Easy Data Analysis, Visualization and Modeling using Datasist

Note: This tutorial originally prepared by Rising Odegua@towardsdatascience.com and slightly modified by Dr. Ali for educational purpose.

A few days ago (December 23, 2019), the first stable version of **datasist** was released. Datasist is a python library that makes data analysis, visualization, cleaning, preparation, and even modeling super easy.
The goal of datasist is to abstract repetitive codes, functions and techniques we use often into simple functions that can be called in one line of code.

The design of datasist is currently centered around 5 modules:

- structdata
- Feature Engineering
- timeseries
- visualization
- model

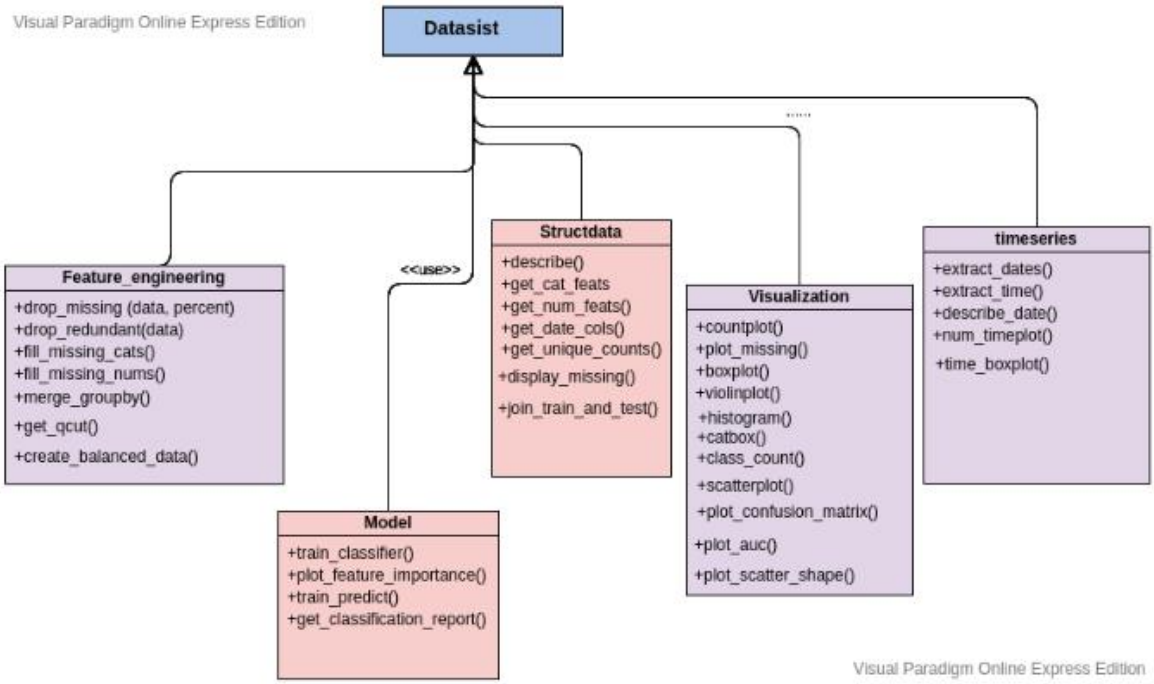


Figure: class diagram for datasist. Source: ArXiv

I have done the tutorial using Pycharm but you can use Jupyter notebook if you want. If you are using Jupyter, make sure to install the datasist library using the python pip manager. Open a terminal or a Jupyter notebook and type:

```
pip install datasist
```

Remember to use the exclamation symbol (!) before the command above if you're running it inside a Jupyter notebook.

Next, you need to get a dataset to experiment with. While you can use any dataset, for consistency, it is advisable to download the dataset we used for this tutorial. You can find it in Canvas or here.

Finally, create a project in Pycharm with a python file, import the following libraries and dataset as shown below. You should save all of your dataset in your project folder. (for Jupyter: open your Jupyter Notebook and import the following libraries and dataset as shown below:)

```
import pandas as pd
import datasist as ds  #import datasist library
import numpy as np
train_df = pd.read_csv('train_data.csv')
test_df = pd.read_csv('test_data.csv')
```

Working with the structdata module

The structdata module contains numerous functions for working with structured data mostly in the Pandas DataFrame format. That is, you can use these functions to easily manipulate and analyze DataFrames. We highlight some of the functions below.

1. **describe**: We all know the describe function in Pandas, well we decided to extend it to support full description of a dataset at a glance. Let's see this in action.

First try without using datasist. Simply using pandas.

```
print(train_df) #print all

print(train_df.head()) #print the first 5 rows

print(train_df.head(10))  # print the first n rows
```

Full description of a dataset at a glance using datasist.

```
print(ds.structdata.describe(train_df))

Note: If you are using Jupyter noterbook, you do not need to use
print function.
```

```
First five data points
  Customer Id  YearOfObservation  ...  Geo_Code  Claim
0      H14663               2013  ...      1053      0
1       H2037               2015  ...      1053      0
2       H3802               2014  ...      1053      0
3       H3834               2013  ...      1053      0
4       H5053               2014  ...      1053      0

[5 rows x 14 columns]


Random five data points
     Customer Id  YearOfObservation  ...  Geo_Code  Claim
4250       H6687               2013  ...     66136      0
6456      H19008               2014  ...     92044      0
4225      H16025               2014  ...     66136      0
96         H2641               2013  ...      5177      0
5307      H16704               2014  ...     76259      0

[5 rows x 14 columns]


Last five data points
     Customer Id  YearOfObservation  ...  Geo_Code  Claim
7155       H5290               2012  ...       NaN      0
7156       H5926               2013  ...       NaN      1
7157       H6204               2016  ...       NaN      0
7158       H6537               2013  ...       NaN      0
7159       H7470               2014  ...       NaN      0

[5 rows x 14 columns]
```

```
Shape of  data set: (7160, 14)


Size of  data set: 100240


Data Types
Note: All Non-numerical features are identified as objects in pandas
                  Data Type
Customer Id          object
YearOfObservation     int64
Insured_Period      float64
Residential           int64
Building_Painted     object
Building_Fenced      object
Garden               object
Settlement           object
Building Dimension  float64
Building_Type         int64
Date_of_Occupancy   float64
NumberOfWindows      object
Geo_Code             object
Claim                 int64


Numerical Features in Data set
['YearOfObservation', 'Insured_Period', 'Residential', 'Building Dimension', 'Building_Type', 'Date_of_Occupancy', 'Claim']


Categorical Features in Data set
['Customer Id', 'Building_Painted', 'Building_Fenced', 'Garden', 'Settlement', 'NumberOfWindows', 'Geo_Code']
```

```
Statistical Description of Columns
       YearOfObservation  Insured_Period  ...  Date_of_Occupancy        Claim
count        7160.000000     7160.000000  ...        6652.000000  7160.000000
mean         2013.669553        0.909758  ...        1964.456404     0.228212
std             1.383769        0.239756  ...          36.002014     0.419709
min          2012.000000        0.000000  ...        1545.000000     0.000000
25%          2012.000000        0.997268  ...        1960.000000     0.000000
50%          2013.000000        1.000000  ...        1970.000000     0.000000
75%          2015.000000        1.000000  ...        1980.000000     0.000000
max          2016.000000        1.000000  ...        2016.000000     1.000000

[8 rows x 7 columns]


Description of Categorical Features
                count unique   top  freq
Customer Id      7160   7160  H678     1
Building_Painted 7160      2     V  5382
Building_Fenced  7160      2     N  3608
Garden           7153      2     O  3602
Settlement       7160      2     R  3610
NumberOfWindows  7160     11     .  3551
Geo_Code         7058   1307  6088   143


Unique class Count of Categorical features
<pandas.io.formats.style.Styler object at 0x10101A50>
```

```
Missing Values in Data
                features  missing_counts  missing_percent
0            Customer Id               0              0.0
1     YearOfObservation                0              0.0
2        Insured_Period                0              0.0
3           Residential                0              0.0
4      Building_Painted                0              0.0
5       Building_Fenced                0              0.0
6                Garden                7              0.1
7            Settlement                0              0.0
8    Building Dimension              106              1.5
9         Building_Type                0              0.0
10    Date_of_Occupancy              508              7.1
11      NumberOfWindows                0              0.0
12             Geo_Code              102              1.4
13                Claim                0              0.0
None

Process finished with exit code 0
```

From the result above, you can have a full description of your dataset at a go and also properly understand some of the features all in one line (amazing right!).

2. check_train_test_set: This function is used to check the sampling strategy of two dataset. This is important because if two dataset are not from the same distribution, then feature extraction will be different because we can not apply calculations from the first dataset on the second.

To use this, you pass in the two dataset (**`train_df`** and **`test_df`**), a common index (`customer_id`) and finally any feature or column present in both dataset.

```
ds.structdata.check_train_test_set(train_df, test_df,
                                   index='Customer Id',
                                    col='Building Dimension')
```

```
There are 7160 training rows and 3069 test rows.
There are 14 training columns and 13 test columns.
Id field is unique.
Train and test sets have distinct Ids.
```

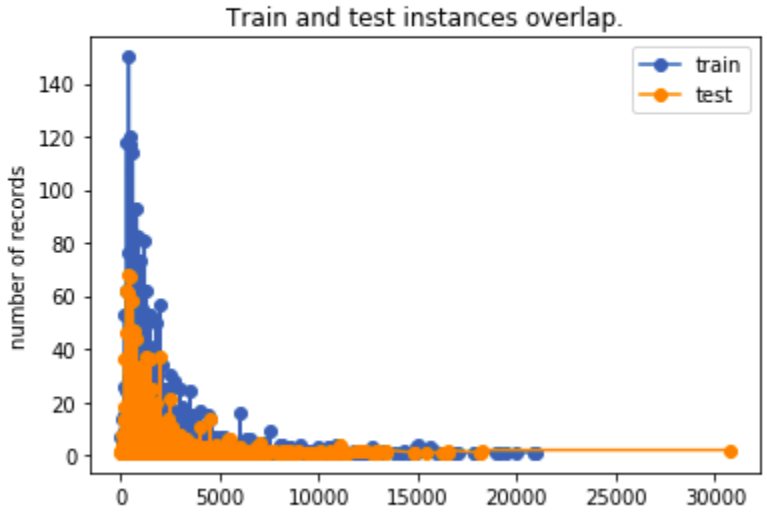


Figure: output from the check_train_test_set

3. display_missing: You can check for the missing values in your dataset and display the result in the well formatted DataFrame using this function.

```
print(ds.structdata.display_missing(train_df))
```

	features	missing_counts	missing_percent
0	Customer Id	0	0.0
1	YearOfObservation	0	0.0
2	Insured_Period	0	0.0
3	Residential	0	0.0
4	Building_Painted	0	0.0
5	Building_Fenced	0	0.0
6	Garden	7	0.1
7	Settlement	0	0.0
8	Building Dimension	106	1.5
9	Building_Type	0	0.0
10	Date_of_Occupancy	508	7.1
11	NumberOfWindows	0	0.0
12	Geo_Code	102	1.4
13	Claim	0	0.0

Figure: missing values in data set

4. get_cat_feats and **get_num_feats:** Just like their names, you can retrieve categorical and numerical features respectively as a Python list.

```
cat_feats = ds.structdata.get_cat_feats(train_df)
cat_feats
```

```
: ['Customer Id',
   'Building_Painted',
   'Building_Fenced',
   'Garden',
   'Settlement',
   'NumberOfWindows',
   'Geo_Code']
```

Figure: categorical features in data set

```
num_feats = ds.structdata.get_num_feats(train_df)
num_feats
```

```
: ['YearOfObservation',
   'Insured_Period',
   'Residential',
   'Building Dimension',
   'Building_Type',
   'Date_of_Occupancy',
   'Claim']
```

Figure: numerical features in data set

5. get_unique_counts: Ever wanted to check the number of unique classes in your categorical features before you decide what encoding scheme to use? well, you can use the ***get_unique_count*** function to easily do that.

```
unique_count=ds.structdata.get_unique_counts(train_df)
unique_count
```

	Feature	Unique Count
0	Customer Id	7160
1	Building_Painted	2
2	Building_Fenced	2
3	Garden	3
4	Settlement	2
5	NumberOfWindows	11
6	Geo_Code	1308

Figure: output from the get_unique_counts function

6. join_train_and_test: Most of the time when prototyping, you may want to concatenate both train and test set, and then apply some transformations to it. You can use the ***join_train_and_test*** function for that. It returns a concatenated dataset and the size of the train and test set for future splitting.

```
all_data, ntrain, ntest = ds.structdata.join_train_and_test(train_df, test_df)
print("New size of combined data {}".format(all_data.shape))
print("Old size of train data: {}".format(ntrain))
print("Old size of test data: {}".format(ntest))

#later splitting after transformations
train = all_data[:ntrain]
test = all_data[ntrain:]
```

```
New size of combined data (10229, 14)
Old size of train data: 7160
Old size of test data: 3069
```

Figure: output from combining the two data set and splitting

Those are some of the popular functions in the structdata module of datasist, to see other functions and to learn more about the parameters you can change, check the [API documentation here.]

Feature engineering with datasist.

Feature Engineering is the act of extracting important features from raw data and transforming them into formats that are suitable for machine learning models.

Some of the functions available in the ***feature_engineering*** module of datasist are explained below. NOTE: Functions in the feature_engineering module always return a new and transformed DataFrame. This means, you should assign the result to a variable as nothing happens inplace.

1. **drop_missing**: This function drops columns/features with a specified percentage of missing values. Assuming I have a set features with say greater than 80 percent missing values, and I want to drop these columns, I can easily do this with the ***drop_missing*** function.

First let's see the percentage of missing values in the dataset

```
print(ds.structdata.display_missing(train_df))
```

	features	missing_counts	missing_percent
0	Customer Id	0	0.0
1	YearOfObservation	0	0.0
2	Insured_Period	0	0.0
3	Residential	0	0.0
4	Building_Painted	0	0.0
5	Building_Fenced	0	0.0
6	Garden	7	0.1
7	Settlement	0	0.0
8	Building Dimension	106	1.5
9	Building_Type	0	0.0
10	Date_of_Occupancy	508	7.1
11	NumberOfWindows	0	0.0
12	Geo_Code	102	1.4
13	Claim	0	0.0

Figure: display of missing values in data set

Just for demonstration, we'll drop the column with 7.1 percent missing values (Date_of_Occupancy).

Note: *You should not drop a column/feature with so little missing values. The ideal thing to do is to fill it. We drop it here, just for demonstration purpose.*

```
new_train_df = ds.feature_engineering.drop_missing(train_df,
                                                   percent=7.0)
print(ds.structdata.display_missing(new_train_df))
```

```
Dropped ['Date_of_Occupancy']
```

	features	missing_counts	missing_percent
0	Customer Id	0	0.0
1	YearOfObservation	0	0.0
2	Insured_Period	0	0.0
3	Residential	0	0.0
4	Building_Painted	0	0.0
5	Building_Fenced	0	0.0
6	Garden	7	0.1
7	Settlement	0	0.0
8	Building Dimension	106	1.5
9	Building_Type	0	0.0
10	NumberOfWindows	0	0.0
11	Geo_Code	102	1.4
12	Claim	0	0.0

Figure: Output of dataset after using the drop_missing function

2**. drop_redundant:** This function is used to remove features with low or no variance. That is, features that contain the same value all through. We show a simple example using an artificial dataset.
First, let's create the data set…

```
df = pd.DataFrame({'a': [1,1,1,1,1,1,1], 'b': [2,3,4,5,6,7,8]})
print(df)
```

	a	b
0	1	2
1	1	3
2	1	4
3	1	5
4	1	6
5	1	7
6	1	8

Figure: sample dataset

Now looking at the dataset above, we see that column **a** is redundant, that is it has the same value all through. We can drop this column automatically by just passing in the dataset to the ***drop_redundant*** function.

```
df = ds.feature_engineering.drop_redundant(df)
print(df)
```

```
Dropped ['a']
```

	b
0	2
1	3
2	4
3	5
4	6
5	7
6	8

Figure: Result dataset from using the drop_redundant function

3. **convert_dtypes**: This function takes a DataFrame as argument and automatically type-cast the features to their correct data types.

Let's see an example. First we create an artificial dataset below.

```
data = {'Name':['Tom', 'nick', 'jack'],
        'Age':['20', '21', '19'],
        'Date of Birth': ['1999-11-17','20 Sept 1998','Wed Sep 19
                          14:55:02 2000']}
df = pd.DataFrame(data)
print(df)
```

	Name	Age	Date of Birth
0	Tom	20	1999-11-17
1	nick	21	20 Sept 1998
2	jack	19	Wed Sep 19 14:55:02 2000

Figure: sample data set

Next, let's check the data types…

```
print(df.dtypes)
```

```
Name            object
Age             object
Date of Birth   object
dtype: object
```

Figure: data types of features

The features Age and Date of Birth are supposed to be integer and DateTime respectively, by passing this dataset to the ***convert_dtype*** function, this can be fixed automatically.

```
df = ds.feature_engineering.convert_dtype(df)
print(df.dtypes)
```

```
: Name                    object
  Age                      int64
  Date of Birth    datetime64[ns]
  dtype: object
```

Figure: datatypes of features in artificial data set after using convert_dtypes function

4. **fill_missing_cats:** As the name implies, this function takes a DataFrame, and automatically detects categorical columns with missing values. It fills them using the mode. Let's see an example. From the dataset, we have two categorical features with missing values, these are Garden and Geo_Code.

```
df = ds.feature_engineering.fill_missing_cats(train_df)
print(ds.structdata.display_missing(df))
```

	features	missing_counts	missing_percent
0	Customer Id	0	0.0
1	YearOfObservation	0	0.0
2	Insured_Period	0	0.0
3	Residential	0	0.0
4	Building_Painted	0	0.0
5	Building_Fenced	0	0.0
6	Garden	0	0.0
7	Settlement	0	0.0
8	Building Dimension	106	1.5
9	Building_Type	0	0.0
10	Date_of_Occupancy	508	7.1
11	NumberOfWindows	0	0.0
12	Geo_Code	0	0.0
13	Claim	0	0.0

Figure: missing values DataFrame after filling the missing categorical features

5. **fill_missing_nums**: This is similar to the ***fill_missing_cats***, except it works on numerical features and you can specify a filling strategy (mean, mode or median). From the dataset, we have two numerical features with missing values, these are Building Dimension and Date_of_Occupancy.

```
df = ds.feature_engineering.fill_missing_num(train_df)
ds.structdata.display_missing(df)
```

:

	features	missing_counts	missing_percent
0	Customer Id	0	0.0
1	YearOfObservation	0	0.0
2	Insured_Period	0	0.0
3	Residential	0	0.0
4	Building_Painted	0	0.0
5	Building_Fenced	0	0.0
6	Garden	7	0.1
7	Settlement	0	0.0
8	Building Dimension	0	0.0
9	Building_Type	0	0.0
10	Date_of_Occupancy	0	0.0
11	NumberOfWindows	0	0.0
12	Geo_Code	102	1.4
13	Claim	0	0.0

Figure: missing values DataFrame after filling the missing numerical features

6. **log_transform**: This function can help you log transform a set of features. It also display the before and after plot with level of skewness to help you decide if log transforming feature is what you really want. The feature B*uilding Dimension* is a right skewed, and we can transform it.

Note: Columns passed to the log_transform function should not contain missing values, else it will throw an error.

```
df = ds.feature_engineering.fill_missing_num(df)
df = ds.feature_engineering.log_transform(df,
                                          columns=['Building Dimension'])
```

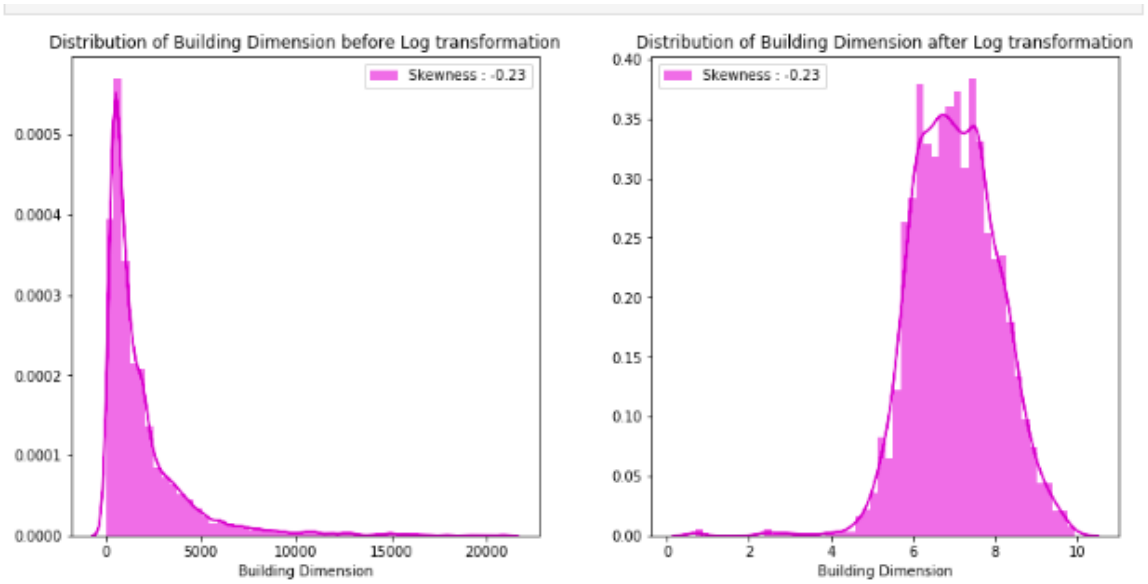


Figure: Log-transform plot

To work with geo features like *latitude* and *longitude*, datasist has dedicated functions like **bearing**, **manhattan_distance**, **get_location_center** etc, also available in the feature_engineering module. To see these and other functions, you can check the API documentation here.

WORKING WITH TIME BASED FEATURES

Finally in this part, we'll talk about the timeseries module in datasist. The timeseries module contains functions for working with date time features. It can help you extract new features from Dates and also help you visualize them.

1. extract_dates: This function can extract specified features like day of the week, day of the year, hour, min and second of the day from a specified date feature. To demonstrate this, let's use a dataset that contains Date feature.
Get the new dataset from canvas or from **here**.

```
new_train= pd.read_csv("Train.csv")
print(new_data.head(3).T)
```

	0	1	2
Order No	Order_No_4211	Order_No_25375	Order_No_1899
User Id	User_Id_633	User_Id_2285	User_Id_265
Vehicle Type	Bike	Bike	Bike
Platform Type	3	3	3
Personal or Business	Business	Personal	Business
Placement - Day of Month	9	12	30
Placement - Weekday (Mo = 1)	5	5	2
Placement - Time	9:35:46 AM	11:16:16 AM	12:39:25 PM
Confirmation - Day of Month	9	12	30
Confirmation - Weekday (Mo = 1)	5	5	2
Confirmation - Time	9:40:10 AM	11:23:21 AM	12:42:44 PM
Arrival at Pickup - Day of Month	9	12	30
Arrival at Pickup - Weekday (Mo = 1)	5	5	2
Arrival at Pickup - Time	10:04:47 AM	11:40:22 AM	12:49:34 PM
Pickup - Day of Month	9	12	30
Pickup - Weekday (Mo = 1)	5	5	2
Pickup - Time	10:27:30 AM	11:44:09 AM	12:53:03 PM
Arrival at Destination - Day of Month	9	12	30
Arrival at Destination - Weekday (Mo = 1)	5	5	2
Arrival at Destination - Time	10:39:55 AM	12:17:22 PM	1:00:38 PM
Distance (KM)	4	16	3
Temperature	20.4	26.4	NaN
Precipitation in millimeters	NaN	NaN	NaN
Pickup Lat	-1.31775	-1.35145	-1.30828
Pickup Long	36.8304	36.8993	36.8434
Destination Lat	-1.30041	-1.295	-1.30092
Destination Long	36.8297	36.8144	36.8282
Rider Id	Rider_Id_432	Rider_Id_856	Rider_Id_155
Time from Pickup to Arrival	745	1993	455

Figure: Transposed head of the dataset

The dataset contains lots of time features which we can analyse. More details about the task can be found on the competition page.

Let's demonstrate how easy it is to extract information from **Placement — Time**, **Arrival at Destination — Time** features using the extract_dates function. We specify that we want to extract just day of the week (dow) and hour of the day (hr).

```
date_cols = ['Placement - Time', 'Arrival at Destination - Time']
df = ds.timeseries.extract_dates(new_train, date_cols=date_cols,
subset=['dow', 'hr')
print(df.head(3).T)
dff=pd.DataFrame(df)#create a new dataframe to save in csv file
dff.to_csv('Train_modified.csv')# save in a csv file
```

	0	1	2
Order No	Order_No_4211	Order_No_25375	Order_No_1899
User Id	User_Id_633	User_Id_2285	User_Id_265
Vehicle Type	Bike	Bike	Bike
Platform Type	3	3	3
Personal or Business	Business	Personal	Business
Placement - Day of Month	9	12	30
Placement - Weekday (Mo = 1)	5	5	2
Confirmation - Day of Month	9	12	30
Confirmation - Weekday (Mo = 1)	5	5	2
Confirmation - Time	9:40:10 AM	11:23:21 AM	12:42:44 PM
Arrival at Pickup - Day of Month	9	12	30
Arrival at Pickup - Weekday (Mo = 1)	5	5	2
Arrival at Pickup - Time	10:04:47 AM	11:40:22 AM	12:49:34 PM
Pickup - Day of Month	9	12	30
Pickup - Weekday (Mo = 1)	5	5	2
Pickup - Time	10:27:30 AM	11:44:09 AM	12:53:03 PM
Arrival at Destination - Day of Month	9	12	30
Arrival at Destination - Weekday (Mo = 1)	5	5	2
Distance (KM)	4	16	3
Temperature	20.4	26.4	NaN
Precipitation in millimeters	NaN	NaN	NaN
Pickup Lat	-1.31775	-1.35145	-1.30828
Pickup Long	36.8304	36.8993	36.8434
Destination Lat	-1.30041	-1.295	-1.30092
Destination Long	36.8297	36.8144	36.8282
Rider Id	Rider_Id_432	Rider_Id_856	Rider_Id_155
Time from Pickup to Arrival	745	1993	455
Placement - Time_dow	Sunday	Sunday	Sunday
Placement - Time_hr	9	11	12
Arrival at Destination - Time_dow	Sunday	Sunday	Sunday
Arrival at Destination - Time_hr	10	12	13

Figure: output from extract_dates function

We can see that datasist created new columns and appended the new features to the dataset.

2. timeplot: The timeplot function can help you visualize a set features against a particular time feature. This can help you identify trends and patterns. To use this function, you can pass a set of numerical columns, and then specify the Date feature you want to plot against.

```
num_cols = ['Time from Pickup to Arrival', 'Destination Long',
            'Pickup Long','Platform Type',
'Temperature']ds.timeseries.timeplot(new_train, num_cols=num_cols,
                        time_col='Placement - Time')
```

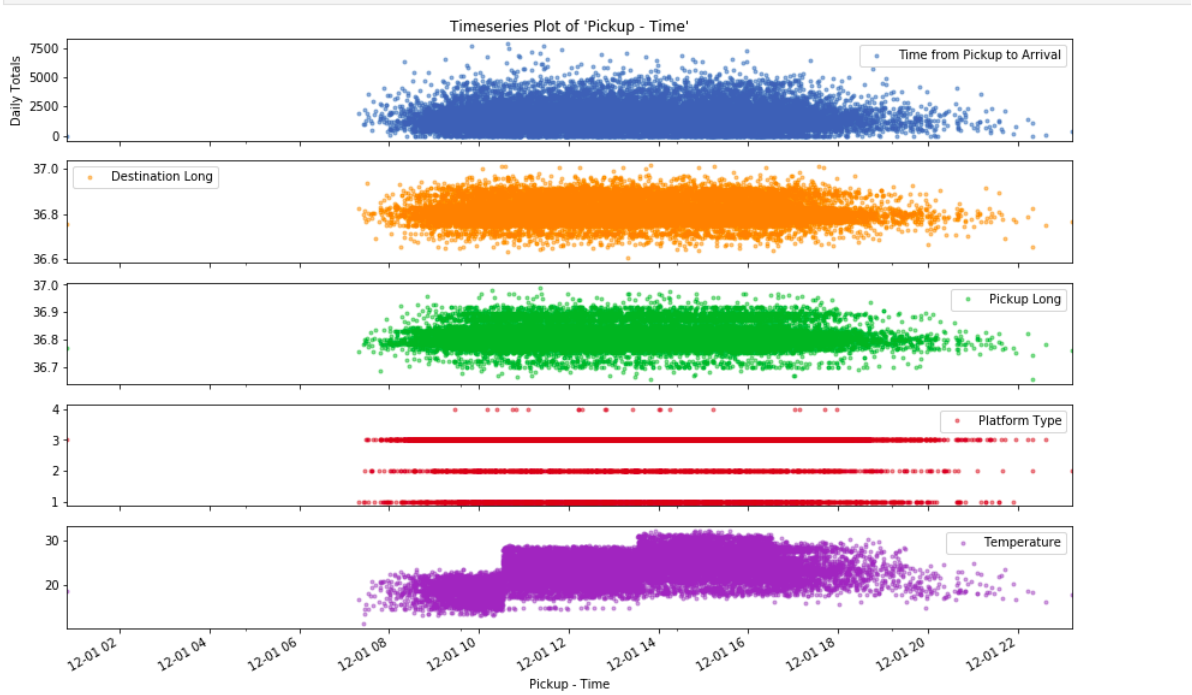


Figure: output of timeplot

And with that, we have come to the end of part 1. To learn more about datasist and other functions available, you can check the **API documentation here.**

In the remaining part we will discuss about Visualization (Visualization for categorical features and Visualization for numerical features.) and Testing and comparing machine learning models with datasist.

Easy visualization using datasist

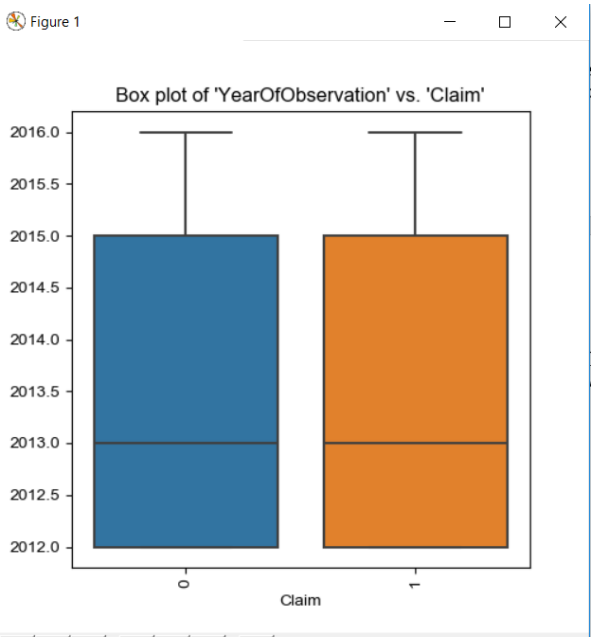
The visualization module is one of the best modules in datasist. There are lots of functions that you can use to create aesthetic and colorful plots with minimal codes. All functions in the visualization module works at data scale not feature scale. This means, you can pass in the full dataset and get visualization for every feature out of the box. You can also specify the features you want to plot.

VISUALIZATION FOR CATEGORICAL FEATURES

Visualization for categorical features include plots like bar charts, count plots, etc.

1. boxplot: This function makes a box plot of all numerical features against a specified categorical target column. A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. *Note: You can save any plot produced by datasist as a .png file in the current working directory by setting the* ***save_fig*** *parameter to* ***True.***

```
ds.visualizations.boxplot(train_df, target='Claim')
```



……..you will see boxplot for all features vs Claim

2. **catbox:** The catbox feature is used to make a side by side bar plot of all categorical features in a dataset against a specified categorical target. This can help in identifying causation and patterns and also identifying features that separates the target properly. ***Note:*** *catbox would only plot categorical feature with a limited number of unique classes.*

```
ds.visualizations.catbox(train_df, target='Claim')# try in Jupyter
```

3. **countplot**: The countplot simply makes a barplot of all categorical feature to show their class count.

```
ds.visualizations.countplot(train_df)# try in Jupyter
```

VISUALIZATION FOR NUMERICAL FEATURES

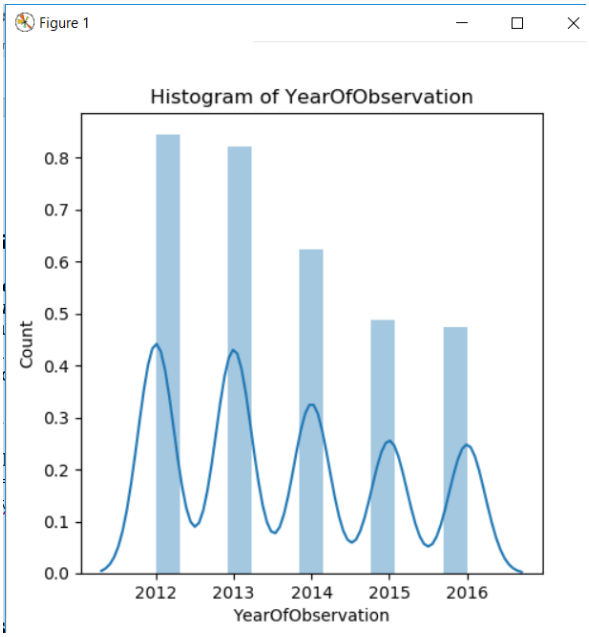
Visualization for numerical features include plots like scatter plot, histogram, kde plots etc. We can use the functions available in datasist to easily do this at data wide level. Let's review some of them below.

1. histogram: This function makes a histogram plot of all numerical features in a dataset. This helps to show distribution of the features. ***Note:*** *To use this function, the specified features to plot must not contain missing values, else it would throw an error.*

In our example below, the features `Building Dimension` and `Date_of_Occupancy` both contain missing values. We could decide to fill these features before plotting or we could pass in a list of features we want.

we'll go with the first option, that is, filling the missing values.

```
#fill the missing values
df = ds.feature_engineering.fill_missing_num(train_df)
ds.visualizations.histogram(df)
```



You will also see the remain histogram

2. **scatterplot**: This function makes a scatter plot of all numerical features in a dataset against a specified numerical target. It helps to show the correlation between features.

```
feats = ['Insured_Period', 'Residential', 'Building Dimension',
         'Building_Type',     'Date_of_Occupancy']
ds.visualizations.scatterplot(train_df,num_features=feats,
target='Building Dimension',save_fig='ali.png')
```

3. plot_missing: As the name implies, this function can be used to visualize the missing values in a dataset. White cells indicate missing and dark cells indicate not missing. The color range at the right hand corner shows missing intensity

6. **autoviz**: Autoviz is the ultimate lazy man's visualization function. With this function, you can visualize any dataset with just a single line of code. To use autoviz in datasist, you first have to install the autoviz package. To install via pip, use the following command.

```
pip install autoviz
```

Now, let's demonstrate how to use autoviz in datasist.

```
ds.visualizations.autoviz(train_df)
```

Testing and comparing machine learning models with datasist

The ***model*** module contains functions for testing and comparing machine learning models. Current version of datasist only supports scikit-learn models. First, we'll get a dataset from the Data Science Nigeria, 2019 Boot-camp competition page **here**. The task is to predict insurance claim (1=Claim, 0=No Claim) from building observations. We will do some basic data pre-processing and prepare the data for modeling.

Note: The goal of this analysis is to demonstrate how to use the model module in datasist, so we would not be doing any heavy feature engineering.

```
import pandas as pd
import numpy as np
import datasist as ds

train = pd.read_csv('train_data.csv')
#test = pd.read_csv('test_data.csv')
```

Next, we do some processing. First, we drop the ID column (Customer Id) and then we fill missing numerical and categorical features.

```
#drop the id column
train.drop(columns='Customer Id', axis=1, inplace=True)
#test.drop(columns='Customer Id', axis=1, inplace=True)

#fill missing values
train = ds.feature_engineering.fill_missing_cats(train)
train = ds.feature_engineering.fill_missing_num(train)

#test = ds.feature_engineering.fill_missing_cats(test)
#test = ds.feature_engineering.fill_missing_num(test)
```

Now that we have a properly filled dataset, we'll encode all categorical features using either label encoding, or one hot encoding depending on the number of unique classes.

```
#check the unique classes in each categorical feature
ds.structdata.class_count(train)
```

We will label encode Geo_Code, since the unique classes is large, and one-hot-encode the rest.

```
import category_encoders as ce

# drop target column
target = train['Claim'].values
train.drop(columns='Claim', axis=1, inplace=True)

enc = ce.OrdinalEncoder(cols=['Geo_Code'])
enc.fit(train)
train_enc = enc.transform(train)
#test_enc = enc.transform(test)


#one-hot-encode the rest categorical features
hot_enc = ce.OneHotEncoder()
hot_enc.fit(train_enc)
train_enc = hot_enc.transform(train_enc)
#test_enc = hot_enc.transform(test_enc)
```

Install category_encoders: `pip install categority_encoders`

Now we will split the to create train and testing samples from train_data.csv. We are going to evaluate random forest model first.

```


from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split


Xtrain, Xtest, ytrain, ytest = train_test_split(train_enc, target, test_size=0.3,
random_state=1)
rf_classifier = RandomForestClassifier(n_estimators=20, max_depth=4)
rf_model=rf_classifier.fit(Xtrain,ytrain)
pred=rf_model.predict(Xtest)
ds.model.get_classification_report(pred,ytest)
```

Now for example we want to compare three ML models such as naive bayes, decision trees and svm. compare_model: This model takes as argument multiple machine learning models and returns a plot showing each model's performance.

Now, let's see this function in action. We will be comparing three models

```
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import LinearSVC

Xtrain, Xtest, ytrain, ytest = train_test_split(train_enc, target, test_size=0.3,
random_state=1)
rf_classifier = RandomForestClassifier(n_estimators=20, max_depth=4)
nb_classifier = GaussianNB()
svm_classifier = LinearSVC()
classifiers = [rf_classifier, nb_classifier, svm_classifier]
models, scores = ds.model.compare_model(models_list=classifiers, x_train=Xtrain,
y_train=ytrain, scoring_metric='accuracy')
```