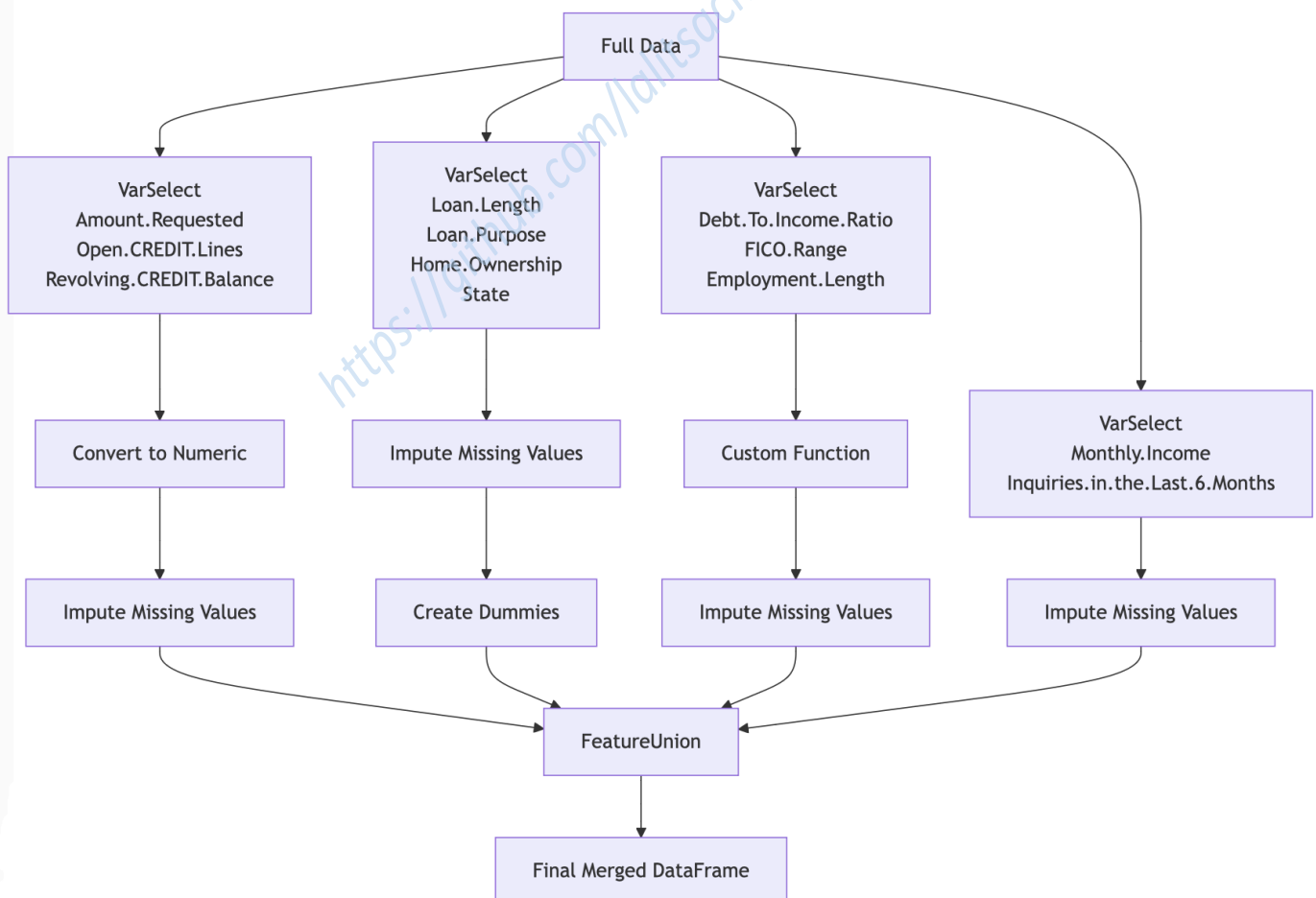
```
1 Amount.Requested : convert to numeric , impute missing values
2 Open.CREDIT.Lines: convert to numeric , impute missing values
3 Revolving.CREDIT.Balance : convert to numeric , impute missing values
```

```
1 Loan.Length : impute missing , create dummies
2 Loan.Purpose : impute missing , create dummies
3 Home.Ownership : impute missing , create dummies
4 State : impute missing , create dummies
```

```
1 Debt.To.Income.Ratio : custom func , impute missing
2 FICO.Range : custom func , impute missing
3 Employment.Length : custom func , impute missing
```

```
1 Monthly.Income : impute missing
2 Inquiries.in.the.Last.6.Months : impute missing
```

You will note that in all these groups there are many common steps that the columns need to go through .



- Each step class inherits from `BaseEstimator` and `TransformerMixin`

- First step in each pipeline receives the entire data as input
- Subsequent steps in the pipeline receive the input from the steps above them which comes from the transform function of the step above
- Eventually `FeatureUnion` combines data from all the pipelines

We will write reusable classes for these steps. General structure that `sklearn` works with looks like this:

```

1 class SomeDataProcess(BaseEstimator, TransformerMixin):
2
3     def __init__(self, arg1, arg2, ...):
4
5         attributes are created and assigned here
6
7     def fit(self, x):
8         x is the input data frame
9         learning from the training data happens here
10
11     def transform(self, x):
12         x is the input data frame
13         here we make changes in the data
14         and return transformed data
15
16     def get_feature_names(self):
17
18         this returns the names of
19         the columns going out of transform
20

```

Every class will have the same structure with the functions of the same name, you can add more functions if required, but these four with these functionalities need to remain as is.

Let's start with writing the first class in each pipeline that is `VarSelect`, there is nothing to learn from the data in this class, its only job is to pass on the specified columns as part of the transform.

```

1 class VarSelect(BaseEstimator, TransformerMixin):
2
3     def __init__(self, feature_names=[]):
4
5         self.feature_names=feature_names
6
7     def fit(self, x, y=None):
8
9         return self
10
11     def transform(self, x, y=None):
12
13         return x[self.feature_names]
14
15     def get_feature_names(self):
16
17         return self.feature_names

```

since the class receives all data as input and list of columns to select at the very beginning we can populate `feature_names` in the `__init__` function itself. It will not be so in other classes that we will write.

Next class that we will write is going to be for imputing missing values that we see in multiple pipelines .

```

1 class MissingImputation(BaseEstimator,TransformerMixin):
2
3     def __init__(self,impute_dict={}):
4
5         self.impute_dict=impute_dict
6         self.feature_names=[]
7
8     def fit(self,x,y=None):
9
10        self.feature_names=list(x.columns)
11        for col in x.columns:
12            if col in self.impute_dict:
13                pass
14            else:
15                if x[col].dtype=='O':
16                    self.impute_dict[col]='__missing__'
17                else :
18                    self.impute_dict[col]=x[col].median()
19        return self
20
21    def transform(self,x,y=None):
22
23        return x.fillna(self.impute_dict)
24
25    def get_feature_names(self):
26
27        return self.feature_names

```

with `impute_dict` in `__init__` we allow for a custom imputation for any column if required. In the `fit` function in the following section :

```

1 else:
2     if x[col].dtype=='O':
3         self.impute_dict[col]='__missing__'
4     else :
5         self.impute_dict[col]=x[col].median()

```

if the column is object type we decide the imputation value to be the string placeholder `__missing__` we discussed earlier . For the numeric column, we learn the median from training data for respective columns and store them in `impute_dict` . we are also storing `feature_names` in `fit` function itself, because in the pipeline , this is the first time this class will get to look at the data it receives from the earlier steps.

When it comes time to transform , we use the `impute_dict` that we learned from training set to fill missing values using `pandas.DataFrame.fillna` function which accepts dictionary of the form `{ 'col_name':fill_value }` for filling missing values.

Another class that I would like to discuss here is for creating dummies :

```

1 class CreateDummies(BaseEstimator,TransformerMixin):
2
3     def __init__(self,freq_percent_cutoff=0.01):
4
5         self.freq_percent_cutoff=freq_percent_cutoff
6         self.var_cat_dict={}
7         self.feature_names=[ ]
8
9     def fit(self,x,y=None):
10
11         self.freq_cutoff=x.shape[0]*self.freq_percent_cutoff
12
13         for col in x.columns:
14
15             col_data=x[col].copy()
16             k=col_data.value_counts()
17             cats_to_be_clubbed=list(k.index[k<=self.freq_cutoff])
18
19             if len(cats_to_be_clubbed)>1:
20
21                 col_data=col_data.replace(dict.fromkeys(cats_to_be_clubbed,'__other__'))
22                 k=col_data.value_counts()
23
24                 selected_cats=list(k.index[:-1])
25
26                 self.var_cat_dict[col]=[cats_to_be_clubbed,selected_cats]
27                 self.feature_names.extend([col+'_'+str(cat) for cat in selected_cats])
28
29             return self
30
31     def transform(self,x,y=None):
32
33         out_data={}
34
35         for col in self.var_cat_dict.keys():
36
37             cats_to_be_clubbed=self.var_cat_dict[col][0]
38             selected_cats=self.var_cat_dict[col][1]
39             out_data[col]=x[col].copy()
40
41             if len(cats_to_be_clubbed)>1:
42
43                 out_data[col]=out_data[col].replace(dict.fromkeys(cats_to_be_clubbed,'__other__'))
44
45
46                 for cat in selected_cats:
47
48                     out_data[col+'_'+str(cat)]=(out_data[col]==cat).astype(int)
49
50                 del out_data[col]
51
52         return pd.DataFrame(out_data)

```

```

53
54     def get_feature_names(self):
55
56         return self.feature_names
57

```

lets look sections of this code one by one to understand what we are doing here:

```

1 k=col_data.value_counts()
2 cats_to_be_clubbed=list(k.index[k<=self.freq_cutoff])

```

`col_data.value_counts()` returns frequency counts of all categories present in the data and since missing value imputation comes before it, if there were any missing values in the data, now they are represented by the category `__missing__` after imputation.

`k.index[k<=self.freq_cutoff]` selects all the categories which have frequencies lesser than the frequency cutoff that we decided, these categories will get clubbed into a new category `__other__` which we were referring to as `__low_freq_cat__` in our earlier discussion about this.

```

1 if len(cats_to_be_clubbed)>1:
2
3     col_data=col_data.replace(dict.fromkeys
4     (cats_to_be_clubbed, '__other__'))
5     k=col_data.value_counts()

```

its not necessary that we will always have low frequency categories in the data, it makes sense to club categories if we have more than 1 such low frequency categories in the data, thats why the condition `if len(cats_to_be_clubbed)>1`.

when we do have more than 1 categories, say `cats_to_be_clubbed` is `['cat1', 'cat2']`, then this part: `dict.fromkeys (cats_to_be_clubbed, '__other__')` will output a dictionary like this: `{'cat1': '__other__', 'cat2': '__other__'}`, which when used by `col_data.replace` will replace occurrences of `cat1` and `cat2` in the column with `__other__`.

`k=col_data.value_counts()` this step then recalculates the frequency counts after clubbing of low frequency categories into `__other__`.

`selected_cats=list(k.index[:-1])` selects top `n-1` categories from this new frequency count. This is what we will create the dummies for. All of this information has been gathered from the training data, we will need to store this so that we can use that at the time of transforming the data.

```

1 self.var_cat_dict[col]=[cats_to_be_clubbed,
2                         selected_cats]

```

this part here is storing information about which categories to be clubbed and then which categories to create dummy for against the respective column names in the dictionary `var_cat_dict` to be used at the time for transforming the data.

```

1 self.feature_names.extend([col+'_'+str(cat)
2                             for cat in selected_cats])

```


each category that we will need to create dummies for will have separate columns in the data, and their name will be for the form `col_name_cat`. This section here is creating the list of those names and adding to `feature_names` sequentially. this will be returned from function `get_feature_names`

```
1 cats_to_be_clubbed=self.var_cat_dict[col][0]
2 selected_cats=self.var_cat_dict[col][1]
3 out_data[col]=x[col].copy()
```

in transform function this bit extracts the information on categories to be clubbed and categories to create dummies for and carries out the operations on the column similar to what we did in fit. We store that column in the dictionary `out_data` which we will convert to a data frame at the time of return. we did this because making changes in the data frame itself raises warnings.

```
1 for cat in selected_cats:
2
3     out_data[col+'_'+str(cat)]=(out_data[col]==cat
4                                 ).astype(int)
5
6     del out_data[col]
7
8 return pd.DataFrame(out_data)
```

in transform function for each column we go through each category and for it create a column named `col_cat` in `out_data[col+'_'+str(cat)]`. we use `str(cat)` in creating them because category in some cases can be a numeric value which will throw error for concatenation.

`out_data[col]==cat` here will create a boolean array containing `True`, `False` whenever that column takes this category as a value. applying `.astype(int)` on top of this boolean array converts `True` to `1` and `False` to `0`, the form in which we want to store data in that column.

once we have processed all the categories of that column for which we needed to create dummies, we delete that column from the dictionary with `del out_data[col]`. Once we have processed all the columns in this way, we convert the dictionary into a data frame and return the data with dummies using `return pd.DataFrame(out_data)`.

There are many more classes that i have included in the file `ml_utils.py` file which enable processing of different kind of datasets.

we can combine these classes using `sklearn's Pipeline` to create a pipeline for any given group of features like so:

```
1 Pipeline([
2     ('var_select', VarSelect(cat_to_dummies)),
3     ('missing_trt', MissingImputation()),
4     ('create_dummies', CreateDummies())
5 ])
```

this is an example of a pipeline for the features we want to create dummies for . However there is one issue with default `Pipeline` implementation that it doesn't contain function `get_feature_names`. To circumvent that we have created our own custom pipeline class named `pdPipeline` which inherits from `Pipeline`, only addition that it makes is the inclusion of function `get_feature_names`.

```
1 class pdPipeline(Pipeline):  
2  
3     def get_feature_names(self):  
4  
5         last_step=self.steps[-1][-1]  
6  
7         return last_step.get_feature_names()
```

What this function does is that it calls the last step in the pipeline and uses its `get_feature_names` function, and since we have implemented `get_feature_names` in all our step classes, now `get_feature_names` function will work on all our `pdPipeline` objects.

All the pipelines objects can then be combined using `FeatureUnion` from `sklearn`. Our class `DataPipe` in `ml_utils.py` does this internally for each group of columns from the data and all that you have to do is to create these groups of columns in the data and pass as inputs to this class.

Please go through the accompanying notebook to better understand the usage of the same for any tabular dataset that you need to work with. Also make sure that you look and try to understand `ml_utils.py` file to appreciate whats happening under the hood.